# The Pain of Infrequent Deployments, Release Trains and Lengthy Sprints

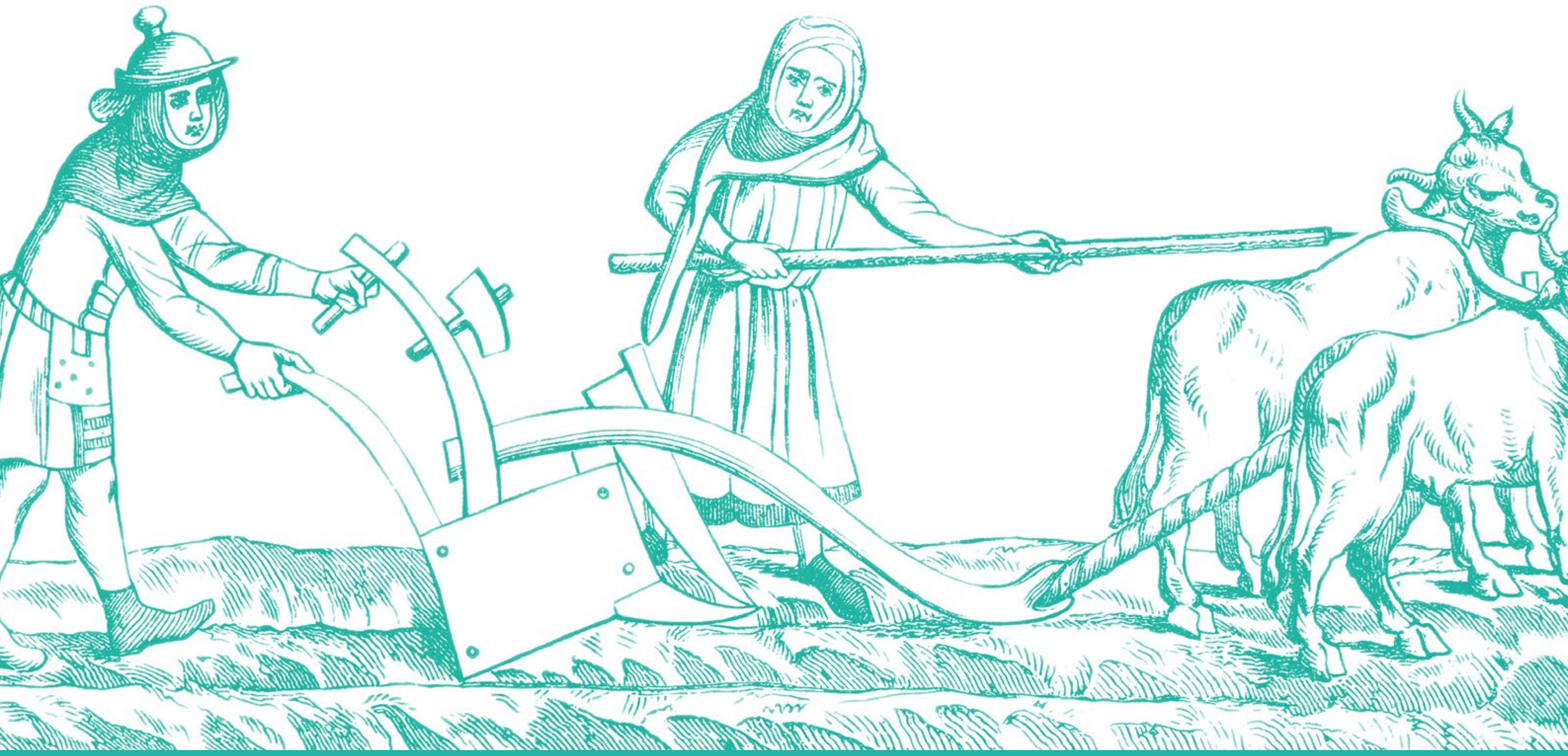
codefresh

One of the most critical metrics when it comes to the software delivery process is deployment frequency, which measures how often releases are happening in production. While in theory all organizations should strive to deploy as often as possible, in practice the benefits for frequent deployments are often overlooked or buried under endless technical debates.

