DevOps

Comp 47480: Contemporary Software Development

What is DevOps?

Development involves writing software.

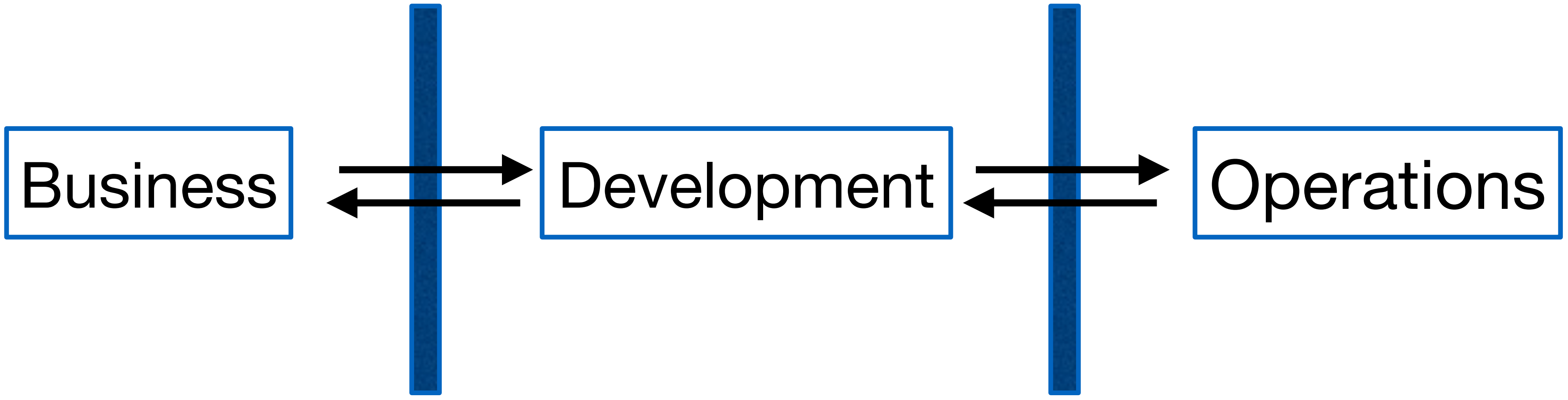
Operations (**Ops**) is the management, monitoring and operation of a system in production.

DevOps is the practice of operations and development engineers working together in the entire software lifecycle, from development through to release.

In this section we look *briefly* at what DevOps is.

Where did DevOps come from?

The traditional view of software development and its context:

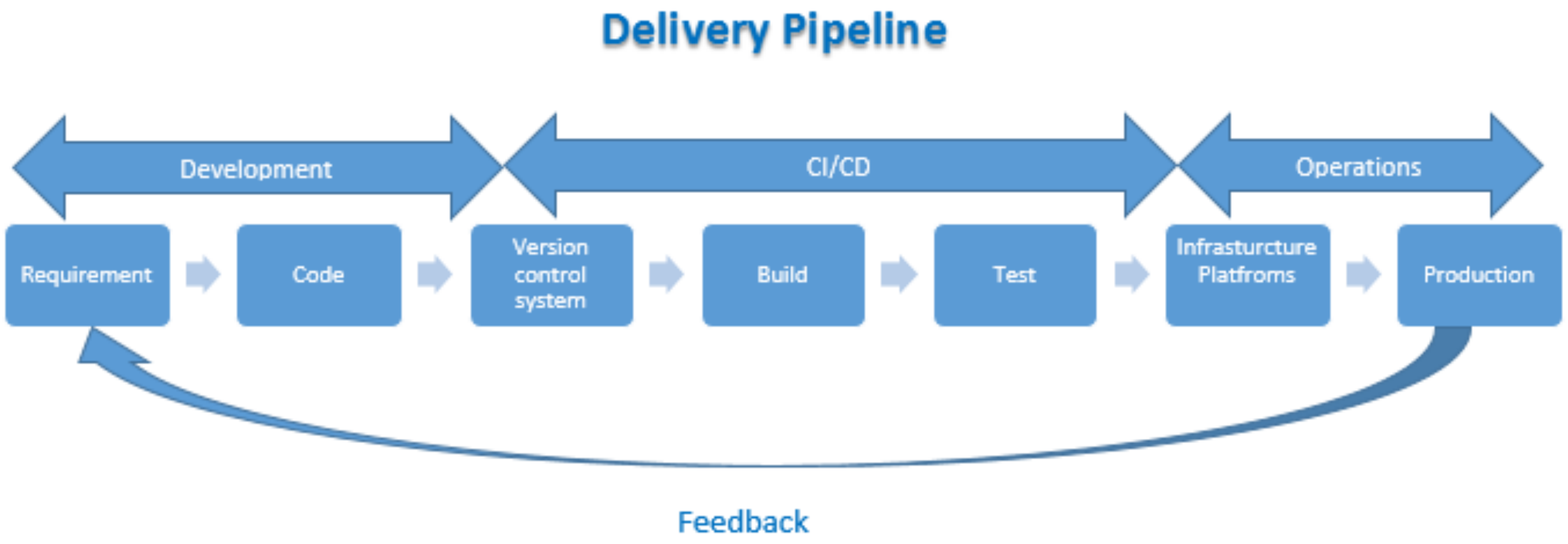


Agile aims to dissolve this wall

DevOps aims to dissolve this wall

DevOps Delivery Pipeline

DevOps is based on a pipeline, e.g.:



There are many versions of this pipeline. The key feature is that, ideally,
the process from commit to production is automated.

DevOps Principles 1

Continuous Integration, Continuous Delivery. Get new releases into production smoothly and quickly.

Microservices architecture. System designed as a number of containers interacting through well-defined APIs, typically using HTTP.

(Not essential for DevOps, but is the most common architecture employed in a DevOps context.)

DevOps Principles 2

Infrastructure as Code. Provisioning and managing servers is performed by code, so version control is possible and the tasks become standardised and repeatable.

Monitoring and Logging. Active monitoring is essential to keeping services running efficiently 24x7.

Automate Everything! Testing, workflows, infrastructure, deployment, monitoring ...

Integration

Integration is where we bring together software components that have been developed separately.

This seemingly innocuous activity has led to huge problems in practice ("integration hell").

Essentially we are testing if multiple developers had **precisely** the same understanding of the contract between them (answer: no).

One of the most famous integration problems is described on the next slide.

NASA's Mars Climate Orbiter

The Mars Climate Orbiter was a robotic space probe launched by NASA in 1998.

On approach, it got too close to the planet and was destroyed by the atmosphere.

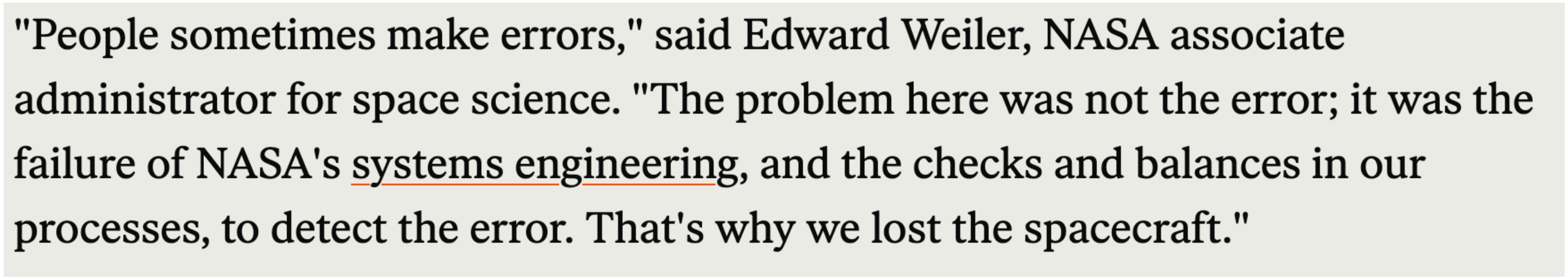
Why?

> The NASA-developed on-board software used the standard *SI* system.
>
> The ground software was developed by Lockheed-Martin used the *United States customary units* system.

Impulse as measured in *pounds-force seconds* and *newton seconds* vary by a factor of ~4, so the orbiter was doomed.

Aside: Blame Culture

The spec clearly stated that SI units (newton seconds) were to be used. NASA used them. Lockheed-Martin didn't. Who did NASA blame?


"People sometimes make errors," said Edward Weiler, NASA associate administrator for space science. "The problem here was not the error; it was the failure of NASA's systems engineering, and the checks and balances in our processes, to detect the error. That's why we lost the spacecraft."

This attitude is evident in all healthy software development environments. In a retrospective, the issue is never "who is to blame" but "how can we do it better?"

The root cause of the loss of the orbiter was the lack of integration testing.

Continuous Integration

The traditional practice is "big bang" integration.

Each feature under development is committed to a different branch ("feature branch") in the (Git) repository.

When the time comes for a new release, a protracted, painful integration phase is required to get the different features to work together.

DevOps demands instead *continuous integration*, where integration happens frequently, perhaps multiple times per day.

Repository Organisation

A large company may store its software in multiple Git repositories.

This may lead to inconsistencies between repos, and make the software harder to refactor, e.g. a method renaming may require multiple, coordinated commits.

"Extreme" DevOps suggests a single **monorepo**, with just **one single branch** (main).

