Access the Data-Driven Layer of Progressive Delivery with Bugsnag

SMARTBEAR

Contents

Access the Data-Driven Layer of Progressive Delivery with Bugsnag

The odds are you've noticed your favorite web applications previewing a new look or feature. For example, Google might encourage you to opt in to a new Gmail interface, or Facebook may subtly change how a comment looks. These small changes represent a huge paradigm shift from the massive overhauls commonly seen in the past.

Tech companies regularly roll out small changes to a tiny subset of users to ensure they improve the user experience and don't introduce any errors. Rather than trying to catch every error in automated tests, they're trying to minimize the blast radius of errors that reach production. As a result, users likely see fewer errors and have a better experience.

In this e-book, we will take a closer look at this concept of progressive delivery, how it works under the hood, best practices to keep in mind, and how Bugsnag can help.

What Is Progressive Delivery?

Progressive delivery is the latest iteration of best practices for software development, building upon the core tenets of continuous integration and continuous delivery (CI/CD). The goal is to ship new features faster, reduce the blast radius of bugs, and improve the user experience by incrementally rolling out software to small groups of users rather than everyone at once.

It's also a shift in mindset: Rather than relying on automated testing to catch every error prior to deployment, progressive delivery acknowledges that mistakes will happen and attempts to catch them early before they reach the majority of users. As a result, there's less hesitation to deploy new code, and fewer users come across bugs in production.

Let's look at the origins of progressive delivery, its two core components, why you might want to use it, and some examples in the wild.

A Brief History

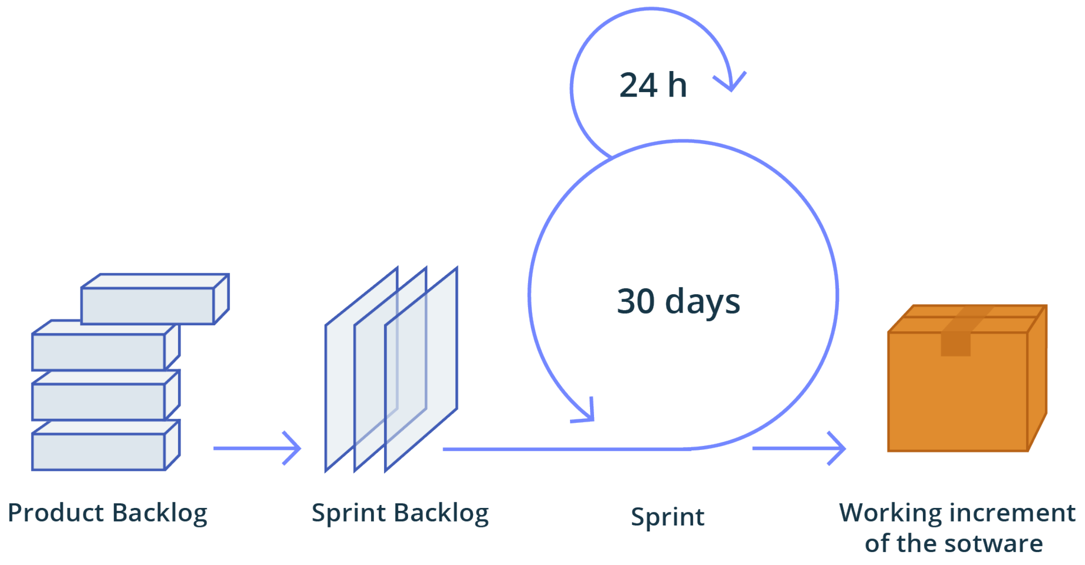
Traditionally, software development involved developing a complete product and then releasing it to the public. Not surprisingly, these "waterfall" development projects often exceeded their budget and failed to meet customer expectations. That's because stakeholders don't really know what customers want without delivering some form of the product..

The Agile Manifesto introduced the idea of “early and continuous delivery of valuable software” – or iterative development. Rather than waiting until the entire project was complete, Agile teams split a project into multiple sprints and release working software at regular intervals. That way, stakeholders have an opportunity to collect real-time customer feedback and make changes in the following iteration.

Progressive delivery builds on the foundation of continuous delivery with gradual feature rollouts, canarying, A/B testing, and observability. Instead of releasing features to all users, progressive delivery exposes features to a small subset. That way, there’s time to identify quality issues and assess user engagement before reaching everyone.

