UC
UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

Fabian Gilson

Software Engineering II

SENG301 - Continuous delivery

Lecture 6 - Semester 1, 2020

University of Canterbury | Te Whare Wānanga o Waitaha
Computer Science and Software Engineering

The Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform.

– Ada Lovelace

An Account of the Analytical Engine – 1842

The deployment problem


?
QUIZ TIME!

Deployment time is stressful

Release dates are often source of anxiety

- does everything **compiles and/or run** on the target?
- did we forget some **dependencies**?
- have we moved all needed **resources**?
- is the **database schema** correctly updated?

If many of those tasks are executed by hand, many things can go wrong

- when new (versions of) **dependencies** are needed
- when the **release procedure** has been amended
- when **manual fixes** are performed by experts
- when you deploy **for the first time** on production

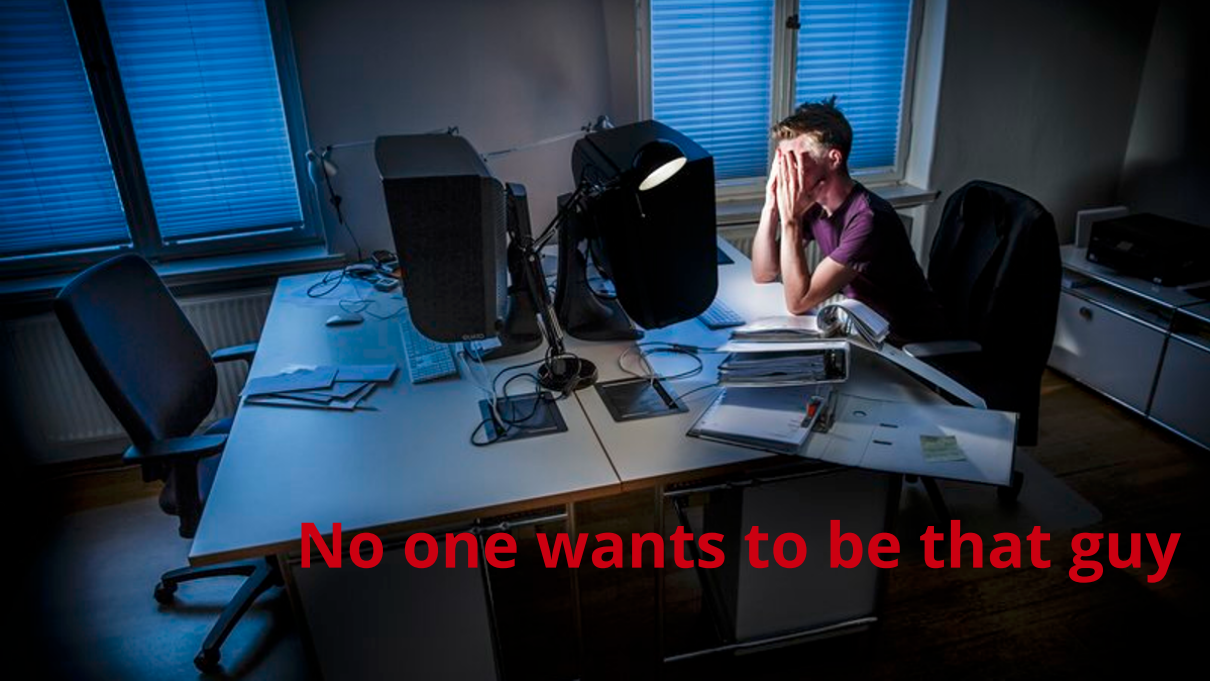
Antipatterns of deployment

Doing it manually

- requires extensive and verbose **documentation**
- rely on **manual testing** during the deployment night or week-end
- calling the development team to **figure out what is going wrong**
- updating the **deployment sheet** since the production environment is different
- deal with manual ***"tweakings"***

Waiting the very end to deploy a release

- the software is **tested** on its new platform **for the first time**
- many organisations are separating developers from IT system engineers
- the bigger the system is, the **more uncertainties** re. side effects exist


No one wants to be that guy


Neither in that situation

Reduce the stress by getting used to it

If the release frequency is increased

- either you suffer more ...
- ... or you find a way to **make it easy**

This is where continuous integration has virtuous side-effects

- you have **automated builds** with testing
- you have enforced **self-contained sources**
- you have written (partial) **deployment scripts**
- you have done it many times for **smaller *deltas***

