OpsMx

Advanced Deployment Strategies

An overview of the most common software deployment strategies

Table of Contents

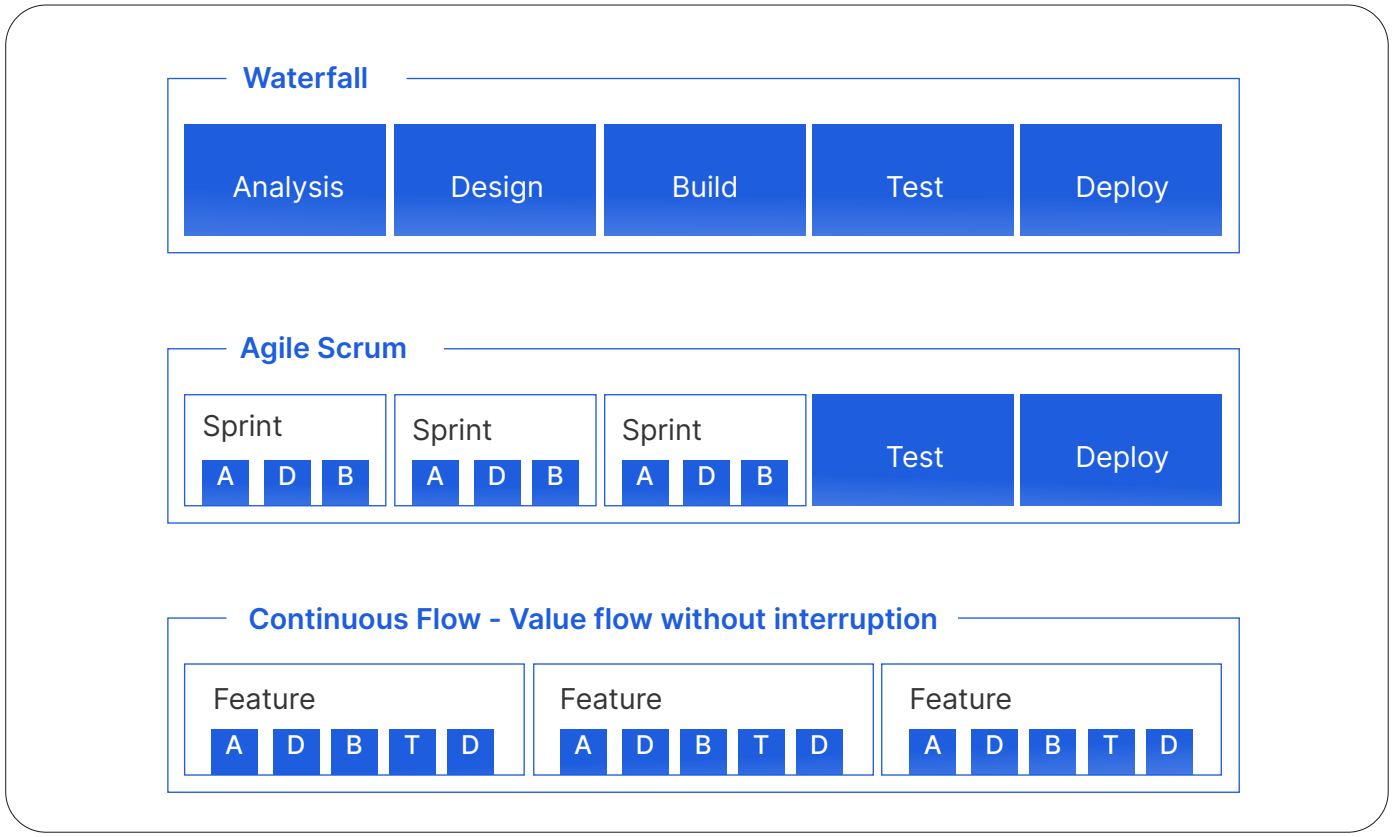
Introduction

Low-Risk Releases are Incremental

Today's high-performance DevOps teams accelerate software delivery and reduce cycle times in a safer, low-risk environment.

Continuous Delivery has enabled companies like Google, Netflix, and Amazon to bring on new revenue streams faster, achieving the agility needed to respond immediately to marketplace opportunities, events, and trends.

Continuous Delivery is a continuous-flow approach associated with just-in-time and Kanban. The goal is an optimally balanced deployment pipeline with little waste, the lowest possible cost, on-time, and defect-free deployment to production.



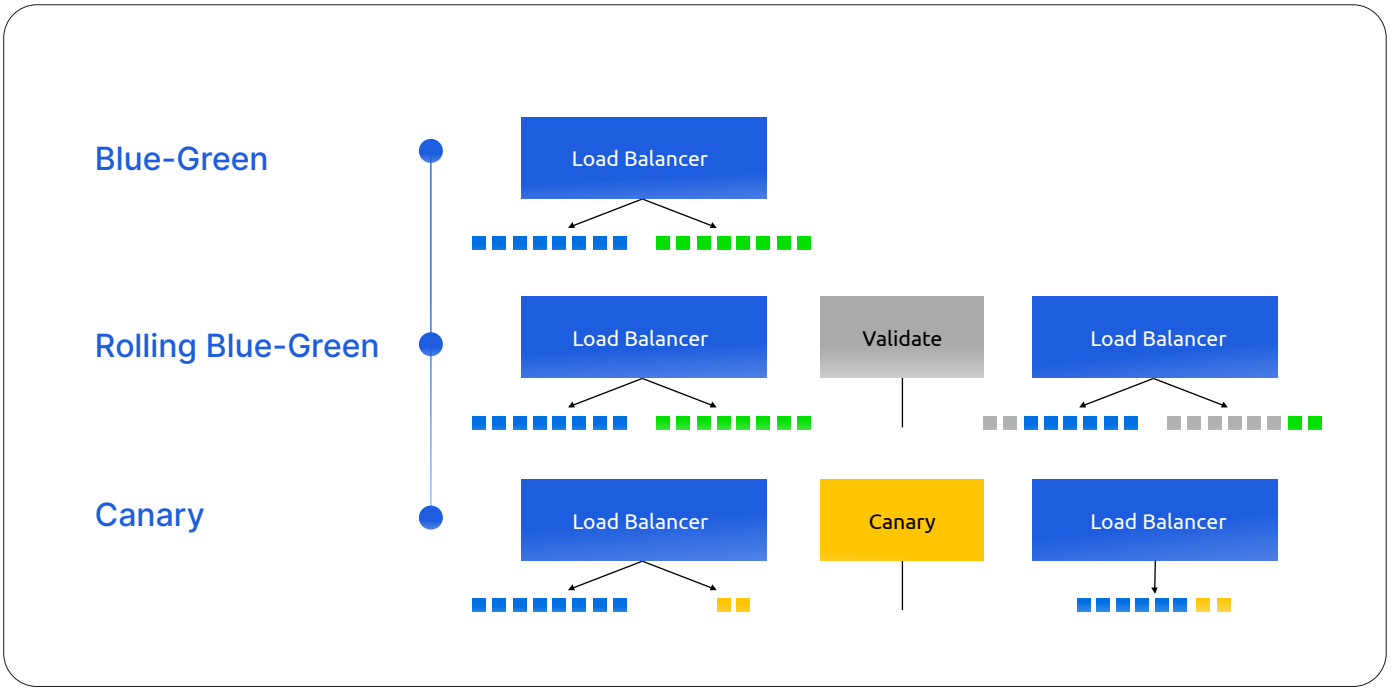
Traditional, big-bang releases are highly volatile:

- Involve multiple dependencies
- Large number of changes
- Harder to rollback
- Higher risk

Contemporary, incremental releases are safer:

- Faster delivery, faster feedback
- Easier rollouts and rollbacks
- Less dependencies
- Lower risk

While Continuous Delivery drastically reduces the time between releases, DevOps teams must implement advanced deployment strategies to ensure that software deployments can be fast, repeatable, safe, and secure.



This ebook provides an overview of the latest deployment strategies and how best to implement them in your software delivery practice.

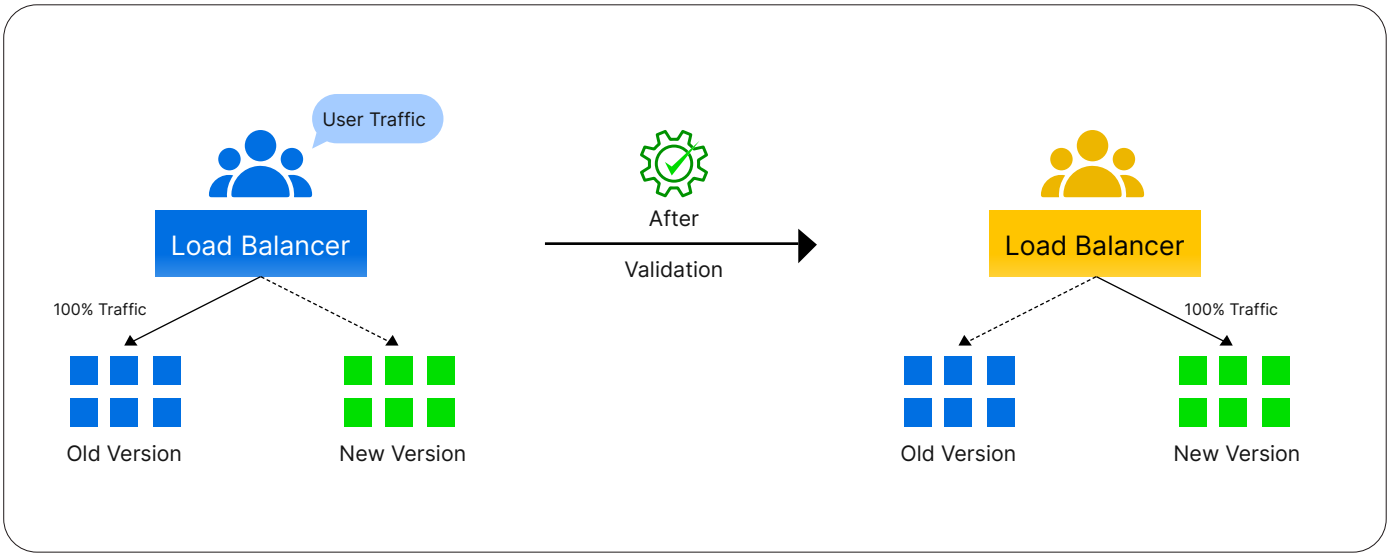
Blue-Green Deployment

Blue-Green

Minimizes downtime during the “cut-over”

One of the challenges with automating deployments is the cut-over, taking software from the final stage of testing to live production. You usually need to do this quickly to minimize downtime.

The blue-green deployment strategy requires you have two production environments as identical as possible. One of them, let’s say blue, for example, is live.



You do your final testing stage in the green environment as you prepare a new software release. When your testing completes, you switch the router so that all incoming requests go to the green environment - the blue environment is now idle.

How to Implement

Four Phases of a Blue-Green Deployment

Load balancers and routers help switch users from the blue instance to the green one. Control is crucial because it may be necessary to quickly switch them back to the blue instance in case of a green instance failure. The implementation assumes that the load balancer currently serves traffic to the blue instance (v1.0).

1 Create and bring the green instance online

The first phase is to create the green instance and bring it online to run in parallel with the blue instance. Initial testing and validation are conducted without live traffic to ensure you have an environment as identical as possible to the blue instance.

2 Execute the traffic switch

Once the new green instance (v1.1) is ready, the traffic is switched from the old blue instance (v1.0). Most users won't even notice that they are now accessing a newer version of the service or application.

3 Monitoring the environments

DevOps engineers can now run smoke tests on the green instance as they need to assess if any issues will impact the users of the new version.

4 Rollback or Continue

During the smoke tests, if any bugs are detected or a performance issue, the users can quickly be rolled back to the stable blue version without any substantial interruptions.

After an initial smoke test, monitoring continues as errors might appear after the new (green) version goes live. The blue version is always on standby, and after an appropriate monitoring period, the green instance becomes the blue instance for the next release. The original blue instance is entirely removed or scaled down to save costs.

Benefits

Seamless customer experience
Users don’t experience any downtime during the cutover.consequat.

Instant rollbacks
You can undo the change without adverse effects.

No upgrade-time schedules for developers
No need to wait for maintenance windows. No weekend work.

Testing parity
The newer versions can be accurately tested in real-world scenarios.

