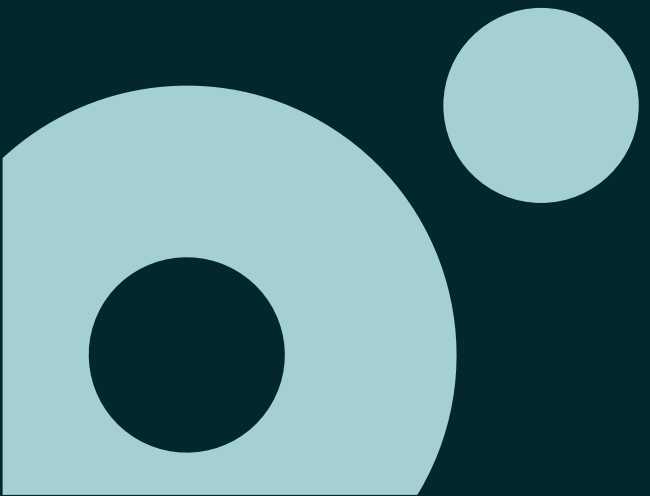
DevOps Fundamentals

Continuous Delivery



Agenda

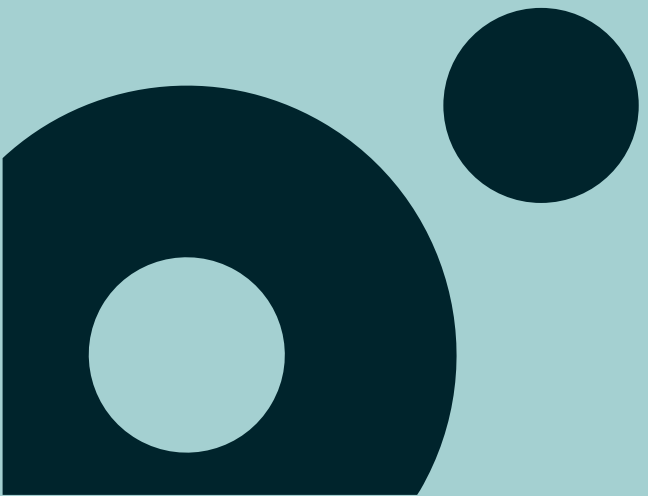
Continuous Delivery

Practices

GitHub Actions

moOngy.

Continuous Delivery



What happens without Continuous Delivery

Manual and Larger Deployments

Higher risk and take a lot longer

Off-Business Hours

Manual steps document

Performed by designated operation team

Harder to rollback if failed

Larger feedback takes longer to implement

How much pain do you suffer to put a new version on production?

What is Continuous Delivery

"**Continuous delivery** is a **software engineering approach** in which teams **produce software in short cycles**, helping to ensure that the software can be **released quickly, reliably, at any time**, following a **repeatable and sustainable process**. It aims at **building, testing, and releasing software with greater speed and frequency**. The approach helps **reduce the cost, time, and risk of delivering changes** by allowing for more incremental updates to applications in production. A straightforward and repeatable deployment process is important for continuous delivery." (Martin Fowler)

Continuous Deployment vs Continuous Delivery

Continuous Deployment

MANUAL

Continuous Delivery

AUTO

What happens with Continuous Delivery

Fast, repeatable, predictable configurable deployments

Lower Risk and higher quality

Early feedback

Faster collaboration, everyone is involved, and anyone can initiate deployments

Adapt and react a lot quickly

Deploy during any business hours

Change delivered without significant delay

Outcomes of Continuous Delivery

According to the 2018 State of DevOps Report, presented by DORA, high performing DevOps companies when compared to low performers achieve:

46x deployment frequency

2555x faster lead time for changes

7x lower change failure rate

2604x faster mean time to recover

Practices

Continuous Delivery Practices

Automated Deployments

One Build Only

Move forward

Buildable and Deployable Trunk

Feature Flags

Shift-Right with Modern Deployments

Practices: Automated Deployments

Deploy process is scripted/automated using automated deployment tool (ex. GitHub Actions, Jenkins, CircleCI, …)

Processes are configurable, repeatable, idempotent and reliable

All artifacts needed for automation (besides CI produced packages) must be stored and maintained using a source control system

The process must be the same for all environments

Automated process must include quality check gates

- Automatic gates: Smoke tests, infra validation, external service check, …
- Manual gates: Human approvals

Practices: One Build Only

Continuous integration process must create a package to be delivered on all needed environments

Between environments you may only change configurations (database connection strings, certificates, services url's, ...)