In this article we will turn things around a bit. Instead of talking about the benefits of frequent deployments we will focus on the pains of NOT deploying as often as possible. Deploying monthly or trying to deploy only when an arbitrary day arrives (often the end of a sprint or a “release train”) is a practice associated with several pains and issues which unfortunately most people take for granted or think that this is the way software is supposed to be released.

# The Dark Ages

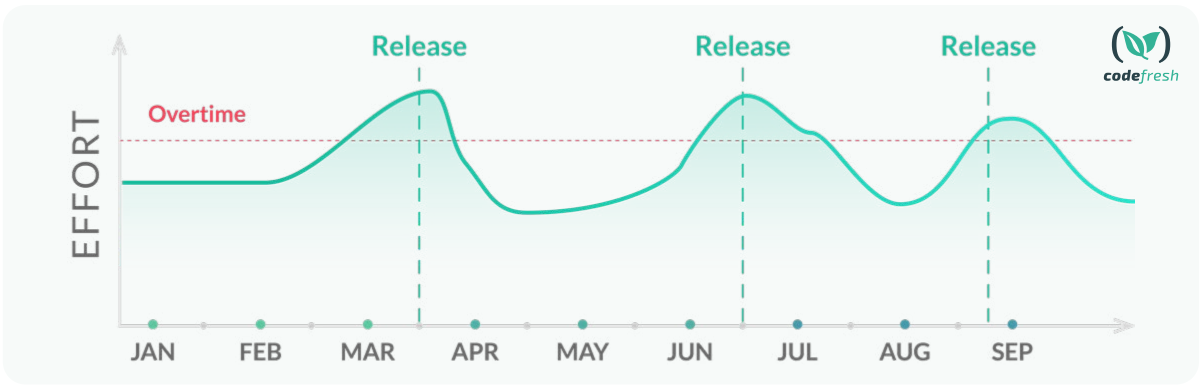
Deploying every month or few months (release trains)



This is the traditional way of shipping software and is followed even today by several companies who are either conservative on adopting new technologies, or are simply accustomed to the pains of infrequent releases. Some illustrative points that reveal your organization is still in the dark ages are the following:

- Your organization deploys to production every 1-3 months
- Deployments are big-bang events where multiple teams are involved
- Code changes must pass from multiple rounds of approval
- There are extensive “code freeze periods” or critical dates when “cutting a release”
- Several departments work overtime during a release and this is considered normal
- Production Rollbacks are lengthy and complex processes
- Your internal Wiki has dedicated pages with checklists for deployments
- Test and staging environments are static and pre-booked for specific teams
- There are well defined hand-off periods between developers, testers and system administrators which delay the deployment process even further.

While organizations that deploy like this might look disciplined and cautious to the outside observer, the sad truth is that most people have roller coaster feelings in the course of a software release and this makes the whole process very stressful.