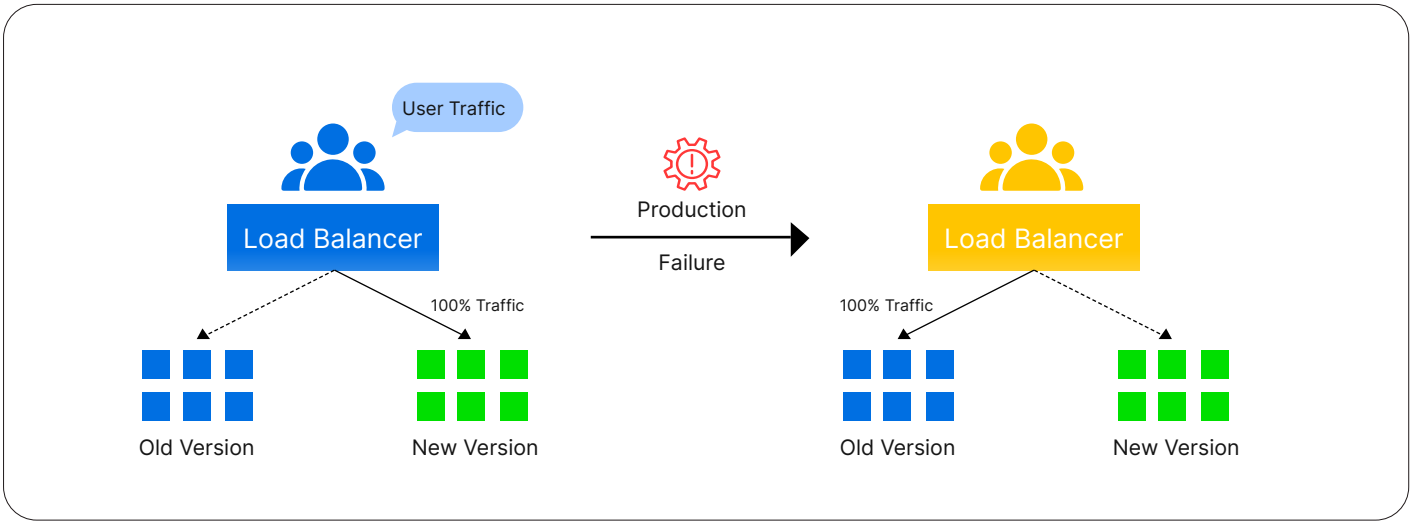
The inherent equivalence of the Blue and Green instances and a quick recovery mechanism is also perfect for simulating and running disaster recovery practices.

Gone are the days when you had to wait for low-traffic windows to deploy the updates. This eliminates the need to maintain downtime schedules. Developers can quickly move their updates into production through the Blue-Green strategy as soon as they are ready with their code.

Challenges

Managing the cutover

Some sessions may fail during the initial switch to the new environment, or users may be forced to log back into the application. This issue can be overcome by using a load balancer instead of DNS to manage new traffic from one instance to another.



High infrastructure costs

Organizations that have adopted a Blue-Green strategy need to maintain an infrastructure that doubles the size required by their application. If you utilize elastic infrastructure, the cost can be absorbed more easily. Blue-Green deployments may be suitable for less hardware-intensive applications.

Code compatibility

Different code versions need to co-exist to support the seamless switching between blue and green instances. For example, if a software update requires changes to a database, the Blue-Green strategy is challenging to implement because traffic may switch back and forth between the blue and green instance. Therefore, you should use a database compatible with all software updates.

Common Practices

Choose load balancing over DNS switching

Do not use DNS to switch between servers. It can take browsers a long time to get the new IP address. Some of your users may still be served by the old environment.

Instead, use load balancing. Load balancers enable you to set your new servers immediately without depending on the DNS mechanism. This way, you can ensure that all traffic comes to the new production environment.

Keeping databases in sync

One of the biggest challenges of blue-green deployments is keeping databases in sync. Depending on your design, you may be able to feed transactions to both instances to keep the blue instance as a backup when the green is live. Or you may be able to put the application in read-only mode before cut-over, run it for a while in read-only mode, and then switch it to read-write mode. That may be enough to flush out many outstanding issues.

Execute a rolling update

A rolling update slowly replaces the old version with the new version. As the new version comes up, the old version is scaled down to maintain the overall count of the application.

Monitor your environments

It is essential to monitor both the production and non-production environments. Make sure your organization sets up an easy way to toggle the alerting between the two environments.

The Blue-Green deployment strategy is one of the most widely used deployment strategies. It is a great fit when environments are consistent between releases and user sessions are reliable even across new releases.

Canary Release

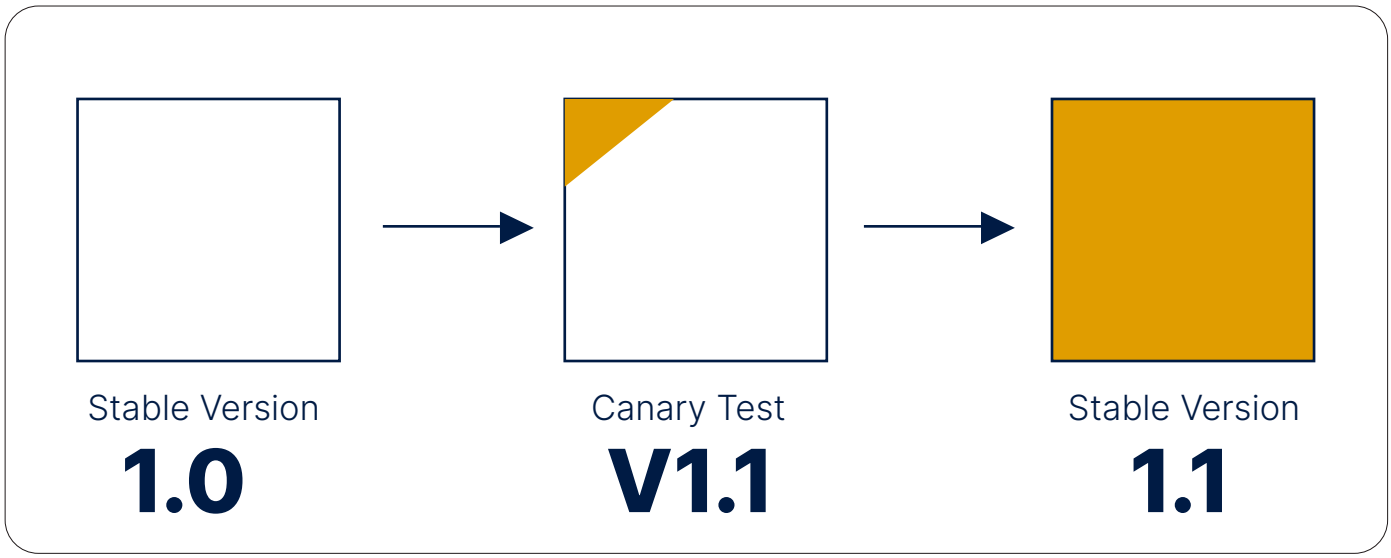
Early warning system to potential issues

Canary Release

Early warning system to potential issues

Canary deployments refer to the practice of releasing a code change to a subset of users and then looking at how that code performs relative to the old code that the majority of users are still running. Essentially becoming a “canary in a coal mine” serves as an early warning system for potential issues.

This is achieved by setting up Canary servers that run the new code. As new users arrive, a subset of them is routed via the load balancer to those canary servers.

