Octopus Deploy

Achieving Continuous Delivery

With trunk-based development, progressive rollouts, and feature toggles

Bob Walker

Contents

Introduction

When production deployments are inconsistent and error-prone, many companies believe the best solution is to reduce production deployment frequency. They move to monthly or quarterly releases. Ironically, doing so *increases risk*.

Application teams become isolated from users. They get requests from "the business," add the feature, and move on. Once a quarter, all the new functionality (and bug fixes) are bundled into a release and deployed to users. Because of many changes, deployments take hours, with verification taking the longest.

The deployment is deemed a success. But, days or even weeks later, users start encountering new bugs. Perhaps it is an integration issue when two new features are used in conjunction, or it's an unknown use case. Sometimes, a feature worked perfectly when created two months ago, but when another feature was added, it caused a race condition. The team must release multiple hotfixes for weeks after the production deployment to fix all those issues.

In doing so, application teams have become feature factories.

The first step to solving this problem is making production deployments *consistent, predictable, and reliable*. To achieve that, application teams were required to automate their deployment pipeline. Developers were required to adopt new tooling, such as Octopus Deploy, git, or containers, or new methodologies, such as storing database migration scripts and infrastructure in version control.

However, application teams were not required to change how they *built and released new functionality*. Teams who automated their deployments and achieved weekly 15-minute deployments still build features like they did when doing monthly or quarterly releases. Features are built in long-running feature branches and delivered in a near-complete state. Attempting to merge multiple work streams causes merge hell and a testing nightmare.

Deployment automation shifted the problem to the start of the pipeline. Before code is merged into main. Almost universally, **DevOps**[1], **Continuous Delivery**[2], **Progressive Delivery**[3] **The Phoenix Project**[4], **The Unicorn Project**[5], **The Goal**[6], **Kanban**[7], and **Agile**[8] all state the same thing:

Reduce your batch size, **limit your work in progress**, and **deploy those smaller changes more often**.

What is TPF?

Curiously, when we talked to current and potential customers, they learned from the above sources that "more automation is required after automating production deployments." Instead, we propose changing how you introduce work into the deployment pipeline, which requires implementing TPF.

- **T**: **Trunk-based development**[9] for making small incremental changes instead of long-lived feature branches.
- **P**: **Progressive roll outs**[10] for frequent zero-downtime deployments of small incremental changes.
- **F**: **Feature toggles**[11] to support multiple work streams and to get fast feedback in production to ensure the feature meets the user's needs.

Understanding the challenges

This article will focus on the principles of Continuous Delivery. The aspects of Continuous Delivery solved by implementing TPF will overlap with DevOps, Progressive Delivery, The Phoenix Project, The Unicorn Project, The Goal, Kanban, and Agile. While those are just as important, including them will belabor the point.

Continuous Delivery[2] focuses on getting changes of all types—including new features, configuration changes, bug fixes, and experiments—into production or the hands of users, *safely and quickly* in a *sustainable* way. The code is *always* in a deployable state, even in the face of teams of thousands of developers making changes daily.

The goal of Continuous Delivery is to make teams user-focused. They achieve that by:

- Get feedback faster from users and check the new features they build are useful and valuable.
- Making it easy for several developers to work in the same code base and coordinate multiple work streams.
- Separating the concept of deploying a new version from releasing new functionality.
- Ensuring bug fixes, performance improvements, and security enhancements aren't blocked by new features and can deploy to production as soon as they are ready.
- Quality is built-in into every process step, not allowing non-useful functionality or critical issues to reach users.

This section will explain why we must focus on solving those challenges.

The risk of adding new features and functionality

Building new features and functionality is risky. Software development is a zero-sum game; a developer working on one feature cannot work on other features. It requires an investment from the company in both time and resources. Building the wrong feature is worse than not building the feature at all, as you could have spent the time and money elsewhere.

In the **Phoenix Project**[4], one of the main characters, Erik, states, "If you're lucky, ten percent will get the desired benefits. So the faster you get those features to market and test them, the better off you'll be." Microsoft's research showed that **2/3 of features they build deliver zero or negative value**[12].

In early 2022, we added **AWS ECS**[13] as a deployment target. A small team of developers spent time researching, designing, and implementing that functionality. They started in the summer of 2021, with our **RFC**[14] being the first time we publicly discussed the new feature. Currently, around 2% of all instances have an ECS target. It isn't because customers aren't using Octopus Deploy for AWS deployments. Around 20% of all instances are deploying to AWS.

The sooner you can get feedback, the better. Trying to pivot after months of work will take more effort than pivoting after a few days or weeks.

Challenges coordinating multiple streams of work

Not every application is a microservice. Many modern and heritage applications have multiple developers making changes concurrently. Changes require coordination to avoid merge conflicts and unexpected results. The larger the team, the greater the chance of problems emerging.

The natural instinct of the teams is to develop different features in isolation in long-running feature branches. That gives the appearance of being more productive, as they spend less time talking to other developers working on different features and more time writing code. But all the "time saved" was immediately consumed resolving dozens of merge conflicts and fixing bugs because of integration errors.

Continuous Delivery Horror Story

Ten years ago, I was an Octopus Deploy customer. After implementing Octopus Deploy (and Redgate for DB changes), my application went from quarterly deployments taking two (or more) hours to taking 15 minutes and deploying every 10 days. Even better, deployment failures and emergency fixes went from a virtual guarantee to never happening.

But we were still making feature changes, like when we had monthly or quarterly releases. Between 5 and 8 developers were on the team, making changes to the code base.

- To "avoid stepping on each other's toes," feature branches lived for multiple days or weeks.
- We delivered features in a nearly complete state. Because of long-lived feature branches, we frequently encountered merge hell.
- To avoid merging into the main branch, we often demoed new functionality to our Business Owner from our local machines to see if we were on the right track.
- Main wasn't always ready to deploy. After merging a feature into the main, finding and fixing all the bugs might take another week or two.
- Often, one feature would create a conflict with another feature. It could be a UX quirk with naming, or it could be on the backend. We wouldn't know until we merged both features into the main branch. It might be a month or two before we knew there was a conflict.
- Because the main branch wasn't ready to deploy at any point, we blocked hot fixes and performance improvements from reaching production for days or weeks.
- We'd implement merge "freezes," preventing new features from being merged after deploying a new feature, to allow a few days for hot fixes and performance improvements.

The Goal introduced the **Theory of Constraints**. One key finding Goldratt made in developing that theory was that any "productivity improvements" made before or after a bottleneck were an illusion. The book focuses on assembly lines, where bottlenecks are clearly visible. If a station on the assembly line was a bottleneck, any improvements to deliver more work to the bottleneck or process the work after the bottleneck was an illusion. If a bottleneck can only process 100 widgets per hour, it won't matter if all the steps before the bottleneck can process 200 widgets per hour. Nor will it matter if all the stations after the bottleneck can process 400 widgets per hour. The maximum number of widgets per hour the assembly line can deliver is 100.

