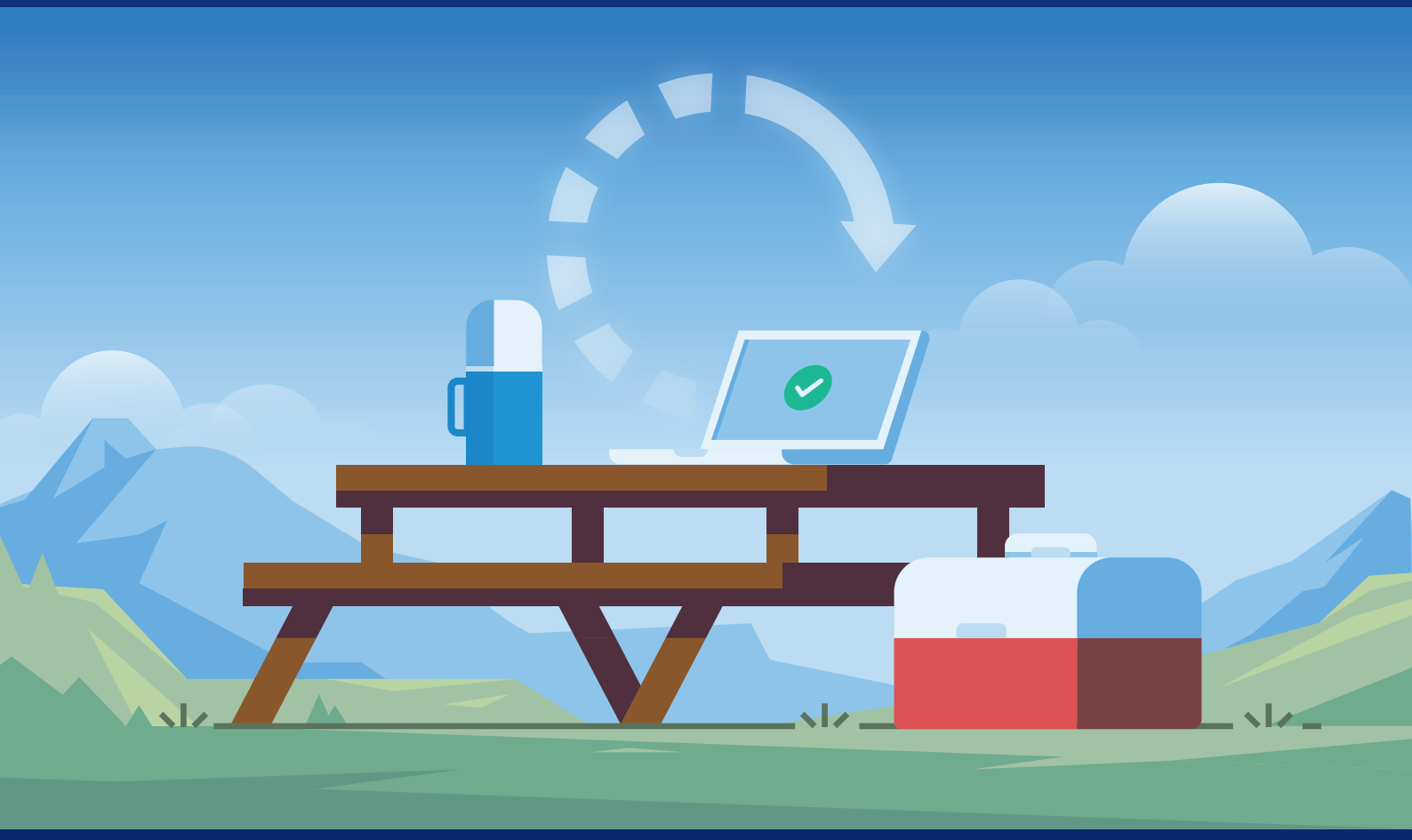
Field Guide to Progressive Delivery + Experimentation



split

by the team at Split.io

A Field Guide to Progressive Delivery and Experimentation

A guide to the terms and concepts you need to build a modern feature delivery and experimentation program

Brought to you by
The Team at Split Software

A Field Guide to Progressive Delivery and Experimentation

by Split Software

Published by Split Software 10 California St.,
Redwood City, CA 94063

ISBN: 978-1-312-78266-2

ID ye9qzp

First Edition

Table of Contents

Part I

Feature Flags and Progressive Delivery

Blue/Green Deployment

Blue/green deployment is a continuous deployment process that reduces downtime and risk by having two identical production environments, called blue and green. The names blue and green aren't special or important – this process is can also be referred to as red/black deployment or A/B deployment.

Let's consider a scenario, say the blue environment is active, while the green is idle. When a developer wants to release new code of any variety – a new feature release, a new version of the application, etc. – the work on the new version is done in the green environment while the old version is maintained in the blue. Once the new release isfinished, the load balancer switches all production traffic to the green version, and the blue version is maintained as a backup.

After the green version is live for a while and all data indicates it is bug-free, performing well, and driving the intended impact, the old blue version is scrapped, the currently-live version becomes the blue, and a new production environment clone is created to become the new green.

Benefits of Blue/Green Deployment

The major benefit of blue/green deployment is that it facilitates simple rollouts, quick rollbacks, and easy disaster recovery.

Have you ever had to deploy a feature release at an insane hour because that was the only time you could take down the system without losing sales? Or maybe you've had a hard time finding any time to release because your business is global enough that the middle of the night in one place is prime traffic time in another? Blue/green deployment allows for zero downtime, so the development team can make the switch and let the load balancing system automatically shift all users to the green version instantaneously – no staying up till 4am required.

Have you ever been called in on a weekend to roll back a buggy deployment? With blue/green deployment, the old version is ready and waiting in case something goes wrong, so all that's required for a rollback is to ask the load balancer to switch users back to the blue version. This way, the programmers can come in at a normal hour during the workweek to fix the issues with the green version, then deploy it again when it's ready.

What's Safer Than Blue/Green Deployment?

There is a strategy even safer than blue/green deployment: the canary deployment strategy. Using canaries, the team will not just create two clones of production and test in only one, they will roll out the new code slowly, testing on only a subset of users before deploying to the entire user base. So in a new release, instead of

an immediate switch from 100% of users seeing version blue to 100% seeing version green, the initial deployment can switch over only 10% of users and leave the rest on blue. This controls the blast radius on blue/green deployment.

Drawbacks of Blue/Green Deployment

There are some drawbacks to blue/green deployment. For one thing, running two identical environments is expensive. Whether you run multiple physical servers or multiple instances in Kubernetes or Amazon Web Services (AWS), maintainining a production environment AND a production-cloned staging environment, which could be pushed to production at any time, is not a simple task.

Furthermore, there is the database problem. The process of maintaining two clones of production and pushing only one of them live can cause all kinds of database problems. Do you clone the database? Don't clone the database? And what if the database schema is going to be changed as a part of the new release? There are no easy answers to be found here. Database refactoring can fix the schema problem, and a mirror database can fix a few other issues but in general, caution is necessary when any blue/green deployment involves a database component.

Blue/green deployment is a great way to mitigate risk and prevent problems from update downtime, but consider both the benefits and drawbacks before diving in.

Canary Deployment

A canary deployment, or canary release, is a deployment pattern that allows you to roll out new code/features to a subset of users as an initial test.

Implement Canary Releases

When you implement a canary deployment you first create two clones of the production environment, then have a load balancer that initially sends all traffic to one version, and creates new functionality in the other version. When you deploy the new software version you shift some percentage – say, 10% – of your user base to that while maintaining 90% of users on the old version. If that 10% reports no errors, you can roll your new feature/code out gradually to more users, until the new version is being used by everyone. If the 10% has problems, you can roll it right back, and 90% of your users will have never even seen the issue.

Infrastructure changes and configuration changes should always be tested with canaries because of their sensitivity.

Why Canary Deployment?

Canary deployment benefits include zero downtime, easy rollout, and quick rollback – plus the added safety from the gradual rollout process. It also has some drawbacks similar to those of a blue/gree deployment – the expense of maintaining multiple server instances as well as the difficult clone-or-don't-clone database decision.

Typically, software development teams implement blue/green deployment when they're sure the new version will work properly and want a simple, fast strategy to deploy it. Conversely, canary deployment is most useful when the development team isn't as sure about the new version and they don't mind a slower rollout if it means they'll be able to catch the bugs.

Where Did the Canary Deployment Concept Come From?

You might be wondering why a little yellow bird is used to indicate a test release of a new feature. To answer that, we'll have to go back to the coal mining days of the 1920s. Miners brought caged canaries into the coal mines because they were highly sensitive to toxic gases like carbon monoxide. When exposed, the canary would become unconscious or die, alerting the miners to evacuate the tunnel immediately.

In a similar vein, when you release a feature to a small subset of users, those users can act as your canary, providing an early warning if something goes wrong so that you can rollback to the previous, stable version of the application.

Change Advisory Board

A change advisory board (or CAB) is a collective of representatives from different departments within the company who run that company's formal change management processes. They are tasked with reviewing and approving or rejecting change requests before implementation is allowed to take place in production.

In some environments, the change advisory board has no explicit decision-making power, but instead makes recommendations to a designated change manager, who makes the ultimate decision about whether to let a change proceed. A change advisory board is not directly involved in designing or implementing the proposed change, which is why the process it conducts is often known as an "external" review.

How to Set Up a Change Advisory Board

A change advisory board should be made up of one representative from each team that may be affected by the change. This group usually includes IT and business leaders that can provide different perspectives on the changes being released. It must meet on a regular basis and have a well-defined process for submitting and reviewing change requests. To be successful, a change advisory board must strike a balance between reducing risk and allowing change to flow to stakeholders with the least possible friction.

The Evolution of Software Processes – How We Got to Change Advisory Boards

Change advisory boards were implemented to improve the visibility and coordination of changes that could impact multiple stakeholders inside and outside of the information technology function. The goal was to avoid situations where service delivery becameunstable because “the left hand didn’t know what the right hand was doing.”

As systems and the larger organization’s dependency upon them became more complex, fears of even an “isolated” change in one part of a system impacting other seemingly unrelated parts grew, accelerating the adoption of change advisory boards and increasingly more stringent change management procedures.

Change advisory boards have been accused of creating significant delays… Delays like weeks or months to get a change into production while procedures are followed and approvals are granted, and the risks of teams missing their change windows, have called these boards into doubt.

Are Change Advisory Boards Going Away?

Rather than layering on more “external” review, many modern software companies that are embarking on digital transformation efforts have sought to reduce the dependencies between systems. Once systems are sufficiently isolated from impacting each other, they expect the individual teams running each component to “self manage” the change process. They may still retain a change advisory board, but allow a wider latitude of decentralized change management.

What’s Better than a Change Advisory Board?

Intuitively, small teams with full ownership of service delivery for an isolated component are more likely to understand the impact of proposed changes to their component than an external review board would.

This decentralized approach delivers greater stability and higher throughput of value than the prior model of managing one or more monoliths through a centralized change approval board. In fact, DevOps Research Associates (DORA) found that organizations with lightweight or no formal change review process significantly outperformed organizations with a change approval board in terms of stability.

Chaos Engineering

In any sufficiently complex software system, failure is inevitable. Given that this is the case, chaos engineering, also known as chaos testing, provides a method and tool-set to deliberately introduce failures and outages in a system.

This approach was pioneered by Netflix, who first created their chaos engineering process in 2010, and shared it in detail in 2014.

Chaos Engineering and the Simian Army

In chaos engineering, a set of automated processes, known collectively as a “Simian Army,” are used to introduce various types of system failures. The colorful naming of these tools evokes the mental image of chaos testing as a group of monkeys wreaking unexpected havoc in a data center, an event for which engineers must prepare as best they can.
Knowing for sure how a complex system will react to failures is practically impossible. The only way to predict the results of failures – especially catastrophic or cascading failures – is to have them happen. Therefore, creating those failures yourself – in a controlled way and at a time of your choosing – via chaos engineering is a valuable learning exercise.

Understanding the failure modes of your system is particularly important if you have high expectations around reliability, or if you are operating in a less reliable environment – on top of cloud infrastructure, for example. However, injecting chaos requires a certain level of preparedness. You might want to try it out in a pre-production environment first!

CI/CD - Continuous Integration / Continuous Delivery

CI/CD is the acronym in software development for the combination of continuous integration (CI) and continuous delivery (CD). It can also be expanded to include continuous deployment, but to avoid confusion we’ll be using CD to refer only to continuous delivery.

CI/CD is an extremely useful agile process for DevOps teams: an effective CI/CD pipeline makes bug fixes easier, eliminates merge hell, and speeds up the development process. Using feature flags, it can even increase the safety of deployments and improve user experience. Today, we’ll explain what each component of CI/CD is, and how to implement them.

Continuous Integration

For most organizations, the standard of continuous integration is that every member of the development team contributes to trunk at least every 24 hours. This process makes it easier to find and fix bugs because doing so in a small code change is easier than in a large one. It also eliminates the merge hell that’s often caused by long-lived feature branches. When each commit is a single day’s work or less, the likelihood of one programmer changing code that another’s code is dependent on shrinks massively.

Continuous integration depends on specialized tools in order to integrate code changes from the myriad of different platforms that developers work in and on. It also needs a set methodology for validating changes.

The most common methodology is Trunk-Based Development, which involves committing every new code change to trunk. This makes satisfying the requirements of CI easy and creates a generally fast-paced development environment. However, it isn't the only process for CI. It's also possible to use a process like Gitflow, so long as the pull request process is quick.

Continuous Delivery

Continuous delivery is the process of automating delivery of an application to any infrastructure environment. (If that happens to be the production environment, it's termed "continuous deployment".) Typically, continuous delivery processes push code to development, staging, or testing environments.

Not every CD tool works the same way, but commonly they will automate the process of creating the infrastructure that the new feature requires, moving code from the version control system to the target environment, defining relevant environmental variables, and otherwise setting up the target environment, executing automated testing, and rolling back if those tests fail.

Many software tools can be used to automate a CI/CD pipeline, the some of most common are Jenkins, CircleCI, Travis CI, and Bamboo.

Using Feature Flags in CI/CD

A feature flag is a piece of conditional code that allows you to turn features on and off without re-deploying. In a CI/CD workflow, it's possible to keep unfinished features behind feature flags, only turning those flags on once the feature is complete. If the feature is turned on and still has a bug, the rollback is as easy as flipping a switch. In this way, using feature flags improves both the speed and safety of your feature releases.

There are a variety of ways to implement feature flags, but for teams implementing CI/CD, a comprehensive feature flag management system is typically the best option. This prevents the accumulation of technical debt when a flag is used to turn off an incomplete feature and then left in the code because the management system will let you know the flag hasn't been flipped on or off in a while.

