CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Monitoring and Feature Toggles

Dr. Philipp Leitner

philipp.leitner@chalmers.se

@xLeitix

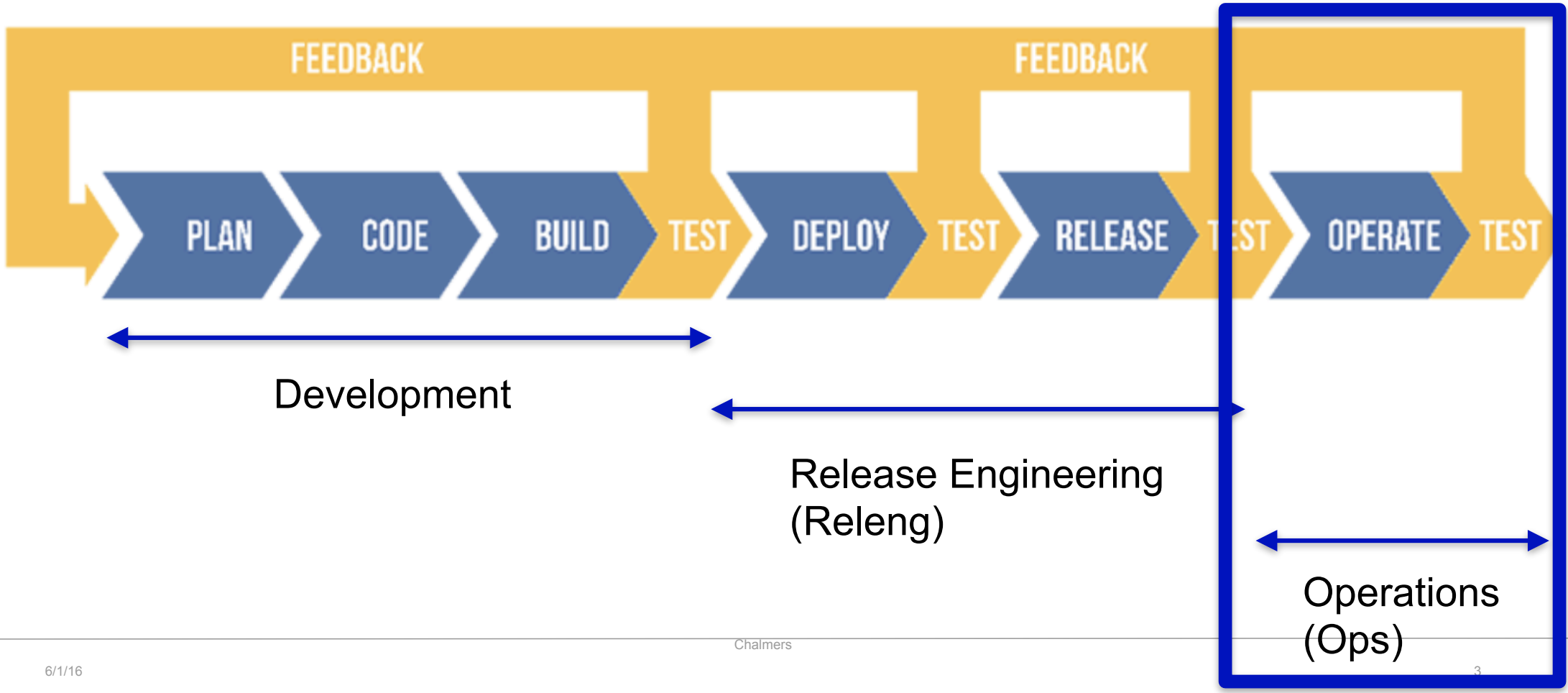
LECTURE 8

Covers …

Monitoring a Running System

Feature Toggles as a Way to Adapt a Running System

SaaS Application Lifecycle



Release Engineering

Traditional Tasks:

- Maintaining the build pipeline
- Quality Assurance (beyond unit testing)

Operations (ops)

Traditional Tasks:

- Capacity planning and capacity management
- Systems administration
- Incident management

Incident Management

"Incident management is the process of responding to an unplanned event or service interruption to restore the service to its operational state."

"Incidents are events of any kind that disrupt or reduce the quality of service (or threaten to do so). A business application going down is an incident. A crawling-but-not-yet-dead web server can be an incident, too. It's running slowly and interfering with productivity. Worse yet, it poses the even-greater risk of complete failure."

Source: https://www.atlassian.com/itsm/incident-management

Sidenote

In much of the rest of this lecture, we will mainly talk about **non-functional problems**.

- Scalability issues
- Slowdowns
- Temporary outages

Traditional functional bugs are identified using standard techniques you already know from other courses

- E.g., unit and regression testing

Incidents are **production problems**

- With a (root) cause **either in development or operations**
- Which may be fixable **either in development or operations**

As a rule of thumb:

Fixing an incident in ops - cheap now, expensive later

"Throwing more hardware at the problem"

Fixing an incident in dev - expensive (and slow) now

"Paying off technical debt"

How do we even **learn** about production incidents?