In software development, merging into main is a bottleneck in the deployment pipeline. Almost every company in the world leverages pull requests requiring someone other than the pull request author(s) to review the change. It takes time for reviewers to approve a pull request. It also takes time to test all the changes once they are merged together. Creating a large batch of changes to make it easier to coordinate batches of work doesn't fix the bottleneck. It exacerbates it.

Impact of new features blocking deployments to production

Octopus users reading through the list of issues and coordinating multiple work streams might ask: "Why didn't you use channels? That'd stop you from needing a merge freeze on the main branch."

Octopus Deploy's **channel**[15] functionality allows users to create different lifecycles (or pipelines) for their applications. A common configuration is to set up "hotfix" and "main" channels. Hotfix deploys to staging -> production, while the main channel deploys to dev -> QA -> staging -> production.

The idea behind a hotfix and a main channel is to provide a path to production that won't impact the features you're testing in the QA environment. There is a false belief that it'll be "faster" to get a fix pushed through this way. But that configuration raises many questions.

- What branch deploys to the hotfix channel?
- How do we know what code is running in production to create the hotfix branch?
- When do we merge the fix into the main branch and all the developers' branches?
- What happens when multiple fixes are needed? Does everyone work off the same hotfix branch? Do you branch off the hotfix branch?
- Are there any verification steps in dev and QA that won't run when you skip past them to staging or production?
- The issue made it through all the gates for the full pipeline. Why would we use just a part of the pipeline for the fix?

As a result, teams implement a complex branching strategy with some form of release, feature, hotfix, and main branches. But that is similar to pouring gasoline on a fire. It only makes the problem worse; it doesn't make it better. A complex branching strategy doesn't address the crucial issues:

- Why was the main branch in a state it couldn't be deployed to production?
- Why were so many changes from features merged into the main branch simultaneously, requiring testing days or weeks?

Dangers of coupling the release of new versions and functionality

The number one cause of critical bugs is releasing new functionality to users. They come from edge cases, conflicts with other changes, or users applying unexpected use cases. Deploying a new feature carries significantly more risk than deploying a bug fix or a security enhancement.

Once users start using the new feature, there is no going back. Rolling back to a previous version risks data corruption, user interface issues, and reputational damage from a substandard user experience. Rolling back will make a problem turn into a catastrophe.

Also, when the launch of a feature is coupled with a new deployment, you have no mechanisms to control when users start using the new feature. If it is a feature on the home page or a primary landing page, the feature will likely be used right away by many users. The feature is already available to all users when you discover any issues.

Continuous Delivery Horror Story

Even after implementing Octopus Deploy, one of our biggest challenges was users using our application during the production deployment window. Despite being an internal application and doing an off-hours deployment, we'd still have users using the application. They'd be doing it to "catch up on work." Users would report all kinds of problems even though we notified them that we were deploying during that time. The analogy I always came back with is, "This is like a random person walking in on an operation and screaming because a person's chest is open."

We created an application offline page and redirected all traffic to it. For fun, we made a gif of a person banging their head against a desk, which our users started calling `headbanger`.

But that only solved the deployment problem. Once we deployed a new version, we needed to remove the offline redirect and verify the new functionality worked as expected. That would take a non-zero amount of time to confirm that the new functionality worked as expected. For minor changes, it'd take five minutes. For bigger features, it'd take 30 to 45 minutes. While we were verifying, our users would be in the application, pushing all the new buttons that just appeared. That killed any chance of a rollback. Users saw and used the new feature. Data was changed by those users. The only way to roll back was to change data directly in the database, bypassing all business rules, or restoring a database backup, resulting in data loss.

We had no control over when users started using the new feature, preventing rollbacks. If we found a critical bug, we had no choice but to fix it that night or ASAP.

A common retort is, "This is what canary deployments will solve." Canary Deployments *do not* de-risk new features and functionality. They de-risk deployment failures and provide a zero-downtime strategy. For new functionality, they ensure the latest version passes smoke tests. Unless the application is being used by tens of thousands of users every hour, there's no guarantee that routing 5%, 10%, or even 25% of traffic will reveal edge cases, feature conflicts, or unintended uses. Those issues might not happen until days or weeks after you deploy the feature.

Concerns of shifting quality gates later in the deployment pipeline

One of the most interesting questions customers have been asking recently is: "Does Octopus have the capability to monitor logs after deployment and automatically roll back if an error budget is exceeded?"

Sometimes, but not always, that question is associated with Canary deployments, where you route a percentage of traffic to the new version, monitor the logs for errors, and increase the traffic to the new version if the errors are within budget.

The core concept makes sense. Create a process that looks for critical errors, and if something is "wrong," automatically roll back to a previous version. But it doesn't work in practice.

- Not all critical errors are the same. They can result from a configuration error, an intermittent network issue, or an expired password. Others might be caused by one user with a specific configuration hitting the button dozens of times.
- It assumes that all critical errors will be logged as errors. Some errors are runtime errors where the software exhibits incorrect behavior.
- You know in advance the critical errors to create a reactive process instead of fixing the code or deployment process to make them resilient and stop them from happening in the first place.
- That there is the possibility of something surprising you during a production deployment that you didn't catch in earlier environments, either in testing or via the deployment process itself.
- The belief that a subset of users could try all the possible use cases during the deployment window. It might be days, weeks, or months before users exercise all the possible code paths and configurations.
- The deployment window is long enough for a rollback to be viable. However, the window to roll back is significantly smaller than most people realize. Once users start using the new version, the risk of data loss or corruption far outweighs the benefits of rolling back.

The simple truth is shifting quality gates to be later in the deployment pipeline increases risk. It puts a lot of pressure on the downstream processes to find those errors. Those downstream processes *will fail*. As a reaction, more people will get involved with a production deployment. The number of sign-offs will increase.

Continuous Delivery Horror Story

Many years ago, I was hired as a developer for a SaaS product. I was added to a team in the middle of creating a new module for a large customer. The module imported data from a comma-separated file created by the customer's propriety inventory management system. I was hired in August and told the module had to be completed by the end of February next year. At that point, the QA team would start their testing. Our leadership had budgeted two months of testing.

We did nominal testing as we developed the module in our developer environments for those six months. When we handed it over to QA, they found 100+ bugs on the first day. I was so embarrassed at the quality of my work that I messaged my manager to apologize. We put our heads down, fixed the bugs, deployed a new release to QA, and they found more bugs. The cycle continued until May.