Core Tenets

Progressive delivery involves gradually rolling out new features and delegating responsibilities for each feature to specific team members. In addition to minimizing the number of errors reaching production users, these processes ensure that the right people



How an Agile development process typically works. Source: Wikimedia Commons / Lakeworks, CC BY-SA 4.0 are responsible for different features or releases at different times during the cycle.

The two core tenets are:

1. **Release Progressions** The number of users exposed to new features at a pace appropriate to your business. By adding more checkpoints for testing and feedback, you can improve the quality of each new feature. The frequency of releases will depend on the organization and its goals.

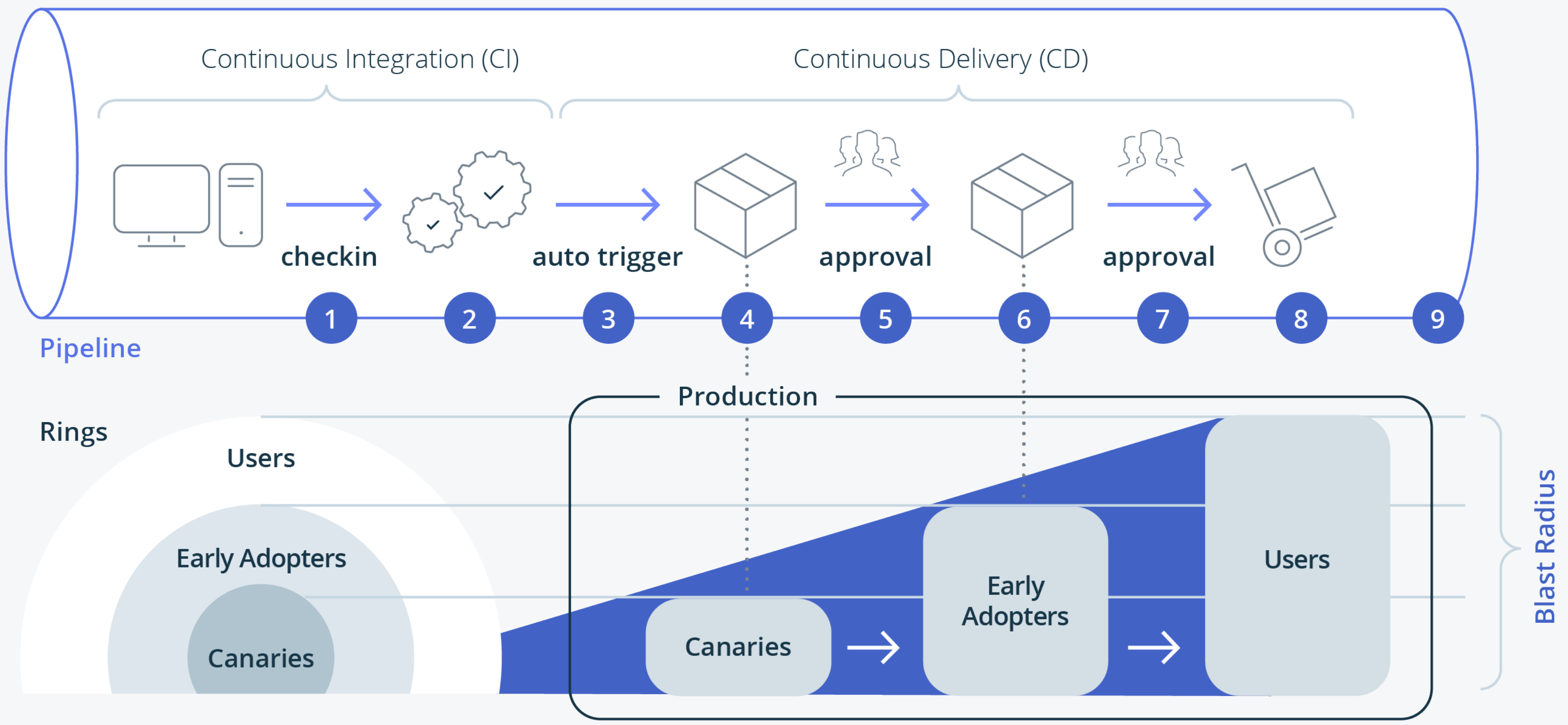
2. **Progressive Delegation** Delegating control of a feature to the owner most responsible for its outcome. While engineers initially control a feature, a product manager may be the most responsible owner after launch. At the same time, different teams may be responsible for different features.

