



CHALMERS
UNIVERSITY OF TECHNOLOGY



UNIVERSITY OF GOTHENBURG

Conducting Quality Assurance in Production

Dr. Philipp Leitner



philipp.leitner@chalmers.se



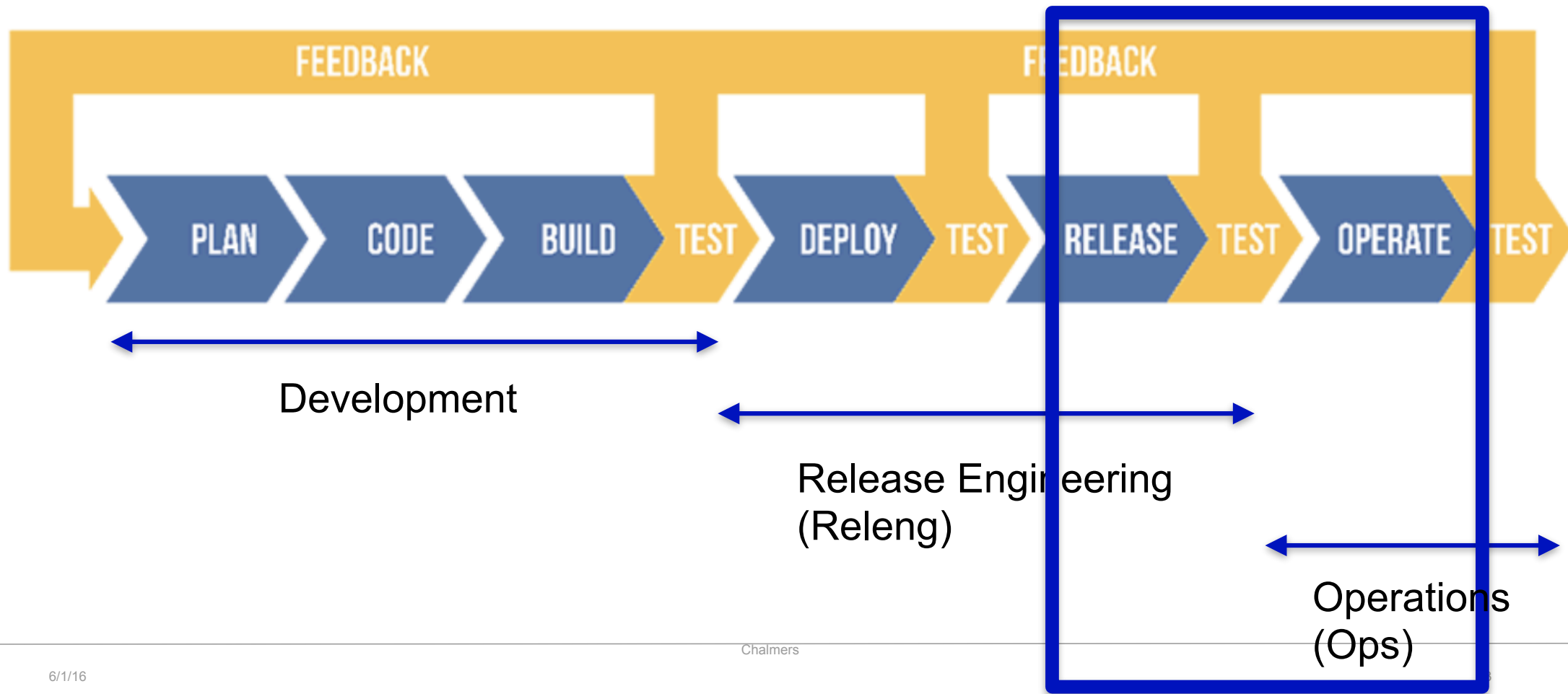
@xLeitix

LECTURE 9

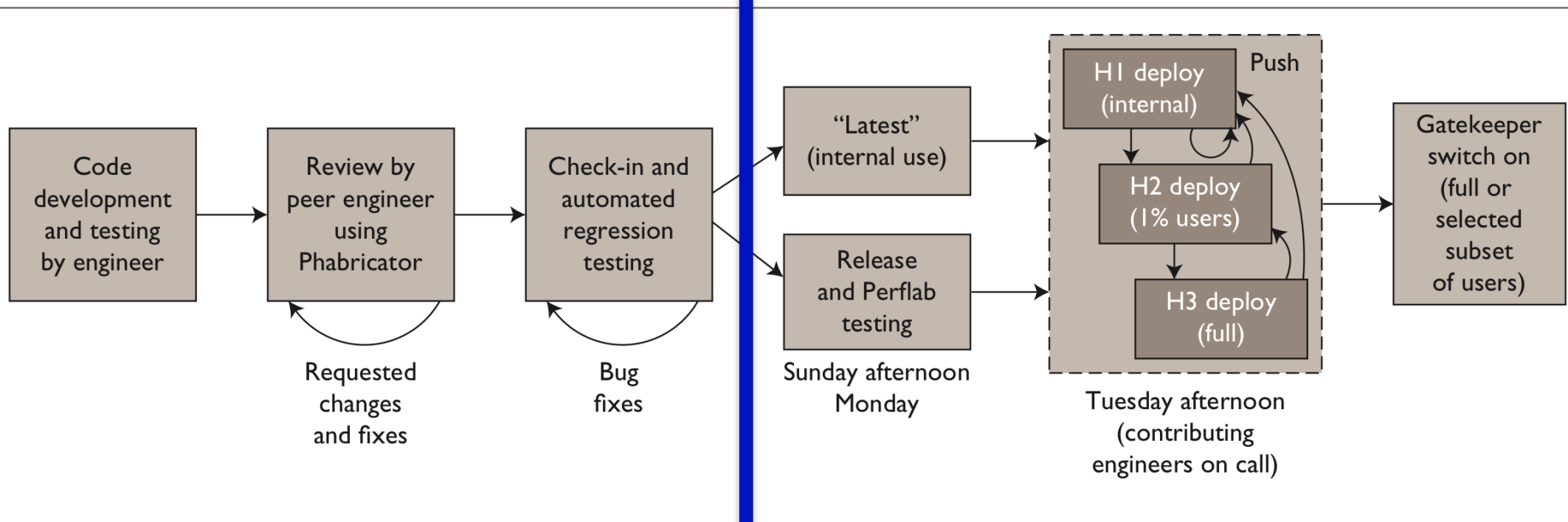
Covers ...

Production Experimentation

SaaS Application Lifecycle



A Real-Life Example (from Facebook, 2013)



Source:
Feitelson, Frachtenberg, Kent. Development and Deployment at Facebook. IEEE Software, 2013.

Environments

Most production SaaS systems use multiple different **environments**

Basically different installations of the same system, used for different use cases

Installations in different environments may be very similar to each other, or very different (depending on use case)

Common Examples (terminology may vary)

Production (Prod)