We then released the module to our user base. This particular SaaS product leveraged feature flags for entitlements. Everyone used the same UI, and all data was intermingled in the same database. The deployment started at 2 am Saturday and lasted until 4:30 am. Most of the deployment was testing the change in production to find any last minute problems. Everything looked good, so we signed off to catch up on some sleep.

Later that afternoon, we started getting reports of incorrect session data from our users. Data from one customer was leaking to other customers. The error was the dreaded runtime error. The application worked and wasn't logging errors. But something was showing incorrect data.

> We could not figure out what was causing the issue. It was impossible to debug as nothing was being logged. We made the decision to roll back, which was something we'd never done before. And it had been 8+ hours since we completed the rollout. I was tasked with going through every file and commit while everyone else on the team was assigned an impact analysis. Could we roll back the code, or did we have to roll back the database? Minutes before the final go/no-go call for rollback I found the change that caused the problem. We fixed the code, pushed it to QA for verification, and then deployed it to production.
>
> We didn't involve QA *for months* until after code was "dev complete." QA spent multiple months testing and didn't find the issue. We spent multiple hours during the production deployment performing last-minute verification. Despite all those gates, this issue still occurred.

The cost to fix a bug substantially increases the further it gets in the deployment pipeline. Once a bug reaches production, it is not just people-hours but reputational damage and churned customers. As a result, the cost increases are exponential, not linear. Positioning quality gates directly in front of, or after, a production deployment does not work. They shouldn't find anything. If those gates are finding problems, then there are much bigger problems in the deployment pipeline to address.

TPF and Continuous Delivery

TPF promotes the principles of Continuous Delivery by controlling how work is put into the pipeline and delivered to users.

- Trunk-based development encourages changes in small batch sizes, making merging easy and letting dozens of developers work in the same code base.
- Progressive rollout helps limit the work in progress by enabling frequent deployments to production.
- Feature toggles and trunk-based development ensure the main branch is always deployable by turning off functionality that isn't ready for users.
- Trunk-based development and feature toggles make getting user feedback on a feature easier while building it and before releasing it to all users.

Trunk-based development

The requirement of the code, which is always in a deployable state, has created several branching strategies. With **feature branches**[16] / **GitHub flow**[17], **GitFlow**[18], **Forking**[19], and **Trunk-based development**[9] being among the most popular.

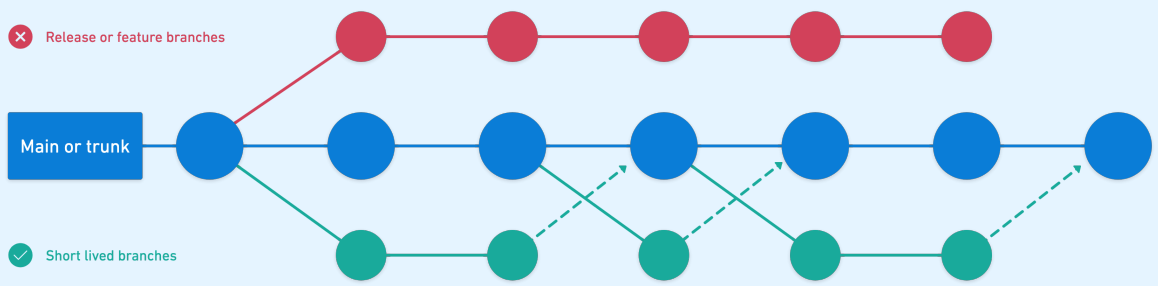
Of the popular branching strategies, only trunk-based development encourages working in small batches and incrementally building features.

> A source-control branching model, where developers collaborate on code in a single branch called "trunk" and resist any pressure to create other long-lived development branches by employing documented techniques.

The goals of trunk-based development are:

- Avoiding merge hell: How often have you worked on a long-lived feature branch, and merging to main took hours or days?
- Small batch sizes: Bigger ones are much riskier to deploy than smaller ones. A bigger batch size means more code has changed, meaning more can go wrong.
- Minimize **yak shaving**[20]. Branching strategies, like **GitFlow**[18], create complexity. Poor branching strategies result in constant yak shaving to handle merging between branches, releasing, and other ceremonies.

I prefer trunk-based development "at scale".



Trunk-based development at scale is best done with short-lived branches.

Applying trunk-based development principles to **GitHub flow**[17] makes for a powerful combination. Changes are made in short-lived branches and merged via pull requests. Each branch should live no more than a couple of days, ideally no more than a few hours.

That doesn't mean merging unfinished code into the main branch. Many features can take weeks or months to finish. Developers and product managers are responsible for creating small units of work that incrementally add functionality (and tests) until the feature is ready to be used. Each incremental change, including database changes, should be deployed to production.

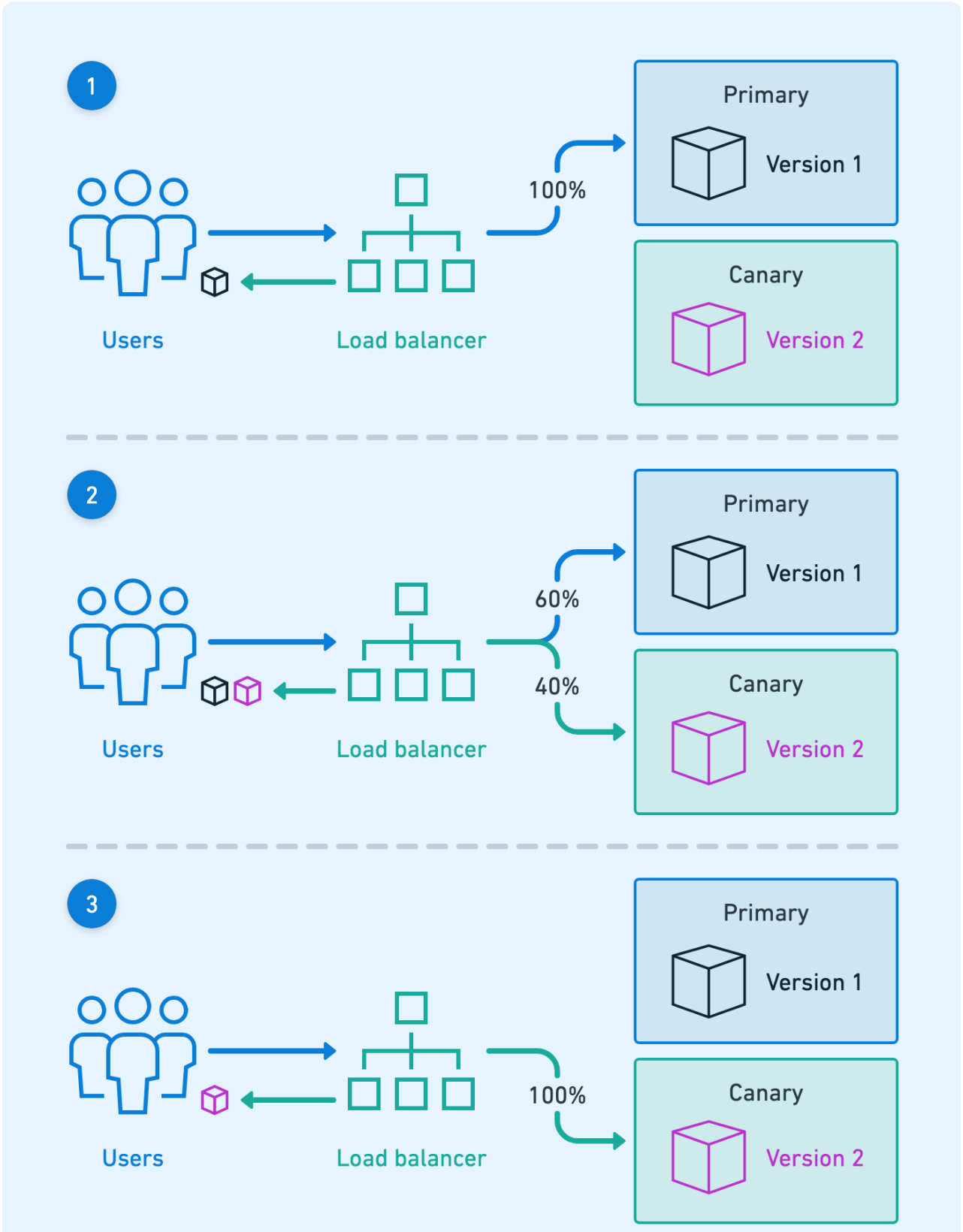
When a feature will likely take you weeks or months to complete, use several dozen short-lived feature branches, each merged to the main branch and deployed to production. As the feature isn't ready for general use, you can use a feature toggle to hide it until it is ready.