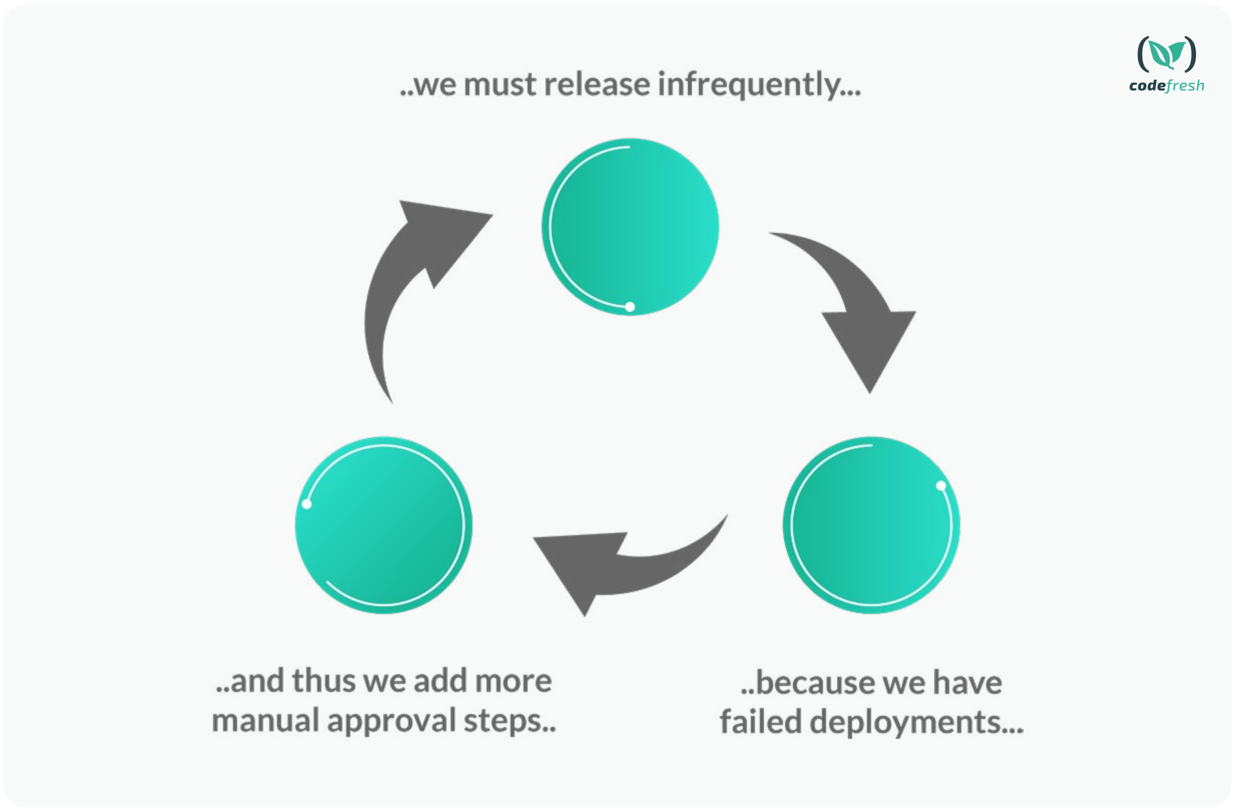
The start of a release train/cycle/sprint starts with enthusiasm and great anticipation by all teams. Teams and their project/product managers are planning new features and making promises regarding the delivery date. Developers start implementation confident that this release will be better than the previous one. However, once teams hit the time period one or two weeks before the deadline of merging/code freeze, several discouraging patterns emerge:

1. A feature is pushed to the next release train because there is simply not enough time to implement/test/integrate it. This often results in various features on hold for several more iterations, leaving them in multiple different testing stages.
2. A feature is deemed “important” enough and MUST be part of the release train. Developers and particularly testers work overtime (often cutting corners) to get it released into production.

This paradigm has three major disadvantages. First of all it makes all production releases a stressful event where people are pressed to deliver a specific feature for no particular reason other than the fact that a delivery date was "promised" in the beginning of the release train.

While most organizations like to think that the dates of releases are important milestones and have to be religiously followed, in practice the decision regarding a specific date was taken in a completely different time period (often months before) with insufficient information of the risks and effort associated with a feature.

But the biggest disadvantage (and the one is that is most damaging to the moral of all teams) is the self-fulfilling prophecy of bad deployments.


..we must release infrequently...
..because we have failed deployments...
..and thus we add more manual approval steps..

