

# Progressive Delivery Patterns and Anti-Patterns

## Faster, Safer, Data-Driven Releases

**DAVE KAROW**  
CONTINUOUS DELIVERY EVANGELIST AT SPLIT

### CONTENTS

- Introduction
- The Rise of CI/CD
- What Is Progressive Delivery?
- Benefits of Progressive Delivery
- Common Patterns
- Implementing the Feature Delivery Pattern
- Final Thoughts

Best practices for developing, building, deploying, and operating software have evolved significantly over the last two decades. Software delivery cycles no longer take 18 months, or even six months; it's now just a matter of weeks, days, or even hours. Two of the biggest developments were the adoption of continuous integration and continuous delivery (CI/CD).

## THE RISE OF CI/CD

### CONTINUOUS INTEGRATION

The original idea behind continuous integration was to find problems faster, and to get away from postponing problem discovery, merging issues, and identifying bugs until late in the process when they are harder to resolve. [As Martin Fowler once said:](#)

*"Continuous integration doesn't get rid of bugs, but it does make them dramatically easier to find and remove."*

Continuous integration required three things:

1. A centralized source code repository.
2. Tests, mostly at the unit-test and integration-test level, that could be run automatically and very quickly.
3. A CI server or service to sweat the details so that you wouldn't have to.

The idea was to check in your code, and then what needed to happen would just happen.

Every. Single. Time.

CI challenged people to focus on building smaller chunks of their

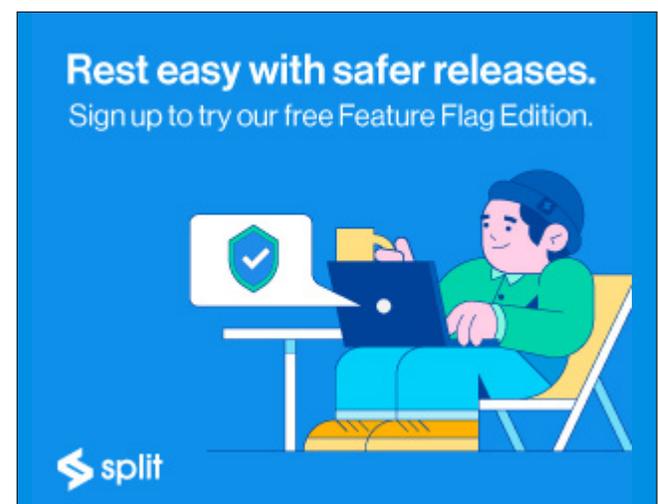
solution at a time and to use mocks or virtual services so that each commit could be checked in on its own, and yet, they could still be tested and validated if other bits weren't ready.