Continuous deployment gives the opportunity to **test deployment processes**

Benefits of automated and continuous delivery

Empowering teams

→ test and deploy **the build you want**

Reducing errors

→ avoid **unversioned configurations**, which are error-prone

Lowering stress

→ **frequent and smaller changes**, accompanied by rollback process

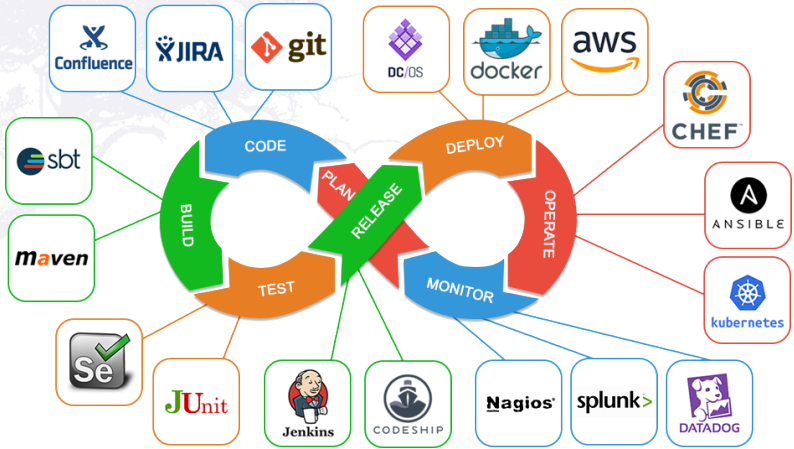
Deploy where you want

→ get the version you want **where you want when you want**

But decide when you deliver

→ this is different that continuous deployment where **CI triggers the deployment**

DevOps or putting it all together



https://medium.com/devopslinks/devops-without-devops-tools-3f1deb451b1c

Deployment strategies

Adequate preparation

Upfront job is needed

- model the **target environment** appropriately
- prepare the **deployment infrastructure** carefully
- master **servers' configuration** (avoid obscure GUI-based configs)
- define the **deployment strategy** collaboratively

Automate as much as possible the process

- many third-party libraries or servers use **textual config files**
- use build chains and **dependency management systems** (*e.g* `maven`, `gradle`, `sbt`)
- preparing **virtual environments** is also a solution

Do not forget a **disaster recovery** process in case of last minute fail

Configuration management

Keep everything under version control

- your **sources**, obviously
- your **configuration** files, including DNS zone files or firewall settings

  but don't put passwords or other **sensitive data** under version control

Use version control in a smart way

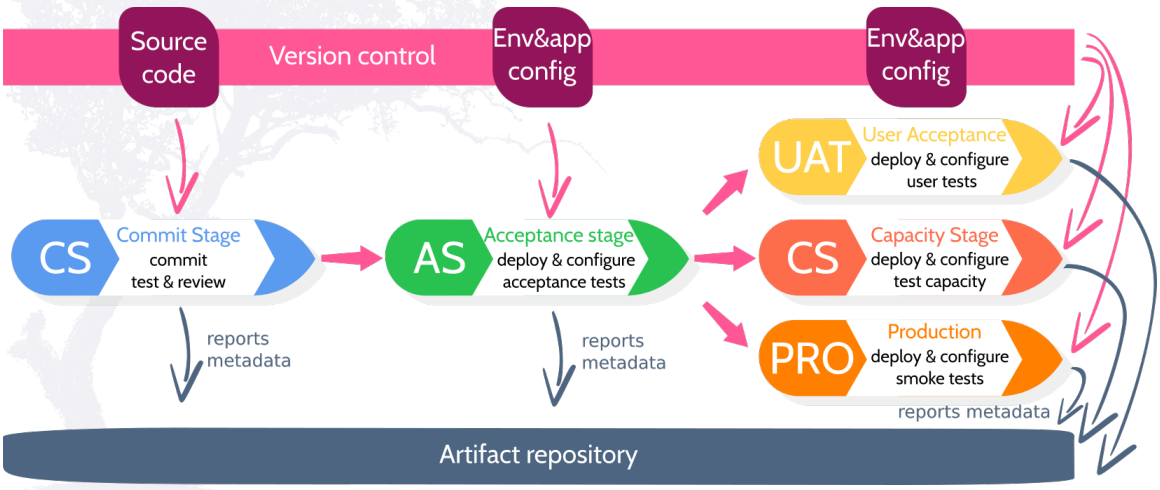
- push **regularly** to the main (develop) branch
- use **meaningful** commit **messages**