Configuration Drift

More and more companies are beginning to understand that using a staging environment to test features causes more harm than good. Because this process separates where end-users will interact with new features and where engineering teams will test new features, something problematic is bound to happen. Configuration drift happens as these two (or more) environments grow to be more and more different.

As engineering teams grow and their product evolves, changes are made to both the configuration and the infrastructure of the application. This change is called configuration drift.

Increasing the Divide Between Staging and Production

Let's look at a typical example of a configuration drift in practice.

An engineer gets paged late one night because of an incident with his mobile application. They look at the logs and identifies the problem. In order to fix it, they need to update a specific configuration in production. They make the change in production and go back to sleep. Although the issue is fixed, they have just created a divide between staging and production because they did not make the same change in staging.

Many times, staging environments are not the same as production because of changes made during incident management. Although it is never anyone's intent to create a difference between the environments, that is generally what happens when there are several environments in play.

As you are increasing the differences between your real-world and test environments, the trust in your staging environment will slowly decline. You will not be able to reliably test in staging because the test results will likely be different in production. Configuration drift can cause unidentified bugs, as well as cost your team time and money.

Automating the Creation and Maintenance of Environments

One way to avoid configuration drift is to apply infrastructure-as-code principles. The idea here is you want to replace manually trying to keep environments in sync with defining the environment with software and code. Then in the code, you can apply the same configurations to all of your environments.

The risk that happens when setting up environments manually is that you don't set them up the same way. It's much more consistent to have the computer make the changes automatically. Ideally, you want to avoid repeating yourself in code. Looking back at our example above with the engineer who made the configuration change for prod, instead of making the change for the one environment, they should have made it to a script that defines all of the environments, then it would have automatically been applied to all of the other environments.

Another way to avoid the issues caused by configuration drift is to set up feature flags to test your code in production safely. With the removal of your pre-prod environments, not only will you not have to worry about the status of your staging environments, but you will be able to release faster.

Continuous Delivery

Continuous delivery, similar to continuous integration and continuous deployment, is a software delivery process centered around improving the speed with which development teams release new features to end-users. But what is continuous delivery exactly, and how is it different from the other two continuous release processes?

What is Continuous Delivery?

The central idea behind continuous delivery (and therefore also continuous deployment) is that of being able to release the current version of the software directly to the production environment and to end-users at any time. The continuous delivery process has two main prerequisites: first, everyone involved in delivery must work closely together – this is frequently called implementing DevOps practices – and second, as much of the delivery process as possible must be automated – this is called a continuous delivery pipeline (or CD pipeline).

Continuous Integration

Continuous integration is simpler to implement, and it frequently serves as the foundation for both continuous delivery and continuous deployment. Where delivery and deployment are both software release processes, continuous integration is a software development process: it happens before the deployment process.

Development teams usually implement continuous integration because of the problems which arise from long-lived feature branches; if they only pull and push to trunk once in a while, it may turn out that one developer has changed the name of a function in his feature that the other developer needs for hers. The technical term for what happens when these two developers both merge their new code to trunk is, aptly, “merge hell”.

Continuous integration is the process of avoiding merge hell by continuously merging every code change to trunk (or “master” in Git). By doing this, developers get used to the process of keeping the trunk code clean while simultaneously pushing to it regularly, and this opens the door to continuous delivery.

Continuous Delivery vs. Continuous Deployment

Continuous integration is not technically required for continuous delivery, though they go together often enough that there’s a single acronym for development teams that do both: CI/CD.
By contrast, continuous delivery is required for continuous deployment. This is because where in continuous delivery you could push to production at any time, in continuous deployment you do.

They both also have different sets of benefits and drawbacks. The benefits of continuous delivery include the potential for more frequent releases, less deployment risk, and more transparency of development practices and progress (if something is deployed

into the real production environment, or even a staging environment cloned from it, that makes it seem more “done” than if a developer just said it was finished).

The main drawback is that, because there is a stage of manual quality assurance (QA) testing before changes can be deployed to users, code can get hung up in those unit tests and not proceed as quickly through the deployment pipeline as in continuous deployment, where automated testing is the only safeguard between the new code and the end-users.

Continuous Delivery Tools

Continuous delivery is the process of systematically keeping code deploy-ready at all times. To shift to a continuous delivery model form another deployment method, several changes are required in the operations of the development team, from improving automated testing to implementing Agile and DevOps processes.

But above and beyond these changes, having the right tools to facilitate your new continuous delivery processes is crucial to your success. In this chapter we’ll explain some of the most popular tools used by software development teams implementing not only continuous delivery but also continuous integration and continuous deployment since there is significant overlap.

Version Control Systems

Continuous integration, the process of merging every new code change back to trunk, is, though not strictly necessary, incredibly useful for continuous delivery and deployment. Version control systems (VCSs) are central to continuous integration because they help development teams to track which changes were made when and by whom, to eliminate bugs or potential problems as early as possible. Perhaps the most well-known version control tool is Git (and its online repository Github), but there are several other popular ones, such as Subversion and Microsoft’s Team Foundation Server.

Continuous Delivery and Deployment Tools

One of the most popular tools for both continuous delivery and integration is Jenkins, an open-source, server-based, plugin-rich application created in Java that automates many parts of the software delivery pipeline. Jenkins is particularly useful for distribution across systems that are on different platforms and its extensible automation allows it to be used as a continuous delivery hub.

Bamboo, an Atlassian product, is an on-premises server for continuous integration, delivery, and deployment which allows you to create multi-stage build plans and run parallel automated tests. It integrates seamlessly with Jira, Fisheye, and Hipchat. The Java Secure Channel is a particularly useful tool for continuous deployment in particular since it provides deployment automation. Other continuous delivery tools include AzureDevOps, Harness, and XebiaLabs.

Continuous Integration Tools

There are many specific tools that provide frameworks for continuous integration, such as Travis CI and CircleCI. Both of these tools integrate with Git to provide a seamless xperience for the thousands of developers already using it as their VCS. In addition, using containerization tools such as Docker and Kubernetes, both of which have integrations to various CI tools, can help with implementing continuous integration at scale.

Other Tools Used by CI/CD Teams

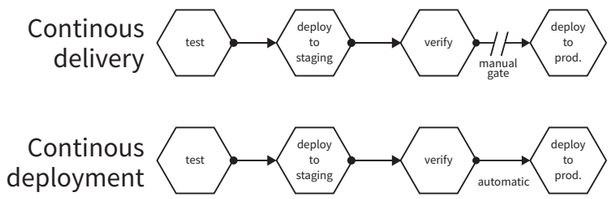
On top of the CI/CD-specific tools, there are horizontal tools that are often used in the CI/CD space. These include Infrastructure as a Service (IaaS) tools such as Microsoft Azure, databases such as MySQL and SQL Server, IDEs such as Eclipse, Visual Studio, and Atom, and issue tracking systems like Jira.

When considering what continuous delivery tools to use in your software build process, think about scope and cost. What do you need your tools to do for you, and how much are you willing to pay for that functionality? Once these questions have definitive answers, you can compare different tools to find the right ones for your particular use case.

Continuous Deployment

Continuous deployment is the practice of automatically promoting code changes to production after they pass all automated tests in a continuous delivery pipeline. In the absence of continuous deployment, changes must be manually approved before they are promoted (i.e., pushed or deployed) to production.

Many continuous delivery implementations automatically promote changes to a staging environment when all automated tests pass. This allows developers, testers, or other stakeholders to perform manual or exploratory testing before manually approving a push to production. Continuous deployment removes that final manual approval gate:



Benefits of Continuous Deployment

Continuous deployment brings significant advantages, but it doesn't come for free. Here are some of the advantages and disadvantages of choosing continuous deployment.

Speed

The most significant advantage of continuous deployment is speed. Teams that practice continuous delivery routinely move code from developer commit to production in just a few days and often in just a few hours or minutes.

Speed, in turn, unlocks greater safety and innovation.

Safety

While it may seem paradoxical or counterintuitive that a faster process without a manual review and approval step would be safer, DevOps Research and Assessment studies have repeatedly shown that teams with a short cycle time from commit to production have significantly lower incident rates and dramatically shorter time to resolve issues. They also achieve better business outcomes, which leads to the next advantage: innovation.

Innovation

Knowing that your team can safely push another deployment in minutes when needed reduces the fear of making changes and rolling them out. This creates a virtuous cycle of faster iteration and faster learning.

Continuous deployment shifts significant amounts of power and responsibility towards the team that is writing code. This leads to another paradox: Instead of leading toreckless, unchecked behavior and decisions that don't reflect business requirements, continuous delivery places developers closer to end-users, and that naturally leads to greater empathy, greater pride of ownership, and a more effective feedback loop when iterating towards desired outcomes.

Before we move on, take a moment to notice the contrast between continuous delivery and legacy development patterns where a development team cuts a release, passes it along to a separate testing team, which in turn hands it off yet again to an operations team to take live. The old ways diffused responsibility and placed a disproportionate amount of it in the hands of teams that knew less about the changes being made in a release and the motivations behind them.

Disadvantages

The disadvantages of continuous deployment over continuous delivery with a manual gate at the end are all related to the amount of rigor you must have in place to get the benefits.

Incomplete Implementation Just Breaks Things Faster
Without effective stakeholder review, code review, automated testing, and observability in place, continuous delivery increases the risk of things going wrong. It can lead you to the same sort of complex multi-layered problems that large batches create. It's all the grief of merge hell but live in production. Incomplete implementations often lead to "this just doesn't work here" reactions and a race back to old ways of doing things.

Implementation Takes Time, Commitment, and Culture
Moving to continuous deployment requires a sustained effort. The cost in time and commitment and the need to shift management and team culture from a short-term deliverable focus to a long-term process improvement focus may be too "expensive," depending on your context. If this is so, your efforts are better spent incrementally achieving continuous delivery to the point where the manual gate at the end is just a mere formality.

Achieve Continuous Delivery

If you are eager to reap the benefits of continuous deployment but your team or organization is not yet ready to implement it rigorously, consider focusing first on achieving continuous delivery. Since there's a manual gate at the end, you can address issues with reviews and testing and overcome the limitations of less robust production observability by having developers obtain approvals and then "manually walk" changes into production one at a time. This requires more coordination, and that probably needs more waiting. Still, once you have orchestrated the automation required to pass the mojito test, you are likely ready to embark on continuous deployment.

It's OK to Push a Button

Something to bear in mind is that even if you've achieved a well-defined and well-operated continuous delivery practice, you may still decide to keep a human-mediated gate before pushing to production.

In his book, Continuous Delivery in The Wild, Pete Hodgson reported that among the dozen or so teams he interviewed that were successfully using continuous delivery practices to achieve daily or more frequent pushes to production, only two had chosen to do continuous deployment. Why? Their continuous delivery practices were giving them almost all of the benefits of continuous deployment, and the large amount of additional work required to allow that automatic push at the end safely wasn't worth the small gain they would capture.

Without Automated Tests, You Don't Have a Pipeline

It's important to remember that any CI/CD pipeline, and especially one that does continuous deployment, cannot function without effective automated testing. Automated tests are the fuel that moves code towards successful deployment, regardless of whether that final deployment to production is automatic or done with the push of a button.

If you can't achieve team consensus that automated testing is essential, you can't implement continuous integration, and you don't really have a pipeline. Don't be fooled by a pipeline that simply kicks off automated builds upon commits or on a schedule.

Continuous Integration

Continuous integration happens when software engineering teams frequently integrate their code into a shared branch, usually referred to as main or mainline, with a goal of discovering merge conflicts as quickly as possible, rather than deferring discovery of issues until some later milestone. After pushing up new code, generally, automated tests will run and block the merge if any test fails.

Continuous Integration: Push More Often in Smaller Increments

As part of continuous integration best practices, the goal is to commit code frequently (daily, if not more often) and to build in smaller increments that each focus on a limited scope. Working this way makes it much easier to quickly identify where a code problem exists when a test fails.

For teams new to continuous integration, the first task is to learn how to break large problems down into multiple small increments that can be coded, built, and tested independently. This is known as incremental feature development.

The Difference Between Continuous Integration and Continuous Delivery

Often we hear the terms of continuous integration and continuous delivery used together. Continuous integration happens when you automate the process of integrating your code to main or master to ensure there are no conflicts. Continuous delivery is when you

automate the process of taking that shared branch (main) and turning it into a deployable release thatyou can push to production whenever you want. You need continuous integration to do continuous delivery, which is why you frequently see them mentioned together as “CI/CD.”

Continuous Integration is Not a Technology; it’s a Practice

If you run automated tests against your code, it doesn’t necessarily mean you are practicing CI. If you are not doing the integration part, then you are not doing CI. Automated testing happens when you integrate your branch with the main, or a shared branch, not your local branch. This is the fundamental principle of integrating changes. Just because you use Jenkins or CircleCI doesn’t mean you’re doing CI. Some people feel like if they pull from the main, they solve the problem, but it doesn’t work that way.

The most common pushback on continuous integration is when engineers are afraid of merge conflicts. Let’s look at an example.

Sally and Joe are two engineers on the same team who both have their own feature branches they are working on. They are afraid they will have merge conflicts. The way they resolve this is by continually pulling in whatever changes are in main. However, all the code on Sally’s branch is not integrated with all of the code on Joe’s branch because Joe hasn’t shared it yet, and vice versa. Until they both merge to main, they will share the risk of having merge conflicts. By keeping pull requests small and pushing to main frequently, you avoid these conflicts.

Trunk-Based Development

Trunk-based development is a modern restatement of CI principles. It takes the original ideas based on continuous integration, development on the shared frequently integrated branch and enables continuous delivery.

There are two types of development in trunk-based development – feature branches and traditional trunk-based development. When developing with a feature branch, a developer or a group of developers will work from a feature branch and merge it to master once the feature is done. In traditional trunk-based development, a developer will divide their work into small chunks and merge that into master many times a day. The difference here is scope.

Controlled Rollout

A controlled rollout is a feature rollout with highly granular user targeting. It allows you to release new features gradually, ensuring a good user experience for smaller groups of users before releasing them to larger groups.

Controlled Rollout Use Cases

There are two basic types of controlled rollouts: those where a feature is released first to a certain percentage of all users, and those where it's released to users according to a specific attribute, like IP address or location.

For the first type, the development team would start with the feature release, in production, to 1% of their real user base. If that randomly-selected 1% responds well – the team's CX metrics have either remained the same or improved, and customer support hasn't seen any significant increase in tickets – they will release to 10% of users. If those users also respond well, they can roll out until every user sees the feature.

For a few examples of the second type: the team could release a feature first to New York, then to the entire United States, and then the rest of the world; or, some number of users could volunteer to beta test new features, and the development team could select those users by a user ID, releasing the feature only to them.

These types of controlled rollouts are often combined: for example, a team could select only internal users by IP and release to them, then release to the beta test user group, then release to 1% of the entire user base, then incrementally roll out to everyone.