Used by end users

Pre-Production (Preprod, Preview)

Prod-like environment used for final sanity checks, sometimes accessible by selected partners

Perf Testing

Focus on simulating high scale / high usage

Staging (sometimes: testing)

Integration test environment, often smaller-scale than prod and preprod

Local

Local developer machines, used during development


.... and so on

Environments in GitLab


Multiple environments can be created easily in GitLab
(Prod exists by default if you use a CD pipeline)

Go to *Deployments* -> *Environments*

▼ production

#131 by 

production #160987

 master → 0223d112

3 weeks ago

Added example Docke...

Open


Stop

⋮

100%

Complete

Instance (1) ?



Perf Testing

No deployments yet

Stop

⋮

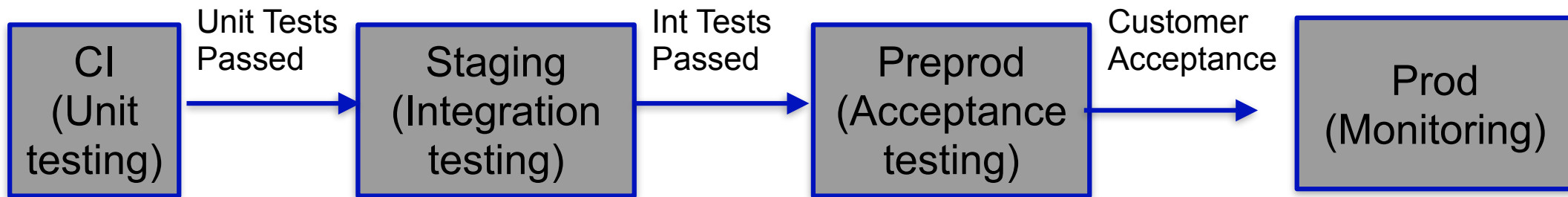
Basic Quality Assurance using Environments