Combining trunk-based development and feature toggles allows you to have multiple features "in flight" and merge them into the main branch. Because developers incrementally check in code, they are less likely to create conflicts with other developers' work, and when they do, they can resolve them much sooner. With changes integrated early and often, you can test and verify sensible feature combinations.

Progressive rollouts

Trunk-based development encourages frequent production deployments. However, you cannot deploy to production daily if each deployment has downtime or a high failure rate. Any downtime might breach SLAs, frustrate customers, or prevent employees from doing their work. In addition, automation isn't perfect; a deployment could still fail. Progressive delivery solves those problems.

Blue/green and canary strategies follow the same core principle. Deploy a new version to a "staging" location in production. Verify the latest version while all the users remain on the previous version. Once verification is complete, route users to the new version. The difference between canary, blue/green, and staging is the percentage of traffic routed.



During a canary deployment, traffic is gradually switched to the new version.

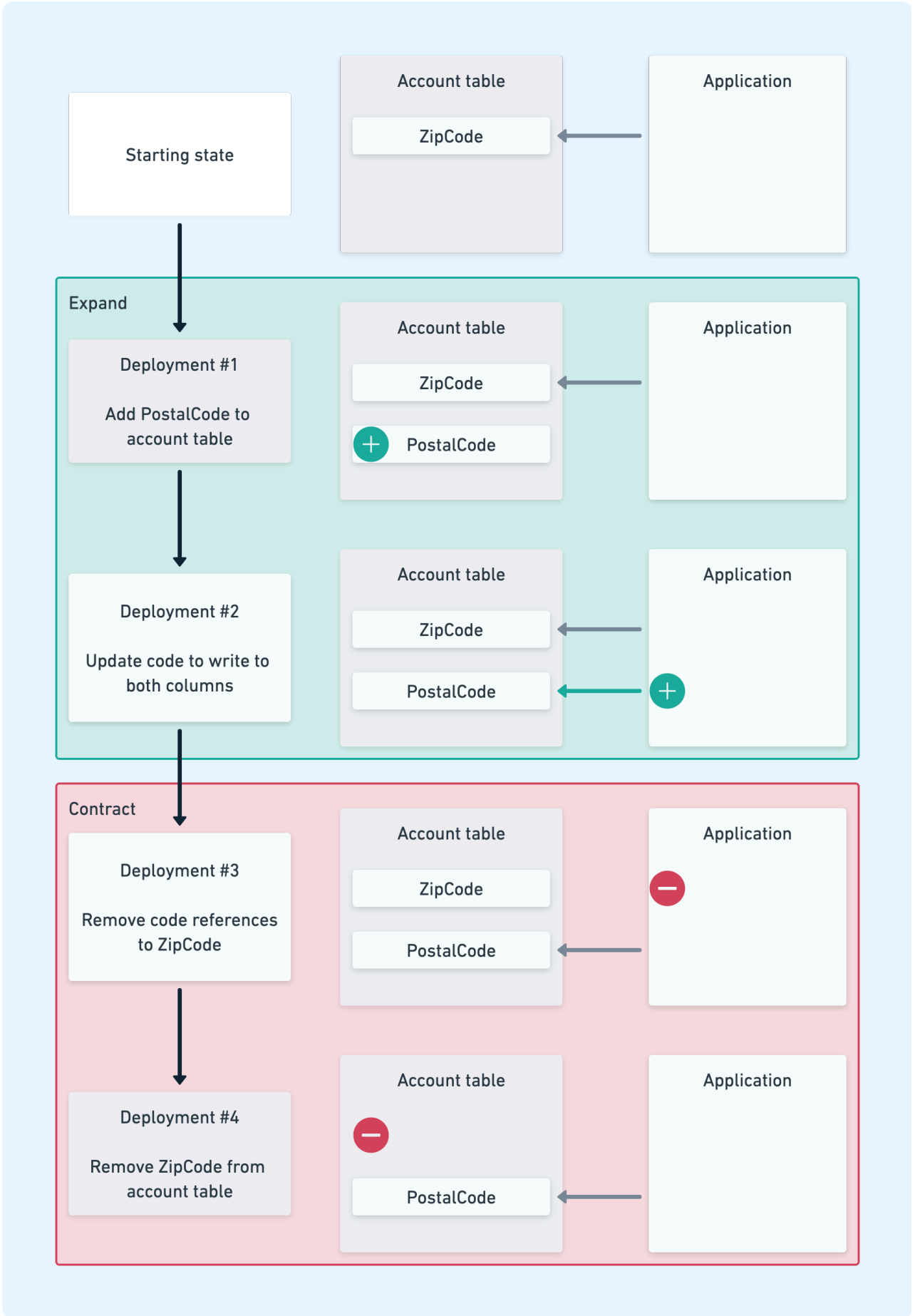
Deployments can fail for several reasons.