If at any point there is a problem in any of these processes, the team should be able to implement a quick rollback to the previous version.

How to Implement Controlled Rollouts

One of the most common ways to implement controlled rollouts is using feature flags and feature management systems.

Many feature flag management systems, like Split, come with built-in targeting capabilities, allowing you to target users based on just about any metric. You can use this capability, not only for controlled rollouts, but also for creating subscription models, hiding features behind paywalls, and merging unfinished features to trunk with their feature flags turned off.

Not only do feature flags make the process of targeting easier, but they also make rollback as simple as the click of a button. You don't even have to re-deploy your code: simply flip off the feature toggle and your application is back to normal. You can now easily fix the bugs without having to worry about the impact on users (you only mildly inconvenienced 1% of them) and re-deploy afterward.

Dark Launch

Dark launching is the term for releasing features to a subset of your users, seeing how they respond, and making updates to your features accordingly. It's somewhat like what every project manager does to monitor application health but focused entirely on a single new feature.

In this modern age of continuous delivery and deployment, feature releases are happening more frequently than ever before. But at the same time, companies must maintain the quality in their applications despite this fast-paced release cycle.

The Benefits of a Dark Launch

A typical dark launch begins by wrapping a new feature in a feature flag. Once the feature is pushed to production, the development team (or product manager, or even marketing team) can begin turning the new version on for users, starting with a small percentage like 1% or 5% and moving up to larger percentages if everything continues running smoothly.

During this process, end-user feedback is being gathered, either with direct methods like surveys or indirect methods like behavior tracking. If the team monitoring the feature notices that it's causing trouble (users are converting less often, submitting more help tickets, spending less time on the page or in the app, etc.), they can turn off the feature flag with the click of a button and have their working application back.

The Drawbacks of Dark Launches

The central drawback of using feature flags for anything is that they can easily turn into technical debt. Unused feature flags can clutter up codebases, in the end making it more difficult, not less, to confidently release new features.

This is a solvable problem, however. Managing your technical debt from feature flags can be done by only wrapping a piece of new code in a feature flag if you're sure you'll need to turn it off and on, and by ensuring your feature flag lifecycle is visible so you remove unused feature flags after a certain period of inactivity.

Dark Launch vs. Canary Release

Dark launches and canary releases are fairly similar: both deal with releasing new features in the production environment to a subset of real users before releasing to everyone, and both decouple deployment from release. However, there are a few key differences.

For one, dark launches typically look directly at user response to features on the front end. They'll be used to release a new option for a shopping cart on an e-commerce store. On the other hand, canary releases are more commonly used to test new features on the back-end. They'll be used to transition slowly to a new infrastructure.

For another, dark launches are commonly released to a group of users that doesn't know they're being tested on and don't have the new feature pointed out to them in any way: hence the "dark" in "dark launch". On the other hand, users can sometimes opt-in to beta test canary releases.

Feature Branch

A feature branch is a copy of the main codebase, where an individual or team of software developers work on a new feature until it is complete. Once work is completed, a merge of the feature branch back into the main codebase (or "trunk") is attempted. The longer a feature branch is open, the more likely that the merge process will prove difficult.

Isolation, For Better or Worse

Feature branching became especially popular with the rise of open-source projects. The isolation it provided allowed independent developers to work on feature contributions at their own pace, leaving the main branch untouched until after a code review and merge. This isolation provided greater stability and a clearly defined quality gate.

In commercial settings, where multiple developers are working full time and the cost in time and lost momentum due to merge conflicts is higher, the isolation created by a feature branching approach can be a significant liability. Worse still, the relationship between the length of feature branches and the complexity of merge conflicts can lead to a perverse incentive where the team avoids merging even longer. This, in turn, increases the chance of a catastrophic "merge hell" situation that can derail a project and/or lead to developer burnout.

Short-lived Feature Branches, Trunk-Based Development, and Continuous Integration

Teams that use feature branches and want to limit the overhead of complex merge conflicts have shifted to the use of "short-lived" feature branches, where changes are scoped down into smaller chunks that can be completed and merged back into the main branch within hours or at most a few days.

As feature branches become shorter they become nearly indistinguishable from trunk-based development and continuous integration. The goal with these practices is to constantly be merging back to the trunk in order to detect merge conflicts as quickly as possible.

When a developer commits an hour of work and is notified of a conflict in less than five minutes, it's much easier to triage and resolve the conflict than if they have to sift through days or weeks of changes by multiple developers. The key benefit is forward momentum and freedom from the fear of multi-hour or multi-day merge conflicts.

Short-Lived Feature Branches and Feature Flags

A large feature may take days or weeks to complete. Decomposing it into smaller chunks that can be incrementally built, tested, and committed in less than a day isn't always practical. Even if that chunking is possible, it might result in various smaller components of the feature being live in the code before the feature is complete and ready for users.

Feature flags provide a consistent and safe way to control access to these partially completed features, allowing developers, testers, and internal stakeholders the ability to execute the code in any environment, including production, without exposing the partially completed work to users. As a result, feature flagging has become a common practice adopted by teams moving to short-lived feature branches, and a must-have for teams that practice trunk-based development and continuous integration.

Short-Lived Feature Branches and Refactoring

Refactoring is the practice of improving the quality and supportability of software by revisiting existing code and making changes. Refactoring fights the natural tendency for entropy and “code smell” to increase over time. It’s most effective when it’s done frequently and when it causes the least interruption or distraction to the team. Since it’s common to run into merge conflicts when refactoring is performed, refactoring isfar easier to accomplish in a short-lived feature branch or full-on continuous integration practice.

Conversely, teams that use long-running feature branches have a strong disincentive to even attempt refactoring, let alone make it a regular and consistent practice.

Short-Lived Feature Branches: The Best of Both Worlds

Feature branches, when kept short, provide the benefit of a well-defined process for review and acceptance of changes that made them popular in the open-source movement and the benefit of continuous integration’s reduction of complex merge conflicts.

Feature Delivery Lifecycle

We believe that modern application development happens at the feature level. In the world of software, the new unit of measure is not the application, but the feature. To move fast, with a high level of control and optimal impact, product development teams need to take a lifecycle approach to feature delivery.

We call this process the Feature Delivery Lifecycle. This lifecycle enables engineering and product teams to beat their competitors to market with innovative products that delight their customers and propel their business forward.

Phases of the Feature Delivery Lifecycle

Let’s start with a trip through the Feature Delivery Lifecycle at a high level. As teams ideate how to solve customer problems, they first develop a plan for a feature. This plan becomes a reality as teams develop and deploy features, employing continuous delivery best practices, developing from trunk, keeping work in progress low, and making small changes that don’t break the development flow.

Once deployed, teams progressively deliver the feature. It’s an iterative target and release process. This reduces risk and unlocks the ability to gather insights from customers in production. These teams enrich feature flag data with other performance and behavioral data, and then monitor individual features and conduct experiments. These steps are critical. They capture

performance degradation and measure success criteria that otherwise are too hard to see. Monitoring captures performance degradation over the minutes and hours after release and experimentation captures success metrics over the days and weeks after the release.

The last phase is learning and deciding. With data from feature monitoring and experimentation, dev teams acquire the insight they need to ideate, plan and start the cycle again.

This entire life cycle requires close management and governance. Managing the state of each feature and how each flows through the cycle is paramount. Governing multiple teams as they contribute to building, releasing, and measuring features is critical to reducing risk. Getting this right makes it possible to iterate through this life cycle rapidly and predictably.

Deliver Value with the Feature Delivery Lifecycle

This iterative feature delivery life cycle, when it happens fast and consistently, allows teams to continuously deliver value to customers and impact to their business.

The Feature Delivery Lifecycle meets enterprises where they are in modernizing theirsoftware development technologies and processes. Whether in the middle of adopting agile and DevOps best practices, building a culture of experimentation, or both, organizations can employ part or all of the lifecycle to help achieve their goals, at any scale.

Centering modern software delivery around this lifecycle is ultimately how software development teams beat competitors to market and innovate features that delightcustomers and propel the business forward.

Feature Flag Management

Feature flag management is the process of, well, managing feature flags. While feature flags are extremely useful for many purposes, they require an organized process to manage.

There are two main types of feature flag management systems: those which are built in-house and those, like Split, which are purchased pre-built and often provide additional features, like measurement and product experimentation. There are benefits and drawbacks to both, and which solution you choose depends largely on your organization's needs.

Regardless of origin, any management system should do these four things: provide a common framework for the whole organization, serve up flags quickly and reliably, manage the testing process, and help avoid technical debt produced by unused flags.

Develop a Common Flagging Framework

The most critical thing a feature management system should do is allow all teams to view and control the state of each feature, ideally through an intuitive GUI. This means that not only developers, but also product managers, sales teams, marketing teams, and any other stakeholders will be able to toggle features on and off.

As an important feature in this GUI, it should be possible, and simple, to assign a certain person or team responsibility for a feature flag. This is useful for implementing progressive delivery, where over time, responsibility for features may transition automatically from the developers to the project manager to the customer success team.

Serve Feature Flags Quickly and Reliably

A well-built feature management system will make sure that the use of feature flags doesn't slow down the application for end-users. In order to do this, it will likely implement client-side caching to be more resilient. It will also likely use a content delivery network (CDN) to ensure that feature flag information gets to a user as immediately as possible

Manage Testing in Production

When your development team is implementing tests for your features, you can use feature flags to test those features in production. Simply add your internal teammates to receive the new treatment, test in production with those users, and then turn the feature flag on for everyone once the feature is ready. This process ensures your features are working correctly in production before your users have access to them.

Avoid Technical Debt

Feature flag management is also critical for avoiding technical debt. Technical debt is caused when feature flags outstay their welcome: a flag that was supposed to be short-lived sticks around a lot longer for any number of reasons; someone implements a flag and then forgets it exists, someone turns a flag on permanently and then forgets to remove it from the codebase, etc.

It's possible to deal with a backlog of technical debt, but it's hard to do and even harder to prioritize so that it gets done. The best way to avoid technical debt is to stop it before it starts and use feature flags responsibly. Use a feature flag management system that provides functionality to track how long it's been since a feature toggle was flipped. If a feature flag is no longer in use but is still clogging up the code, you can see that and delete it promptly. Check out these tips for feature flag maintenance.

Feature flags are an immensely useful tool for any DevOps or continuous delivery team, but without proper management, they can require additional effort, create technical debt and complicate testing. As such, you should implement a feature management system that will help your team to mitigate these problems.

Feature Flags

A feature flag, or feature toggle, is a software development tool used to safely activate or deactivate features for testing in production, gradual release, experimentation, and operations.

In trunk-based development, changes to the code base are consistently merged into the main trunk and pushed through to testing and production in a systematic way. Feature flag best practices are essential to maintaining the integrity and stability of code deployment, as many different features and even multiple feature branches can be included under unique feature toggles to be turned on and off when necessary.

What is a Feature Flag?

In its most basic sense, a feature flag is a section of code governing the execution of a specific software feature, allowing that feature to be "toggled" on and off without a new deployment. When a developer wishes to add new functionality, the feature can be implemented under a feature toggle in order to avoid impacting the user experience until the functionality is complete and verified in production.

How Feature Flags Work

By wrapping new feature code blocks with feature flags, developers can merge new code into the main trunk without affecting the release as a whole. New features can be activated selectively and their effect on the overall platform is monitored.

Features can be selectively enabled or disabled for specific groups of users, allowing for individual features to undergo a phased rollout that occurs in stages. Feature flags can also be used as A/B testing tools by deploying two different versions of a given feature at once, restricting each to a certain environment, and enabling them for different groups of beta users to see which performs better.

Importance of Feature Flags

With feature flags, it's no longer necessary to bundle multiple new features together for testing and release in a single, large periodic software update. Instead, it's possible to perform continuous delivery, an iterative development approach where features are deployed, rolled out, and tested in a larger number of smaller payloads. It's not uncommon for release cadence to increase by 10x or 100x as teams go from quarterly or monthly "big bang" releases to daily, weekly, or even hourly continuous delivery of smaller, more easily observed rollouts.

Feature flags are an essential tool in modern software development. They enableorganizations to move faster while safely and securely implementing, testing, and delivering new features. You can try feature flags for free on Split's Feature Delivery Platform.

Feature Flags Framework

A feature flags framework is a revolutionary tool for software development that allows individual features of a software product to be individually enabled or disabled. Feature flags allow features to be centrally managed from outside of the application, meaning they can even be turned on and off after they've already been rolled out to end-users.

Why Use a Feature Flags Framework?

Any new software feature needs to go through rigorous testing before being rolled out as the default user experience. To determine the optimal design, an experimentation period takes place, during which separate test groups are provided with different candidate versions of the planned changes. Each test group's usage is tracked against predetermined metrics that will be used to identify the best-performing option.

The feature flag framework allows code deployment to occur separately from the live rollout, without any interruption in service. The code for all versions of a new or updated feature can be deployed simultaneously to all users, and the feature toggles can then be enabled or disabled for the
version associated with each user test group.

How to Use a Feature Flags Framework

An experimentation platform provides you with the tools to perform a controlled rollout of new features through the use of feature flags. After code deployment into the production environment has been completed, the feature flags can be individually enabled to roll them out to end-users. This allows development teams to monitor the impacts of each feature, and turn a feature off if it negatively impacts the customer experience.

Benefits of Feature Flags Framework

The feature flags framework offers many benefits even beyond the pre-release testing of new features. Because they make it possible to deploy code into a live production environment in a controlled state, site maintenance and upgrades can happen without the scheduled downtime they previously entailed. New features can then be rolled out to a larger set of customers, or even globally when success and safety metrics indicate the feature is ready.

Feature flags make it easy to quickly and smoothly disable individual features, including decommissioning older features that are being phased out. If a major bug or other issue is discovered in a feature, a kill switch can take the feature flag offline while the issue is corrected.

These are just a few of the reasons the feature flag framework is proven to improve the software development process while minimizing the potential for the risks traditionally associated with deploying features to a live production environment.

Feature Rollout Plan

A feature rollout plan is a process that allows the introduction of a set of new features to a group of your user base. A good rollout plan gives developers control of the releases in the development cycle of a limited set of features. This way is possible to ship and test specific changes into a control group before deploying to all your users.

In the early stages of development, a small team could release multiple changes to production every week or two. Testing the impact of specific changes was difficult because of the noise generated by releasing a large number of features at the same time. Another problem development teams faced with the early approach, was the risk of a full rollback if anything went wrong, which eats up the company's time and resources.

In response to this issue, many product teams started planning and releasing specific changes to a set of users instead of full feature releases. This approach allows for copious testing of the new feature.

Advantages of Feature Rollout Plans

A good feature rollout plan can provide a solution. One of the advantages is that it encourages collaboration and considered planning. Instead of pushing new features to all users, development teams are able to do controlled releases in the development cycle by incorporating a release strategy.