As we discuss later, the release progressions and progressive delegation involve different strategies and tools. For instance, release progressions might rely on ring deployments and feature flags, while progressive delegation requires observability and team communication. But it's tying these concepts together that is the biggest challenge for most organizations.

Examples in the Wild

Most large tech companies use progressive delivery. For example, Zend reports that Netflix deploys 100 times per day, and Amazon deploys code every 11 seconds! Big companies aren't always the best resource for smaller organizations, but organizations like GitHub, Microsoft, and other tech companies provide a helpful example of what's possible, as we'll see below.

GitHub's "staff ships" approach involves unveiling new functionality to internal users via canary deployments. Since the company leverages its own source control tools in development, it's in a unique position to test its features and identify any problems before releasing them to external users. And as a result, they can be more confident with their deployments.



Microsoft's Ring Deployments involve several different cycles. Source: Microsoft

Microsoft's "ring deployments" involve dividing users into relevant groups, where each ring represents a different stage of a rollout with well-defined performance criteria. For example, the company's first ring might have relatively few beta users with higher fault tolerance, whereas later rings may have more enterprise users with much lower fault tolerance.

Pros and Cons

Progressive delivery has several obvious benefits but also some important risks. Before adopting progressive delivery, it's essential to understand these pros and cons. That way, you know what success metrics to watch from a business standpoint and, simultaneously, be mindful of some common pitfalls along the way.

Shifting testing into production has several benefits:

- **Faster Deployments:** Progressive delivery provides safeguards and control levers that give development teams the confidence they need to deploy code faster. Development teams don't have to spend undue time stressing about the stability of a release since they know it will only impact a handful of users (although stability is still important).

- **More Stability:** The safeguards that progressive delivery introduces also help preserve system stability by limiting the impact of any individual bug. Rather than clicking the deploy button and praying that nothing goes wrong, organizations can rest easy knowing that they will only impact a few users and can quickly roll back a release.

- **Better Collaboration:** Progressive delegation also helps increase operational efficiency and collaboration between engineers and non-engineers. With the right observability tools in place, anyone on the team can see how new features or releases impact stability while working together to create the best possible user experience.

The biggest drawback of progressive delivery relates to scale. Small or complex applications without many users or much traffic may require a longer time for users to explore a sufficient number of code paths to be confident in a release. As a result, teams using progressive delivery on smaller applications must be patient to reap the benefits.

Progressive delivery may also introduce complexity without the right strategies and tools. If you don't already have a high level of observability with a well-oiled deployment process, progressive delivery could end up causing more problems than it solves.

What's Next?

This next section will look at how progressive delivery works in practice, including phased rollouts, feature flagging, and experimentation. We'll also look at some key challenges that you might come across and some potential solutions to consider.

How Progressive Delivery Works

Progressive delivery may sound complex and difficult to implement. For example, you need to segment users into relevant groups and write extra code to display features to specific groups without impacting all users. Fortunately, a handful of well-established strategies and robust tools help simplify and streamline the process.

Let's look at common progressive delivery strategies, popular approaches to implementing them, and how to overcome challenges you might encounter.

Creating the Right Culture

Progressive delivery isn't a set-and-forget strategy where you can install tools and put your team on autopilot. As we mentioned earlier, a shift in mindset requires the buy-in from technical and non-technical team members. As a result, it's a good idea to ensure that your organization is ready before implementing strategies and tools.

There are two key steps to this process:

1. **Set Clear Goals** – Start by ensuring everyone understands the benefits of progressive delivery and how they're involved in the process. For example, product teams can get faster feedback from customers, quality assurance will have more time to test, and developers have a faster feedback loop.

2. **Create Communication Channels** – Many organizations operate with technical and non-technical silos, which can quickly become problematic when implementing progressive delivery. Whether you use in-person meetings or communication tools like Slack, it's imperative to keep everyone on the same page throughout the process.