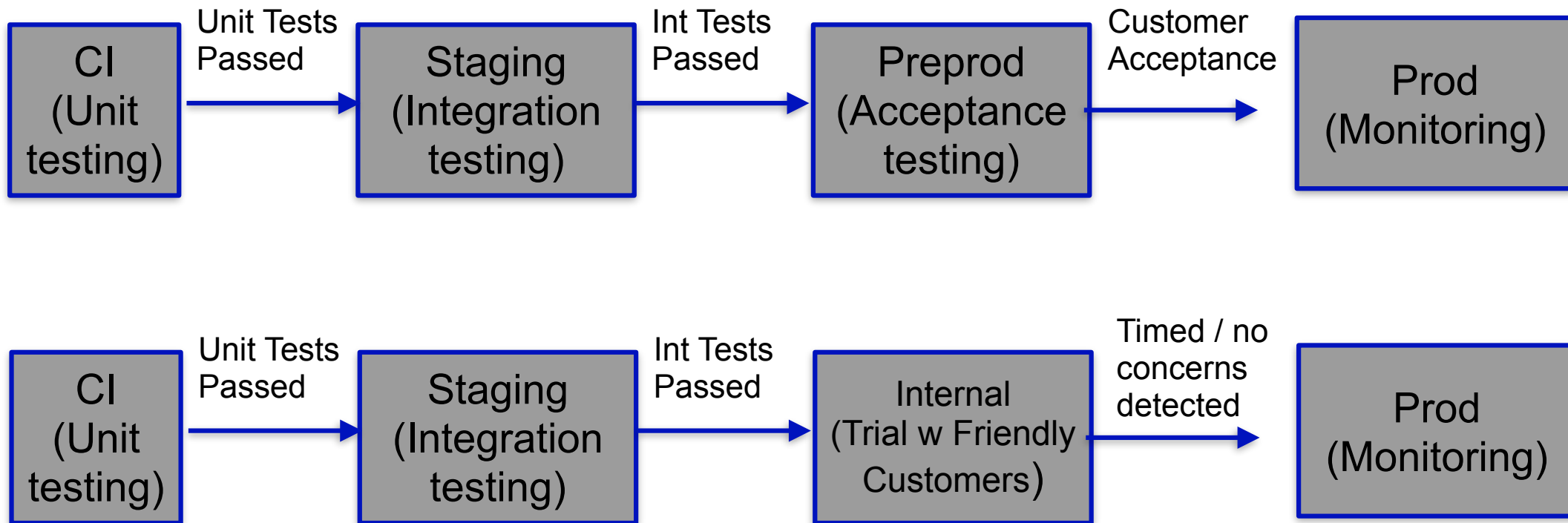
Environments are inherently a (basic) QA mechanism

Use CD pipeline to move changes across environments (until prod)

Every env has specific tests and criteria for progression

Progression can be **automated**, **timed**, or **manual** (even in CD!)





Experimenting in Production

In addition to this basic QA activities, advanced Web companies use environments (especially preprod and prod) as the basis for **continuous experimentation**

Continuous experimentation is a **data-driven, rigorous** approach to quality assurance (or, in some cases requirements engineering) using **real users** and **real traffic**.

Essentially:

- Realistically reproducing a large, scaled system is impossible

- The only way to really test is to deploy and see what happens

- But: have mechanisms in place to minimise impact on end users if things go wrong

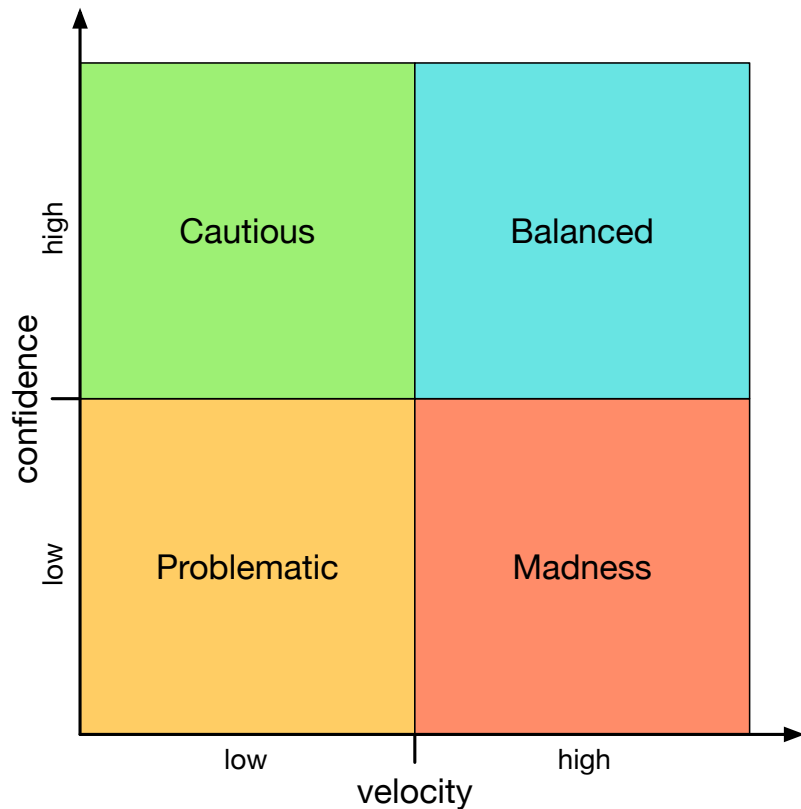


Image Credit: Facebook

MOVE FAST WITH STABLE INFRA



Image Credit: Facebook



Gerald Schermann, Jürgen Cito, Philipp Leitner, Harald C. Gall (2016). **Towards Quality Gates in Continuous Delivery and Deployment.** In Proceedings of the 24TH IEEE International Conference on Program Comprehension (ICPC), Best Short Paper Award.