Sometimes a product with a large number of users will need major changes. In this situation, a rollout plan makes a significant impact on the success of those changes. Rolling them out in phases or segmenting users into groups to test different features gives companies the freedom to fully test the user interface and user experience, as well as to run more tests. This creates a faster feedback loop, which allows dev teams to spend less time debugging and more time building features.

Rollout Plans with Feature Flags

There are multiple ways to deploy new feature rollouts; one useful method involves feature flags. Feature flags are a software development technique that lets dev teams turn features on and off, without having to deploy new code. Using feature flags gives companies the ability to perform more incremental rollouts. It also fixes bugs in the code without redeploying, creating a smoother, more streamlined development cycle.

The Feature Rollout Process

To properly execute a feature rollout plan, you'll need to implement careful planning, scheduling, controlling, and testing a feature every step of the way until its release. The process goes like this:

1. Design the new feature, examine the use case, and develop a timeline for completion.
2. Develop a release strategy that sets the parameters of release and a plan for incorporating feedback from your end-users.
3. Further develop the feature and manage its progress as it passes through various development environments.
4. Using feature flags to manage rollout and user targeting, test the feature. Then assess the quality of its performance with feedback from your users.
5. Launch the feature with the feature toggle off, then implement your rollout strategy.
6. Gather feedback so that you set in place a constant feedback loop.
7. Work with your team and product manager to monitor the feature's continual release throughout the development cycle. This will allow you to make incremental changes based on user feedback and continually optimize the product as you release your feature to your entire user base.

Kill Switch

In software development, a kill switch is a button or toggle that disables a feature if needed. This enables stakeholders to turn off broken features in production simply and immediately, which is often accomplished via a feature flag.

Before feature flags, if something were wrong, developers would have to look through the logs, pinpoint the issue, analyze how to fix it, write the code fix, test it, and then push it live to production, all while the feature remains broken in production. Not a great customer experience.

Incident Management Made Easy with a Kill Switch

With a kill switch, when a feature breaks in production, you can turn it off immediately while your team analyzes the issue.

This is ideal, especially when you get paged in the middle of the night and find yourself less than eager to take on a code fix. Having a kill switch allows you to quickly disable the feature, then work on a fix at your leisure, and push it when ready, rather than working under pressure to fix a production issue. Not only is this great to promote a healthy engineering culture, but it also allows anyone on the team to control the state of a feature, regardless of if they code.

If there is an issue with your code, you won't have to go through an entire code review process or revert the change that caused the problem. All you'll need to do is log in to the feature mangement platform, like Split, and click on the kill switch for that feature.

Isolate Code Changes

Let's look at an example of a typical agile team using feature flags in their software development lifecycle. Once the team deploys the code to production, and the product owner turns the flag on, then based on your configuration, the entire user base, or a subset of the user base will be able to see and interact with the new feature. However, in a few weeks, if something goes wrong and there is a bug with the new feature, what do you do?

In this exmaple, you can avoid rolling back an entire version of your application because you isolated the change to a specific feature. This also allows development on other features to continue without forcing a complete rollback. In effect, you're isolating the change while your codebase evolves around it.

The Cost of a Kill Switch

Any additional implementation to your codebase comes with a cost. Kill switches are no exception. The first cost is the management cost. To maintain a codebase with feature flags, you must continually be aware of the different flags' states. If not, you will likely be overwhelmed with the sheer volume of flags and unable to maintain them properly. There is also a code cost. The more extra code you add to your codebase, the more complex it gets, and the harder it gets to reason about that code. There is also a testing cost that comes with testing each new feature and all its different variations. With all of that in mind, the benefits clearly outweigh the cost.

Progressive Delivery

Progressive delivery is the logical next step for teams who have already implemented agile development, scrums, a CI/CD pipeline, and DevOps. It includes many modern software development processes, including canary deployments, A/B testing, and observability.

It is essentially a modified version of continuous delivery – in fact, before the term "progressive delivery," many people called it "continuous delivery ++" – with two core differences. First, progressive delivery teams use feature flags to increase speed and decrease deployment risk. Second, they implement a gradual process for both rollout and ownership.

Software Development with Feature Flags

The essential difference on the development side between CI/CD and progressive delivery is the use of feature flags. A continuous delivery team may do A/B testing, it may do blue/green deployments, it may implement DevOps or GitOps. But unless the development team is using feature flags, they're not doing progressive delivery. At best, they're doing really well at CI/CD.

The reason feature flags are so important is because they provide the opportunity for zero-risk deployment. By using a feature flag management system, even junior devs and new hires can push code to production: if the new version doesn't work, you can rollback with the click of a button. With feature flags, you can test in production – not a very similar test environment, not a clone of it, but production itself. This means that not only do you get to use your real architecture, you also get to test on real users.

Some developers get antsy about this idea but remember, you can roll back the feature instantly. So far as your users are concerned, you're not causing a massive shutdown, but a trivial inconvenience. And this inconvenience is even further mitigated by the second aspect of progressive delivery.

Gradual Rollout Process

Good feature flag management systems provide extremely granular user targeting. This means software delivery teams have the ability to roll out to a small subset of users first, make sure their feature works as expected, and if it does, gradually roll out to everyone else. So in stage one, you release to the developers only; in stage two, you release to a small set of users, and then if nothing goes wrong, you slowly release to more users until eventually, you've rolled out to everyone. This is useful because not only can you switch a feature off at any time, but you cause extraordinarily minimal inconvenience in the process. So not only do you cause only a trivial inconvenience to your users – you cause it to only 1% of them! This process of ensuring a minimal number of users are impacted by a failure is commonly termed "controlling the blast radius" of new features.

Gradual Ownership Change

Along with the process of gradually releasing a feature to more and more users, propagating it outward from the dev team, progressive delivery teams also propagate ownership outward. When a feature is first released internally, the dev team owns it and is responsible for fixing any bugs that might be present. After its initial release to production, maybe the project manager owns it. And after it's been released to all users, the customer success team probably owns it.

While some execute this process manually, the most successful teams automate the majority of this handover process, checking metrics and events to tell the system when to switch the release's owner.

Gradual, and especially automatic, changes of ownership help to ensure the feature is always being tracked by the most appropriate team for the job. On initial release, the development team that built the feature should be monitoring it to ensure it's working. After the feature is solid and released for everyone, the customer success team can answer user questions and gather feedback.

If you already have a great CI/CD pipeline, progressive delivery may seem trivial. But having a separate word for "continuous delivery with feature flags and canary deployment with gradual outward propagation of ownership" is useful, because this delivery process makes a major improvement on standard CI/CD. It creates an environment that not only fails quickly but comes back from failure quickly. And given that no system is perfect, building your system to do well at handling failure is the next best thing.

Trunk-Based Development

Trunk-based development (TBD) is a branching model for software development in which developers merge every new feature, bug fix, or other code change to one central branch in the version control system. This branch is called "trunk," "mainline," or in Git, the "master branch."

Trunk-based development enables continuous integration – and, by extension, continuous delivery – by creating an environment where commits to trunk naturally occur multiple times daily for each programmer. This makes it easy to satisfy the "everyone on the development team commits to trunk at least every 24 hours" requirement of continuous integration, and lays the foundation for the codebase to be releasable at any time, as is necessary for continuous delivery and continuous deployment.

Styles of Trunk-Based Development

Depending on the size of the development team, two different styles of trunk-based development emerge: small teams will tend to simply merge every new change to trunk, while larger teams may use short-lived branches, owned by one person/pair, or a small team. These branches will be merged back to trunk within days of being cut from it. (Any changes that require more than a few days to make should be done using feature flags in a branch by abstraction method in order to prevent "merge hell" from long-lived feature branches.)

Release Branches

The only long-lived branches in trunk-based development are release branches, managed by a release engineer. Developers don't make commits directly to release branches, although the release engineer may sometimes take a particular developer's commit and merge it into the release branch.

Release branches are never merged back to trunk. They are created at the beginning of a major version and merged into from trunk for minor versions, but when it's time to begin another major version, the existing release branch is deleted and a new one is created from trunk.

Pull Requests in Trunk-Based Development

Many people think of pull requests and imagine GitFlow, which is almost the polar opposite of trunk-based development (slow and fault-tolerant, contrasted with TBD which is fast-paced and developer-trusting). But pull requests do have a place in TBD – under specific circumstances. Using feature branches responsibly, a developer will, at any given time, still have some code that has not been merged to trunk yet. A pull request could be made to initiate a code review (especially an automatic one by a CI tool) on this new code.

When to Use Trunk-Based Development

There are two key features of TBD to consider when deciding whether or not to implement it. First, TBD has the ability to move very quickly. Second, it's very trusting of developers: no matter what they do, they are trusted to not break the build. These are commonly espoused as benefits of trunk-based development – and they are – but no system works perfectly for everyone.

For example, a brand-new company that needs to create version 0.0.1 of its product as soon as possible and has a team comprised of experienced engineers will work perfectly with TBD: everyone who would be committing to trunk is trusted, and speed is critical. However, a group of developers maintaining an established open-source project will not: speed is less important for them, and they can't possibly trust every random person who opens a Github pull request. The latter team would be better off using a more fault-tolerant process (such as GitFlow).

Part II

Measurement and Experimentation

A/A Testing

A/B testing is the process of split testing two different variations of a web page or feature by serving different versions of the feature to specific percentages of users, gathering data over time until the sample size is large enough, then finding whether there are significant results for a key metric, such as conversion rate. A/A testing involves running an A/B testing process with two identical versions in order to ensure the testing process is in working order.

Why Run A/A Tests?

A/B testing is an immensely valuable process for making data-driven decisions about everything from web pages to feature releases. A hunch that your conversion rate optimization could be improved by making the CTA button larger is all well and good, but if you've split your userbase into two groups and the one that saw the larger button made 5% more conversions, that's a very different (and much better) thing. But an A/B test can be a complicated process. How can you tell that your testing process is operating properly?

This is where A/A tests come in. By running two identical features through your A/B testing software or other process, you can ensure that the testing tool works as expected. With an A/A test, you can answer these questions:

- Are users split according to the percentages you planned?
- Does the data generally look how you expect it to?
- Are you seeing results with no statistical significance 95% (or whatever your confidence level is) of the time?

Let’s discuss that last point a bit further. If the two versions are identical, why are the results statistically insignificant only 95% of the time? Shouldn’t they be insignificant all the time?

If you have a 95% confidence level, that means you’re still wrong 5% of the time. Not all your data is identical – there is some variation – and that variation causes “significant” results 5% of the time, even when the versions are identical. This is called a false positive.

A/A tests can help you to ensure that your A/B testing process is working properly – you understand your data, the users are being split into groups as you wanted, and your significance levels are appropriate – so you can ensure that your A/B test results are telling you exactly what you think they are.

A/B Testing

A/B testing, otherwise known as split testing, is the process of testing two different versions of a web page or product feature in order to optimize conversion rate or improve upon a certain business metric. The two versions can be very similar, with only a change in button color, or very different, with a total change in the way a feature behaves.

How Does A/B Testing Work?

A/B testing is based on the scientific method, and the process is very similar. To start with, gather relevant data on your current features and see which ones have the most potential to improve key business metrics. After you have the baseline data, look at those features to see how customers are utilizing them, and hypothesize a variation that could improve it.

Since the next step is to build the new version, you’ll want to make improvements with similar expected user experience improvement and prioritize them by how easy they are to build. Then, pick a test to start with and build the new version. The old version will be what scientists call the “control” and what we’ll call Version A; the new version is the “experiment” or: Version B.

With front-end A/B testing, people typically assign Version A and Version B to different sets of users and measure which set of users, if either, had a higher conversion rate with statistical significance. But there is a different kind of A/B testing that happens at a much deeper level.

A/B Testing with Feature Flags

While typical A/B testing happens on the front end, choosing which version of the page is shown to website visitors, there is a way to A/B test your product features as well: using feature flags.

Feature flags allow development teams to release a feature to only a subset of users, which satisfies the necessary step of creating two versions of a feature. All that's left to do is to integrate the team's analytics platform with the feature flag management system, such that the team can correlate the users' behavior with which version of the feature they used.

After these things are done, the A/B testing process can be used to find the expected user experience change when any new feature or code change is implemented. Development teams can then look at this information and adjust the feature accordingly. If the change is significantly negative, they can find out what's wrong and roll back the feature so it performs as it did before running the test. If it's a positive impact, they can release the product feature to a larger percentage of their customers.

A/B/n Testing

A/B/n testing is the process of A/B testing with more than two different versions. The little "n" doesn't refer to a third test, but to any number of additional tests: A/B/n encompasses A/B/C, A/B/C/D, or any other type of extended A/B test.

Despite these additional variations, though, A/B/n testing works the same way as standard A/B testing: split users into groups, assign variations(typically of landing pages or other webpages) to groups, check the change of a key metric (typically conversion rate), check the test results for statistical significance, deploy the winning version.

A/B/n Testing vs. Multivariate Testing

Though they're often confused, A/B/n testing is not the same as multivariate testing. The key difference lies in how the variations are controlled. Let's use a webpage as an example. Say we have an image and a call to action (CTA) button, and we have three variations of each. If we run a multivariate test, it will automatically test all possible combinations – in this case, 6. However, if we run an A/B/n test, we hand-select which variations we want to test, which is frequently less than every possible combination. If we had a large number of different resources we wanted to test, the number of different variations in a multivariate test would grow exponentially – quickly requiring massive amounts of traffic and time it would take to get statistically significant results – but in an A/B/n test, we can manually choose how many variations to deploy.

A/B/n testing is more helpful in situations where getting results is more important than learning or generalizing from them. Multivariate testing, because of its granularity, is more helpful where knowing the precise cause of an increase or decrease in traffic is worth waiting for.

A/B/n Testing vs. Multi-Armed Bandit Testing

Another experimentation method, which happens to be most commonly used in machine learning, is the multi-armed bandit algorithm (MAB). Pardoning the esoteric, gambling-inspired name, multi-armed bandits basically use a different set of assumptions on how long an experimentation algorithm should spend on exploring possible alternatives versus how long it should spend exploiting those it has already found.

The process of A/B testing in general, and A/B/n testing, in particular, explores possible alternatives and their effectiveness for the test period before spitting out an answer and letting the user exploit the opportunity it has decided is best. By contrast, MABs dynamically explore and exploit in much shorter phases, relying on the past effectiveness of explored opportunities to decide on their next actions.

MAB testing is applicable to a broader range of problems than A/B testing. A MAB can produce significant results more quickly than an A/B test, and it can also automatically adapt to a changing environment and provide the best alternative in each, where several sequential A/B tests would need tobe run manually to achieve the same result. However MABs are not perfect: if there is any significant time between a change and its result – like an email campaign taking a few days to convert a prospect – A/B testing is far superior. Not to mention, MABs are more computationally difficult than A/B tests.