### CONTINUOUS DELIVERY

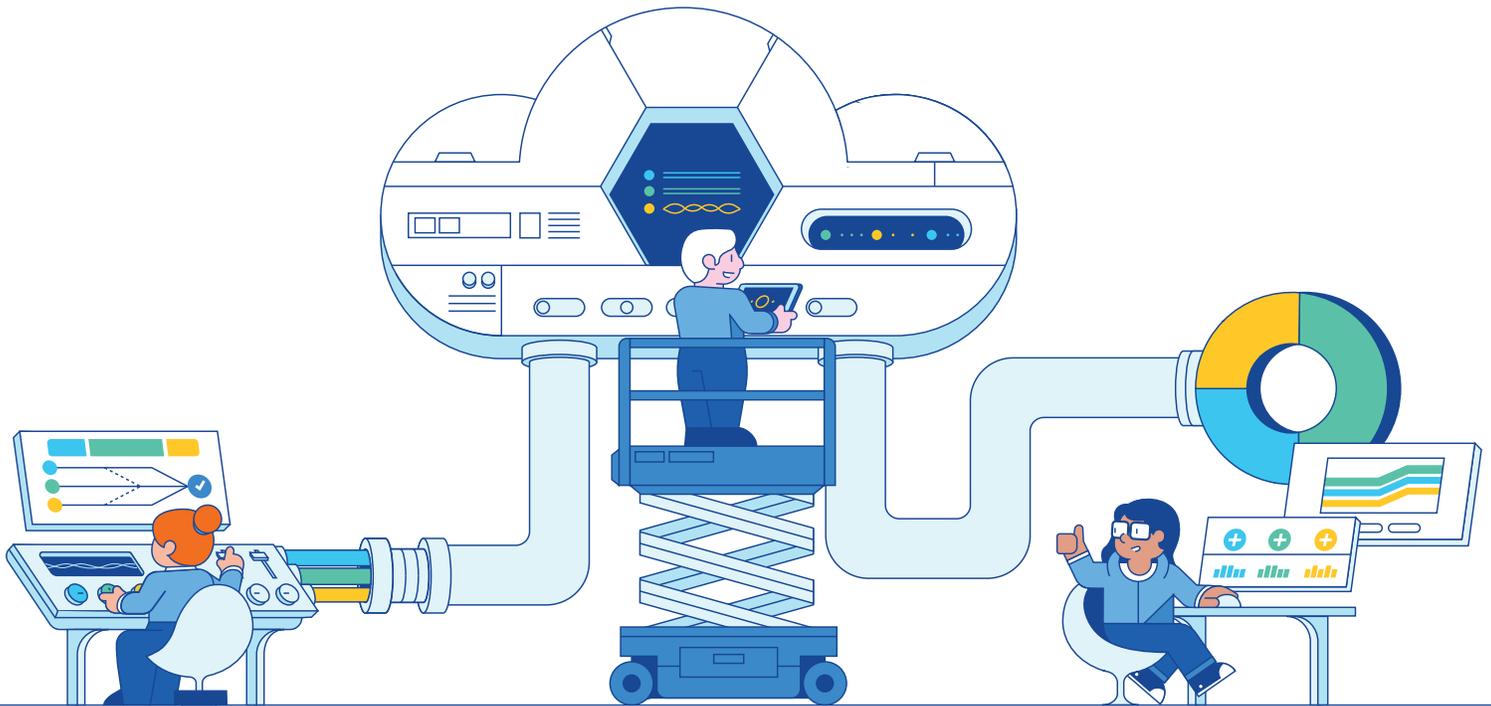
Then along came continuous delivery, the "radical" idea that deployments shouldn't be labor-intensive, high-drama, multi-hour events that must happen outside of normal business hours.

In an [interview with Jez Humble](#), co-author of the book on continuous delivery, [Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation](#), Humble said:

*"We just didn't want to spend our weekends in data centers doing releases anymore. We thought it was a shitty way to spend our time and it was miserable for everyone. We actually want to enjoy our weekends. It was really about making releases reliable and boring."*



# Skip the hotfixes and rollbacks with Split's **Feature Delivery Platform.**



**Manage**  
feature flags

**Monitor**  
release errors

**Experiment**  
with A/B tests

Confidently release features as fast as you develop them.  
Keeping your customers (and engineering teams) happy.

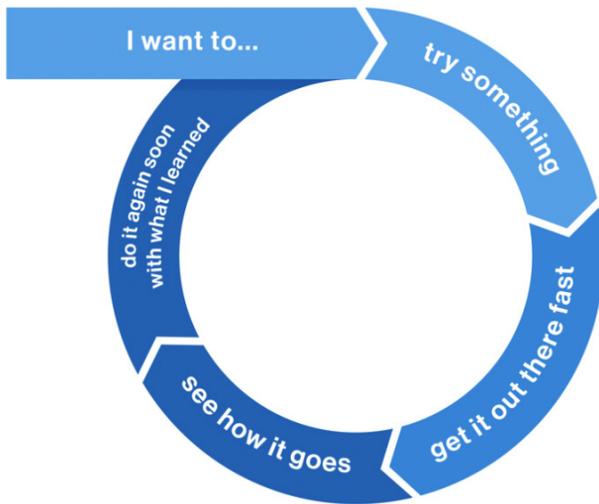
Try it for free at [Split.io/signup](https://split.io/signup)



*"[Continuous Delivery] reduces the ongoing cost of evolving your software because what you're fundamentally doing is reducing the transaction cost of pushing changes. So, you can put changes out more often, at a lower cost."*

Continuous delivery sought to lower the cost (in time and talent) of delivering change. This goal of frequency and low drama required better ways to limit risk and observe the business impact.