This approach guarantees that you delivery exactly what you tested and have direct link with your codebase on source control

Practices: Move forward

If the application breaks after a deployment, fix it, commit the fix and redeploy

Rollback must be really a last (last, last) resort

“Definition of Done” must be around of having your work deployed and working as expected, on production environment

Don’t do quick fixing on target environment. Any change must come from your CD process

Having this in mind (as a goal to your time) will automatically bring you better DevOps metrics

Practices: Buildable and Deployable Trunk

Source control as single source of truth (again)

Keep branching strategy simple

Main/trunk branch always buildable and deployable (you need to feel comfortable to start a CI/CD process anytime)

Short-lived branches (small batches)

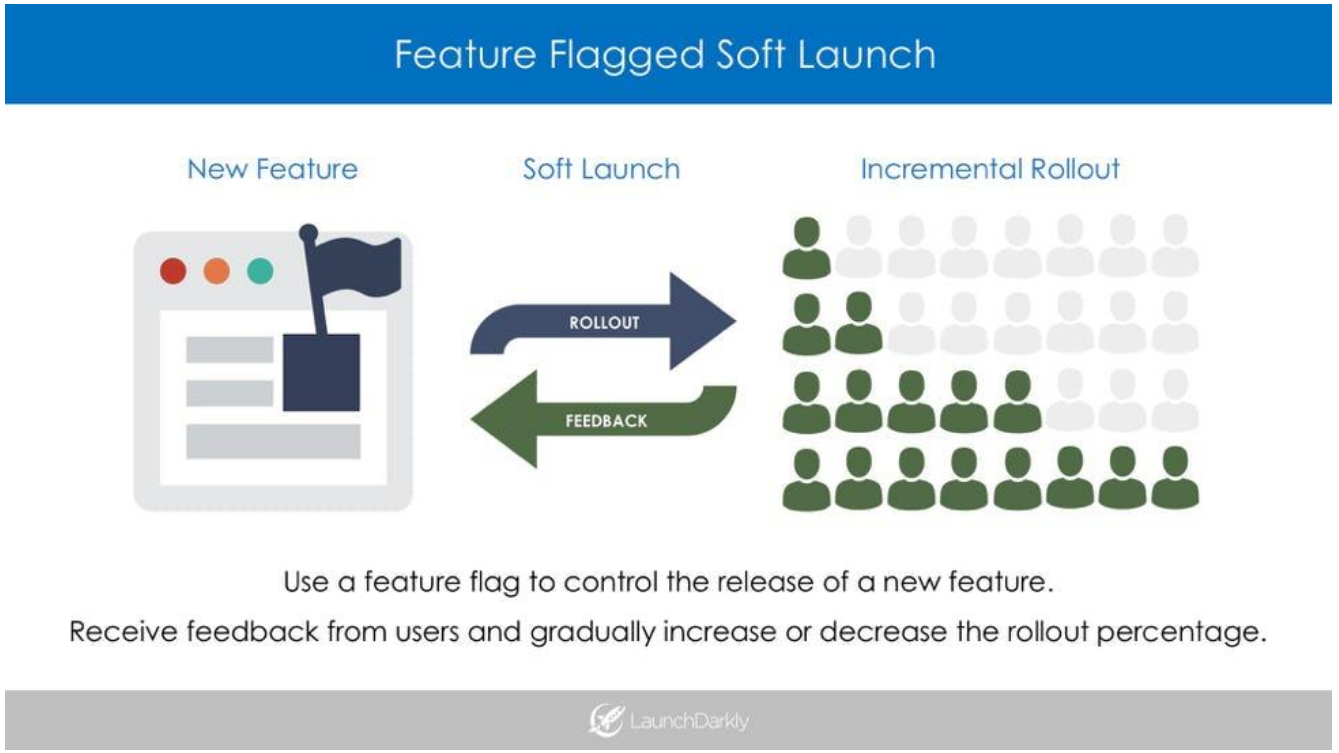
Do merge on trunk as fast as possible

Practices: Feature Flags

A check on your code that makes a feature available or not

Allow to include partial developments without manual intervention (cherry-picking)

Can be a good way to make beta/experimental feature available to a subset of your users



Practices: Shift-Right with Modern Deployments

Traditional approach is to do a “Big Bang” deployment, every user is automatically affected by new version

Before reaching production several environments are used to test your application: quality assurance environment, user acceptance test environment, load/capacity test environment, staging, …