Types of Experiments

Blue/Green Deployments

Canary Releases (gradual rollouts)

Dark Launches

A/B Testing

Chaos Engineering

(We'll go into detail what these are in a minute)

“Hypothesis-Driven Development”

<https://barryoreilly.com/explore/blog/how-to-implement-hypothesis-driven-development/>

Hypothesis-driven development

We believe *<this capability>*
Will result in *<this outcome>*
We will have confidence to
proceed when
<we see a measurable signal>

@barryoreilly, <http://barryoreilly.com/2013/10/21/how-to-implement-hypothesis-driven-development/>

(+ guardrails, oftentimes)

“Listen to your customers, not the opinion of the highest-paid person in the room.”

Even more:
Listen to what your customers do,
not what they say!

Ron Kohavi, Randal M. Henne, and Dan Sommerfield. 2007. **Practical guide to controlled experiments on the web: listen to your customers not to the hippo**. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '07)*. ACM, New York, NY, USA, 959-967.

Basic Tenets of Continuous Experimentation Techniques

Continuous experimentation techniques are all fairly unique, but they share some common characteristics:

- Involve **real users** and/or the **production environment**
- Are based on **statistical testing** and data analysis
- Are enabled by **scale**
- Require a **mature** process, technology, and product

Two broad classes of continuous experimentation techniques:

(a) Regression-Driven Experiments

Main goal is to find problems in a change
Quality Assurance Techniques

(b) Business-Driven Experiments

Main goal is to evaluate the business value of a change
Requirements Engineering / Validation Techniques

	Regression	Business
Goals	Identify regressions (technical problems)	Validate business value of change
Experiment Duration	Days	Weeks
Metrics	System and app metrics	Business metrics
Techniques	Canaries, dark launches, chaos experiments	A/B testing

Conducting Experiments

Two primary strategies:

(1) Traffic splitting

(2) Feature toggles

(Applies to everything exception chaos experiments, they work a little differently ...)

Traffic Splitting

Customers get (randomly) routed to either the production or canary environment
Experiment orchestration is part of the load balancing / traffic routing infrastructure

Advantages:

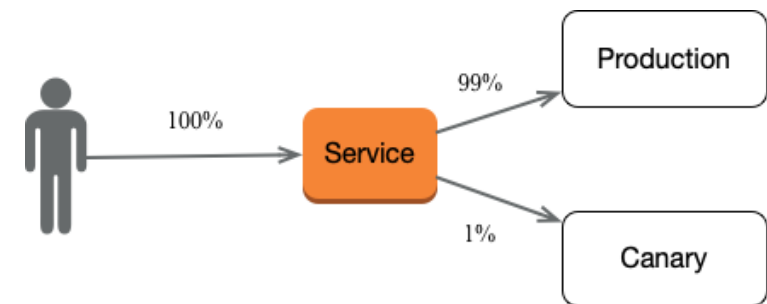
- No changes on code level required

- Easy to stop experiment

Disadvantages:

- Experiments are coarse-grained (service level)

- Most useful in combination with microservices arch



Feature Toggling

Customers get feature toggle on or off randomly

Experiment orchestration is part of the configuration management strategy

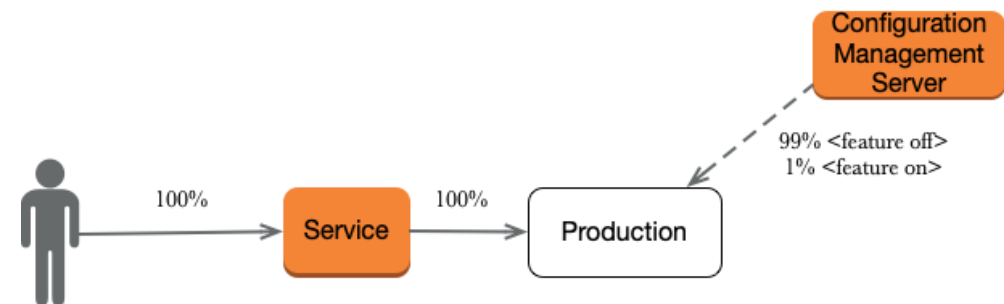
Advantages:

- Fine-grained experiments possible

- Possible for any type of system

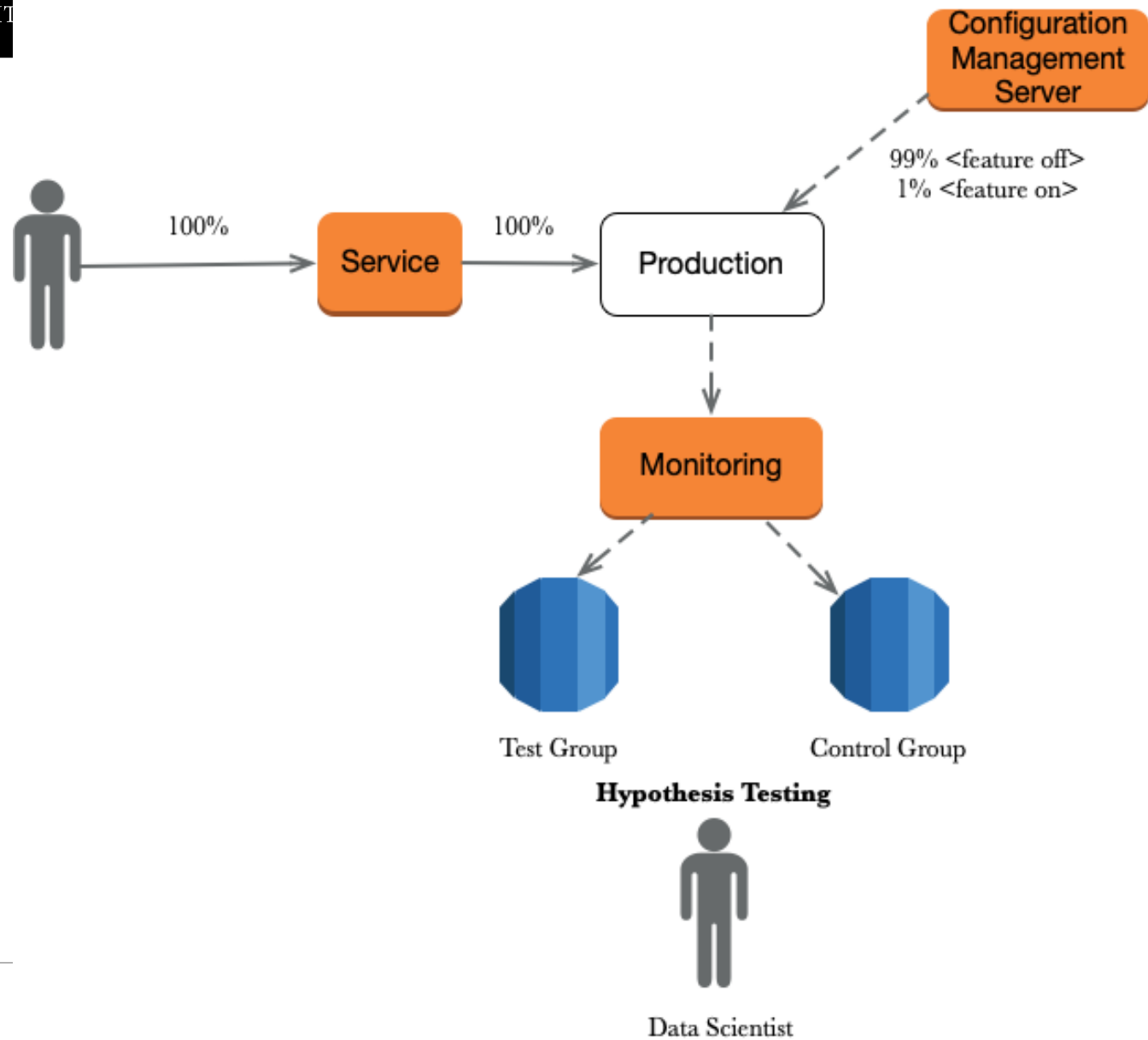
Disadvantages:

- Requires feature toggles in place





Analysing Results



Types of Experiments

Blue/Green Deployments

Canary Releases (gradual rollouts)