If you can do releases often with less effort, then it's much easier to achieve a fast feedback loop, which is the fundamental objective we're aiming for in the first place:



## WHAT IS PROGRESSIVE DELIVERY?

This Refcard will dive into greater detail on what progressive delivery is, why it's being adopted, and how you can get started. Let's begin with this [brief definition by Carlos Sanchez](#), Sr. Cloud Software Engineer at Adobe:

*"Progressive delivery is the next step after continuous delivery, where new versions are deployed to a subset of users and are evaluated in terms of correctness and performance before rolling them to the totality of the users and rolled back if not matching some key metrics."*

### PROGRESSIVE DELIVERY

Progressive delivery emerged as a natural response to concerns raised by the idea of "continuous" anything; if teams were going to move faster and release more often, then the surface area for things going wrong would likely be bigger. How could that be managed? And better still, how could risk be reduced while simultaneously increasing the value of moving fast?

The actual term was born out of a conversation between Sam Guckenheimer, head of product for Azure DevOps, and James Governor, Redmonk analyst. Sam was describing Azure DevOps' staged deployments around the world and how they used feature

flags to gradually expose functionality to particular users. Sam called that practice "[progressive experimentation](#)":

*"Well, when we're rolling out services, what we do is progressive experimentation because what really matters is the blast radius. How many people will be affected when we roll that service out and what can we learn from them?"*

Before we consider several strategies for implementing progressive delivery, let's take a deeper look at the benefits each provide. In other words, what goals are met by implementing progressive delivery?

## BENEFITS OF PROGRESSIVE DELIVERY

### REDUCE DOWNTIME

We used to accept planned outages for "system upgrades" as mildly annoying but normal. Not anymore. No one enjoys logging onto a website to see a message declaring that the system is temporarily down for maintenance, or to launch a mobile app and find it mysteriously unresponsive.

For services that are consumed by other services (such as credit card processing, shopping cart providers, and authentication providers), planned downtime isn't just annoying, it's simply unacceptable. This is the first goal of progressive delivery — to get away from bringing down an entire service just to install a new release.

### LIMIT THE BLAST RADIUS OF UNINTENDED CONSEQUENCES

No matter how much planning, testing, and simulation is put into a release, it's likely that something will eventually go wrong. When it does, we want to limit the blast radius (the extent of the impact), both in terms of scope (the number of users impacted) and duration.

### FACILITATE HIGH CADENCE (FLOW)

Years of research by DevOps Research Associates (DORA) have proven that higher cadence delivery, or "flow" as some like to call it, is closely related to lean manufacturing concepts like limiting work in progress (WIP), small batch sizes, loosely coupled architecture, and empowering individual teams.

In his latest book, [The Unicorn Project](#), Gene Kim advances the ideal of "locality and simplicity," which best sums up these ideas:

*"If a team needs to schedule a deployment and it requires 40 to 50 other teams to work with them into the schedule, nothing will ever get done."*

With that in mind, think about this goal of progressive delivery as improving the ability for smaller, independent deliverables to make it to production and to remain as isolated as possible from the progress and/or stability of other teams' work.

## LEARN FASTER

As we saw above, Sam Guckenheimer used the term “progressive experimentation” and focused on two key goals. The first (the one most people think of) was limiting the blast radius, and the second one (less often thought about) was to learn as much as possible from users exposed to a new release.

Learn faster is seldom “built-in” to tooling by those just starting out on the progressive delivery journey. Instead, ad-hoc “checking to see if everything’s OK,” i.e. nothing is burning the system to the ground, often suffices, and subtle changes are missed altogether.

If the learning component of your implementation is left to manual work done by highly skilled (and scarce!) data scientists, you can bet it won’t be performed every time you release. To make matters worse, a gradual release can make it more difficult to see changes in KPI’s because there is no sudden fluctuation as there would be in a big-bang, all-or-nothing release.

Learn faster is about discovering the real-world, in-production impact of a release on system health and business KPIs before calling the release “done” and rolling it out to users. Automating this capability is a bit like moving from manual runbooks to SRE-built automation.