Getting buy-in from the entire team makes an enormous difference in long-term success. Implementing new processes often fails because the organization wasn't ready for it. As a result, progressive delivery requires that everyone understand the tools and processes, including non-technical team members (e.g., business analysts).

Deployment Strategies

The first step is segmenting users into relevant groups. While there are many different approaches to segmentation, the most common methods involve segregating groups using certain attributes or randomly selecting a subset. Ultimately, the correct choice depends on the specific application and user base, as well as your organization's risk tolerance.

Targeted Rollouts

Targeted rollouts involve creating a select group of users with certain attributes and exposing them to new features or releases. If there aren't any issues with the small group, you may then decide to deploy the feature to every user. If there is a problem, you can quickly address the change and deploy a fix until the feature or release is stable enough for everyone.

Some examples of targeted rollouts include:

- An organization deploying new releases to a small group of beta users that are comfortable trying new features and willing to provide insightful feedback
- A food delivery app releasing new features in a small city before moving on to larger markets where scalability becomes a concern

Ring Deployments

Ring deployments are a type of targeted roll-out where user groups are assigned to rings. In addition, ring deployments may only release to a percentage of a target cohort initially and then increase it over time. As a result, ring deployments are one of the most common strategies across large tech companies.

Some examples of ring deployments include:

- A food delivery app may roll out changes to beta users in a single city for the first ring, a group of small cities in the second ring, and then larger markets in the third ring.

How to Deploy Features

The next step is developing the technical capabilities to implement these deployment strategies. Again, there are countless strategies that organizations might use to manage releases, but the most common approaches include feature flags and blue-green deployments. And, like deployment strategies, the right decision depends on the organization's goals.

Feature Flags

Feature flags are a software development tool enabling teams to turn that functionality on or off. In other words, they separate code deployment from a feature release. And they open the door to the A/B testing and experimentation necessary to make the most out of progressive delivery.

From a technical standpoint, feature flags range from a simple IF statement to a complex decision tree involving multiple variables. Static feature flags are hard-coded into a release, whereas dynamic feature flags can change at runtime using an admin interface or third-party tool. But adding feature flags to an application can quickly become complex.

Fortunately, several third-party platforms have popped up to simplify the process. For example, Split.io, an official partner of SmartBear, provides an easy-to-use SDK that makes it easy to add feature flags within the application and toggle them using a web-based interface. That way, business-facing teams can test features while developers focus on building features rather than feature flags.

How It Fits with CI/CD

Feature flags and blue-green deployments must integrate with existing continuous integration and deployment services. For example, a common approach involves developers checking in a new feature to a continuous integration process that triggers a canary release. After a manual review, the continuous delivery server deploys it to early adopters and then to every user.

From a technical standpoint, these processes involve a combination of continuous integration and deployment servers, such as Jenkins or CircleCI. If you're using blue-green deployment, these platforms may also need to integrate with load balancers, DNS tools, or other solutions to ensure that deployments reach the correct users.

Monitoring Progressive Rollouts

The final step in progressive delivery is monitoring deployments with a standard set of tools and metrics. After all, if you don't have a well-defined success metric, you'll never know when it's time to roll out a deployment to the next ring or when it's ready for everyone. In addition, you need to have a plan for handling rollbacks and mitigating any issues that arise.

Choosing Metrics

There are countless metrics that organizations might monitor to assess stability and performance, but they typically fall into stability, performance, or business categories. Like the deployment strategies and technical considerations, the best metrics will depend on the specific application and its users and requirements.

- **Stability** – Application stability metrics are essential to understanding when a release is ready for the next stage. For instance, you may want to look at error reports, crash logs, refresh rates, and other signals to evaluate stability. In the next section, we'll look at how Bugsnag can help automate stability analysis.
- **Performance** – Application performance looks at speed and resource consumption. For instance, you might look at CPU or memory usage to determine if a new feature or release is causing scalability issues. And, of course, speed is a critical consideration when creating the best user experience.
- **Business** – Business metrics are often an after-thought when evaluating new features or releases, but they're essential to minimize code bloat and ensure a great user experience. For instance, you might measure click-through rates, time on page, or other usage metrics to assess if the feature adds value.