- A database migration fails.
- Infrastructure cannot be created or updated correctly.
- Environmental problems, like network issues.
- The application version won't start after deployment.
- The new application version crashes minutes after the deployment.

The advantage of progressive rollouts is that every user remains on the old version until the new version is verified. Users won't know if the deployment or verification fails because they continue to use the previous version.

There is no rule to say you must immediately route users to the new version when verification is complete. You could deploy at 2 pm and wait until 9 pm to route traffic when there are fewer active users.

It is important to remember that two versions of your application will be running in production concurrently. Unless you are keen to solve **the split-brain challenge**[21], those two versions will point to the same database. That means that all database changes must be backward compatible. That requires implementing the **expand and contract pattern**[22].


Starting state
Account table
ZipCode
Application
Expand
Deployment #1
Add PostalCode to account table
Account table
ZipCode
PostalCode
Application
Deployment #2
Update code to write to both columns
Account table
ZipCode
PostalCode
Application
Contract
Deployment #3
Remove code references to ZipCode
Account table
ZipCode
PostalCode
Application
Deployment #4
Remove ZipCode from account table
Account table
PostalCode
Application

Just like code changes, you should make incremental database changes. In the example in the diagram, the goal is to rename a column from `ZipCode` to `PostalCode` to support non-US-based addresses. Running the rename stored procedure would be the fastest solution, but that is not a backward-compatible change. Instead, that change requires multiple deployments.

1. Add the new column `PostalCode` as a nullable field to the table, update the necessary views and stored procedures, and deploy the change.
2. Update the code to write to `PostalCode` and `ZipCode`. Deploy that change.
3. Run a migration script to populate missing `PostalCode` columns from the `ZipCode`
4. Update the code so it no longer reads from or writes to `ZipCode`. Deploy that change.
5. Remove the column `ZipCode` from the table. Deploy that change.

Each of those changes is backward compatible. The challenge to this approach is inspiring the appropriate discipline from all the developers on the application team. The expand and contract pattern fits nicely with trunk-based development, as both encourage incremental changes.

Feature Toggles

All too often, I hear from people who think feature toggles are a simple binary on or off for all users. You might turn it on in testing environments but leave it off for production. Feature toggles are far more robust than a simple binary. **OpenFeature**[23] provides specifications and libraries to create robust feature toggles that let you get early feedback and progressively roll out changes.

Instead of an application-level binary switch, you create multiple user segments. As you roll out, the toggle switches on for each segment.

The segments can be whatever you want them to be:

- Staff or internal users
- Early adopters or beta users
- Users in different timezones
- Users in specific locations, like country, state, or province
- Users with specific entitlements, like enterprise customers

With feature toggles, the focus is often on how you deliver features to users. It's why they are frequently lumped in with canary and blue/green deployments as part of **Progressive Delivery**[3].

However, feature toggles can do more than progressively deliver user functionality. They can help you answer crucial questions like, "Am I building the right thing?" You can use them for days, weeks, or months to get user feedback while building a feature. Once the feature is complete, you can progressively roll it out to the entire user base.

Combining feature toggles with trunk-based development lets you incrementally add functionality to a feature until it is complete. But you can also let users try the new functionality *in production* before it's finished. Users can provide feedback on what works, what doesn't, and what can be improved. Concepts such as pre-alpha, alpha, and beta are possible. Each stage can last a few weeks to months, depending on the feature.

1. Pre-alpha - internal or staff users can **dogfood**[24] new functionality.
2. Alpha - early access to a subset of (willing) users for early feedback. These users have to be willing to accept incomplete functionality and be willing to provide detailed feedback.
3. Beta - test near-complete functionality with a larger group of users before everyone. Ideally, users have an easy way to volunteer. Beta testers ensure the feature works "at scale" and no unknown show-stopping bugs exist.

The advantages of this approach are numerous for building new features:

- Application teams get feedback faster, allowing them to verify assumptions and, if necessary, make changes to build a more useful feature. This increases the likelihood that the feature will be valuable to other users.
- Features that take months to build can be used *in production* for weeks by a subset of users. Unknown use cases, bugs, or integration issues will be found and reported before reaching the entire user base.
- Multiple features can be worked on simultaneously, with each feature having different alpha and beta testers.
- Developers can test how features interact with each other much sooner. Either manually or via automated tests. If a critical collision occurs, they can turn off a specific combo of features and get unblocked while the issue is fixed.

Evolving the Software Delivery Lifecycle

TPF will help application teams evolve their Software Delivery Lifecycle (SDLC) by unlocking the potential of Agile, eliminating rollbacks to previous versions, and improving developer experience. We've seen this happen at Octopus when we implemented these practices.

Moving beyond Scrum

Scrum[25] is a popular Agile team collaboration framework. An oversimplification is that teams break work into goals within time-boxed iterations (1 to 4 weeks), called sprints. At the end of the sprint, the team demonstrates the new functionality to stakeholders for feedback. The cycle repeats until a feature is completed.

The most common anti-pattern (and implementation) of Scrum is the "multiple waterfall" method. With **waterfall**[26], all development work is broken into linear sequences. For example, testing cannot start until the development of all new functionality is complete. Development cannot begin until the design, with every possible use case, is ready.

Continuous Delivery Horror Story

I worked for a company that used the waterfall model for decades. My team all worked on the same application. But we were split into two groups. Each group deployed a new release every 6 months. We were staggered, so a new release occurred every quarter. One group might be in the "QA Phase" while the other was in the "Development Phase." Each phase had a budget of one to two months! QA didn't start testing until the development work was complete. Generally, once they completed testing one group's work, they'd move on to the next group's work.

When we first implemented Scrum, we used the "multiple waterfall" method, which meant that instead of 6-month cycles, we worked in 3-week sprints. However, the work was still linear. The first 2 weeks focused on development work, with 1 week of testing. The goal was to have something to deploy at the end of three weeks.

- There is tremendous psychological weight on everyone involved to meet the sprint deadline. There's also another sprint after the current one to worry about. That substantially increases stress and leads to burnout.
    - Developers still work in long-lasting branches and only merge into the main branch once a feature is ready for QA. They want to be as "feature complete" as possible to avoid unnecessary back-and-forth with QA.
    - QA is idle for days or most of the sprint before they have something to test. They must test frantically at the last minute, increasing the risk of missed use cases.
    - Planning meetings would devolve into arguments over story sizes and capacity.