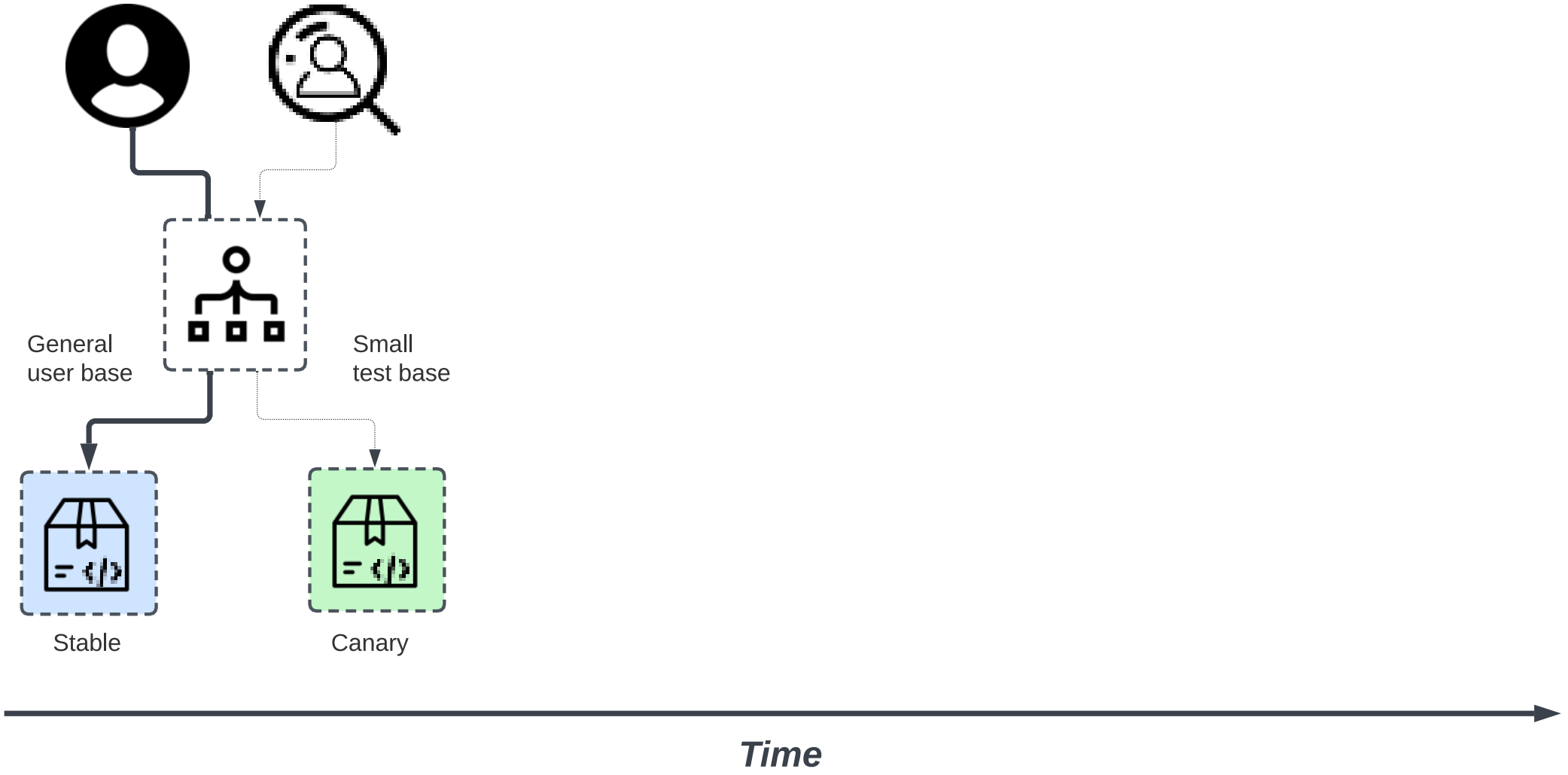
Modern approaches have the goal to reach production as soon as possible

Shift-right on testing doing it on production environment (only environment the same as production ☺)

Different techniques: Canary or Ring Deployments, Blue-green or red-black or A/B deployments

Infra as Code (detailed on next session) takes a crucial role on these techniques

Practices: Canary Deployments



Practices: Canary Deployments

Pros of Canary Deployment

Components can be **updated independently** in real-time with the safety of monitoring for performance consistency in live production before a full-scale rollout.

Least risk for platforms or services handling hundreds of thousands of users each day (gaming, banking, e-commerce, social media, and so on) where a faulty upgrade might result in significant financial losses.

"Real-world" testing on dynamic client traffic is the best possible test and only feasible on live production servers.