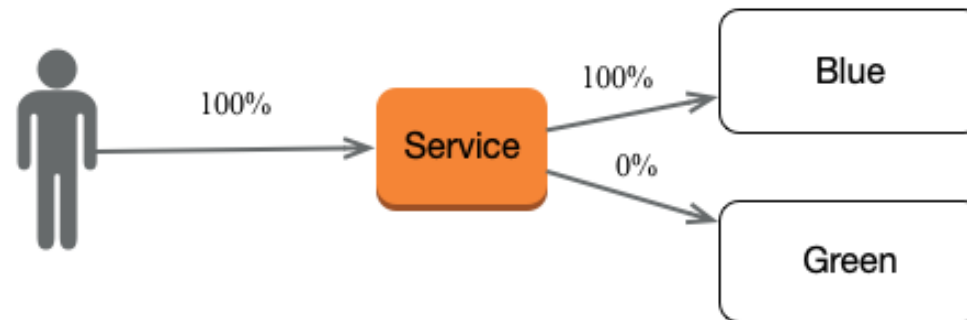
Dark Launches

A/B Testing

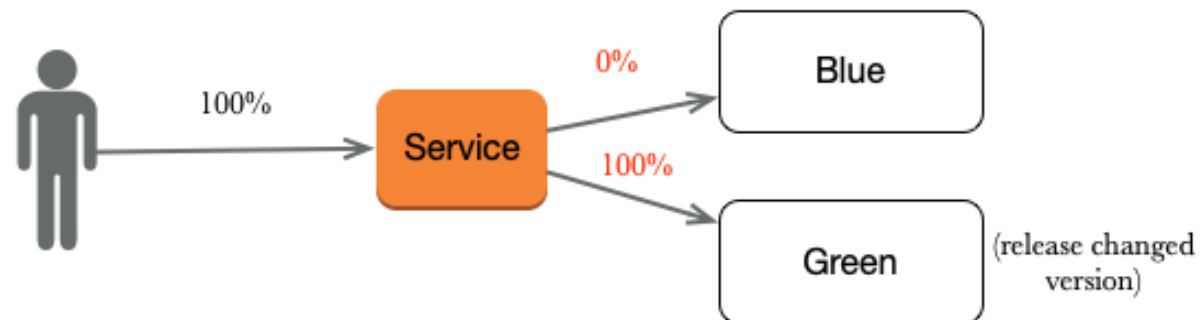
Chaos Engineering

Blue/Green Deployment

Before Experiment



During Experiment



Blue/Green Deployment

Advantages:

- Very easy to do

- Allows you to have a functioning “backup” version

Disadvantages:

- Only works well for software with clear release cycles (not really for CD applications)

- Requires to maintain two production environments

Types of Experiments

Blue/Green Deployments

Canary Releases (gradual rollouts)

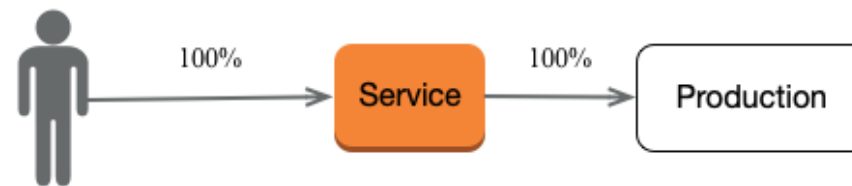
Dark Launches

A/B Testing

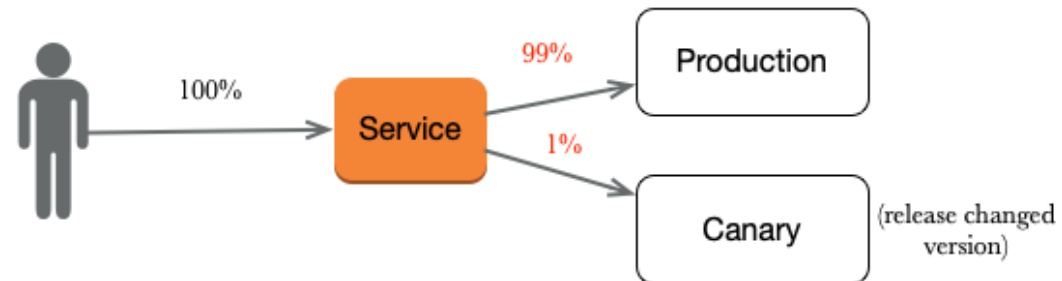
Chaos Engineering

Canary Releases

Before Experiment

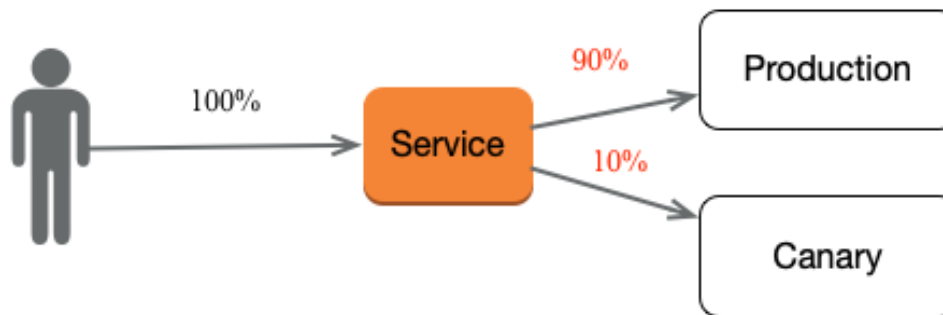


During Experiment - Step 1



Gradual Rollouts

During Experiment - Step 2



...