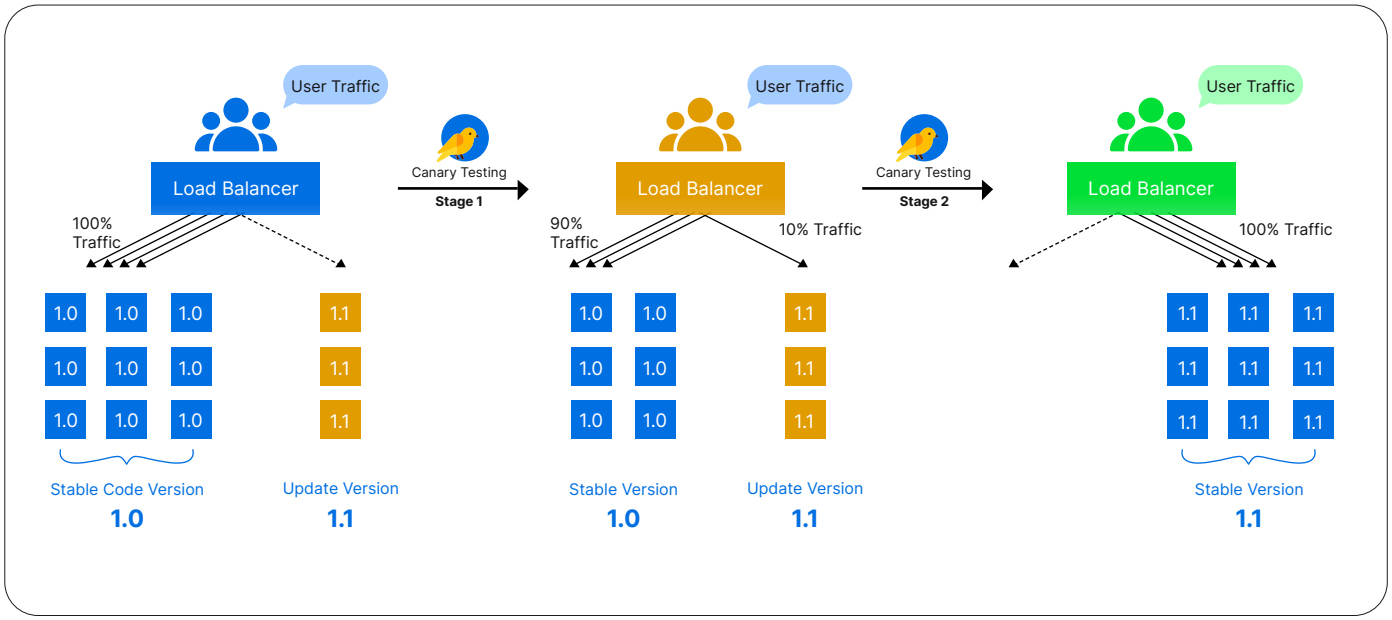


You can then use standard performance and monitoring tools to detect whether the new code is working correctly. For instance, you might monitor the compute load of the Canary servers relative to those servers running the old code. If the load increases substantially, you know that’s a potential issue.

Equally, if you see a much higher rate of I/O, that might also indicate a problem. Canary Testing is ideal when you wish to test the performance of your backend.

How to Implement

In a simple structure, the canary deployment has three stages.



Plan and Create

1

The first step involves creating a new canary infrastructure where the latest update is deployed. Rolling out canary instances can take two forms:

1) You are controlling traffic using load balancer rules to the new version. A small amount of traffic is sent to the canary instance, while most users continue to use the baseline instance.

2) Adding new version instances to the existing version. For example, if ten server instances are running version 1, you would bring up version 2 instance and add it to the server pool sending ~10% of the traffic to the new canary instance.

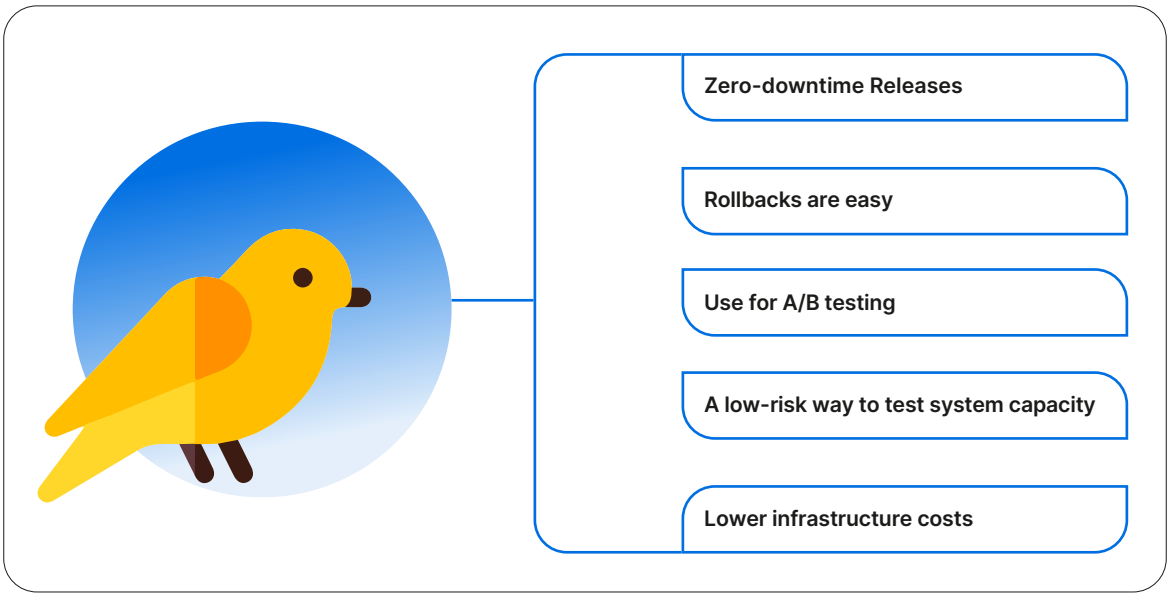
2 Analyze

Once traffic is diverted to the canary instance, the team collects data: metrics, logs, information from network traffic monitors, results from synthetic transaction monitors – anything that helps determine whether the new canary instance is operating as it should. The team then analyzes this data, comparing it to the baseline version.

3 Roll

After the canary analysis is completed, the team decides whether to move ahead with the release and roll it out for the rest of the users or roll back to the previous baseline state.

Benefits



Zero-downtime releases
Switch users from one release to another instantaneously.

Rollbacks are easy
Just stop routing users to the bad version, and you can debug and break/fix at your leisure.

Use for A/B testing
By routing some users to the new version and some to the old version, companies can measure and compare different feature versions and kill the new features if not enough people use them. A/B testing can also measure actual revenue generated or can be rolled back if the more recent version generates lower revenue. Just route a small representative sample of users to the new version.