Client-Side Testing

Client-side testing refers to any type of testing – commonly A/B testing, but also multivariate testing or multi-armed-bandit testing – that occurs in the user's browser. This is contrasted with server-side testing, where the test cases are decided on the back-end (in the webserver) before they're served to the end-user.

Benefits of Client-Side Testing

There exist a variety of testing tools that can make it easy to implement client-side A/B testing. Many of them include a WYSIWYG editor that lets you easily change components in a visual editor without needing to reach into the code at all. This type of testing framework makes running tests on the client-side extremely easy and intuitive.

It also makes it possible for marketing teams to run experiments without needing to employ a front-end developer. Not a single line of code needs to be written, not a single actual deployment needs to happen until the experiment is complete. Once that happens, the developers only need to be brought in if the winning variation was one of the alternatives: otherwise, the alternate variations can simply be scrapped and a new experiment can begin.

Another benefit of client-side testing is the additional user data available. Because the variation hasn't been decided until the page loads in the visitor's browser, more data can be gathered about the user to determine which variation to serve. On the other hand, server-side testing has less user data to work on, so it's less able to segment users.

There are some drawbacks to client-side testing, though. The most common is that, since the test is implemented using client-side JavaScript, the user experience can suffer. Depending on the specific implementation, the page load time can get higher as it takes a second to determine what variation the user should see, or the user could see a "flickering" effect on the webpage as the original version is displayed before the test variation displays in its place. While load time issues are harder to fix, flickering can be tactically improved by only using client-side tests for elements below the fold.

The Lifecycle of a Client-Side Test

A client-side A/B test, like any other A/B test, begins with a hypothesis. "We think changing the color of this CTA button will improve conversion rate" is a classic example. Once the hypothesis is determined, the variations can be created using the visual editor and displayed to users using the testing tool.

After the test is complete, significance is calculated, and the winning variation is determined, it's time to implement the winner. This is a key difference between client-side and erver-side testing: when an alternate version wins in an experiment, the actual deployment process is slower than in client-side testing because the variations have not yet been built. With a server-side test, the variations have to be built in order to be tested, so the rollout process is extremely fast. However, on the converse, if a test fails to produce significant results, the variations that cost developer effort for a server-side test will have to simply be scrapped, whereas no developer effort went into creating variations in a client-side test. There is less cost to doing more experiments if they're done on the client-side.

Client-Side Testing Use Cases

By now, you've probably realized that the question is not "server-side or client-side testing, which is better?" – it's "server-side or client-side testing, which is better for you?"

You should probably use client-side testing if:

- You only need to test the front-end "look and feel" of your website or web application
- You're either testing below-the-fold elements, or you'd like to run low-cost experiments only visible to internal users (aka, you're in a situation where the flickering or page load issues don't make a significant difference)
- You don't want to expend the developer resources to do a deployment for each experiment
- You want to collect more user data before displaying different variations

If your use case doesn't fit all or some of these criteria, you might want to consider server-side A/B testing instead.

Customer Experience Metrics

Customer experience (abbreviated CX) is the experience your customers have with your brand and application. Customer experience metrics are organizational KPIs that help you monitor your customer journey to see if there are touchpoints where you are letting customer churn rate increase. In this chapter, we'll discuss some of the most commonly-used customer experience metrics and explain the benefits and drawbacks of each.

Net Promoter Score (NPS)

The best way to find out what customers think of you is to ask them directly. As such, the majority of CX metrics are self-reported customer feedback – Net Promoter Score is no different. NPS is designed to find not only your customer loyalty but more specifically how inclined your customers are to promote your brand through word-of-mouth.

The standard way to measure NPS is to find the number of "promoters" (people who rate 9-10 on a scale of 1-10 for questions like "how likely are you to recommend our brand to a friend"), find the number of "detractors" (people who rate 1-6 on the same scale), then subtract the percentage of the latter from the percentage of the former. Higher numbers are better here.

Note that we've skipped out on the "neutrals" – people who would rate 7-8. These people are not counted for purposes of NPS; the reason being that they are not especially likely to promote nor to spread bad word-of-mouth about your product, therefore, from apromotion/detraction standpoint, they are neutral.

One may wonder why we've skewed the entire distribution of results to the positive end. Should the results not be categorized more equally (ex. 1-3 = detractor, 4-7 = neutral, 8-10 = promoter)? As it turns out, no – and here's why.

When someone absolutely loves a product, they'll tend to rate towards the very top of the scale – 9 or 10, in other words. Some people are sparing with 10s, but anyone who loves your product will rate it at least a 9 of 10. When someone finds the product pretty good but not worth raving about, they won't rate it in the middle, because the middle translates to a feeling of "ehh, it was okay, I guess." Still, they won't give it a 9 or a 10. Therefore, "neutrals" will tend to give ratings of 7 or 8. Anyone who gives a 6 or below will have a feeling about your product somewhere between "ehh, it was okay, I guess" and "it was awful." Both of those count as bad word-of-mouth, so we count both as "detractors."

Customer Satisfaction (CSAT)

Customer satisfaction is a simpler metric: you just ask customers to rate on a Likert scale how satisfied they were with the product. The labels for this scale should be, in order, "very unsatisfied," "somewhat unsatisfied," "neutral," "somewhat satisfied," and "very satisfied." The Customer Satisfaction score is the total number of customers who were either "somewhat satisfied" or "very satisfied." With this metric, higher numbers are better. A CSAT score measurement is often accompanied by a series of other, more open-ended questions to determine each individual customer's experience. This is useful for tworeasons: first, listening to the voices of your unsatisfied customers can help you find problems that hamper customer satisfaction so you can fix the issues, and second, you can use the responses of your satisfied customers to find what exactly your customers loved about your product and do more of that.

Customer Effort Score (CES)

Customer effort score measures how hard your customer had to work to achieve their goal, either with your customer support team, with your website, or with your product itself. It's an extremely useful metric because it provides immediately actionable insights.

Previous measurements of CES had the customer rate effort on a 1-5 or 1-10 scale, but this was confusing, so nowadays most measurements of CES rely on a question like "do you agree or disagree with this statement: 'it was easy to handle my issue,'" and a Likert scale from "strongly disagree" to "strongly agree." Your CES score is the total number of customers who rated either "somewhat agree" or "strongly agree."

Drawbacks of Customer Experience Metrics

All these CX metrics are extremely useful and can provide very useful insight into your customer engagement, loyalty, satisfaction, and lifetime value. However, there is a drawback f or all of them.

These metrics do a pretty good job at measuring what we want them to, but they aren't perfect. When you rely on an imperfect metric, you need to be careful not to prioritize improving the metric over improving the result the metric is supposed to measure. In the end, your CSAT score is not what really matters; your actual customer satisfaction is.

Data Pipeline

Data pipelines automate the flow of data from one point to another. In a data pipeline, you start with defining how data is collected, and in what schema it should be collected. Then, you can automate the process of how to extract the data you need from the inbound pipeline, combine it with other data, and validate your team's KPIs by comparing the data to your baseline metrics. This automated process reduces the risk of not collecting the correct data and having to manually sort through the data you have collected.

Use Your Data for Experimentation

When software development teams run experiments for their products, the first thing they need to do is collect baseline data. This can include current conversion rates, average order value, click rates, etc. Once you have a baseline, you can then set a hypothesis of what you think will happen when you add a variant to the existing experience.

If you are running an A/B test, for example, half of your population will have the existing experience (the control) and half your population will have a new experience (the experiment). When data starts coming in through the data pipeline from both the control and the experiment, you can compare the baseline data from the control to the experiment. If the experiment gives you

better results for your KPIs, you can confidently release that experiment to the rest of your population knowing you are bettering the user experience. However, if the data coming in from the data pipeline shows a decrease in metrics, then you can confidently end the experiment knowing it would have been harmful to your user experience.

Establishing Causality Between Features and Metrics

By using a powerful data and experimentation dashboard, like Split, you are able to drive deeper insights with advanced analytics. Because all business decisions should be based on data, you need to have a visualization of what your users are doing, and how they are performing based on the experience they get.

With Split's statistics engine, you can establish causality between feature releases and company metrics, and you can add as many variants as you want.

A Powerful Data Pipeline Means A Better User Experience

The more powerful your data pipeline is in handling your data, the better your user experience will be. The best data pipelines will automate the influx of the data from your customers, transform it into the schema you need, and make it easy to assess how your features are performing.

For example, with Split's integration with Segment, you can collect the data you need, and send it to analytics, marketing, and any other stakeholders. You can ingest the user data you collect from Segment to power your A/B tests and feature release alerts. This data can also be used to send Split impression data to your warehouse or third-party applications. You should also be able to store data for future use in case you want to collect baseline data for another experiment later on. These properties of a strong data pipeline make for a solid foundation for A/B testing and experimentation.

Do No Harm Metrics

Do no harm metrics are metrics that teams use to ensure nothing bad is happening to your team's metrics due to a feature rollout. Many times in product experimentation, you release a feature through a canary release and monitor your metrics throughout. If your metrics show higher conversion rates and higher engagement, you can continue to roll out the feature to your entire user base. However, sometimes product managers monitor metrics that are not necessarily tied to a specific feature release.

Product experimentation is a way to increase engineering impact and progressive delivery while reducing the risk of moving fast. A/B testing and multivariate testing can reveal user behavior and user trends that you did not foresee. However, the most crucial part of running an experiment is not the variants but understanding why you're running the experiment in the first place.

Testing to Learn vs. Testing to Launch

According to Sonali Sheel of Walmart Labs, there are generally two reasons to run an experiment: Testing To Learn and Testing To Launch. Testing to learn is about the iterative discovery of what works and what does not, understanding customer behavior, and validating or invalidating a hypothesis. On the other hand, Test to Launch is about gradually rolling out a new feature to the entire population while keeping a close eye on metrics. Product owners launch a feature that's expected to have a long-term, strategically significant impact and run an experiment to (hopefully) show that conversions and KPIs are not negatively impacted.

The Impact of Do No Harm Metrics

Many times in experimentation, product owners and business stakeholders want to make sure that releasing a specific new feature does not have a negative effect on any existing metrics. They accomplish this with Do No Harm Metrics. This approach is used by product owners to ensure that if they make a change, it won't make any existing user behavior worse.

The goal here is to watch your team's do no harm metrics and stop the rollout early if metrics degrade. Your team can accomplish this with a percentage rollout. These metrics can include time to load, conversion rates, click rates, etc. Test to launch is first and foremost about mitigating risk. The idea here is to launch this feature unless it does something unexpected that you don't want.

Test to launch uses the same underlying capabilities of an experimentation platform, including managing selective exposure of a new feature and observing the system and user behavior differences between those who get a feature and those who do not. Suppose you're performing a canary release, or percentage rollout, as a Test To Launch. In that case, your experimentation and analytics systems must be connected so that you can specifically compare what your Do No Harm metrics are for the canary cohort vs. the control. In the analytics system, you should be able to differentiate between the traffic coming in for the experiment and the traffic coming in from the existing state. When you can make this differentiation, you can see what impacts your metrics and make more informed decisions.

It's essential to not just release features for the sake of proclaiming them as "done" but to ensure that features deliver impact and don't do any harm to key businessmetrics. Whether you are building a business or widening an existing business, you can use the same tactics to ensure your engineering efforts make a difference you can be proud of.

Event Stream

An event stream is a series of data points that flow into or out of a system continuously, rather than in batches. Event stream processing (ESP) refers to the task of processing event streams in order to identify the meaningful patterns within those streams.

Use Event Streams as Telemetry for Experimentation

In the context of an experimentation platform, an event stream is the incoming telemetry of user and system behavioral data, and event stream processing is the process of deduplication, attribution, and statistical computation that transforms events into the metrics upon which subsequent conclusions are made.

Examples of events as telemetry consumed by an experimentation platform might include:

- At time “T” user “U” clicked the “show more info” button on the property listing page
- At time “T” system returned 6 rows to user “U” from a search query, taking 2.4 seconds
- At time “T” user “U” upgraded from “basic” to “pro” tier

Note that all events above have both a timestamp and an association with a specific user. These two event attributes are essential in order to associate the user with a particular cohort and to know what the active experiment state was at that time. Consider this example:

- A series of experiments are being run, at two-week intervals, to determine the optimal configuration parameters to pass to a recommendation engine in order to best meet the needs of your site’s user population.

- Users are randomly split into three cohorts, with each cohort being treated to a different set of recommendation engine parameters.
- User behavior (i.e. purchases, upgrades, unsubscribes) and system performance (i.e. response time, errors) are observed for two weeks.
- Based on the results of the first experiment, parameters are changed and another two-week experiment is run.

If we didn’t know which user the events were associated with or exactly when the event occurred, we would not be able to allocate the behaviors to the right cohort or know which version of the parameter sets the behaviors occurred under. This is one reason why data aggregated across different time boundaries (i.e. monthly gross sales) isn’t useful as an event stream for experimentation.

Prioritize Event Stream Selection

“If we have data, let’s look at the data. If all we have are opinions, let’s go with mine.”

–Jim Barksdale, (CEO of Netscape Communications from 1995-1999)

Experimentation is about using data, rather than mere opinions, to inform decisions. If you’ve read this far, you probably agree with that. That said, you don’t want to take that idea too far. Rather than attempting to create an event stream from every possible data point in your environment before you begin experimentation, consider working back from the most important metrics you will need to inform your decisions.

For example, “Bookings Per Platinum Member” and “Average Booking Price Per Platinum Member” are metrics calculated from a stream of booking events that contain a timestamp, a user identifier, the users membership type, and the booking amount. That stream doesn’t need any data about clicks, scrolls, or page counts. If “Ratio of Booking to Room Selection” is a metric you wish to track, you’ll need to add an event stream of room selection events. Working backward from the most important metrics will ensure that you source the most important event streams first, clearing the way for your most important experiments early on.

Source the Needed Event Streams

The ideal event stream for establishing or expanding an experimentation practice is a stream that already exists and can be routed to your experimentation platform without custom development work. Customer data platforms (CDPs) have simplified the process of discovering and integrating these existing streams, even to the point where a non-technical user can configure and manage event stream flows. If you have access to a CDP, by all means, start there.

In the absence of a CDP, you’ll either need to build an integration thatextracts, transforms, and streams existing data to your experimentation platform, or you’ll need to add new instrumentation to create streams in cases where the data isn’t yet being captured. Most platforms have a variety of options for this, including SDK endpoints you call from inside your code, REST API endpoints you call per-event or to bulk-load events, and integrations that simplify the creation of event streams from other platforms such as Google Analytics.

Source Event Streams from Batch Data

It’s worth noting that an event “stream” can be created periodically from batched data (i.e. data that is only available after a nightly or weekly processing cycle). Sure, the “stream” may only flow now and then, but as long as the timestamps within the batch are preserved, calculations of attribution and impact can be accurately performed.