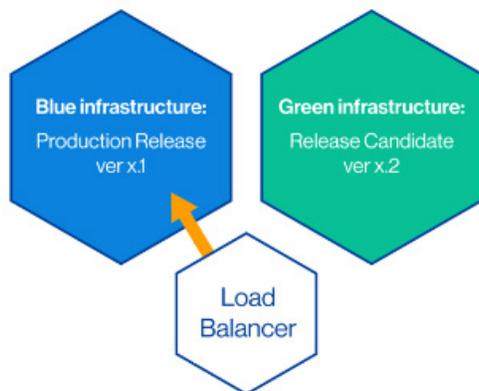
## COMMON PATTERNS

Now that we have a better understanding of where progressive delivery came from and the goals it helps achieve, let’s move on to four common patterns for implementation and see how each helps us meet one or more of those goals.

### BLUE-GREEN DEPLOYMENTS

#### PATTERN

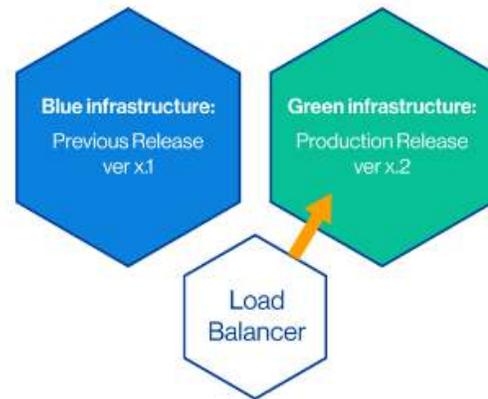
With blue-green deployments, you have your production running on the “blue” infrastructure, and then you stand up “green,” which is a copy of your production infrastructure. You take your time installing the new release on green, do your smoke tests, and make sure everything is good to go.



When you believe you’re ready, it’s time to make a clean cut over from blue to green, routing your production traffic there.

If you missed something in testing that shows up when you go to green, reverting is really easy: just switch back to blue.

If things go well, you stay on green. Then, you recycle the blue environment to become the next staging area. That’s blue-green.



#### GOALS MET

##### Avoid Downtime

Blue-green is great at avoiding downtime because you can take as long as you want for preparation, and then you can instantly cut traffic over when you are ready.

##### Limit the Blast Radius (Half Credit)

For limiting the blast radius, blue-green gets half credit, and that’s for the duration of issues. Since blue is still up and unchanged before the release, you can switch right back to it in minutes if you have to. In terms of scope, it gets no credit. You may have shown all your users something horrible for a few minutes or been entirely down.

##### Achieve Flow (No Credit)

Blue-green doesn’t advance the goal of flow because it’s still an all-or-nothing, “big-bang” release of all payloads in the deployment. Everything either goes live all at once or gets turned off to await the next deployment. In other words, the “working” portions of a deployment get shut off along with the broken bits.

##### Learn During the Process (No Credit)

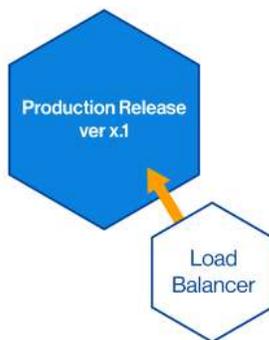
There’s nothing inherent to blue-green that helps you learn. It doesn’t help you focus on each of the things you just changed in this release. How are they impacting system health or user behavior? It’s hard to tell because you’ve got one big release that you’ve exposed all users to at once.

| Benefits                 | Blue/Green Deployment |
|--------------------------|-----------------------|
| Avoid Downtime           | ●                     |
| Limit the Blast Radius   | ◐                     |
| Limit WIP / Achieve Flow | ○                     |
| Learn During Process     | ○                     |

**CANARY RELEASES**

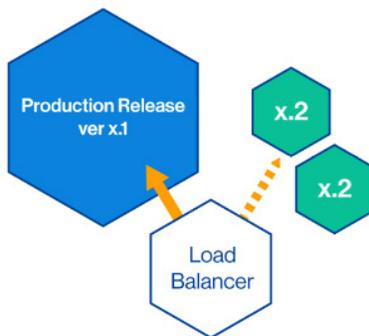
**PATTERN**

In a canary release, your infrastructure is already up and running.



Let's say there are 100 servers and you want to try out your new release on just two of the servers, sending 2% of your population there.

You will need to create logic somewhere to figure out how to route those users, and you'll need to decide whether the user routing decision needs to be sticky (it probably does).