**This is an endless cycle:**

1. Releases happen very infrequently
2. When they do happen several things can go wrong (often caused by the differences between the production environment versus the staging environments)
3. The idea that production deployments should happen infrequently is further reinforced by the bad experience of past failed releases
4. More manual checks and approval steps are added that stand as roadblocks to frequent deployments

This cycle is then evolving on its own and touches all aspects of the software delivery process in a negative way. Some of its manifestations are:

- Precaution: Developers do not get access to production machines. Actual result: developers don't know how production looks and cannot match their staging systems
- Precaution: Test environments are statically created and must be prepared in advance in order to "match" production. Actual result: developers are blocked until the test environment is ready pushing further back the feature implementation
- Precaution: Endless layers of approval steps that try to predict bad deployments. Actual result, development time is wasted on pull requests that are queued and by the time they reach production might need more fixes to match the current state
- Precaution: Big checklists with the steps needed for a production release have to be followed manually by developers/operators. Actual result, the presence of manual steps needed for a release is a big reason why a release might fail
- Precaution: Developers, Testers and operators communicate strictly via ticket assignments. Actual result: teams are siloed and unaware of they can improve the workflow of their neighbor

The last important point of infrequent releases is the fact that they set the pace for the whole organization and can limit your teams, especially in the presence of particularly experienced people.

For example if your organization has decided that it will deploy to production every three months, then even if tomorrow you hire the best developers/testers/operators in your area and they are 4x as productive, you put an artificial brake on them and still force them to release every three months.

This is a waste of human talent and in several cases some people will simply leave the company and try to find another one with less bureaucracy and more freedom on how deployments are performed.

In summary, this deployment workflow is detrimental to the morale of your teams and makes every production release a very stressful event for all involved stakeholders.

## The Renaissance

Deploying at the end of the sprint (weekly or biweekly)

codefresh

The previous paradigm of monthly releases is now being challenged by the weekly (or biweekly) deployment scheduled. This is often seen as the agile evolution of a company and is based on short-lived feature branches and the dynamic creation of test environments as they are needed.

The appearance of cloud services that can scale as needed, as well as the containerization of services along with their dependencies has boosted the popularity of this paradigm.

You know that you work in such company if:

- Your organization deploys to production at the end of the sprint (every one or two weeks)
- Sprints are organized so that complex/risky features do not occur in the same sprint
- Developers have self-serve access to test environments
- Rollbacks are performed manually but are relatively fast
- Cloud services are preferred when scalability and resource spikes can affect infrastructure provisioning (compared to static on-prem deployments)

Obviously, releasing every week or every other week is a certain improvement over the dark ages delivery methods. Because releases are happening often, they need to be automated as much as possible. Certain approval steps might be used for the final deployment (i.e. just before it reaches production) but the bulk of the deployment process should be fully automated.

Developers can create test environments on demand and deploy their features in an isolated manner. The test environments should be very close to the production one (but with much less resources assigned). This means that developers are not blocked by waiting for infrastructure to be created. Tickets between developers and operators are only for defects and not for the creation of test environments.

The appearance of microservices as the new deployment unit, along with containers in the form of Docker and schedulers in the form of Kubernetes has further helped with isolation and easy deployment of web services moving away from the “big-bang” releases of the past.

The end-result is that developers are no longer blocked by stalled Pull Requests, lengthy deployment processes and too much bureaucracy with the creation of test environments. Also because the sprint period is much shorter, there isn’t that much pressure on following release dates.

Test engineers are the big winners here as well, because rather than facing a very stressful period (right before the end of the release train), they can instead split their workload during the whole sprint.

**But here is the catch!**

In order to reach this level of operational maturity several things must be in place for developers such as:

- A platform that offers self-service creation of test environments
- A way to see what is deployed on test environments and if it is passing required tests
- A way to clean-up test environments that are no longer used
- A way to easily spin up new project infrastructure when a new microservice is added to an existing application
- A way to manage build nodes in a dynamic environment covering the need of different developer teams (often using different programming languages and tools)
- Dashboards for monitoring build infrastructure, artifact management, cluster status, deployment metrics etc.