Look Outside the Box

Event data may need to come from a source outside the application you are focused on. Consider the case where an e-commerce team is experimenting with a free shipping offer for customers who buy three or more items in a single online session. If the same company’s brick-and-mortar stores have a return policy allowing in-store returns of online purchases, then it would be nice to know if the “buy three, get shipping free” cohort returns more products than the norm, right? For that reason, sourcing the event stream of in-store returns is critical for determining the business value of the experiment results. Bottom line? Don’t limit your thinking to your application’s data model when considering data stream candidates.

False Discovery Rate

False Discovery Rate (FDR) is a measure of accuracy when multiple hypotheses are being tested at once, for example when multiple metrics, or variations, are being measured in a single experiment.

False Discovery Rate Definition

In technical terms, the false discovery rate is the proportion of all 'discoveries' which are false.

When running a classical statistical test, any time a null hypothesis is rejected it can be considered a 'discovery.' For example, any statistically significant metric is considered a discovery since we can conclude the measured difference is highly unlikely to be due to random noise alone and the treatment is directly influencing the metric. On the other hand, metrics which did not reach significance are statistically inconclusive – they are not a discovery as it wasn't possible to reject the null hypothesis.

In the context of online experimentation and A/B testing, the false discovery rate is the proportion of statistically significant results which are false positives.

Or to write it algebraically:

$$FDR = N\_falsely\_significant / N\_significant$$

Where N_falsely_significant is the number of statistically significant metrics that were not truly impacted (false positives) and N_significant is the total number of metrics that were deemed statistically significant.

For example, if you see 10 statistically significant metrics in your experiment and you happen to know that 1 of those 10 significant metrics was a false positive and wasn't really impacted, that gives you a false discovery rate of 10% (1 out of 10). In this way the FDR only depends on the statistically significant metrics, it doesn't matter if there was 1 or 1000 other statistically inconclusive metrics in the example above, the FDR would still be 10%

Other Measures of Accuracy

Another common measure of the accuracy of a test is the False Positive Rate (FPR). This is the probability that a null hypothesis will be rejected when it was in fact true. In other words, it is the chance that a given metric, which is not impacted at all by your experiment, will show a statistically significant impact.

The important distinction between the false positive rate and the false discovery rate is that the false positive rate applies to each metric individually, i.e. each non-impacted metric may have a 5% chance of showing a false positive result, whereas the false discovery rate looks at all hypotheses that are being tested together as one set.

The Family Wise Error Rate (FWER) is another measure of accuracy for tests with multiple hypotheses. It can be thought of as a way of extending the false positive rate to apply to situations where multiple things are being tested. The FWER is defined as the probability of seeing at least one false positive out of all the hypotheses you are testing. The FWER can increase

dramatically as you begin to test more metrics. For example, if you test 100 metrics and each has a false positive rate of 5% (as would be the case if you use a typical 95% confidence or 0.05 p-value threshold), the chance that at least one of those metrics would be statistically significant is over 99%, even if there was no true impact whatsoever to any of the metrics.

When you are testing only one hypothesis (ex.: a test with only one metric) these three measures will all be equivalent to each other. However it is when multiple hypotheses are being tested that they differ; in these situations, the false discovery rate can be a very useful measure of the accuracy as it takes into account the number of hypotheses being tested, yet is far less conservative than the FWER.

The false discovery rate is a popular way of measuring accuracy because it reflects how experimenters make decisions. It is (usually) only the significant results – the discoveries – which are acted upon. Hence it can be very valuable to know the confidence with which you can report on those discoveries. For example, if you have a false discovery rate of 5%, this is equivalent to saying that there is only a 5% chance, on average, that a statistically significant metric was not truly impacted. If you know your false discovery rate is 5%, you can rest assured that 95% of all the statistically significant metrics you see reflect a true underlying impact.

Controlling the False Discovery Rate

As well as simply measuring the accuracy of a test, there are ways to control and limit the accuracy to the desired rate through the experimental design. The false-positive rate can easily be controlled by adjusting the significance threshold that is used to determine statistical significance. Controlling the false discovery rate is more complex as it depends upon the results, which cannot be known in advance. However, there are statistical techniques, such as the Benjamini Hochberg Correction, which can be applied to your results to ensure that the false discovery rate is no larger than your desired level.

False Positive Rate

The false positive rate (FPR) is a measure of accuracy for a test, be it a medical diagnostic test, a machine learning model, or something else. In technical terms, the false positive rate is defined as the probability of falsely rejecting the null hypothesis.

False Positive Definition

Imagine you have an anomaly detection test of some variety. Maybe it's a medical test that checks for the presence or absence of a disease; maybe it's a classification-based machine learning algorithm. Either way, there are two possible real-life truths: either the thing-being-tested-for is true, or it isn't. The person is sick, or they aren't; the image is a dog, or it isn't. Because of this, there are also two possible test outcomes: a positive test result (the test predicts the person is sick or the image is a dog) and a negative test result (the test predicts the person is not sick or the image is not a dog).

Because there are two possible truths and two possible test results, we can create what's called a confusion matrix with all possible outcomes.

Here are the possibilities:

- True Positive: the truth is positive, and the test predicts a positive. The person is sick, and the test accurately reports this.
- True Negative: the truth is negative, and the test predicts a negative.The person is not sick, and the test accurately reports this.

- False Negative: the truth is positive, but the test predicts a negative. The person is sick, but the test inaccurately reports that they are not. Also called a Type II error " in statistics.
- False Positive: the truth is negative, but the test predicts a positive. The person is not sick, but the test inaccurately r Reports that they are. Also called a Type I error in statistics.

Measuring the Accuracy of a Test

By calculating ratios between these values, we can quantitatively measure the accuracy of our tests.

The false positive rate is calculated as FP/FP+TN, where FP is the number of false positives and TN is the number of true negatives (FP+TN being the total number of negatives). It's the probability that a false alarm will be raised, that a positive result will be given when the true value is negative.

There are many other possible measures of test accuracy and error rate. Here is a short rundown of the most common ones:

- The false negative rate – also called the miss rate – is the probability that a true positive will be missed by the test. It's calculated as FN/FN+TP, where FN is the number of false negatives and TP is the number of true positives (FN+TP being the total number of positives).
- The true positive rate (TPR, also called sensitivity) is calculated as TP/TP+FN. TPR is the probability that an actual positive will test positive.
- The true negative rate (also called specificity), which is the probability that an actual negative will test negative. It is calculated as TN/TN+FP.

If you're on the patient side of a medical test being analyzed like this, you may care a bit more about two additional metrics: positive predictive value and negative predictive value.

Positive predictive value is the likelihood that, if you have gotten a positive test result, you actually have the disease. It's calculated as TP/TP+FP. Conversely, negative predictive value is the likelihood that, if you have gotten a negative test result, you actually don't have the disease.

Feature Experimentation

Ordinary software development is widespread with guesswork. Some amount of data, plus some amount of the product manager's hunches, drives the decisions of what new features the development team will create. Without some fairly major shifts in how development is done, this is almost how it has to be. You have to guess how your users will react to a feature because you can't just show it to them and gauge their reaction.

With feature experimentation (often called product experimentation, or digital experimentation), this dynamic shifts significantly. The product manager is able to make data-driven decisions based on the actual performance of the features in production because the new features are actually deployed to (a small percentage of) the real user base. Instead of guessing user reactions, the product team can deploy the feature in production to a small number of users, using customer experience surveys and KPI tracking to determine the feature's effectiveness, or even A/B testing or multivariate testing different versions of features to find the best one.

The Benefits of Feature Experimentation

There are two central benefits of feature experimentation: increased understanding of the user experience, and increased ability on the part of the development team to improve it.

No matter how well you know your user base, you won't be able to predict their reactions perfectly. Instead of trying to do this impossible thing, many teams have switched to testing new features in production. When you can gather real-time data on

how a new feature has impacted your KPIs – be they conversion rate, page load time, or API response time – you can choose which features to release to all your users based on a high probability that those features will produce a positive business impact.

Implementing Feature Experimentation

If experimenting with features is so great, how do you go about doing it? The most common way to do any type of product experimentation is to use feature flags, which allow you precise, granular targeting on who sees what features. Using a feature flag-based experimentation platform (like Split), anyone – the marketing team, the product team, the development team, the product management, or anyone else – can toggle features on and off.

This has an important benefit, when anyone can run experiments and test product features, this creates a culture of experimentation, inspiring everyone from all teams to test their ideas and gather data. Feature flags have a host of other use cases as well. They enable continuous delivery, promote DevOps, and allow you to monitor features even after you deploy them to your whole user base.

Hypothesis-Driven Development

If a software engineer wakes up in the morning and hears something that sounds like rain outside her window, she would likely think it might be raining. Her hypothesis, that it's raining, drives her decision to look out her window. She knows ahead of time that if she sees rain, it's actually raining, whereas if she sees a sprinkler running, it's not. When she actually looks out her window and sees rain instead of a sprinkler, she decides she should bring an umbrella to work.

When she gets to the office, if she notices people aren't clicking on her website's CTA button, she thinks the button might need to be more visible. Her hypothesis, that the button isn't visible enough, drives her development process. When she wants to verify that the button's visibility is causing the low conversion rate, she creates a new UI with a larger CTA button and tests it alongside her previous UI (probably using A/B testing). She knows ahead of time that if she sees a statistically significant increase in clicks from the users who see the bigger button, that was the problem, whereas if she doesn't see an increase, it wasn't. When she actually runs the test and sees a statistically significant increase in conversions, she decides to roll out the larger button to all users.

This is experimentation, using the scientific method to solve problems, test hypotheses, and create effective solutions. We do it all the time, often without even realizing it. In fact, many recent technology-related processes use this model: agile, DevOps, and the lean startup business model are based on the experimentation mentality. Hypothesis-driven development (HDD) is just the name we give to experimenting on the software development process.

The exact steps of hypothesis-driven development are:

1. **Set up user tracking** - Running experiments is impossible without tracking, so make sure that you have a way to track the impact of any changes or tests. A common way to track experiments is with a feature toggle-based experimentation platform like Split.
2. **Define a hypothesis** - When you define your hypothesis, you'll also define your validation criteria – aka, how much evidence you'll need to make a decision. Ensuring you know upfront what outcomes would cause you to make which decisions will prevent a significant degree of bias. Ask, "what will tell me this new product or feature is successful?"
3. **Test the hypothesis** - Set up the test and run it. In the software development world, most tests take longer than the short period of time it takes to looking out your window to see if it's raining: you'll need to run the experiment for a while in order to gather enough data for statistical significance.
4. **Act on the experiment results** - Once you have a statistically significant result, act on it. Roll out, or rollback, the experimental feature. Note what worked and what didn't, and keep running experiments.

Turning every new feature proposition into an experiment means all your feature releases are driven by data. You'll know what your users want, and how the form of that desire shifts over time. You'll know what features your users use and which they don't, which they want and which they only say they want. And because you know these things, you'll be able to create the best product for your customers.

Mobile A/B Testing

A/B testing is the process of testing two variations of a resource by showing different versions to different users, then comparing the test results (aka, the differences in user behavior between the two groups) for statistical significance. The process is essentially the experimental method as applied to software development.

There are two things that people mean when they say "mobile A/B testing," app store A/B testing and mobile app A/B testing. In this chapter, we'll focus primarily on the latter.

A/B testing for mobile apps is about as similar to standard A/B testing as mobile app development is to standard software development. There are some key differences (for example, mobile apps have unique features like push notifications that developers can and should use), but the overall methodology is the same.

The Mobile A/B Testing Process

Mobile A/B testing, as with any experiment, begins with data gathering. What are your current metrics and what are some key areas of your app that could improve along with those metrics? For example, if you have an in-app purchase feature, how many of your app users are using it? If your app has complex functionality, are too many people dropping off in the user onboarding phase?

After data gathering comes hypothesis formulation. What could you change to fix these user engagement problems? What re-engagement strategies could you use? After deciding on what hypothesis you'd like to test, you build the new variant, split your total user base in two, and serve one variant of the feature to each group.

Many standard A/B testing tools can be used for mobile applications, but ideally, developers should buy or build tailor-made mobile A/B testing tools, or select a platform, like Split, that can support both regular and mobile A/B testing. This will let them take advantage of the unique aspects of the mobile experience which are some of the reasons they built a mobile app in the first place. It will also let them account for unique draw-backs of mobile development. For example, somebody in the city will have lower network latency than someone in the country, but the app should work as well for both of them. Further, a lot of mobile A/B testing tools have built-in visual editors which make it easier for developers to design something that actual mobile users will want to use. After all, how many times have we all designed new features for our apps which looked really cool on a simplistic desktop simulation, but that ended up looking awful on a real smartphone screen?

Though the process is similar for A/B testing mobile apps as it is for any other software, understanding the unique aspects of the mobile development process and keeping those in mind during your testing process is paramount to designing and maintaining a great app.

Multi-Armed Bandit

A multi-armed bandit problem is any problem where a limited set of resources need to be allocated between multiple options, where the benefits of each option are not known or are incompletely known at the time of allocation, but can be discovered as time passes and resources are reallocated. The name comes from a particular visualization of this problem.

Imagine a gambler playing several different slot machines (sometimes called "one-armed bandits"), each of which has a different possible return (aka, some arms are superior to others, but the gambler doesn't know which ones). The gambler wants to maximize his total reward and to do this, every round he can choose an arm to pull from whatever number of arms he has. Resulting from this predicament iterated over many rounds, the gambler has two choices: he can either keep playing whichever arm has had the greatest return so far, or he can take a random action to pull some other arm, knowing that while some may be more optimal than his current best arm, some may be less. In machine learning, the tradeoff between these options is called the exploration/exploitation tradeoff.

This may seem like a highly specific, non-generalizable problem, but its applications range from clinical trials to financial portfolio design to adaptive routing to feature experimentation. The exploration/exploitation trade-off is seen in any agent incapable of simultaneously planning and executing.

And in general, multi-armed bandit algorithms (aka multi-arm bandits or MABs) attempt to solve these kinds of problems and attain an optimal solution which will cause the greatest returns and the lowest total regret.

Types of Multi-Armed Bandits

There are different approximate solutions to the multi-armed bandit problem. The simplest such solution is called the "epsilon-greedy" algorithm, and all it does is, given a small decimal value epsilon (ε), it spends ε% of the time exploring and (1 – ε)% exploiting. This algorithm is called "greedy" because of all the exploiting.