Think about dependencies

- **automated** retrieval **dependencies** is handy

  but avoid having to *"download the Internet"*

Deployment pipeline



Deployment pipeline practices

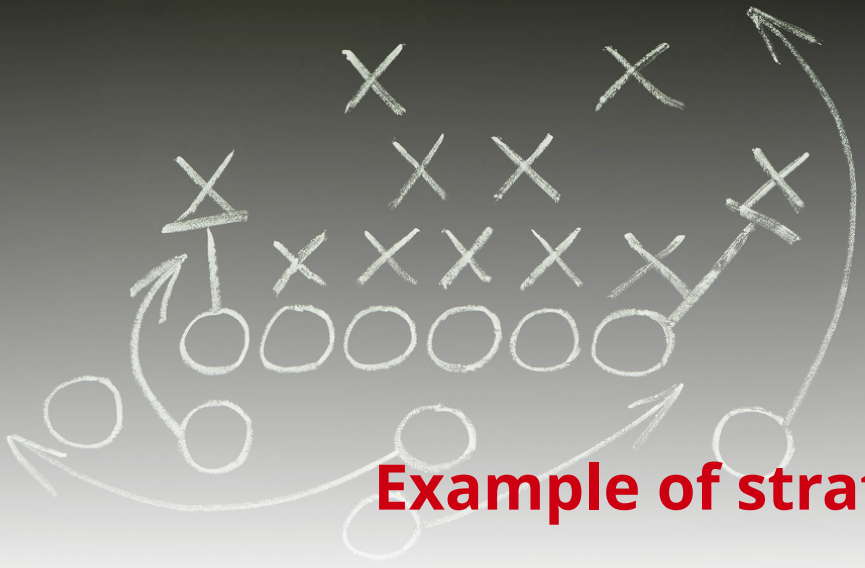
Build only once
→ even a slight difference in building environment may cause problems

Deploy the same way everywhere
→ following the same deployment procedure for actual testing on multiple environments

Smoke-test your deployments
→ also script the automated start-up of the system with a few simple requests

Deploy into a **copy of production**
→ mimic network and firewalls, operating system and application stack

Propagate changes into the whole pipeline as they appear
→ as changes trigger the pipeline, the latest available version is built or tested



Example of strategies

Creating deployment strategies

To prepare your strategy, you need

- **all parties** in charge of the various environments
- a deployment **pipeline plan** and a **configuration** management strategy
- a list of **environment variables** and a process to move from one to the other
- a list of **monitoring** requirements and solutions
- to discuss when **third-party systems** are part of the testing
- to discuss the **disaster recovery** plan
- to agree on a *Service Level Agreement* (**SLA**) and on **support**
- an **archiving** strategy of outdated data (for auditing)

The strategy also encompass **data migration**, **upgrades**, **patches** and **bug fixes**

Zero downtime releases

Negate as much as possible downtime

- downtime is **frustrating** for users
- must be minimized or during **off-peak time**

Many *hotplug* facilities exist

- decouple the whole system in pieces migrated **one after another**
- offer **roll-back** to previous version if necessary

Zero-downtime is not always possible

- web-based systems may be **rerouted**, even temporarily
- but **data migrations** often need some time

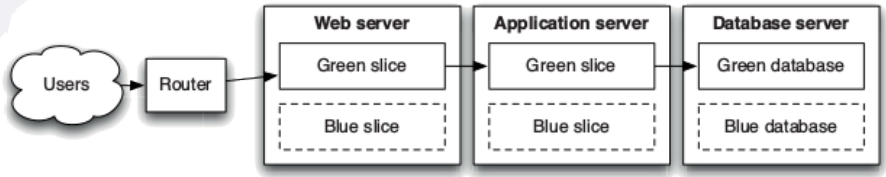
Blue-green deployment

In distributed systems, *e.g* web applications

- **rerouting** is rather easy
- possibility to let **multiple versions** available at the same time

Two **identical** environments

→ **simple rollback** of routing rules, if necessary



Humble and Farley, *Continuous Delivery*, 2010

Canary deployment

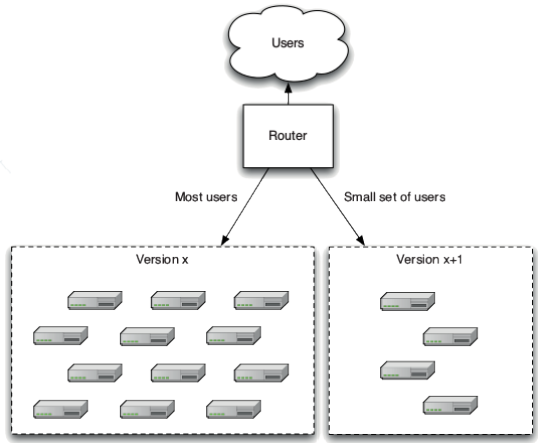
Real systems are not always easily *cloneable*

- real-world constraints are **not fully testable** upfront
- **scalability** and **response-time** are crucial features

As *blue-green*, needs a router

- deploy a **subset of servers** with new version
- use a (well identified) **subset of users**
- gradually **increase the charge**

→ developers collect **feedback** faster



Humble and Farley, *Continuous Delivery*, 2010


七国山行
稲荷前
がんばって　ください

Third round

Pitfalls of code reviews

https://www.michaelagreiler.com/code-review-pitfalls-slow-down/

Remember to keep notes

- what is it about? was it **relevant**? what are your **takeaways**?
- how does it relate to this course and SE in general?

All readings are on Learn in the **Resources** section (with optional ones)


TO BE
CONTINUED...

Next episode

Next week is testing week!

We'll talk about tests and resilience engineering

... aka what do we do when we got hacked

... come next lecture