After Experiment



Canary Releases

Advantages:

- Regressions only impact a small number of users

- Possible to configure on a fine-grained level how conservative / fast you want to release

Disadvantages:

- May take weeks for a change to get fully released

Types of Experiments

Blue/Green Deployments

Canary Releases (gradual rollouts)

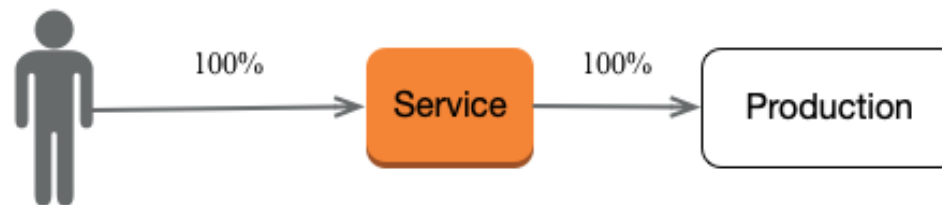
Dark Launches

A/B Testing

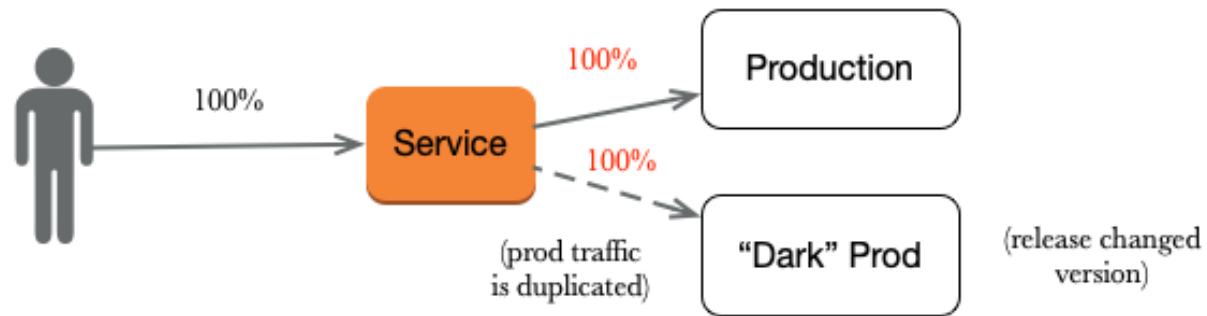
Chaos Engineering

Dark Launches

Before Experiment



During Experiment



Dark Launch

In a dark launch, a copy of prod is produced which can be used to simulate the impact of a (large, important) change

Disadvantage:

High-effort

Expensive

(You do this mostly if you have a high-risk change that cannot be tested through a canary release)

Types of Experiments

Blue/Green Deployments

Canary Releases (gradual rollouts)