You build a replica of production, install changes on it, and run smoke tests. Next, you route part of your real traffic to these canaries and pay close attention to system health (error rates, response times, CPU/memory stats, etc.).

If production traffic succeeds on your canary, you expand it to replace production. If anything goes wrong, you just drain the traffic from the canary and route back to production.

**GOALS MET**

**Avoid Downtime (Full Credit)**

Canary is great for avoiding downtime. As with blue-green, you can set up your new release on your own time and only direct traffic to it after you are sure everything is ready and checked out.

**Limit the Blast Radius (Full Credit)**

Canary does a great job of limiting the blast radius in terms of both duration and scope. You can limit the duration by simply updating routing away from the canaries and letting the current canary traffic drain off. Since you are only going out to 1% or 2% at the beginning, you are also doing a great job limiting the scope.

**Achieving Flow (No Credit)**

Canary doesn't really do a great job helping with flow because as with blue-green, we are exposing all of the release payloads in the deployment as a single whole. If one of those payloads has an issue, we "kill" the entire canary and start over.

**Learn During the Process (1/4 Credit)**

Canary gets a quarter credit here because if you are going out to just two servers, you can pay a lot of attention to those two servers and you are probably going to see obvious things like CPU, network traffic, etc. That's a big red flag.

The 3/4 credit that Canary doesn't get for this goal is due to the fact that if you have multiple changes in that deployment, you still don't know which one (or more) is causing problems or leading to undesirable changes in user behavior. Bottom line? We may know "something is not right" but the feature that is causing the problem doesn't shout, "Hey, it's me!"

| Benefits                 | Canary Release |
|--------------------------|----------------|
| Avoid Downtime           | ●              |
| Limit the Blast Radius   | ●              |
| Limit WIP / Achieve Flow | ○              |
| Learn During Process     | ◐              |

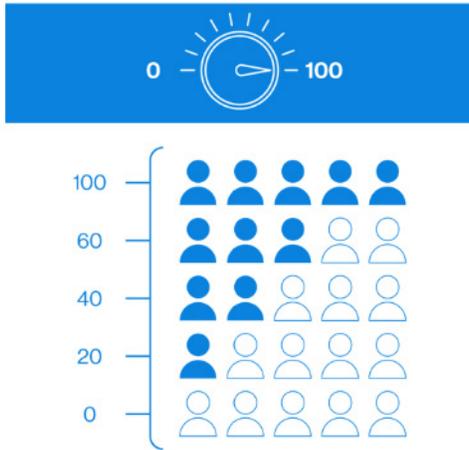
**FEATURE FLAG ROLLOUTS**

This was a big leap in the evolution of the idea of progressive delivery. It's no coincidence that the term emerged after many teams started using feature flags to roll out features gradually, essentially doing a canary release at the feature level. Another important difference we'll see below is that you can use user attributes to decide who should get new features first.

**PATTERN**

With feature flag rollout, you deploy the code with the new features turned off. Once the code with flags is in place, you can turn it on and off whenever you want, for as many users or as few as you want.

To begin your rollout, you might first expose the code just to the dev team for a final smoke test on production. Next, you might dogfood, exposing the new features to your employees only (again, this is on production).



If things are still going well, you start to ratchet up the rollout into your actual user population. This gives you the chance to expose new features to 5%, 10%, 20% of your users in steps as you go, and to be able to see how it's going before ramping higher.

Reverting your release is quite simple: You just ratchet back the feature flag setting. If you had it out to 10% of your real users and you find some unexpected issues, you might go back to just dogfooding or even back to just the dev team. More importantly, you don't need to patch, re-deploy, or even change network routing rules. You are still deployed to production, but you're not exposing the new code to your customers.

**GOALS MET**

**Avoid Downtime (Full Credit)**

Feature flag rollouts avoid downtime because there is no deployment needed to turn them on or off. To get the code in place with the flags turned off, you might still use blue-green or canary.

**Limit the Blast Radius (Full Credit)**

Feature flag rollouts check the box really well here in terms of both duration and scope. Duration is very short because it does not take a deployment or a hotfix to undo a feature flag rollout. You just send a different signal to the system. In terms of scope, the typical pattern is to roll to non-customers and then very few customers at the start, so the chances of a broad-scoped impact are very small.