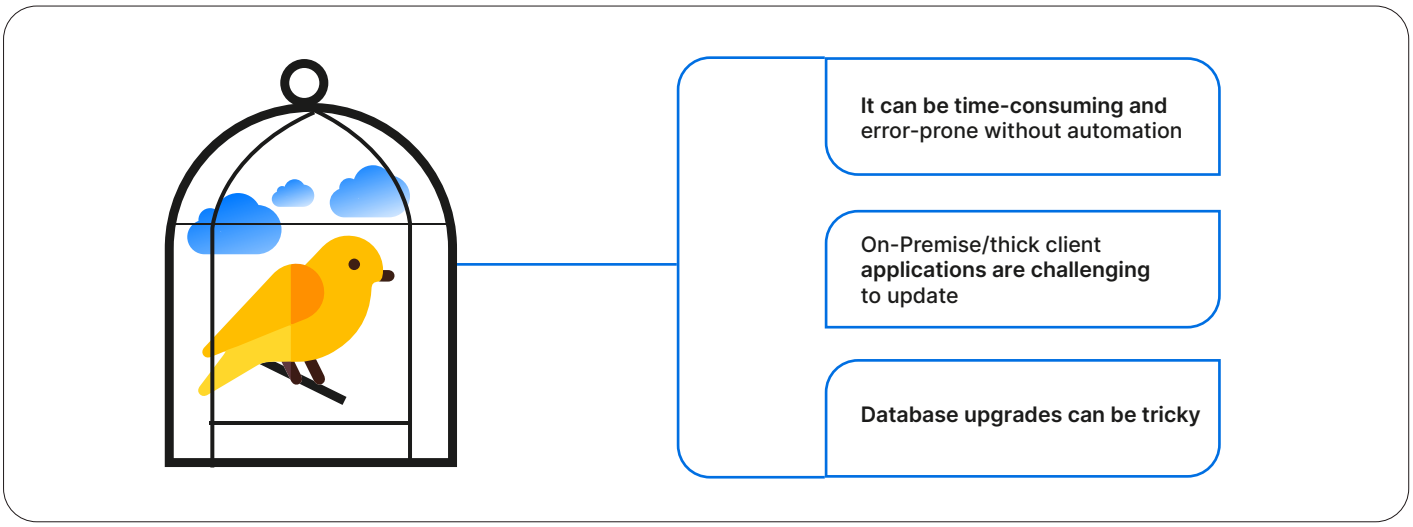
A low-risk way to test system capacity
Gradually ramp up the load by slowly routing more and more users to the application while measuring application response time metrics like CPU usage, I/O, and memory usage and watching for exceptions in logs. This is a relatively low-risk way to test system capacity in your production environment.

Lower infrastructure costs
Unlike the Blue-green strategy, where you need nearly an identical copy of the infrastructure, canary releases require a small infrastructure initially to deploy and test that your code changes are running fine.

Challenges

It can be time-consuming and error-prone without automation

Today, many companies execute the analysis phase of canary deployments in a siloed and non-integrated fashion. You need an automated and integrated toolchain.



If a DevOps engineer is assigned to manually collect monitoring data and logs from the canary version and analyze them. It will not be scalable for rapid deployments. Decisions on whether to roll back or roll forward will be delayed and may be based on incorrect data.

On-Premise/thick client applications are challenging to update

It becomes challenging for a business to perform a Canary deployment in an environment where the application is installed on personal devices. One of the ways around this can be setting up an auto-update environment for the end-users.

Database upgrades can be tricky

Databases (or any shared resource) need to work with all versions of the application you want to have in production. You might need to create a very complex deployment process if you try to modify the application to interact with the database or change the database schema.

To perform the canary, first, change the database's schema to support two or more instances of the application. This will allow the old and new versions of the application to run simultaneously. Once the new architecture of the database is in place, the latest version can be deployed and switched over.

Common Practices

Compare canary against a baseline, not against production

You might be tempted to compare the canary deployment against your current production deployment. Instead, always compare the canary against an equivalent baseline deployed.

The baseline uses the same version and configuration that is currently running in production but is otherwise identical to the canary:

- Same time of deployment
- Same size of deployment
- Same type and amount of traffic

In this way, you control the version and configuration only, and you reduce factors that could affect the analysis, like the cache warmup time, the heap size, and so on.

Ensure to give your Canaries enough time

Slower rollouts mean better data but reduced velocity. It's a balancing act, but canaries deployed for critical services should live longer. Longer canary durations will help detect issues. For highly critical services, recommendations are for 4 to 24 hours.

Time canary deploys with your traffic cycles

If you see regular workload peaks and troughs, time your canary period to begin before peak traffic and cover a portion of the peak traffic period.

Your canary process should cover 5% to 10% of your service's workload

For example, if a service tier typically comprises 100 instances, you should canary on 5 to 10 of those instances. A canary covering only 1% to 2% of the workload is more likely to miss or minimize some important cases; a canary covering more than 10% of the workload may have too much impact if it doesn't work as expected.

Conclusion

The canary deployment strategy is widely used because it lowers the risk of moving changes into production while reducing the need for additional infrastructure. Organizations using canary can test the new release in a live production environment while not simultaneously exposing all users to the latest release.

Dark Launches & Feature Toggles

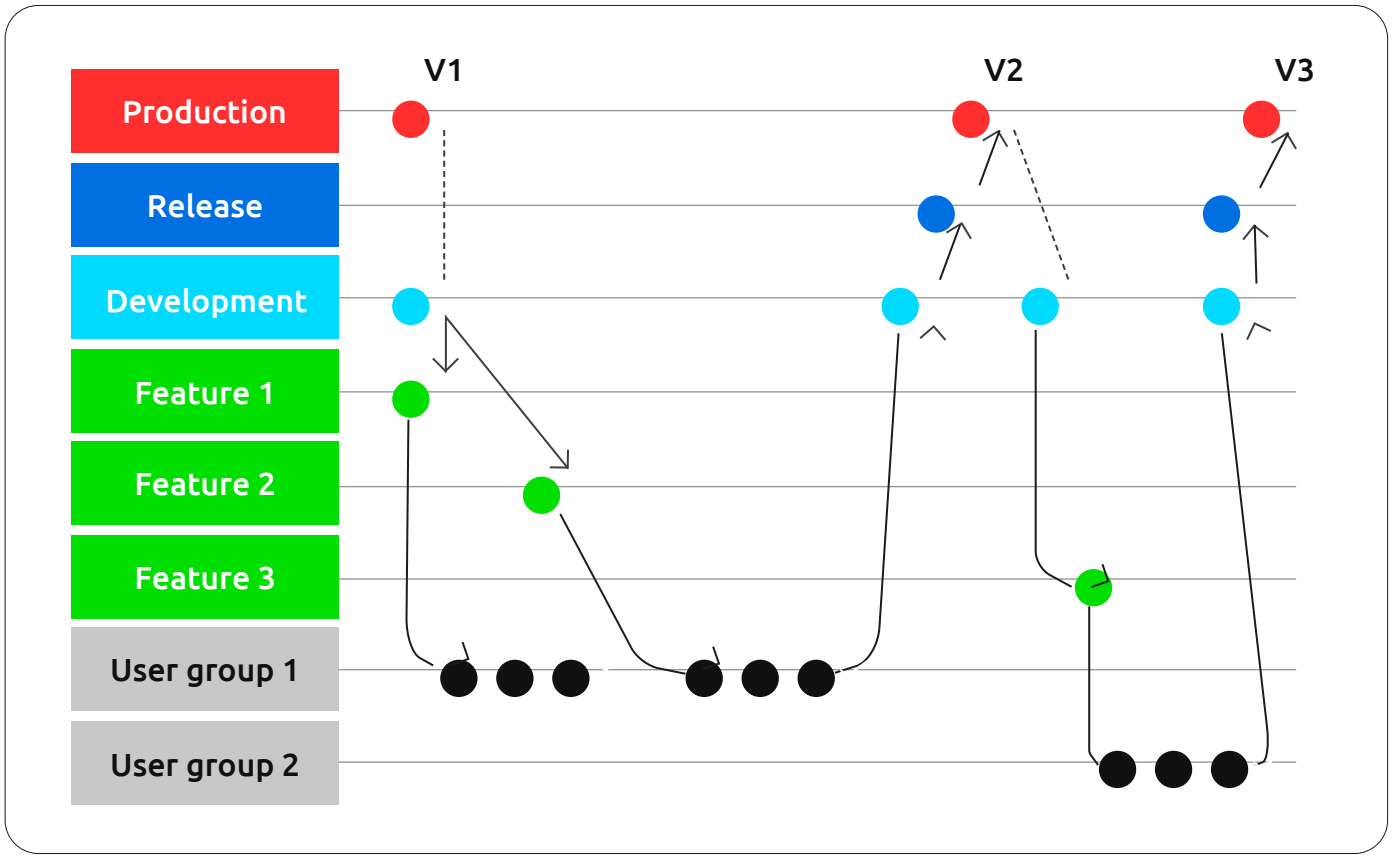
A safe way to gauge interest in a new feature