There are many variations on the basic epsilon-greedy algorithm: strategies for finite experiments such as epsilon-first (pure exploration followed by pure exploitation) and epsilon-decreasing (decreasing value of ε over the course of the experiment), as well as strategies which can be used on infinite or continuous experiments, such as value-difference-based epsilon (automatically reduced ε based on machine learning process) and contextual-epsilon-greedy (value of ε computed based on situation). There are also probability-matching (also called Thompson sampling or Bayesian Bandits) strategies which involve matching the number of pulls to the probability of a certain arm being the optimal one. You may note similarities to A/B/n testing in the process of finding the optimal alternative among many for the purpose of exploiting it.

Benefits and Drawbacks

Multi-armed bandit algorithms are best used for two use cases: either very shortexperiments where the time it would take to gather significant data in an A/B test is prohibitive (like finding the best headline for a hot new article), or else in very long or ongoing experiments where waiting for a "final answer" from an A/B test doesn't make sense (like optimizing each user's news feed).

The main problem with bandit algorithms is their difficulty to implement. If an organization is falling at all short in their DevOps practices, trying to implement a bandit will bring that out. Further, because there aren't many data scientists who are also excellent programmers, bandits are frequently more expensive since they take more people.

Multivariate Testing

Multivariate testing is a method of experimenting with different variations of particular elements in a feature implementation, such as the headline, images, copy, etc in a landing page or application launch screen, or other critical moments of truth in a customer journey, in order to determine which variations of said elements are best suited to improve conversions.

The method is similar to A/B testing, however, multivariate testing experiments are run with a higher number of variables and generally provide deeper insight on how to optimize your page. In multivariate testing, your feature implementation becomes a combination of elements which can be decomposed and tested simultaneously.

To help break down this process, let's assume you're working with the following elements: header, page images, and copy.

If you were doing a multivariate test of these elements, you'd create variations of them:

Header 1 – Header 2 – Header 3
Image 1 – Image 2 – Image 3
Copy 1 – Copy 2 – Copy 3

The purpose of a multivariate test is to try out different versions of these variations, as illustrated below:

Header 1 + Image 1 + Copy 1
Header 1 + Image 2 + Copy 3
Header 3 + Image 3 + Copy 2
Header 2 + Image 1 + Copy 1
Header 2 + Image 1 + Copy 1

As you can see, the number of possible variations can stack up quickly, and this is using only three elements. The complexity of a multivariate test can grow exponentially, making it difficult for your team to manage. Software solutions allow you to run multivariate tests more efficiently as they can experiment with a multitude of possible combinations.

As the multivariate test gathers data over time, you'll be able to separate the wheat from the chaff and discern which combination of the variations performed the best. For instance, maybe Header 2 + Image 1 + Copy 3 gets the most conversions, making it the winning combination that you decide to run with from then on.

How To Create A Multivariate Test

When creating a multivariate test, it is best not to include too many elements since every element you include more or less doubles the number of combinations you'll have to experiment with.

Not to mention that all elements aren't created equal. For instance, if your test includes headers, call to action buttons, and footers, you may discover that footer variations make little impact on conversions.

Some good steps to follow when creating a multivariate test (using a landing page example):

1. Use your analytics data to do an evaluation of the page and identify what is and isn't working with it.
2 Once you know which elements are hampering performance, order them based on the amount of damage they're dealing to the page's quality.
3. Formulate a hypothesis regarding the elements you want to test. Ask questions like: If I fix these issues, what impact will it have on the page's conversion rate? What about the page's overall performance?
4. Launch the test, and as it is going, document it. Doing this formalizes the process and makes it easier for others to provide feedback on it later.
5. Once the test is complete, analyze the results. Pay attention to what did or didn't work and conclude whether your hypothesis was correct. You can use the data generated by the test to make appropriate changes to your web-page/app, or you can use it to create follow up tests.

Observability

Observability is defined as the ability of the internal states of a system to be determined by its external outputs. With the unknown unknowns of our software's failure modes, we want to be able to figure out what's going on just by looking at the outputs: we want observability.

"Observability" is the hot new tech buzzword. But is this actually a new concept, separate from monitoring? Or is it just a fancy new term? Today, we'll be explaining observability: what it is, how it differs from monitoring and alerting, and why you should care.

One of the benefits of working with older technologies was the limited set of defined failure modes. Yes, things broke, but you would pretty much know what broke at any given time, or you could find out quickly, because a lot of older systems failed in pretty much the same three ways over and over again.

As systems became more complex, the possible failures became more abundant. To try to fix this problem, we created monitoring tools to help us figure out what was going on in the guts of our software. We kept track of our application performance with monitoring data collection and time series analytics. This process was manageable for a while, but it quickly got out of hand.

Modern systems – with everything turning into open-source cloud-native microservices running on Kubernetes clusters – are extraordinarily complex. Further, they're now being developed faster than ever. Between CI/CD, DevOps, agile development, and progressive delivery, the software delivery train is speeding up.

With these complex, distributed systems being developed at the speed of light, the possible failure modes are multiplying. When something fails, it's no longer obvious what caused it. We cannot keep up with this by simply building better applications. Nothing is perfect, everything fails at some point, and the best thing we can do as developers is to make sure that when our software fails, it's as easy as possible for us to fix it.

Unfortunately, many modern developers don't know what their software's failure modes are. In many cases, there are just too many. Further, sometimes we don't even know that we don't know. And this is dangerous. Unknown unknowns mean you won't put effort into fixing the problem, because you don't know it exists.

Standard monitoring – the kind that happens after release – cannot fix this problem. It can only track known unknowns. Tracking KPIs is only as useful as the KPIs themselves are relevant to the failure they're trying to detect. Reporting performance is only as useful as that reporting accurately represents the internal state of your system. Your monitoring is only as useful as your system is monitor-able.

This concept of monitor-able-ness has a name: observability.

Implementing Observability

The key tools for implementing observability are metrics, logs, and tracing.Metrics are a central part of any monitoring process, but even when you have the right ones, you're necessarily limited by the constraints of linear time. People decide on metrics based on failures they've already found and fixed in the past. But there may be unknown unknowns: failures you haven't seen before, and therefore can't anticipate.

Preemptively checking your metrics to find patterns is an option, but this isn't a replacement for being able to come back quickly from a failure. In short, metrics are necessary, but t sufficient. While metrics should be constantly tracked, you only look at logs when your metrics are showing something strange you'd like to investigate. They're more specific and detailed than metrics, and they exist to show you what happened in each event. Having understandable, queryable, comprehensive logs is a significant component of what separates the observable from the non-observable system.

Tracing is really just a type of logging that's designed to record the flow of a program's execution. Typically, tracing is more granular than standard logging: while logs may say that a program installation failed, a trace will show you the specific exception that was thrown and when during the runtime it happened. Tracing is frequently used to detect latency issues or find out which of many microservices is not working. It's especially useful for error detection in distributed systems, to such an extent that this use case has its own name: distributed tracing.

The biggest problem with all logging, including tracing, is that the volume of data storage becomes prohibitive, fast. Sampling is a possibility, as was implemented in Google's Dapper project, but it's not a perfect solution. For one thing, sampling is not easy or simple: different logs may need to be sampled in different ways and the sampling strategy will need to change over time. For another, sampling is too rigid for some use cases. So while it may be tempting to be like Google, using Google's development processes is only reasonable for companies on the same order of magnitude as Google – if you're smaller, you have much more flexibility.

Different companies implement observability differently. Some track dozens of metrics and some track only a few; some keep all their logs and some downsample them aggressively. Which solution works for you depends heavily on your company, your system, and your resources. But one thing is clear: observability is a real thing, it's important, and systems that implement it from the get-go will be uniquely situated to spring back quickly from failure when it happens.

Server Side Testing

Server-side testing refers to any type of testing – commonly A/B testing, but also multivariate testing or multi-armed bandit testing – that occurs on the web server instead of in the user's browser. This is contrasted with client-side testing, where the test cases are rendered (typically using some type of testing tool) using J avaScript during the loading of the web page.

Benefits of Server Side Testing

One of the main problems with client-side A/B testing is that it's almost certain to impact the user experience in some negative way. If you use asynchronous JavaScript, the original page will load first and then be replaced by the test variation, causing a "flicker effect". If you use synchronous JavaScript, however, the page load time will suffer, and your end user will stare at a blank page until the content loads.

Server-side A/B testing eliminates this problem completely. Since the variation is determined before the resource is served to the visitor's browser, there is no flickering and no negative effect on load time.

Another benefit of server-side testing is the ability to use it for mobile apps, or other environments serving dynamic content. This ability to be "omnichannel" helps businesses test variations across multiple platforms.

The most critical benefit, though, is the ability to test the full stack. Client-side testing tools may be simpler for marketing teams to implement, but they can only test the look and feel of

the website. If a development team would like to A/B test anything on the back-end, like a new algorithm or a different database query, they will need to use server-side testing.

The Lifecycle of a Server Side Test

To run experiments with server-side testing, just like any other A/B testing, you begin with a hypothesis. “We think this new search algorithm will be more effective,” or “we think this simplified user interface will improve conversion rates,” or whatever it is. Once the hypothesis is determined, all variations have to be built.

This is a key difference between server-side and client-side tests. Using the type of WYSIWYG editor provided by many client-side testing tools, marketing and product teams are able to perform tests without having to actually implement the alternate variations. On the other hand, all the different variations in a server-side test must be actually implemented by the developers.

After the test is complete, significance is calculated, and the winning variation is determined, it’s time to roll out the winner. There is another key difference between server- and client-side testing here: in a client-side test, once the winner is determined, it has to be built and implemented, whereas in a server-side test, all variations have already been implemented (and are probably just behind feature flags), so it’s a simple matter of rolling out the winner to all users.

So in short, server-side testing is slower when the alternate versions don’t win (because the developers built them for nothing), but faster when they do (because they’re already built and just need to be rolled out).

Server Side Testing Use Cases

By now, you’ve probably realized that the question is not “server side or client side testing, which is better?” – it’s “server side or client side testing, which is better for you?”

You should probably use server side testing if:

- You want to be able to test across multiple platforms, from web applications to mobile apps
- You want to be able to test across the full stack
- You want to test without it impacting your users on the front-end, either from longer page load time or “flickering”
- You have the developer resources to do a deployment for each experiment

It’s much easier to implement this if you use a server-side solution like Split, which can help you manage all your experiments in one place.

Simpson's Paradox

Simpson's Paradox, otherwise known as the Yule-Simpson Effect, is a reversal paradox where the correlation found in each of several groups either disappears or even reverses when the groups are combined. It's relevant in the context of many non-experimental studies, including A/B tests.

Examples of the Paradox

One classic example of Simpson's Paradox is from a 1975 study of sex bias in graduate admissions at the University of California, Berkeley. When considering the overall figures, there was a significant difference between the percentage of male versus female applicants admitted (44% out of 8,442 men were admitted, vs. 35% out of 4,321 women). MHowever, when considering individual departments, the data showed a statistically significant bias in favor of women (6 out of the 85 departments were significantly biased against men, while only 4 were significantly biased against women).

How can this be? It seems counterintuitive that this result is even possible. Perhaps an example will help.

A Real Life Example

In a medical study, two types of kidney stone treatments were compared: Treatment A (including all open surgical procedures) and Treatment B (a specific, less invasive surgery involving a small puncture). The result was that Treatment A was more effective for small stones (93% success rate vs. 87% for Treatment B) and large stones (73% vs 69%), but Treatment B was more effective when considering both groups together (83% vs 78%).

In this situation, the confounding variable is the severity of the case: doctors are more likely to prescribe the overall less effective but also less invasive Treatment B for less severe cases, whereas more severe cases are commonly prescribed Treatment A. Since more severe cases have a lower base recovery rate, this pulls down the perceived effectiveness of the treatment disproportionately used for them.

A/B Testing

In a hypothetical example, say that a development team is running two variants of a feature for their web application. The metric they're focused on is conversion rate, and they're running the test across two different browsers (Firefox and Chrome). They assign 80 of 100 Firefox users to Variant A (and the remainder to Variant B), and assign 20 of 100 Chrome users to Variant A (and the remainder to Variant B). The conversion rate of Variant B is found to be superior in each browsers individually (100% in Firefox compared to Variant A's 87.5%, and 62.5% in Chrome compared to Variant A's 50%). However, when considering both at once, Variant A is the winner (80% overall).

The confounding variable here is sample size in each browser. The number of users assigned to each variant is significantly different (80 vs 20). As such, the total conversion rate numbers for Variant A is dominated by Firefox, which has a higher conversion rate, whereas the numbers for Variant B are dominated by Chrome, which has a lower conversion rate. If the numbers were equal, we would find that Variant B is the overall winner

Considering Simpson's Paradox

When considering any instance of Simpson's Paradox, it's critical to look for an un-controlled-for third variable (such as department competitiveness, case severity, or sample size), which explains the paradox. Further, since the mathematics allows perfectly well for a difference between aggregate associations and associations in individual groups, the seeming "paradox" of Simpson's paradox arises from inappropriate use of causal inference in non-experimental studies – that is, mistaking correlation for causation.

When considering whether an admissions program is biased, we don't care only about the correlation between the sex of a student and the state of their admission, we care about whether a university department may be biased against certain students.

Determining which of the two relationships – that of the department or that of the whole university – is spurious is dependent, not on statistics, but on other information about the problem. No generalizable conclusion can be drawn about which relationship is relevant, in instances of Simpson's Paradox.

In A/B testing in particular, sample size is the most common confounding variable. This makes it easier to detect Simpson's Paradox and correct it in your feature experimentation. So long as users are evenly distributed between variants, browsers, and any other potentially-relevant categories, Simpson's Paradox is unlikely to show up and confuse your results.

Smoke Testing

Smoke tests are a type of regression test which ensure that your most important, critical functional flows work. These tests should not encompass your entire testing suite, rather they are a subset of tests that cover your top prioritized user flows. These are usually end-to-end tests that are executed in the build pipeline and are blocking, which means if any of them fail in a pull request, they will block the pull request from being merged. This blocking aspect of smoke testing is important because if the code you're trying to push breaks a critical flow, you either need to update the test if the requirements have changed, or fix your code to ensure proper functionality of the feature.

Why Implement Smoke Testing?

Smoke tests should be put in place to ensure that your new code has not broken any existing functionality, and ideally they should be run in production. In software testing, you don't want to be reactive – you don't want to wait until a feature breaks, have a user report it to you, and then push a fix. You want to know if something is wrong before your users ever experience anything wrong. As a developer, smoke testing gives you that confidence you need to know you are releasing new features without breaking existing functionality.

Smoke Tests vs. Unit Tests

Both smoke tests and unit tests should be implemented in your build pipeline. Smoke tests should cover high-level end to end functionality, where unit tests should coversingle component testing. Both should be present, and one should not replace the other.

Statistical Significance

When people discuss A/B testing, they commonly throw around the term “statistical significance” a lot. But what does this mean? In short, getting a statistically significant result means that the result is highly unlikely to be the product of random noise in the data, and more likely to be the result of a legitimate, useful trend.