So every commit, by any developer, is to the head of main.

Sounds insane, but this is the practice in several of the best-known software companies.

Automation

Automation is key. CI is very challenging if human intervention is required.

Automated builds.

Automated unit testing.

Automated integration testing.

If e.g., performance is a concern, testing this should also be automated.

CI Challenges

CI may be hard to achieve in certain contexts.

End-user tests: This type of test may be script-based and not automatable.

External Mandates: Software developed in certain domains, e.g. medical, finance may be required to go through certain non-automatable checks.

Open source: Developers are typically less reliable and not available on a daily basis.

Code Reviews

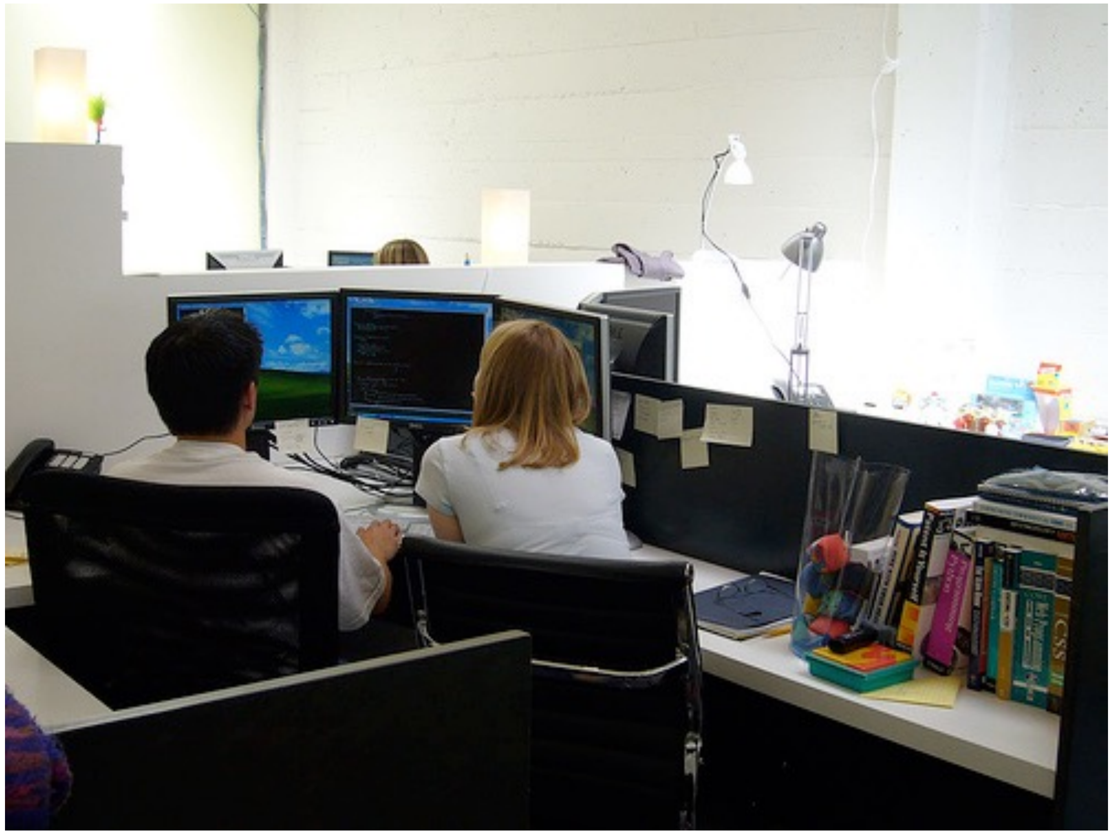
How can code reviews fit into a a CI process? They require another developer to review your code.

Extreme Programming proposed **pair programming** as a solution (see next slides).

In a CI context, code reviews may be replaced with automated tools like SonarQube that try to measure the quality of the committed software.

Pair Programming

Pair programming: Code is written by two people; one writes the code, the other considers the effect of the code on the larger system.



One of the more controversial XP practices.

Pair programmers are peers!

Continuous Deployment

Continuous Deployment takes CI a step further: once a developer makes a commit, the process to deployment to the customer is fully automated.

This is natural extension to CI.

It has many benefits, but also some risks:

- Possibility of deploying buggy software
- Developer stress (every commit may lead to a "production system down" event)

Feature Flags

What if you commit code that only partially implements a feature. How do you stop this going to production?

You can hide the code behind a **feature flag**, e.g.

```
featureX = false;
...
if (featureX)
   "here is my incomplete code for X"
...
if (featureX)
   "more incomplete code for X"
```

`featureX` is set to `true` for development

In a real context, feature flags will be managed centrally.