Dark Launches and Feature Toggles

A safe way to gauge interest in a new feature

Dark Launching is similar to Canary deployments. However, the difference here is that you are looking to assess users' responses to new features in your frontend rather than testing the performance of the backend.

The concept is that rather than launch a new feature for all users, you instead release it to a small set of users.



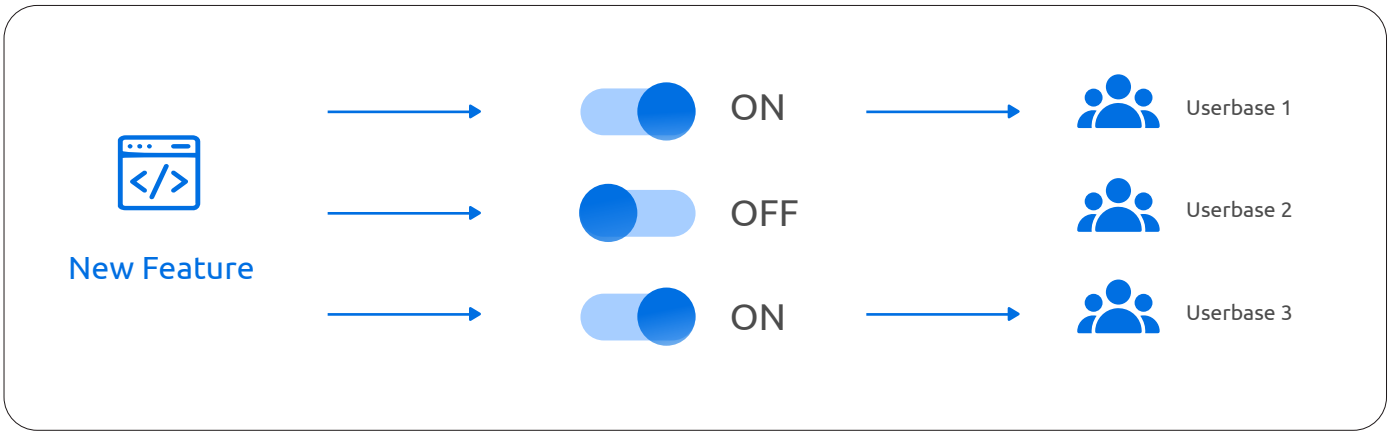
Usually, these users aren't aware they are being used as guinea pigs for the new feature, and often you don't even highlight the new feature to them, hence the term "Dark" launching.

How to Implement

You can use UX instrumentation to monitor if the feature improves the user experience or increases your revenue (e.g., the new feature may encourage them to spend longer using your app and thus consume more ads or make more in-app purchases).

This process is precisely what any product manager is doing when assessing how well an app performs. The only difference is that you are now looking at the performance of a single new feature.

Use feature toggles to incrementally roll out a new feature to more and more users and assess performance.



A feature toggle is a technique that attempts to provide an alternative to maintaining multiple source code branches. Feature toggles hide, enable or disable the feature during run time. A feature can be tested even before it is completed and ready for release.

Dark Launching enables product teams to roll back features that are not performing well or fully launch features that users love.

Benefits

Empower DevOps team to safely experiment

Dark launching empowers DevOps teams to safely experiment with new features and new software versions at a lower risk. This creates faster feedback cycles so that product teams can adjust features and software versions to the needs and wants of the users and the markets they serve faster than their competitors.

Gauge interest in new features

With dark launches and feature toggles, product teams can gauge interest and adoption of new features. They can make rollout decisions based on how the current subset of users accept the new feature and determine how it will help with the success of the product and company.

Allows you to deploy a new feature in a controlled way

Like canary releases, dark launching and feature toggles will enable you to roll out new features in a controlled manner. However, dark launches and feature toggles do not require you to run multiple versions of an application in an environment simultaneously, which you must do for a canary release.

Reduces pre-production testing

Organizations can save time and money since features can safely be tested in production and by actual users instead of QA engineers. Feedback would be directly from the users in a real-world production environment.

Challenges

It might be challenging to adopt

Dark launches and feature toggles require you to change the code in the application you want to deploy. Development teams will need to design, code, build, maintain, and deploy this support. Implementing this support for legacy applications with large codebases could be problematic.

Feature Toggles could lead to more technical debt

Feature toggles require code updates to implement. Typically these toggles are temporary software updates to support the new feature. Once the feature has been tested and accepted, the feature toggle is no longer required. If you don't have a process to maintain, update, and remove old temporary feature toggles, technical debt will increase.

May make the system harder to understand and less secure

As more feature toggles are added, the code can become more fragile and brittle, harder to understand and maintain, and less secure. Feature toggling is about your software being able to choose between two or more execution paths based on a toggle configuration. This increases the complexity of the code and makes it harder to test, support, and secure.

Common Practices

Encapsulate feature toggles with the business logic it supports

As a general rule of thumb, you should try to encapsulate your feature toggle with its supported business logic. This will help avoid having other areas of your codebase be aware of the context needed for toggling the feature. Sometimes, this is impossible as the core business logic might be broken up into several different services. If that is the case, the toggle should be placed closest to the service call, passing a parameter to the target services.