Some in-house feature flagging solutions only support simple on/off toggles and thus are not capable of percentage-based rollouts. In that case, the behavior is more like blue/green and the ability to limit the blast radius applies to duration only (half-credit). In terms of scope, those systems get no credit. You may have shown all your users something horrible for a few minutes or been entirely down.

**Achieving Flow (Full Credit)**

Feature flag rollouts really shine here. Since each feature has its own flag, each one is independent. You may have dependencies you want to enforce, but no team has their deliverable stuck on a release train with 15 other pieces of payload. Barring any dependencies, if one feature in the deployment has issues in production, it can be ramped back to developers only while the other features can continue to ramp up.

**Learn During the Process (No Credit)**

Feature flags, by themselves, don't really help here. There's nothing inherent to using feature flags that let you know which feature is causing which problem. If you take 10 features live in a release and you're ramping them up and things start to go wrong, you still don't know which one is causing it without running an incident and doing triage work. Observation of these rollouts is challenging with traditional tools because unlike canary, small populations running the new code are mixed in with larger populations running the status quo on the same infrastructure.

| Benefits                 | Feature Flag Rollout |
|--------------------------|----------------------|
| Avoid Downtime           | ●                    |
| Limit the Blast Radius   | ●                    |
| Limit WIP / Achieve Flow | ●                    |
| Learn During Process     | ○                    |

**FEATURE DELIVERY PLATFORMS**

**PATTERN**

Feature delivery platforms marry the gradual release capabilities of feature flag rollouts with the automated ingest and statistical computation of KPI differences between the status quo and new code. Simply put, feature delivery platforms provide both a control mechanism to determine "who gets what" and a "sensemaking" mechanism to answer the question: "Did we accomplish what we set out to do without making something worse in the process?"

Contrast this with the feature flag rollout pattern where the “sensemaking” (if done) is performed separately by ad-hoc query or exploratory log analysis. When the stakes are high, that often means a war-room full of highly skilled experts standing “on alert” to determine if everything is going as planned, and if not, to figure out where the issue(s) are.

### Key Success Criteria

To be successful, feature delivery platforms must have control and sensemaking mechanisms that are automatic, proven, and repeatable. In the most mature implementations, these capabilities are used for every single release payload as the standard operating procedure. This is a proactive rather than reactive approach. It’s also vastly more scalable because it allows experts to focus on novel problems, not watching every release payload as it goes live.

### Guardrail Metrics

Mature feature delivery platform implementations also have a key capability known as guardrail metrics or “do-no-harm” metrics.

In the physical world, a guardrail provides “feedback” if your car gets off course. Your car will bang into it and may get a minor scrape, but you won’t fall off a cliff.

Guardrail metrics do the same thing for us in fast-moving continuous delivery environments. The idea is to automatically calculate “do-no-harm” metrics without asking individual development teams to perform extra work. This allows teams to focus on their objectives but still learn whether they are negatively impacting the business before rolling out to 100% of users.

Imagine that your development team is working to drive users to create more “tasks” in their solution. Initial results are quite good, with an 11.2% increase in task creation. Here, the guardrail metric might be average initial page load time, alerting the team that they have unintentionally increased response time by 25.64%. Without automatic calculation of guardrail metrics, this impact might have been missed, and the new code could have been rolled out to all users, leading to churn of users due to unacceptable latency.

### GOALS MET

The first three goals (Avoid Downtime, Limit the Blast Radius, and Achieving Flow) are all met with the same “full credit” score earned by feature flag rollouts because the feature delivery platform pattern is a superset of the feature flag pattern.

### Learn During the Process (Full Credit)

Learning during the process is where feature delivery platforms differentiate themselves from all of the other patterns we’ve covered so far. The difference is the built-in feedback loop at the individual

feature level, allowing easy side-by-side comparison of populations running the new code and running the status quo. This increases the value of each build-measure-learn iteration your teams perform.

Consider again a deployment containing 10-15 separate release payloads (i.e. features or bundles of new/changed code). Unlike blue-green, canary, or feature flag rollouts, each payload is continually evaluated to determine if it is accomplishing its goal or doing harm. When issues arise, instead of running manual triage to look for a needle in the haystack, the needle “phones home” shouting out: “It’s me! I’m the one hurting users!”