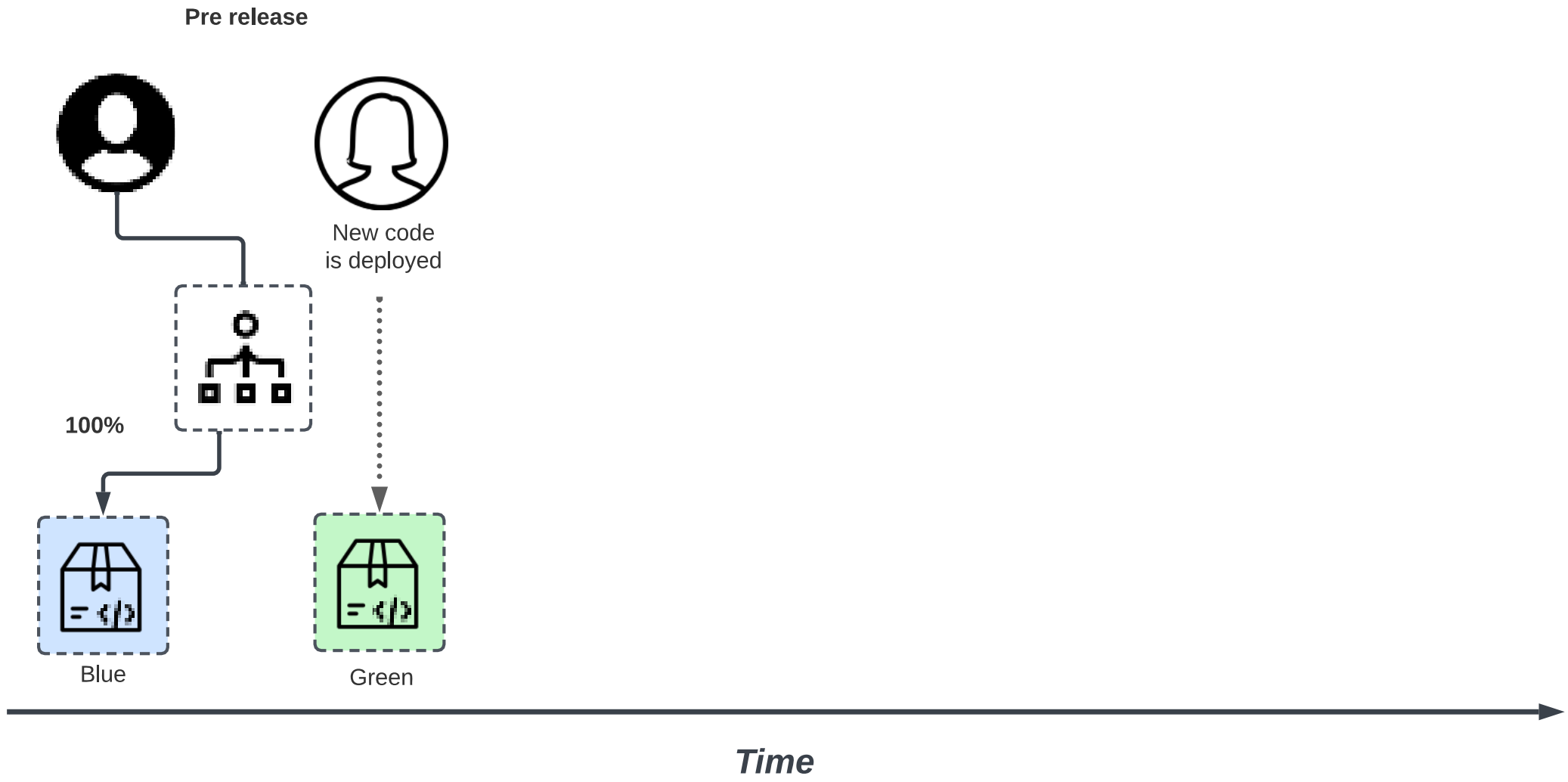
Cons of Canary Deployment

Traditional infrastructure is static and incurs the **expense of maintaining two production environments**.

Real-time analytics and insights are required for rollback decisions.

Database management might be complex since the two environments are supposed to operate simultaneously (even if they aren't receiving any traffic).

Practices: Blue/Green Deployments



Practices: Blue/Green Deployments

Pros of Blue/Green Deployment

Low risk of exposing a defective feature to public traffic because we testing the new codebase in the production-like staging environment.

Rollback is easy because we have a complete set of Production environments.

Deployment is fast and straightforward compared to other deployment strategies..