Establish a baseline of service levels to monitor and test for success

Ensure that you are setting service-level objects and determining service level indicators to track the service or feature impacted performance. It's critical to have the tools and infrastructure to assess the system's performance, monitor for unexpected responses to client requests, and compare any system deviation to a baseline.

Create a retirement plan and process for feature toggles

Development and delivery teams are asked to add toggles for various reasons but aren't often asked to remove a toggle after it has served its purpose. Teams need to put processes in place to ensure that toggles are eventually retired. Whether adding a toggle retirement task to the team's work backlog or creating an expiration notification event when a toggle's expiration date has passed, you should have an automated system that manages the lifecycle of a feature toggle.

Progressive Delivery

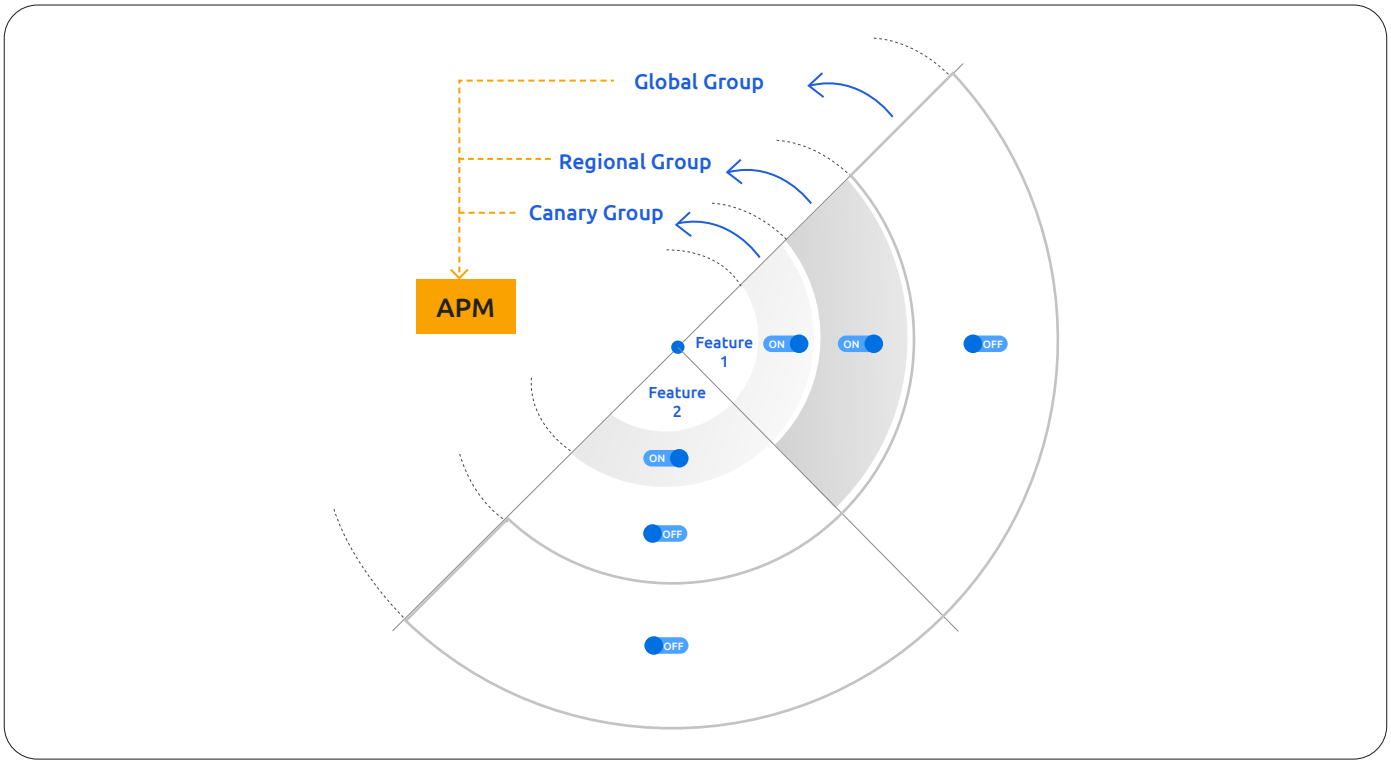
Control which users see which feature, and when

Progressive Delivery

Control which users see which feature, and when

Progressive Delivery is not a deployment strategy, but we have chosen to include an overview in this ebook because it is a new DevOps practice that leverages the deployment and rollout strategies previously discussed.

Progressive Delivery extends Continuous Delivery by enabling more control over feature delivery. The process deploys features to a subset of users, then evaluates key metrics before rolling out to more users or rolling back if there are issues. Progressive delivery introduces two core tenets, release progressions and progressive delegation.



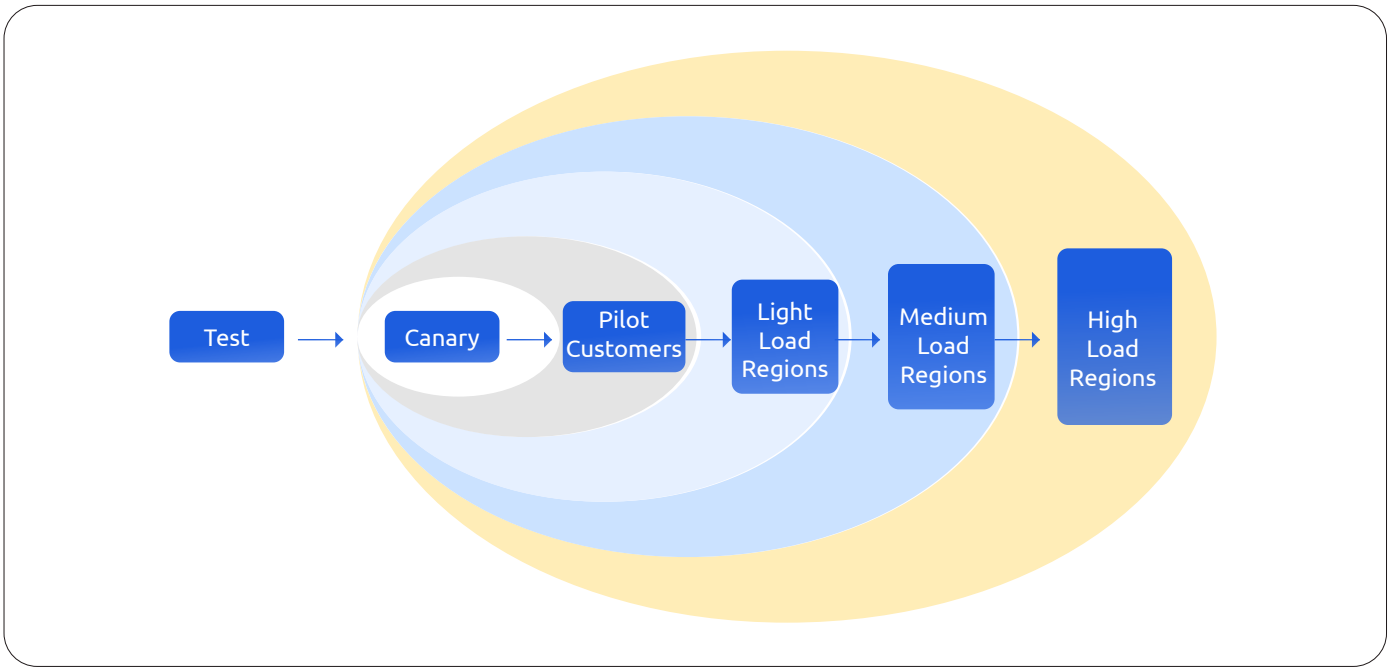
Release Progressions uses a variety of deployment strategies to deploy features to a subset of users at a pace sustainable for the business. This facilitates the creation of checkpoints for testing, experimenting, and gathering user feedback.