To understand statistical significance in detail, we’ll need to explain three key concepts: hypothesis testing, the normal distribution, and p-values.

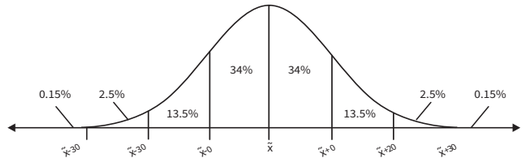
Hypothesis Testing

Statistical hypothesis testing is just a formalization of something simple we do all the time. You begin with an idea about how things might be – we call this the alternative hypothesis – and test it against its opposite – which we call the null hypothesis. For example, when you hear water falling on your sidewalk, your alternative hypothesis might be that it’s raining outside, whereas your null hypothesis would be the opposite: that it is not raining outside.

In a more complex experiment with a larger sample size and less clean-cut results, it’s important to verbalize your null and alternative hypotheses so that you can’t trick yourself – or anyone else – into believing you were actually testing something else.

The Normal Distribution

You may have already heard of or seen the normal distribution: it’s also called a bell curve, though it looks a bit more like a rollercoaster than a bell. The center of the normal distribution is the mean (average) of the data set. The steepness of the curves on either side are determined by the standard deviation, which is a measure of how far away the data gets from the mean. If most of the data is close to the mean, the standard deviation will be small and the curve will be narrow; if most of the data is farther from the mean, the standard deviation will be large and the curve will be fat.



The normal distribution is most useful for finding how anomalous a data point is. The height of the curve at any given point is the probability that a randomly-chosen data point will be that distance from the mean: for example, the probability of finding a data point that is +1 standard deviation from the mean (otherwise known as a z-score of 1) is 34%. So as the curve becomes shorter toward the ends, it becomes less and less probable that any given data point would be found that far away from the mean. In the same way, more standard deviations, or a higher z-score, means a less probable result.

P-Values

The p-value is the probability of observing results as extreme, or more extreme, than those measured, in a world where the null hypothesis is true.

Before we begin any experiment, we should decide on a p-value todetermine our minimum significance level. This value, commonly called alpha, is most commonly set at 0.05 (which is why you'll commonly hear scientists and statisticians say that a result is significant "at $p<0.05$"). However, values between 0.1 and 0.001 are commonly used, depending on the discipline.

When setting alpha for your experiment, be aware that it directly corresponds to your confidence interval. If you have a study done at $p<0.05$, you can be 95% confident those results are significant, and not a result of random noise. If you use $p<0.0000003$, as the physicists who discovered the Higgs Boson did, you can be 99.99997% confident the results are significant. Consider the risk of your experiment: you may be alright with a confidence level of 90% if you're testing a relatively low-impact low-risk feature, whereas if you're testing something mission-critical you may want a confidence level closer to 99.7%.

How Statistical Significance is Calculated

Putting all these three pieces together, let's do a simple example with minimal math. Let's say we have a coin, and our alternative hypothesis says it's weighted towards heads (which means our null hypothesis is that it's balanced, aka, not weighted toward heads or tails). We choose an alpha of $p<0.05$ for our test. Now let's say we flip the coin 10 times, and 9 times it comes up heads.

The p-value for that outcome, according to the normal distribution, is 0.01 – so we have significant evidence to reject the null hypothesis.

That's it – this really is that simple. Even with more complex experiments, you'll need about three lines of Python or R code for the calculation of the p-value, and a little bit of algebra for the other pieces.

T-Test

A t-test is a type of hypothesis test which assumes the test statistic follows the t-distribution. It can be used to determine if there is a statistically significant difference between two groups.

The t-test is the hypothesis test of the t-distribution. The t-distribution is a particular kind of probability distribution, similar to the normal distribution but the variance is estimated rather than known. There are various different types of t-tests; any hypothesis test which relies on the assumption that the parameter of interest follows a t-distribution falls under the t-test family. A t-test results in a t-score, which can then be translated to a p-value for easier interpretation and to determine statistical significance.

There are different versions of the t-test that can be used in different scenarios, the three main types are:

Independent Samples T-Test

This is the type most commonly used in online experimentation. It compares the means for two independent groups. For example, when you randomly assign all visitors to a website into one of two groups, you are creating two separate samples of visitors who are independent from each other. The independent samples t-test can be used to test for differences between the average behavior of users in those two groups.

Paired Samples T-Test

This type of test is used for paired data, when each measurement in a sample is paired with a measurement from the other sample. For example in a repeated-measures design, each pair may contain measurements for the same unit before and after a treatment, or in a matched-pairs design each unit may be matched with a similar unit from another sample.

One Sample T-Test

The one-sample t-test can be used to determine whether the mean of a single sample differs from a particular value. For example, it could be used to determine whether the average exam score for a class of students differs from a particular target.

Testing in Production

Testing in production is the process of testing your features in the environment that your features will live in. It does not mean releasinguntested code to users and hoping it works, but rather using feature flags to test the different treatments. This is best implemented in addition to pre-production testing processes, and should not replace all testing.

Deployment vs. Release

In order to explain testing in production, we should first explain the difference between deployment and release.

Deployment means pushing a piece of new code live to the production environment. It does not mean it is actually handling production traffic. Deployment, as such, is a near-zero-risk activity to end users. Release, on the other hand, is the act of exposing end users to a new version. This is what actually impacts user experience, and as a result, it can be risky.

From these definitions, technically we should be saying “blue green release” instead of “blue green deployment”, and “canary release” instead of “canary deployment”, because what these tools are actually impacting what the customers see, not just what code is in production. And likewise, it’s not technically a bad deploy; it’s a bad release that causes outages and angry customers.

The Shortcomings of Staging

For software testing to be effective at predicting how a rollout will perform under the stress of production traffic, the test environment has to be as close to the production environment as possible. One way to attempt this is to maintain a staging environment, and try to keep it as in-sync with production as possible.

However, the trouble with this is that differences between staging and production systems occur regularly, often of necessity. For example, different instances of stateful systems like databases must be run in order to maintain data integrity. Further, the staging environment is commonly on a differently-sized cluster than production, with different configurations for things like load balancers and queues, and with less monitoring.

Again, most of this can be fixed or mitigated, but trying to do that necessitates having a group of software engineers spend a lot of time ensuring staging is as close to production as possible. And because production is constantly changing, this time has to be spent continuously.

This isn’t to say that staging is wholly unnecessary, or not useful. It is saying, though, that testing in staging should not be the only testing you do before you release to your end users. Some tests can be done perfectly fine in staging – but other tests work much better in production with production data. Ideally, testing in staging should be a precursor to testing in production. Neither one should replace the other.

How to Test in Production

One of the best ways to eliminate the risks of testing in production is with feature flags. Essentially, a feature flag is an if/then statement which is wrapped around a new feature, allowing the software development team to turn that feature on and off without deploying any code changes.

This prevents a lot of the worst failure modes of testing in production. It enables disaster recovery by providing a kill switch for each feature, allows for near-real-time monitoring of feature releases to check for performance degradation, and prevents testing from creating a poor user experience. Further, feature flags allow for more in-depth A/B testing, easy canaryreleases, unique benefits for DevOps organizations, and even increased observability.

With a feature flag management system like Split, it's easier than ever to effectively manage production testing and gain the benefits of testing on real users with production data.

Type I Error

A type I error (or type 1 error), also called a false positive, is a type of statistical error where the test wrongly gives a positive result, when a perfect test would report a negative. It is one of four possible results from a hypothesis test.

What is Hypothesis Testing?

Hypothesis testing is the process of testing a hypothesis against its opposite to find whether it's true or not. If we want to test if two variables are related, the alternative hypothesis states that there is a significant relationship between them, and the null hypothesis states the opposite, that there is no relationship. [Note that we don't test two hypotheses at once. If our alternative hypothesis is that there is a positive correlation between two variables, our null hypothesis is not that there is a negative correlation: it's that there is not a positive correlation (so, there is either a negative or no significant correlation). The null hypothesis is always the mutually exclusive converse of the alternative hypothesis.]

In any hypothesis test, there are two possible results: we accept the null hypothesis (ex. if there is no significant difference between our variables), or we reject the null and accept the alternative (ex. if there is a correlation of statistical significance).

The ideal statistical test always accepts a true null hypothesis and rejects a false null hypothesis, but every test has some margin of error (corresponding directly to your confidence interval/p-value). As such, there are really four possible outcomes: we accept the null correctly, we accept it incorrectly, we reject the null correctly, or we reject it incorrectly.

A type I error is the incorrect rejection of the null hypothesis; a type II error is the incorrect acceptance of it.

When a Type I Error is Acceptable

Completely eliminating type I and type II errors is a statistical impossibility, but the design of a test can impact the amounts of each. Depending on your situation, a type I error might be an acceptable trade-off.

In medical screenings, for example, a simple and inexpensive test is administered to a large number of people, none of whom present any symptoms. These are designed to bring to the doctors' attention anyone who has any significant change that would indicate they have the condition, which means the rates of false negatives (type II errors) must be minimized. As a result, a large number of type I errors are made. The false positives from medical screenings are typically sorted out afterward by more complex and expensive tests.

Airport security in the United States is another example of a situation where type II errors are minimized by producing a large number of type I errors. The overwhelming majority of times that a detector goes off, it's because of something inconsequential: a watch, a shirt made of a peculiar fabric, a metal lunchbox, a belt buckle. But since airport security minimizes false negatives, it creates a lot of false positives.

When a Type I Error is Unacceptable

In any situation where type II errors must be minimized, an abundance of type I errors is typically the by-product. Conversely, if type I errors are unacceptable, then type II errors typically take their place.

In email spam detection systems, for example, type I errors (pushing non-spam email to the spam folder) are much less acceptable than type II errors (mistakenly leaving spam alone in the inbox). The former causes significant annoyance to the user, because they are missing potentially important emails; the latter causes minor inconvenience, because they must manually delete the spam. An optimized spam filter will filter out as many spam emails as possible while ensuring no non-spam emails get marked as spam.

In general, what types of errors are more acceptable to you determines the setup of your test. To some extent, both errors can be minimized at once, but it's impossible to completely eliminate either one, and minimizing one tends to increase the rates of the other. Building in a system to handle the types of errors your test throws (such as the additional tests administered to people who test positive in medical screenings, routine pat-downs and bag searches in airport security, or a "mark this as spam" button) will improve your overall user experience.

Type II Error

A type II error (type 2 error) is one of two types of statistical errors that can result from a hypothesis test (the other being a type I error). Technically speaking, a type II error occurs when a false null hypothesis is accepted, also known as a false negative.

In any hypothesis testing situation, the null hypothesis states that the subject of the test is not significantly different in the experimental versus the control group, and so any difference observed is the result of some error. The alternative hypothesis, by contrast, states that there is a significant difference.

As a result of this setup, there are four possible outcomes from any hypothesis test:

1. We reject a false null hypothesis
2. We reject a true null hypothesis
3. We accept a true null hypothesis
4. We accept a false null hypothesis

1 and 3 are correct inferences; 2 is a type I error (a false positive), and 4 is a type II error (a false negative).

When Type II Errors are Acceptable

Since it's statistically impossible to entirely eliminate both type I and type II errors, individuals performing experiments must decide which type of error is more acceptable to them and structure their experiments to eliminate the less acceptable one as much as possible.

As an example of when a type II error might be more acceptable than a type I error, let's look at email spam checking. The alternative hypothesis is that the email is spam, and thus the null hypothesis is that the email is not spam. Committing a type I error means marking a legitimate email as spam, preventing its normal delivery. Committing a type II error means a spam email being marked as legitimate and sent to the user's inbox.

A significant number of type II errors points to an ineffective spam filter, but a significant number of type I errors means the spam filter is overall doing more harm than good by preventing users from seeing legitimate communications. Therefore, the goal of email spam filtering systems should be to bring down the number of type II errors while keeping the number of type I errors at near-zero.

By contrast, in a biometric security system, such as a fingerprint scanner on a mobile phone or facial recognition software on a personal computer, then the alternative hypothesis is "the scanner doesn't identify the person on its list of authorized users" and thus the null hypothesis is "the scanner does identify the person on its list of authorized users".

In this situation, a significant number of type II errors would mean an insecure device, whereas a significant number of type I errors would mean some minor user inconvenience of needing to demonstrate their authorization another way (such as with a password or pin code). Therefore, the syem should be designed to bring down the number of type I errors while keeping the number of type II errors at near-zero.

How to Minimize Type II Errors

Because they arise from the design of the test, minimizing a certain error type requires altering the test. To minimize the number of type I errors, decreasing the p-value increasing the confidence interval) is an easy way. To minimize the number of type IIerrors instead, either increase the sample size (or run the experiment for a longer time,in some cases), or increase the p-value.

Usability Testing

A key part of the software development process, usability testing provides invaluable feedback on the user experience of a product. Usability testing involves conducting real-world tests with a segment of your customer base. The goal is to conduct real-time test sessions asking the end user to complete tasks using your product for evaluation of its ease of use, as well as to identify problems that might negatively impact the user experience.

There are many advantages of usability testing beyond the benefits of other testing methods. Internal pre-production, unit tests or alpha testing can help you identify technical issues or bugs, but conducting user testing with real users will provide you with much more qualitative feedback. This can include:

- Comments on user experience from real users directly to the development team
- Resolution of internal debates about usability issues
- Identification of unforeseen usability problems ahead of launch
- Reduced risk of product failure through experimentation

There are many ways to conduct usability testing, but there are a few common features of the testing process in all usability studies. First, there are two groups: observers (typically one or more usability experts), and participants. The observers can be physically present, or sessions can be recorded for later analysis. Participants can be chosen in a variety of ways:

Hallway Testing

This type of testing relies on people chosen randomly from passing foot traffic, e.g. in a hallway or outside on a sidewalk. This testing method is more useful earlier in the design process, as it can help development teams identify major issues or “brick walls” that create major usability issues for the target audience.

Remote Usability Testing

Remote testing is most commonly chosen via a third-party service or software and conducted via interactive meeting tools such as Zoom, with the observer and end user located in different locations and/or time zones. Alternatively, the test sessions can be conducted at different times, at the participant’s convenience, for later review by the observer or development team.

Expert Review

This testing method makes use of an outside usability expert or consultant to assess the user experience. These researchers often evaluate user experience using the 10 usability heuristics developed by Jakob Nielsen, including factors such as user control and freedom, error prevention, efficiency of use and more.

A/B testing

Already a common type of testing for web design, product features, e-commerce and more, A/B testing exposes the target audience to two different versions of a website or feature and records the user experience with each. Test results are compared to determine which version, A or B, exhibited more or fewer usability issues.

How you conduct usability testing will depend on the method you choose for user research. In general, you should choose one part of your product for testing, choosespecific tasks for the end user to accomplish, and set standards for success.


split