> - The amount of time available for feature work decreases with each sprint.
>   - Multiple changes are pushed to production at once, increasing the likelihood of critical bugs. Developers must then fix those bugs in the next sprint, reducing the time available to work on new features.
>   - Production deployments still involve a lot of ceremony and consume much of everyone's time instead of being predictable and routine.
>   - Production deployments must occur "off-hours" because so much has changed.
>
> Eventually, the tech debt piles up, and a "recovery sprint" is required, during which developers work on bug fixes and improvements.
>
> The developer experience was terrible, as the stress of delivering and meeting those arbitrary deadlines was immense. When I didn't get everything done, I felt like a failure. The stress was so bad that I jumped at the opportunity when a recruiting firm contacted me. Anything to eliminate the stress.

Nothing is wrong with committing to completing work over a 1, 2, or 3-week period. The deadline can be a forcing function to reduce scope and focus conversations.

The common misconception, and as a result of the downfall of Scrum, is that "a production deployment happens" at the end of a sprint. While fewer changes exist than a quarterly or 6-month release cycle, not much else is improved. In addition, the rigid nature of the sprint, specifically always following a set 1, 2, or 3-week sprint, causes a "junk drawer" of requests to be added to fill out capacity.

TPF promotes deploying to production *throughout the sprint*.

- Trunk-based development allows developers to make incremental changes to the code base. Once merged into the main branch, that change should go to production. QA gets involved much sooner, decreasing the chance of missing a use case.
- Progressive rollouts make it easy for those changes to go to production by de-risking deployments and providing a zero-downtime strategy. Deployments can occur in the middle of the day and are routine and predictable.
- Feature toggles hide the functionality from users until it is complete, which could take 2 or 6 sprints. However, developers can enable toggles for a subset of users to get feedback on the new feature.

Because the changes are rolled out throughout the sprint, teams can move beyond the rigid 1, 2, or 3-week cycle. Imagine a team works on via 3-week sprints. During planning, they see one milestone or feature takes 1 week to deliver. Another milestone or feature might take 2 weeks to deliver. With a rigid 3 week cycle, they'd try to fit both items into the same sprint. Both features would be deployed at the end of the sprint. But what happens if the first feature or milestone takes 1.5 or 2 weeks to deliver? Or the first milestone/feature takes a half week to deliver, but the second feature/milestone requires 3 weeks? Or, if a developer is needed for both features?

With TPF, that doesn't need to happen. Changes are made incrementally and deployed frequently. The features can be worked on concurrently and delivered incrementally to production. If one feature/milestone takes 1.5 weeks and the other takes 2.5 weeks, it won't matter. Teams aren't trying to fit work into a bucket.

No more rollbacks

When we refer to rollbacks, this means redeploying a previous version of the application and database. Rolling back a database requires "downgrade" scripts or restoring a backup. In that context, rolling back to a previous code version is a "break glass in case of emergency," not a default recovery strategy. There is no such thing as a risk-free rollback. Restoring a previous version is often *significantly more risky* than fixing the issue.

There are many problems with rolling back.

- Rolling back would turn off the new features *for all users*, even if the issue impacted a small (but important) group of users.
- Any other change included in the new release is rolled back, including other features, bug fixes, security enhancements, or performance improvements.
- Rollback plans are often untested. The first time the plan is executed is when you do the rollback, typically in production.
- Restoring a database from a recent backup will result in data loss. Alternatively, running the previous application version with the latest database version increases the chance of data corruption.
- The window to roll back is much smaller than most people think. For a lot of applications, it's less than an hour. After a few hours, a rollback is not a viable option.

Continuous Delivery Horror Story

As a lead developer, my team was responsible for our bank's loan origination system. One evening we deployed a new version of our application that included a fix to our dashboard. The dashboard would refresh once and never again. It was caused by using `setTimeout` instead of `setInterval` in our JavaScript. That one line of code increased traffic by 1800%. We were requesting data faster than the database could handle. We crashed our application and every other application running on that SQL Server.

Unfortunately, the new version included a database change. We deployed the new application version at 7:00 pm, and it wasn't until 9:15 am that the application finally crashed. We were faced with a choice:

1. Redeploy the previous application version and don't modify the database -> We never tested and weren't sure would work.
2. Redeploy the previous application version and restore a database backup -> That would result in data loss. We'd likely spend weeks reconciling any "data funk". That's not acceptable to a bank or for loans.
3. Change the `setInterval` back to `setTimeout` -> That would cause the bug to reappear but give us time to find a permanent solution.

We opted for option 3. We pushed that change through the standard deployment pipeline and up to production in 45 minutes. It went through our entire suite of unit tests. Weeks later, we found a better solution to the bug using a unique caching solution. Not only that, we also improved our monitoring and testing.

Treat the deployment pipeline like an assembly line. Always moving forward. Never backward. By adopting TPF, redeploying a previous version is no longer needed.

- Incrementally adding functionality instead of merging everything as a "big bang." Smaller changes mean less can go wrong, and they are much easier to test.
- With pre-alpha, alpha, and beta release rings via feature toggles, the functionality will have been used and tested by a subset of users in production. It decreases the likelihood of an unknown configuration causing a critical issue.
- Progressive rollout patterns allow you to deploy and verify changes in production while users remain on the old version. If the verification fails, users stay on the old version.
- With a canary approach, stop routing more traffic to the new version if you discover a critical issue impacting all users. Fix the problem, then start again.
- Providing the capability *from the start* to turn off new features to a subset of users (or all users) in case something goes wrong. This prevents remedial action from impacting other changes included in the deployment, like bug fixes, performance improvements, security enhancements, and other features.
- All fixes go through the same pipeline. Once the fix is deployed, it can be tested with a subset of users impacted.

That is not to say every feature will be bug-free; far from it. You'll always encounter unknown use cases, random configurations you weren't expecting that cause your feature to fail, or substandard performance. But you'll have a ready-to-use, battle-tested mechanism to turn off the feature. Imagine saying to a user, "We've identified the root cause. We'll push out a fix overnight. In the meantime, we've turned the feature off to unblock you."

Improved developer experience

TPF improves the developer experience by providing a framework for incremental changes and fast feedback.

> ***Continuous Delivery Horror Story***
>
> Early in my career as a developer, I was working on an internal application used by thousands of users. The project was a rewrite of a heritage application on a mainframe, and I was creating a reporting module for our users.
>
> I was new to the team, and the other developers said, "Oh, the users don't know what reports they want until they use them. Let's make it super easy for users to create custom reports." So that's what I created.
>
> When I pushed it out, the users *hated it*. What they wanted was a standard set of reports with some filters. As a developer, that is very frustrating. I put a lot of effort into creating a custom report module. But it wasn't useful to users, so I scrapped 80% of the code and started again.

While TPF doesn't explicitly call out Developer Experience (DevEx), it is a key benefit.