Cons of Blue/Green Deployment

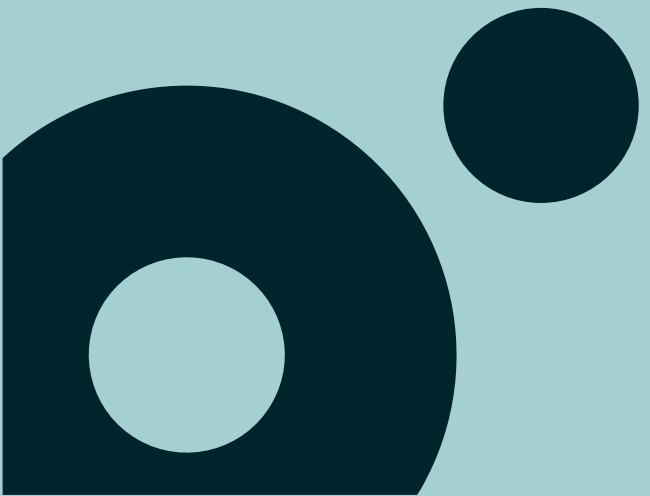
High effort and cost of managing/maintaining two Production environments.

Long-lived transactions in the Blue environment make switching over to Green difficult because you have to wait to close in-flight transactions gracefully.

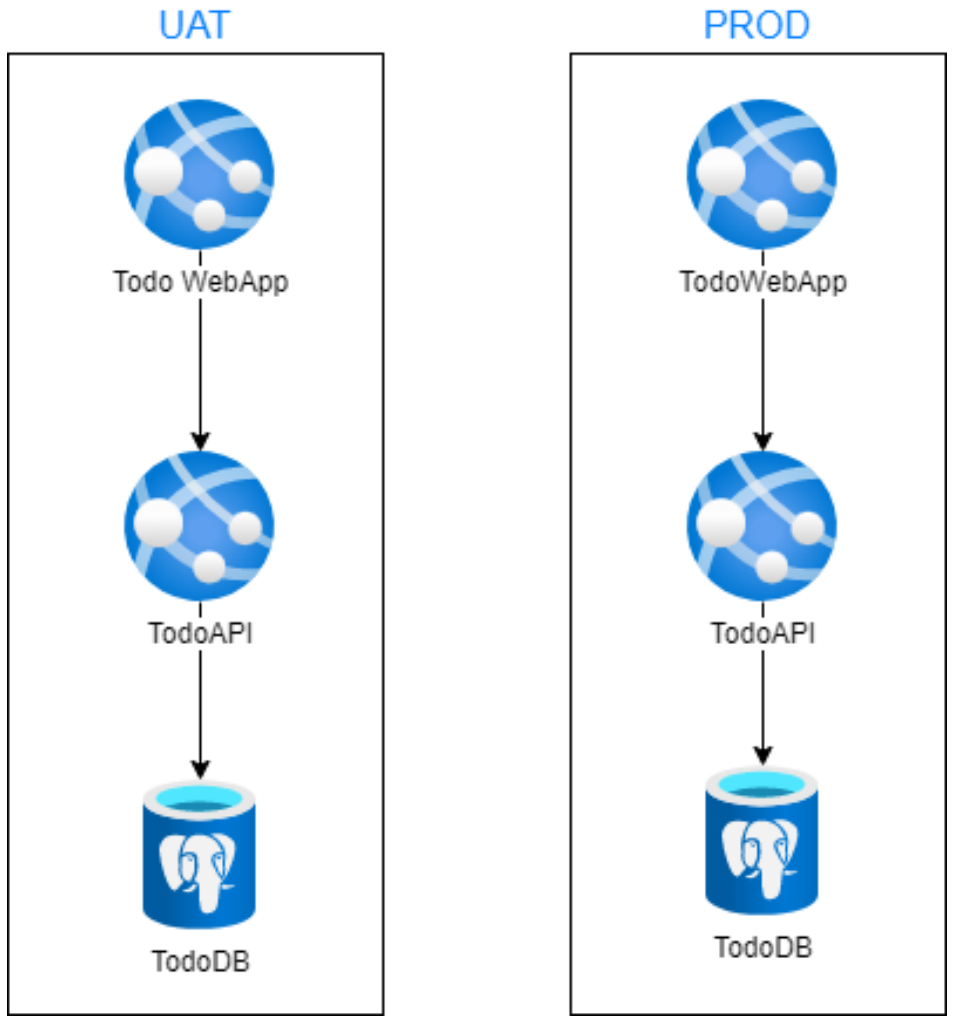
Database management might be complex since the two environments are supposed to operate simultaneously (even if they aren't receiving any traffic).

Hard to test experimental features to a small audience of live users.

GitHub Actions



Final Architecture



moOngy.

Demo: CD on GitHub Actions

moOngy.
Minds on the move

Rua Sousa Martins, nº 10
1050-218 Lisboa | Portugal