These platforms amplify the impact of engineering and operations resources by separating signals from noise so teams can focus on lessons learned, not manual observation and triage.

### EXAMPLES IN THE WILD

To learn more about early pioneers that built these platforms to improve their outcomes, look into Microsoft’s internal platform, [EXP](#), and the in-house systems at [LinkedIn](#), [booking.com](#), and [Wal-Mart](#).

| Benefits                 | Feature Delivery Platform |
|--------------------------|---------------------------|
| Avoid Downtime           | ●                         |
| Limit the Blast Radius   | ●                         |
| Limit WIP / Achieve Flow | ●                         |
| Learn During Process     | ●                         |

## IMPLEMENTING THE FEATURE DELIVERY PATTERN

Use the checklists below to establish or extend the feature delivery platform pattern in your own environment. It is essential these capabilities be accessible to any member of your team. This should not require re-inventing the wheel by separate teams or ad-hoc investigation by a subject matter expert or “on-call” resource.

### FOUNDATIONAL CAPABILITIES CHECKLIST

#### DECOUPLE DEPLOY FROM RELEASE

Create a consistent, organization-wide mechanism for controlling exposure of new code:

- Allow changes of exposure without new deploy or rollback
- Support targeting by UserID, attribute (population), random hash

### AUTOMATE REPORTING OF RELEASE EXPOSURE

Automate a reliable and consistent way to answer, “Who have we exposed this to so far?”

- Record who hit a flag, which way they were sent, and why
- Confirm that targeting is working as intended
- Confirm that expected traffic levels are reached

### AUTOMATE REPORTING OF RELEASE IMPACT ON SYSTEM HEALTH & USER BEHAVIOR

Automate a reliable and consistent way to answer the question, “Did we accomplish what we set out to do, without making something worse in the process?”

- Compare system health (errors, latency, etc.) between populations exposed to new code and status quo.
- Compare user behavior (business outcomes) between populations exposed to new code and status quo.
- Automatically compute comparisons of “guardrail metrics” between populations exposed to new code and status quo to avoid the local optimization trap

### THE TWO PRIMARY USE CASES

The three essential capabilities can be applied to address these two key use cases:

#### RELEASE FASTER WITH LESS RISK

Limit the blast radius of unexpected consequences so you can replace the “big-bang” release night with more frequent, less stressful rollouts.

- Ramp in stages, starting with dev team, then dogfooding, then % of public
- Monitor at feature rollout level, not just globally (vivid facts vs. faint signals)
- Alert at the team level (build it/own it)
- Kill if severe degradation detected (stop the pain now, triage later)
- Continue to ramp up healthy features while “sick” are ramped down or killed

### ENGINEER FOR IMPACT (NOT OUTPUT)

Focus precious engineering cycles on “what works” with experimentation, making statistically rigorous observations about what moves KPIs (and what doesn’t).

- Target an experiment to a specific segment of users
- Ensure random, deterministic, persistent allocation to A/B/n variants
- Ingest metrics chosen before the experiment starts (not cherry-picked after)
- Compute statistical significance before proclaiming winners
- Design for diverse audiences, not just data scientists (buy-in needed to stick)

## FINAL THOUGHTS

When considering new ways of doing work, it’s useful to be clear on the “why” before figuring out the “how.” In this Refcard, we took time to identify the goals for progressive delivery before exploring the available implementation patterns.

Higher cadences of delivery simultaneously increase the chance for things to go wrong, but also the surface area for learning. Progressive delivery patterns are a proven way to reduce the risk of unforeseen consequences, and depending on your implementation choices, can increase the value of each release iteration.

#### Written by Dave Karow,

*Continuous Delivery Evangelist at Split*

Dave has three decades of experience in developer tools, developer communities and evangelizing sustainable software delivery practices. Dave grew up just “off-campus” from Stanford as Silicon Valley morphed from defense to chips to software and finally internet services.

Dave’s front-row seat to those changes equips him with a unique point of view on the long arc and repeating themes of technology evolution. As CD Evangelist at Split Software, Dave speaks on aligning progressive delivery (i.e. gradual rollouts of new code) with observability of system health, user experience, and user behavior. Before Split.io, he democratized “shift left” performance testing at BlazeMeter.