Multiple feature flags can greatly increase code complexity.

Other uses for feature flags

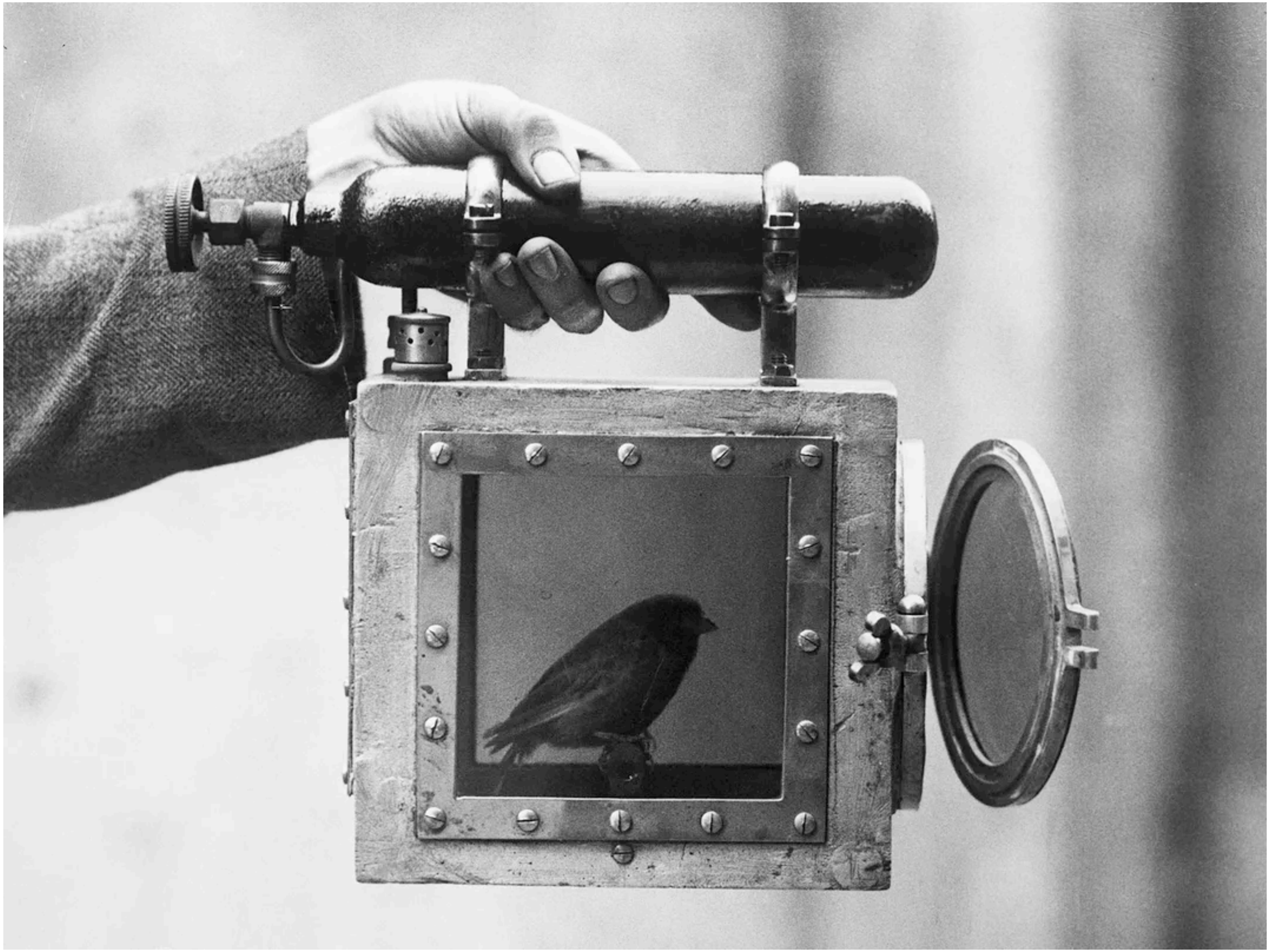
Canary release: release a new version to a small number of users to see if it works.

Mimics the use of canaries to detect poisonous gases in coalmines.

A/B Testing: release a new version to a small number of users to see the new feature performs better than the previous one.

In A/B testing, the new version may provide a new feature, or may simply have a button in a different place in the UI.

In case you feel bad for the canaries...



Typical DevOps Tooling

Source Code Management: Git, GitHub, Bitbucket.

Build: Gradle, Maven, Ant

CI/CD: Jenkins, Bamboo

Containerisation: Docker (for deploying microservices), Kubernetes (for managing containers at scale)

Infrastructure as code (configuration management): Puppet, Ansible, Chef

Monitoring: Nagios, Prometheus, Zabbix