Handling Rollbacks

Rollbacks should be relatively rare, but it's essential to have a plan in place for when they're needed. While rolling back a release isn't necessarily challenging from a technical standpoint, the management overhead required to diagnose the problem, assign a team member to fix it, and integrate any changes into the next release can be daunting. Even the most well-oiled machines struggle with it.

Fortunately, observability tools can help streamline the process. For instance, Bugsnag makes it easy to identify and prioritize the most impactful errors, assign them to the right people, and generate enough data to debug problems quickly. We'll take a closer look at how it works in the next section, where we dive into the platform's unique capabilities.

Common Challenges & Solutions

Progressive delivery is a complex process with several moving parts. As a result, development teams face many challenges when implementing these strategies. The good news is that progressive delivery has been around for a while, and most challenges have solutions or workarounds that you can use to stay on track.

Starting Too Early

Progressive delivery is a complex process that's not right for every team. If you don't have robust continuous integration and deployment (CI/CD) processes, trying to implement progressive delivery could cause more headaches than it solves. As a result, it's imperative to have a solid deployment setup before diving into a new paradigm.

You also need the right observability solutions in place to detect problems. For instance, a blue-green deployment may face "race conditions" where data is read or written by multiple versions of the application simultaneously. Without observability solutions capable of drilling down into the production stack, it's impossible to trace these bugs.

Releasing Too Quickly

Progressive delivery involves deploying new features or releases to small groups of users, meaning it can take a long time to become confident in a release. If you have a complex application with little traffic, it could take hours or even days to collect enough data to garner the confidence to deploy a new release to all users.

Grouping several changes into a single release helps mitigate these issues. While it may be harder to debug and roll back larger releases, they may be necessary to maintain the right release cadence for your development team. Otherwise, you could end up with a huge backlog of features while waiting for enough data from earlier ones.

Failing to Write Tests

Progressive delivery shouldn't be a substitute for automated tests. While integration tests can be boring or challenging to write, skipping them and trying out untested features on small groups of users in production is never a good idea. Progressive delivery should be a safety net to roll back unexpected problems rather than a first-line testing strategy.

When a rollback occurs, it's a good idea to, as a team, conduct a post-mortem meeting discussing what happened and how to prevent it in the future. Ideally, progressive delivery should involve progressively fewer rollbacks. In the words of Dan Lorenc, you should treat it as a parachute rather than a hammock.

What's Next?

In the next section, we'll look at how Bugsnag can help enable progressive delivery – and progressive delegation – with its unique observability capabilities.

How Bugsnag Enables Progressive Delivery

Feature flags and blue-green deployment simplify release progressions, but it's challenging to integrate error detection with progressive delegation. For example, you don't want to roll back a release unless necessary, and assigning bugs can quickly become a considerable challenge in multi-team apps. Fortunately, observability can help address these issues and streamline the process.

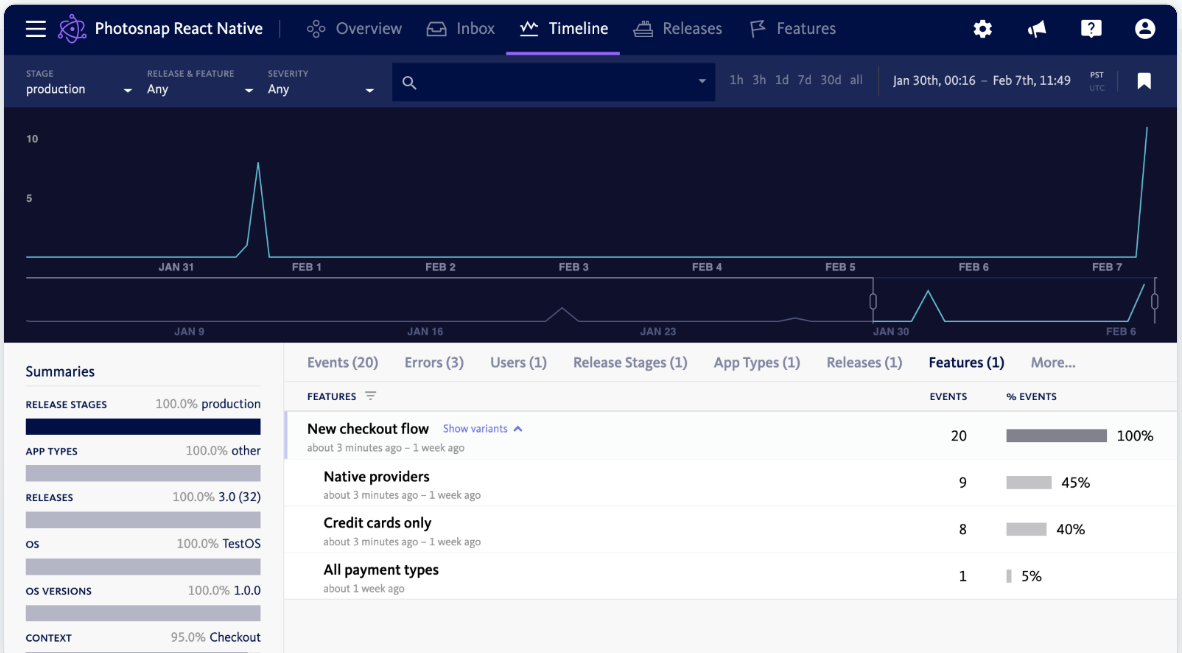
Bugsnag makes it easy to identify and prioritize the most impactful errors, assign them to the right teams, and empower developers with the debugging tools they need to implement quick fixes and keep progressive delivery on track. Essentially, it provides the data-driven layer of progressive delivery, delivering actionable insights to your team.