Progressive delegation refers to progressively delegating the control of a feature to the owner who is most closely responsible for the outcome. As the feature transitions from development, test, and production, the ownership changes from engineering to a role closer to the end-user, usually the product manager.

How It Works

Incremental delivery and feature management are the key enablers of Progressive Delivery. Both of these together provide fine-grain user exposure control to a new feature. This exposure is referred to as the “blast radius.” By limiting the blast radius, you restrict the set of users exposed to a possible bad outcome.

The decision to proceed or fall back is based on testing criteria and careful monitoring. You might use canary analysis, A/B testing, observability, or other methods to meet the service objectives and success criteria.



For example, using a release progression with canary, you can select a small blast radius, 1-5% of the entire user population, and then gradually increase to 10%, 20%..etc., based on the feature performance and user feedback in each stage. If there are critical issues, you can either turn that feature off or roll back to the baseline version.

The progressive delegation lifecycle would start in pre-production environments. Developers and test engineers would own the feature and associated toggle support, ensuring that it performs to its expected specifications. Once the feature goes live, ownership transitions to the product manager to review and determine if the feature is delivering the expected outcomes.

Once the feature is deployed to all users, the management of the feature returns to development so they can remove the feature toggle and ensure the new feature is part of the new baseline version.

Benefits

Release features faster and with control

Progressive delivery helps DevOps teams deploy features faster to production. It provides safeguards and controls to “Go Live” incrementally and rollout features at a pace that supports the business. Development teams can work independently, delivering different features and different release frequencies.

Lower deployment risks and improves quality

Limits blast radius and users affected if features have problems or don’t work as expected. Only a small subset of users are impacted by limiting the blast radius. Developers get faster feedback and reduce break/fix cycle times, improving the quality of the features as they get deployed across the user community.

Faster decision-making and better collaboration

With progressive delegation, real-time feedback and control of a feature are routed to the team most responsible for the outcome. This improves both technical and business outcomes.

Reduce pre-production testing

Organizations can save time and money since features can safely be tested in production and by actual users instead of QA engineers. Feedback would be directly from the users in a real-world production environment.

Challenges

Moving fast without breaking things

Testing in production poses real risks. You cannot rely on automated testing to catch every issue before it hits production. Still, testing in production can be complex, degrade the user experience, and slow down your development team. It can take a long time to be confident in your release. You might have a new team of developers that are not that familiar with the application. Perhaps traffic is slow, or there are many features and code paths to exercise. Aligning the product team to a fast release cadence may force a time constraint, and impatience will lead to poor quality and broken glass.

Becoming complacent and deferring to the production environment

Once Progressive Delivery has been adopted, it's easy for teams to get lazy and wait until the feature is deployed into production to do all of their testing. Fixing a defect is cheaper if found on the developer's desktop or during integration testing in a pre-production environment. Progressive delivery enables you to get additional validation, but it should not be an excuse to cut corners.

Adds complexity to your release process and production environments

Progressive Delivery is complex and adds another practice layer to Continuous Delivery. You will have multiple application versions running simultaneously - potentially for hours or even days. Your code and shared resources need to be forward and backward compatible. If you don't have the basic CI/CD fundamentals down, it could be a recipe for disaster. A strong Continuous Delivery practice and a tightly integrated DevOps toolchain are required.

Common Practices

Collaborate when defining release progressions

When you are in the planning phase of developing your Progressive Delivery practice, make sure it's a collaborative exercise with the product team. Development, product management, and marketing should be involved in defining how to expose a new feature to the users. You want to make sure everyone is clear about why you are releasing the feature and what a positive or negative outcome is.

Automate your workflows for faster decision-making during deployment

The key to progressive delivery is quickly making data-driven decisions about whether to roll back or roll forward a feature. The data to drive that decision is typically dispersed across many different tools within the DevOps toolchain. It might take hours to approve or reject a canary. That is why it's widespread for organizations and teams to use a solution that automates workflows, securely deploys codes while meeting all compliance requirements, and provides an automated risk assessment.

Eliminate human intervention as much as possible

Implementing a progressive delivery practice is a relatively large undertaking. The main principles of Continuous Delivery are a prerequisite. Integrated toolchains and automation lay the foundation for the incremental delivery of features. Leveraging orchestration tools that can aggregate data-driven analysis and approval cycles are required to scale delivery at the pace of the business.

With OpsMx you can set up a production ready pipeline in under 60 minutes

STEP -1

Duration: 10 mins

Provision Infrastructure

Provision your infrastructure and define your pipelines in minutes using pre-defined templates. Automate your pipelines; no scripts required!

STEP -2

Integrate all of your required DevOps tools like Jenkins, Artifactory, and Splunk through our self-service integrations module.

Integrate Tools

Duration: 10 mins

STEP -3

Duration: 5 mins

Automate Compliance

Automate your compliance checks out of the box with standard static and dynamic policies. Standard open protocols are pre-defined and can be turned on at the flick of a button.

STEP -4

Set up control gates and assign ownership at each pipeline stage. Automated dashboards ensure nothing is pending on a queue for too long.

Setup Pipeline

Duration: 10 mins

STEP -5

Duration: 5 mins

Setup Deployments

Leverage ready-to-use support for on-prem, hybrid, and multi-cloud deployments. Utilize canary, blue-green, and highlander strategies with automated rollbacks. Use our unique architecture to deploy across security zones.

STEP -6

Quickly and safely make informed decisions with DORA metrics, real-time insights, and audit reports. AI-driven risk assessment automates the pre-check of software releases at every stage.

Actionable Insights

Duration: 10 mins

OpsMx

About OpsMx

Founded with the vision of “delivering software without human intervention,” OpsMx enables customers to transform and automate their software delivery process. OpsMx’s intelligent software delivery platform is an AI/ML-powered software delivery and verification platform that enables enterprises to accelerate their software delivery, reduce risk, decrease cost, and minimize manual effort. Follow us on Twitter @Ops_Mx and learn more at www.opsmx.com