- Trunk-based development means small incremental improvements to the main code base, not big-bang changes. It reduces the possibility of merge hell or wasting unnecessary cycles merging changes from the main branch into a developer's branch. Small changes also reduce the time a developer spends figuring out which change caused a critical bug.
- Progressive delivery uses zero-downtime deployment strategies to deploy frequently during the day. Deployments no longer need to occur off-hours or on the weekends. Once you've verified the latest version, you could delay traffic routing until off-hours, but the risk is already minimal, thanks to the validation steps and small batch sizes.
- Feature toggles help you make sure the features you are building are useful to the user. Developers can get feedback from a subset of users using the feature in production over the days, weeks, or months the feature is being built. Users will also try various configurations and use cases and will help you find the rough spots. That'll significantly reduce the chance of a critical problem when it goes live for everyone.

Easier context switching

One of the main challenges developers face is context switching. In *The Goal*, this was demonstrated by critical stations in an assembly line. When a different type of widget needs to go through a critical station, setup time is required. Imagine an assembly line for Fender Guitars. Fender makes solid-body electric guitars and hollow-body acoustic guitars. The materials, shape, and construction of each guitar is very different. If both types of guitar must pass through the same station, a great deal of setup would be required to change from electric to acoustic and back. When asking developers to context switch, they encounter the same thing. Unlike an assembly line, there is no visible indication of the change. When they switch context, they first have to flush all the code they've been thinking about out of their heads. Then, load all code for the next task.

One of the biggest challenges developers encounter is needing to "put down" what they are working on to fix a critical bug. If a developer is in a long-lived feature branch, they must flush a lot of code out of their heads to look at the code representing production. They have to spend a lot of time loading production code into their heads. Once they fix the bug, they then have to spend a lot of time re-loading the feature branch code.

Continuous Delivery Horror Story

The loan origination system I was responsible for as the lead developer had a unique position in the company. It was used by over 1200 users, with a diverse set of user roles, from loan officer to financial analyst to risk analyst and more. The application interacted with over a dozen internal truth center API services for financial information, customer PII, collateral, risk rating, and credit rating. When a user encountered an error, the request was typically routed to my team first, then redirected where appropriate.

The business owner and business analyst did their best to perform tier-1 support for user issues. But for many tickets, they'd need to escalate the issue to a developer. We initially tried a round-robin approach to tickets. Unlike other teams, we'd have a consistent flow of 2 to 5 issues a day to investigate. Not all issues required a code fix. Sometimes, the problem could be solved by changing a value in the UI. Or, proving the problem happened with an internal service and handing it over to the appropriate team.

The round-robin didn't work. With a round-robin, it was guaranteed every developer would be interrupted to research an issue every two days. We noticed a nosedive in our ability to release features. Developers would say, "If I could just get a couple of hours of focus, I could finish this task and get feedback."

Our solution was to create an "on-call" rotation. For one week, a developer, including myself, would be "on-call" for any issues that needed to be escalated. I put on-call in quotes because we didn't carry a pager. It was a "if you need to bother someone on our team, bother this person." My team name was Team Avengers. As a visual cue, I bought Thor's Hammer, and we placed it on the desk of the "on-call" person.

TPF makes that much easier. With small incremental changes, the delta between production and what the developer is working on is much smaller. Additionally, unless the application is crashing in production, the bug can likely wait an hour or two as the developer finishes their work. With TPF, the chance of the application crashing in production is much lower. Small incremental changes are made and pushed to production frequently. It is much easier to test and find that potential show-stopping bug.

More thorough reviews

Long-lived feature branches mean large pull requests. Imagine a feature that has been worked on for three weeks in a single branch. When the pull request was submitted, 60 files were changed, covering 3,000 lines of code and database migration scripts. The developer(s) submit the PR to the tech lead or senior engineer. They are busy as well, but they know the change is important, so they carve out the entire afternoon, 3.5 hours, to review and submit comments. That means the tech lead can devote around 3.5 minutes to reviewing each file changed. Or about 4.2 seconds per line of code.

The tech lead is human. They will lose focus the longer the review goes on. What are the possibilities for the tech lead to find every potential issue with the pull request? They could miss a rename column statement in one of the SQL Scripts. Or a potential race condition.

> ***Continuous Delivery Horror Story***
>
> I'm the author of two very complex Octopus Deploy Community Library steps, **Run Octopus Deploy Runbook**[27] and **Deploy Child Octopus Deploy Project**[28]. Every community library step undergoes a review from someone at Octopus Deploy. If an Octopus Deploy employee wrote the step, a different person must approve it.
>
> The steps are close to 2,000 lines of PowerShell code in a single file. When originally submitted, the reviewers didn't know where to start. I had to provide guides on how to test the step, what to look for, and more. Not only that, but once I submitted the PR, much of the context was flushed from my memory. When I was asked about a particular function or a bug was found, I had to spend time reloading all that information. It was difficult for everyone involved to review such a large change. Despite our best efforts, problems still get in, requiring a frantic fix. There is a reason why those steps have had over 25 revisions when most steps have had fewer than 10.

The solution is not to have AI review the code, as it will likely make the same mistakes as the tech lead can. It can speed up the review by looking for common problems and ensuring standards are met. But AI will not have the context for what the code *should be doing*.

With TPF, the same feature is built incrementally and merged into the main branch every couple of hours or at the end of the day. The feature isn't made available to users during the same three weeks. The difference is how often the code is reviewed and how long the reviewers spend reviewing the change. A 15-minute review of a change with 3 files and 100 lines of code will be much more thorough. Each file gets around 5 minutes of review, or about 9 seconds per line of code changed. The tech lead is less likely to lose focus, and it will be much easier to find critical problems.

With TPF, code is reviewed much more often and with greater scrutiny.

Case Study: Octopus Cloud

In the early days of Octopus, we implemented canary deployments using release rings.

- We'd deploy to our staff instances and wait a day to see if anything blew up.
- After a day, we'd deploy to our Octopus Insiders and get their feedback.
- If Octopus Insiders found no problems, we'd deploy to our early adopters after 7 days.
- We'd wait 7 more days; if the early adopters didn't report anything, we'd deploy to the stable customers.
- Then we'd deploy to any laggards (very rarely did we have any).