Let's look at some of the most significant features that can help you implement an effective progressive delivery strategy across your organization.

Real-time Observability

Bugsnag makes it easy to understand how feature flags and experiments impact the user experience. For example, you can configure alerts that notify teams responsible for developing and rolling out a particular feature when an error occurs with them.

Bugsnag makes it easy to drill down into what features are causing errors. Source: Bugsnag



Summaries

RELEASE STAGES	100.0% production
APP TYPES	100.0% other
RELEASES	100.0% 3.0 (32)
OS	100.0% TestOS
OS VERSIONS	100.0% 1.0.0
CONTEXT	95.0% Checkout

FEATURES	EVENTS	% EVENTS
New checkout flow Show variants — about 3 minutes ago – 1 week ago	20	100%
Native providers — about 3 minutes ago – 1 week ago	9	45%
Credit cards only — about 3 minutes ago – 1 week ago	8	40%
All payment types — about 1 week ago	1	5%

You can also zoom into the errors occurring within individual feature flags to diagnose problems.

In addition, the platform makes it easy to determine if unusual application-wide error activity is associated with a feature flag. Using a timeline view, you can set up pivot tables for feature flags to isolate errors that occurred when the feature flag was enabled. That way, you can assign problems to relevant teams to fix quickly.

Go/No-Go Stability Scores

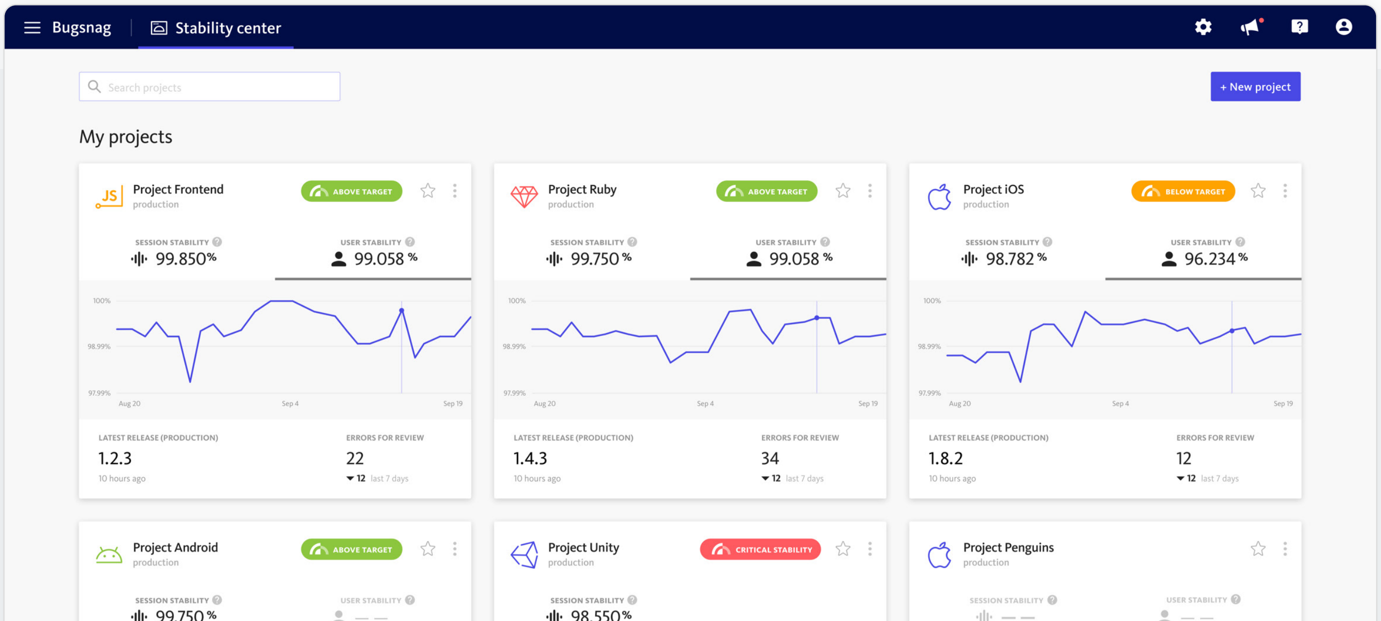
Progressive delivery is all about balancing bugs with features. Rather than pursuing perfect software before release, development teams try to release imperfect software and fix bugs before they impact several users. But, to do that, you need an easy way to determine if the number and severity of errors cross an acceptable threshold.

Bugsnag's stability scores provide a definitive metric for deciding when a release is ready for the next ring or group of users. When a stability score turns green, developers can focus on building the next feature rather than troubleshooting existing features. And, when a stability score turns red, the platform shows you the top errors occurring in a specific release.

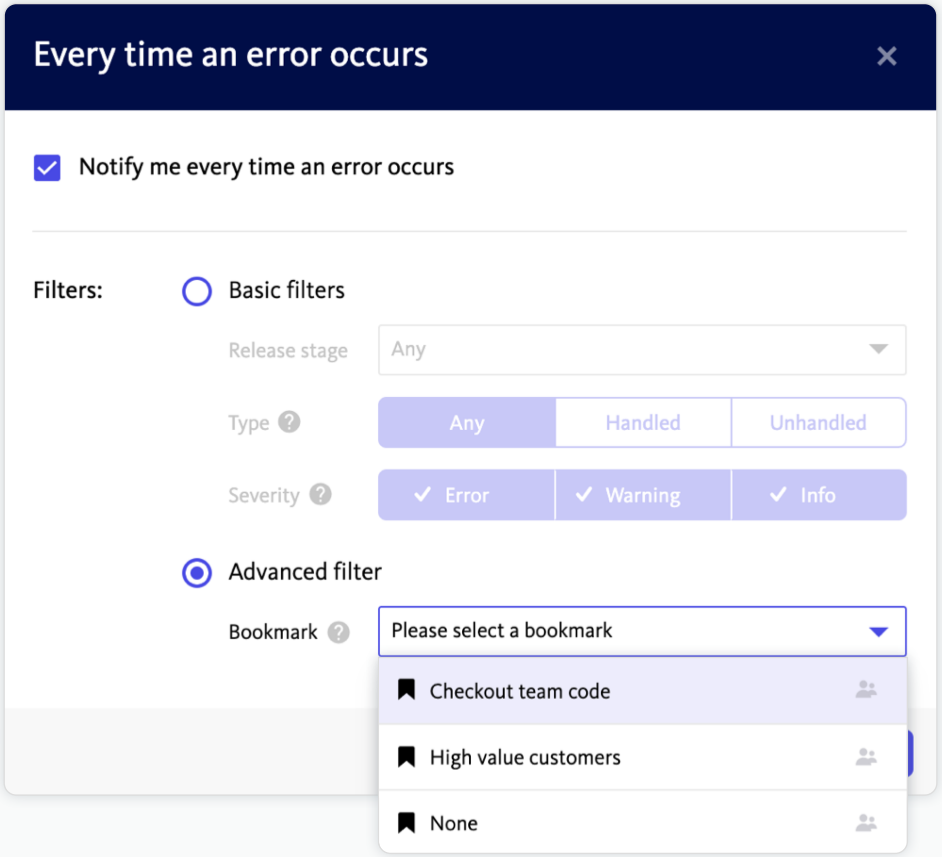
Search and Segment

Prioritizing errors is only half the equation. After identifying errors, you need to alert the relevant teams or developers to implement a fix. This process can be a challenge in multi-team apps where different teams have different responsibilities – especially if you want to avoid "notification fatigue" by notifying every developer of every problem.