In most cases system administrators/operators are the ones who are tasked with all these prerequisites.

This means that any company that wishes to deploy weekly and adopt microservices, will need to continuously invest on operator personnel simply to scale along with the developer teams. The end result is that while on the surface, developers are happy and feel productive, operators often feel like firefighters who must respond to incidents while still trying to create helper tools for the development teams.

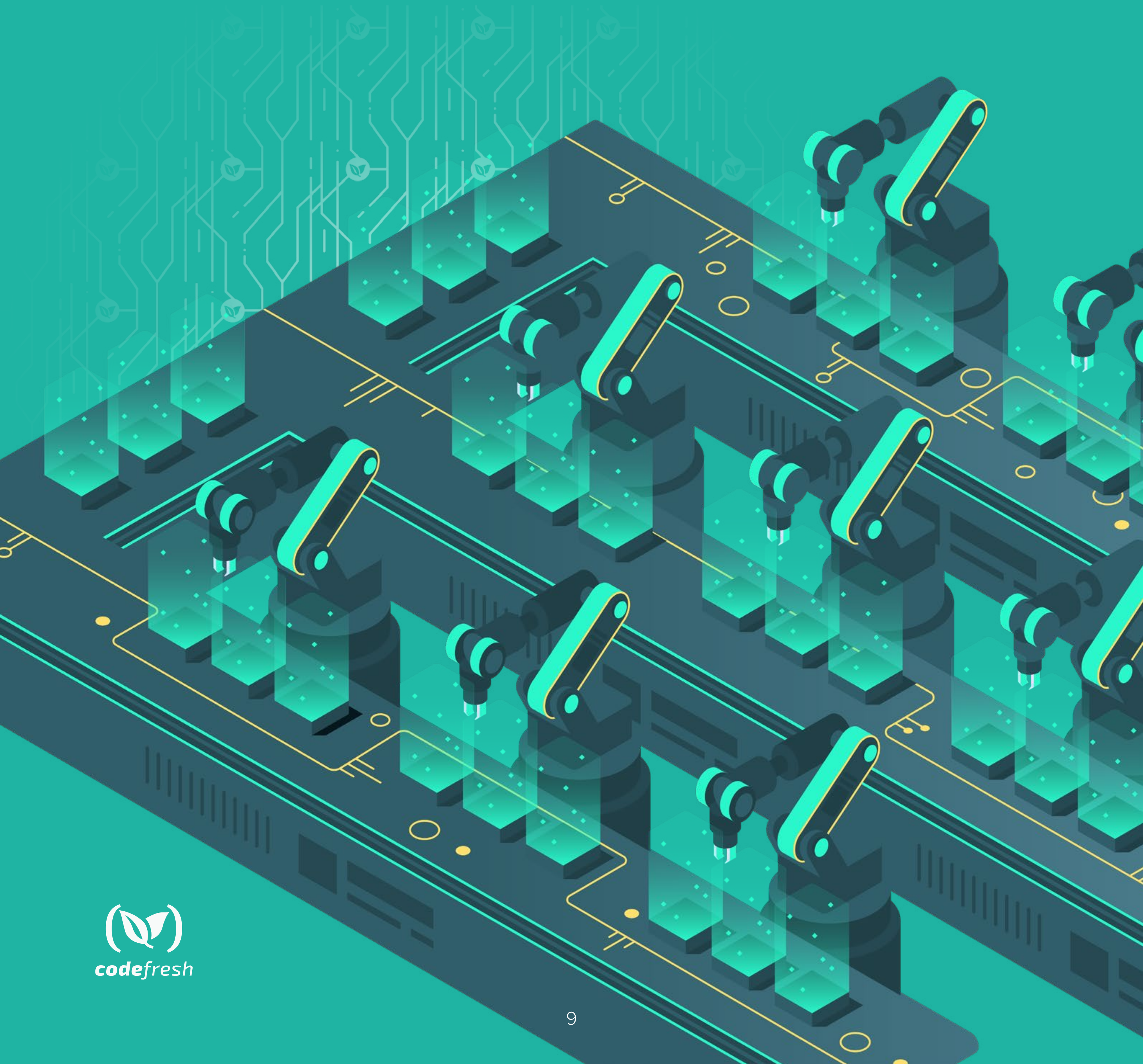
To understand if your organization suffers from this pattern you need to understand some internal metrics (which are not always measured officially) such as:

- How much time people spend managing build nodes (and especially upgrading their tools)
- How much time is spent on creating/managing/removing test environments
- How much time is needed to create a brand new project and all its associated infrastructure (git repository, pipelines, deployment environments, manifests, dbs etc)
- How easy is for developers to bring a completely new tool or programming language into an existing project

In summary, deploying weekly or biweekly allows for developers to be productive, but requires a well oiled operator/system team that needs to be disciplined and automate all repetitive infrastructure scenarios that will allow developers to keep the weekly schedule.

# The Industrial Revolution

Deploying on demand (i.e. right now)



Achieving the weekly release cycle is an important feat on its own. A lot of companies unfortunately think that this is the end goal, without realizing that this is only a stepping stone to true deployment nirvana (i.e. deployment of any feature on demand where it is ready).

Here are the characteristics of this approach

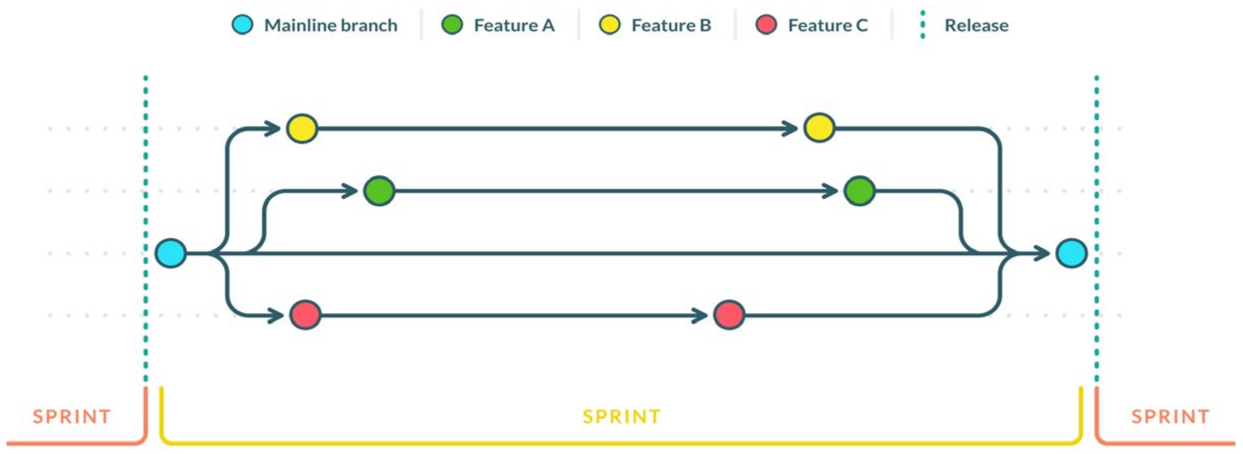
- There are no specific “sprint release” dates. Individual features are deployed in production as they are finished (completely unrelated to any other features)
- Sprints are centered around business needs and do not have any real technical impact (i.e. the end of a sprint is nothing special)
- Failed deployments are detected before production rollouts (i.e. in canaries)
- Rollbacks are happening automatically
- Deployments and releases are not always happening at the same time (usage of feature toggles)
- Developers, testers and system administrators work in tandem for the duration of the sprint

Sending a feature to production as soon as it is ready (on its own) is a very big advantage that cannot be overstated.