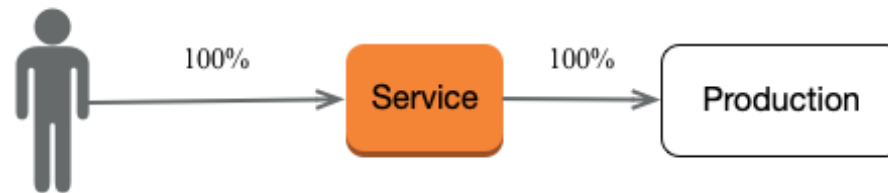
Dark Launches

A/B Testing

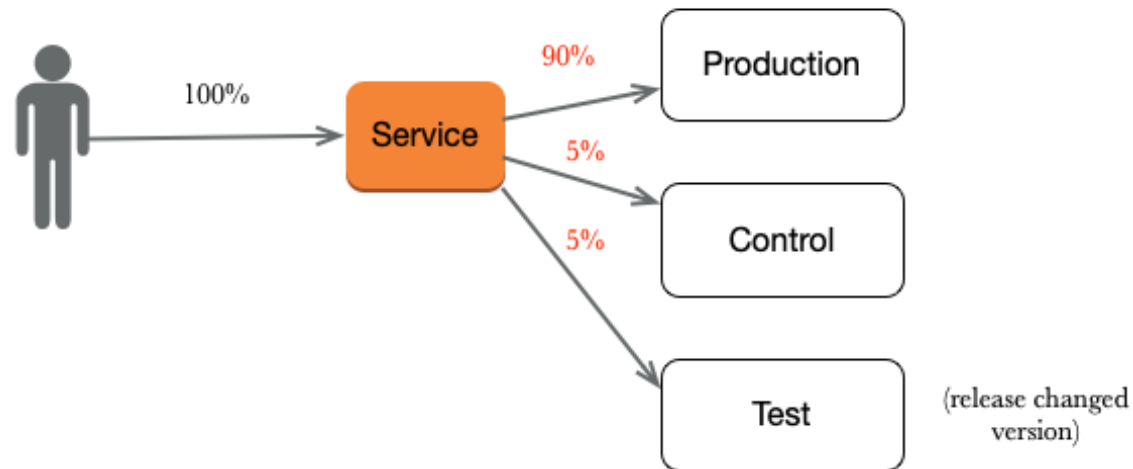
Chaos Engineering

A/B Test

Before Experiment



During Experiment



A/B Test

In an A/B test, a fraction of production traffic is funnelled into an explicit controlled experiment

2 groups:

- Test (or treatment) group has the change installed

- Control group is identical to prod

Used mostly to evaluate business impact of a change (as measured through business metrics)

Often UI changes with little danger of technical regression

Types of Experiments

Blue/Green Deployments

Canary Releases (gradual rollouts)

Dark Launches

A/B Testing

Chaos Engineering

Chaos Experiment

Unlike previous experiment types, a **chaos experiment** targets **resilience** of a current deployment

Not the impact of a specific change, like the ones discussed so far

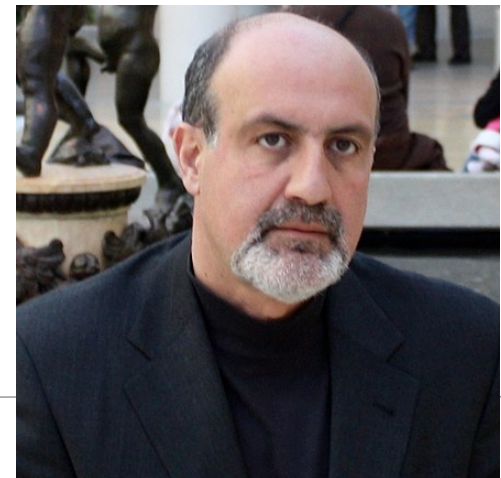
Observation:

Systems at scale will always experience unexpected faults and problems

No matter how careful we try to plan for everything

“Black Swans”

https://en.wikipedia.org/wiki/Black_swan_theory



Black Swan Events

Providers of scaled systems accept “black swans” as a fact of life

- *Requests sometimes fail or get massively delayed for no clear reason*
- *Pods die and need to be restarted*
- *Network connections are unstable for a brief period*
- *Software fails for no reproducible reason*
- *...*

Recall: Kubernetes is Sealf-Healing

Back when we discussed k8s we emphasised “self-healing” a lot

A large motivation for this is to deal with “black swans”

Example:

User: “My application consists of 3 MongoDB shards”

k8s: starts three pods with MongoDB running

k8s: observes that one MongoDB shard crashed, starts another

Chaos Engineering

Chaos engineering is the systematic evaluation of our production environment w.r.t. how well it deals with black swan failures

Basic Procedure

1. **Define a steady state.** How do our ops metrics look like if everything is going as planned?
2. **Define a hypothesis** what will happen in case of a specific disturbance. A disturbance might be the temporary outage of a specific network link, or a server crashing, etc.
3. **Enact the disturbance.** Kill the server or network link.
4. **Validate the hypothesis.** Use statistics to validate that your system reacted in the way you anticipated.

<https://principlesofchaos.org>

Some Observations

- Chaos engineering is technique reserved for **very mature** projects. You do not do a chaos experiment if you already know your system will crash!
 - Consequently, if your hypothesis is not *“the system will react gracefully with minimal interruptions to end users”* you are not going to go forward with the experiment.
- Even in mature orgs, chaos experiments tend to be high-pressure, with ops teams on stand-by to quickly intervene if things go off rails.

Chaos Monkey



Well-known (simple) tool in the space of chaos engineering
[Netflix]

“Chaos Monkey is responsible for randomly terminating instances in production to ensure that engineers implement their services to be resilient to instance failures.”

<https://netflix.github.io/chaosmonkey/>

Summary - “Hypothesis-Driven Development”

Scaled deployments enable us to use our customer base for **quality assurance** and **requirements engineering** in ways not possible with traditional applications.

Mostly an application of the scientific method to the analysis of **production monitoring** data.

Only becomes a factor once you have sufficient customers to do **reliable statistical testing**.