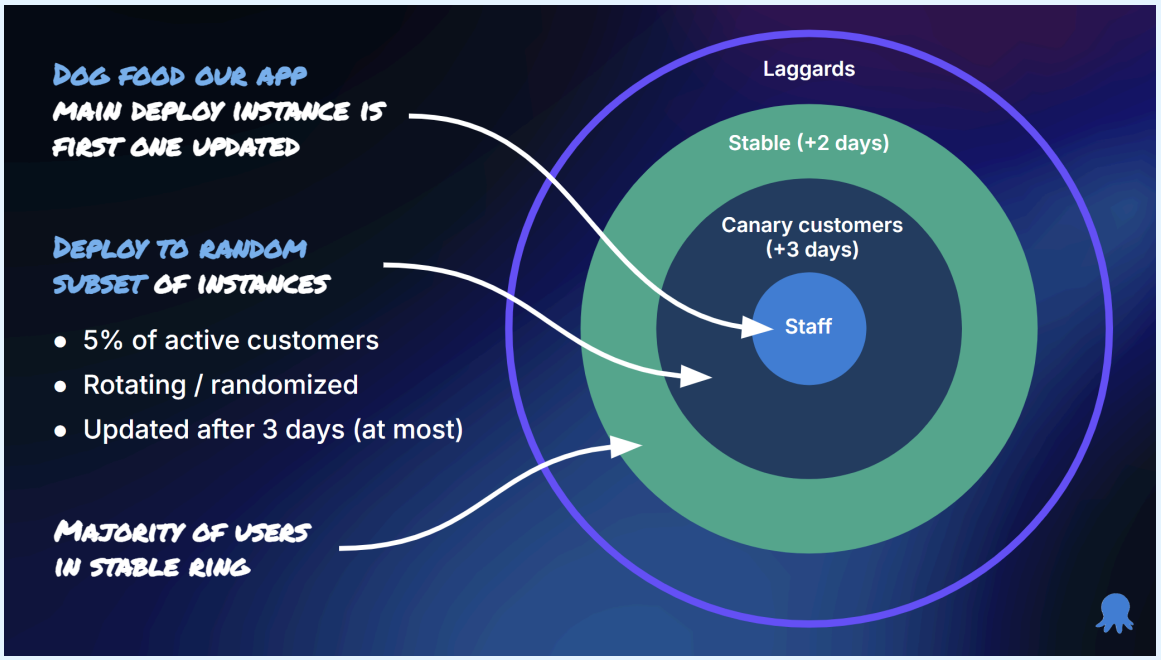
We believed canary deployments would be a panacea. In reality, it had many limitations.

1. There is no guarantee a subset of users executes all code paths. Unless you are a FAANG-sized company, a subset of traffic is unlikely to execute every code path in a few hours or days.
2. Long deployment durations. The natural reaction to the code path limitation is to extend the canary deployment duration to hours or even days.
3. Indiscriminate traffic routing. Targeting a specific set of users via a load balancer is nearly impossible. In the best case, you can use HTTP headers, but that is often too broad.
4. Disjointed user experience. When canary deployments take days or weeks, and traffic routing is indiscriminate, you cannot rely on "sticky sessions" to keep a user on a particular version.
5. User expectations. I have a laptop and an iPad. When I go to a website, I expect the same experience when using either device (even when used within minutes of one another).

Even with a 15-day canary cycle, critical bugs still got to the stable ring on Octopus Cloud! The 15-day cycle was the best case, as we might find a show-stopping bug. At one point in 2022, customers in the stable ring didn't get updates for almost 50 days!

Our engineering leaders (managers and senior engineers) realized we needed to change. We adopted TPF.

- Trunk-based development with short-lived branches, no more than a few days.
- Feature toggles that we could turn on or off for a specific customer instance.
- Shorter release rings deployed to our main instance, which also deploys Octopus (we use Octopus to deploy Octopus).



There have been numerous benefits to this approach.

- We have four concurrent early access programs for features on our **roadmap**[29]. Each program has a different list of customers.
- We can incrementally add new functionality to features and gather feedback. For example, I get early access to many features and see the feature taking shape every other day. It allows us to get feedback from our customers (not just from me) and adjust before releasing it to everyone.
- We can eat our own dog food and find the rough spots. For example, internally, we used the updated Octopus Deploy UI released last year for months.
- Rolling out new functionality is an update to a feature toggle. Even better, if a customer reports an issue, we can turn it off to unblock them. Meanwhile, everyone else can use the new functionality.

Conclusion

TPF, trunk-based development, progressive delivery, and feature toggles are part of a tripod. All three are needed to achieve true Continuous Delivery, getting changes of all types—including new features, configuration changes, bug fixes, and experiments—into production and into the hands of users, *safely and quickly* in a *sustainable* way.

- Without trunk-based development, you cannot iteratively change a feature to gather feedback faster.
- Without progressive delivery, you increase the risk of a deployment failure or prolonged downtime.
- Without feature toggles, you cannot get feedback for the feature from a subset of users, or if something wrong happens, you cannot turn off the new feature.

Until next time, Happy deployments!!

References

1. https://www.amazon.com/DevOps-Handbook-World-Class-Reliability-Organizations/dp/1942788002
2. https://continuousdelivery.com
3. https://www.amazon.com/Progressive-Delivery-Build-Right-People/dp/1950508978/
4. https://www.amazon.com/Phoenix-Project-DevOps-Helping-Business/dp/0988262592/
5. https://www.amazon.com/Unicorn-Project-Developers-Disruption-Thriving/dp/1942788762/
6. https://www.amazon.com/Goal-Process-Ongoing-Improvement/dp/0884271951/
7. https://en.wikipedia.org/wiki/Kanban
8. https://en.wikipedia.org/wiki/Agile_software_development
9. https://trunkbaseddevelopment.com/
10. https://octopus.com/blog/common-deployment-patterns-and-how-to-set-them-up-in-octopus
11. https://martinfowler.com/articles/feature-toggles.html
12. https://ai.stanford.edu/~ronnyk/ExPThinkWeek2009Public.pdf
13. https://octopus.com/blog/using-ecs-deployment-step
14. https://github.com/OctopusDeploy/StepsFeedback/issues/5
15. https://octopus.com/docs/releases/channels
16. https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow
17. https://docs.github.com/en/get-started/using-github/github-flow
18. https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow
19. https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow
20. https://en.wiktionary.org/wiki/yak_shaving

21. https://en.wikipedia.org/wiki/Split-brain_(computing)
22. https://www.tim-wellhausen.de/papers/ExpandAndContract/ExpandAndContract.html
23. https://openfeature.dev
24. https://en.wikipedia.org/wiki/Eating_your_own_dog_food
25. https://en.wikipedia.org/wiki/Scrum_(software_development)
26. https://en.wikipedia.org/wiki/Waterfall_model
27. https://library.octopus.com/step-templates/0444b0b3-088e-4689-b755-112d1360ffe3/actiontemplate-run-octopus-deploy-runbook
28. https://library.octopus.com/step-templates/0dac2fe6-91d5-4c05-bdfb-1b97adf1e12e/actiontemplate-deploy-child-octopus-deploy-project
29. https://roadmap.octopus.com/tabs/2-planned

Octopus Deploy

Octopus Deploy
Level 4, 199 Grey St
South Brisbane, QLD 4101, Australia

Email: sales@octopus.com

Phone: +1 512-823-0256

octopus.com