First of all this means that features get to production right away giving value to the company as fast as possible. In the case of biweekly releases a feature that needs 3 weeks for implementation, will actually be deployed in 4 weeks (increasing its lead time by 33%).

But the biggest advantage of deploying outside of sprints is the isolation of risky features. Let’s say that a company is releasing bi-weekly and there are 3 “risky” features in the current sprint right now.

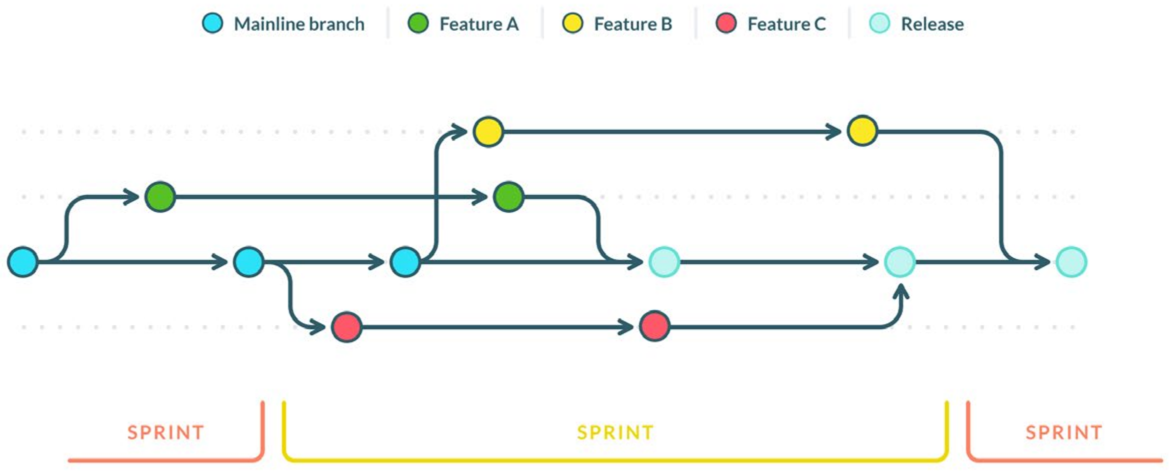
Unfortunately these 3 features are of large size and each one will take the duration of the sprint to be implemented (i.e. 2 weeks). This means that even if your testing strategy is perfect and features A, B, and C are well tested on their own, in practice all of them will land in the mainline branch at the exact same time period.



Any seasoned developer knows that just because features A, B and C work fine on their own, their late integration in the main branch can cause problems due to the way programming dependencies work. For example feature B and C might create significant issues when deployed together.

If you only deploy at the end of sprints, this means that your development/testing team has only a very small time frame (some hours) to evaluate the behavior of all features in production. And by then, it might be too late. For example if there is an issue when all 3 features are sent into production and feature C is reverted, there is no guarantee that features A and B will not have an issue on their own (after the removal of C).

Now let's take the same scenario of 3 features that each take 2 weeks to implement. If your company deploys on demand, you have the flexibility to split the risk of each feature by deploying them further apart from each other.



The end result is the same (all 3 features are in the mainline branch) but the risk is now much lower. First of all each feature is merged incrementally so it is much easier to evaluate its impact on a release. Test engineers have a much easier time to test, because:

- They can test for the duration of the sprint rather than at the end of it
- They only need to evaluate each feature against the mainline, rather than all 3 features at once.

It should also be evident that now you have days between each feature deployment (instead of hours) which means seeing the true impact is easier if you have more metrics on it without any other interference..

In summary releasing in production as frequently as possible actually decreases the risk of failed deployments.

Now you know what is our vision here at Codefresh! When you are ready to leave the dark ages behind and follow us through Renaissance to the industrial revolution (and beyond) we can tell you all about it.

Join us at https://codefresh.io