Bugsnag makes it easy to segment errors using bookmarks. For instance, you might segment front-end and back-end errors and configure alerts that only show relevant errors to relevant teams. That way, developers can quickly see the errors that apply to them and prioritize them using stability scores to be as efficient as possible.



Stability scores make it easy to determine what's good to go and what needs work. Source: Bugsnag


Every time an error occurs

[✓] Notify me every time an error occurs

Filters: ○ Basic filters

Release stage: Any

Type: [✓ Any] Handled Unhandled

Severity: [✓ Error] [✓ Warning] [✓ Info]

● Advanced filter

Bookmark: Please select a bookmark

- Checkout team code
- High value customers
- None

Easily create alerts for specific features or segments to avoid notification fatigue. Source: Bugsnag

Getting Started

Bugsnag's Releases Dashboard and new Features Dashboard make it easy for developers to understand if they need to roll back a release version or feature to preserve stability. In addition, they can quickly assess if a feature or release is responsible for a specific exception. A search builder lets you filter your inbox for errors exclusive to a particular release or feature flag.

Currently, the Features and Releases dashboard is available for client-side applications, including Android, iOS, React Native, Flutter, Unity, Unreal Engine, Expo, JavaScript (browser-based and Node.JS-based), and Electron. Bugsnag also works constantly to add new platforms. Contact us to discuss your platform and if we can support it.

If you're a current Bugsnag user, you can start monitoring features and experiments in the dashboard by configuring the Bugsnag library in your application. You can learn more about how to set up the Features and Releases dashboard by reading our documentation with specific guides for each popular platform.

Best Practices and Bottom Line

Progressive delivery represents an evolution from continuous delivery. Rather than trying to catch every error with automated tests, progressive delivery incrementally rolls out new features to a small subset of users to test in production. As a result, development teams can move faster while improving the overall user experience.

Some best practices to keep in mind include:

- **Start with Culture** – Most progressive delivery implementations fail because the organization wasn't ready for change. Before installing any tools, start by getting buy-in from team members and understanding what approaches and tools might work best with existing workflows to minimize disruption.
- **Nail the Prerequisites** – Progressive delivery requires a solid continuous integration and deployment workflow. Rather than trying to implement feature flags from the get-go, many organizations may need to step back and fix things like source code management before even discussing progressive delivery concepts.
- **Focus on the Outcomes** – Many organizations don't fully understand how to use progressive delivery. While shipping faster is a common goal, the most powerful benefits include long-term experimentation, A/B testing, and personalization. Implementing these will often require an outcome-focused mindset.

While the process sounds complex, Bugsnag and other new technologies have made it easier than ever. Development teams can quickly deploy code with feature toggles that product managers can use to roll out changes progressively. At the same time, Bugsnag and other tools enhance observability to identify problems and route them to the right teams quickly.

bugsnag

A SMARTBEAR COMPANY

Clarity on what to do next.

Monitor your application stability non-stop, and get the visibility you need to confidently make your next move without digging for information.

Start 14-Day Free Trial

Request a Demo

Resources Articles

https://www.bugsnag.com/blog/appstabilityseriesgainprecisionandreducenoise

https://www.bugsnag.com/blog/acc-progressive-delivery

https://www.bugsnag.com/blog/progressive-delivery-accelerate-app-releases-while-minimizing-bugs

https://www.bugsnag.com/blog/features-experiments-dashboard

https://launchdarkly.com/blog/what-is-progressive-delivery-all-about/

https://dlorenc.medium.com/pitfalls-of-progressive-delivery-114c6e3f9dbb

https://d1.awsstatic.com/whitepapers/AWS_Blue_Green_Deployments.pdf

SMARTBEAR