(1) Because users complain about them (**bad**)
(2) Because of monitoring (**much better**)

—> Pro-active (monitoring can alert us before users are impacted)
—> Allows us to distinguish between acceptable and unacceptable runtime situations

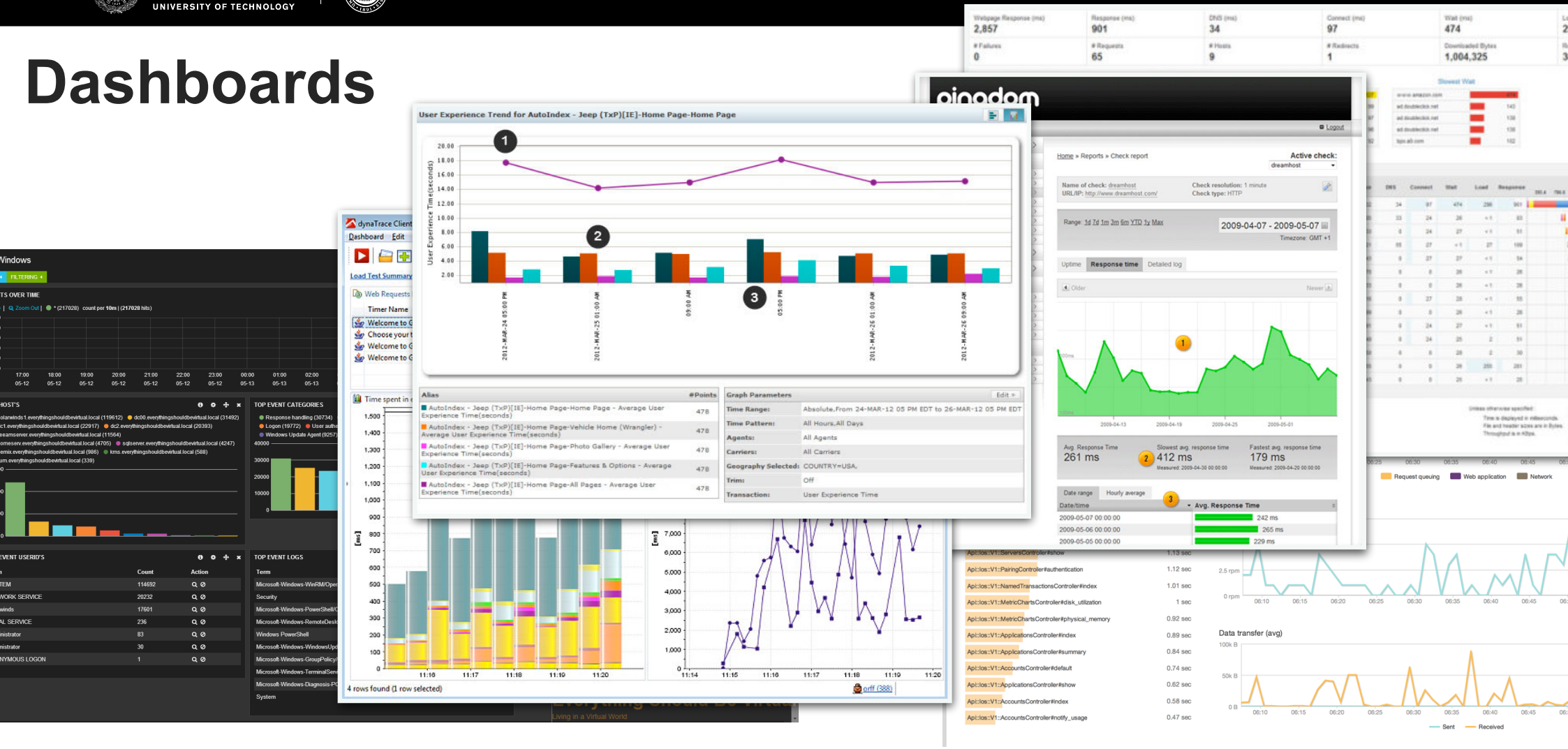
(Runtime) Monitoring

Systematic and automatic tracking of **metrics**

Metrics are quantitative data about the state of our system

- CPU load
- Error rates
- Conversion rates
- … hundreds more …

Dashboards


User Experience Trend for AutoIndex - Jeep (TxP)[IE]-Home Page-Home Page
dynaTrace Client
Dashboard Edit
Load Test Summary
Web Requests
Timer Name
Welcome to G
Choose your t
Welcome to G
Welcome to G
Time spent in
Alias
#Points
Graph Parameters
Time Range:
Absolute,From 24-MAR-12 05 PM EDT to 26-MAR-12 05 PM EDT
Time Pattern:
All Hours,All Days
Agents:
All Agents
Carriers:
All Carriers
Geography Selected:
COUNTRY=USA,
Trim:
Off
Transaction:
User Experience Time
4 rows found (1 row selected)
pingdom
Logout
Home » Reports » Check report
Active check:
dreamhost
Name of check: dreamhost
URL/IP: http://www.dreamhost.com/
Check resolution: 1 minute
Check type: HTTP
2009-04-07 - 2009-05-07
Timezone: GMT +1
Uptime
Response time
Detailed log
Avg. Response Time
261 ms
Slowest avg. response time
412 ms
Fastest avg. response time
179 ms
Date range
Hourly average
Date/time
Avg. Response Time
2009-05-07 00:00:00
242 ms
2009-05-06 00:00:00
265 ms
2009-05-05 00:00:00
229 ms
Request queuing
Web application
Network
Data transfer (avg)
Sent
Received
Windows
TOP EVENT CATEGORIES
TOP EVENT LOGS
EVENT USERID'S

Types of Metrics

System Metrics vs. **Application Metrics** vs. **Business Metrics**

Types of Metrics

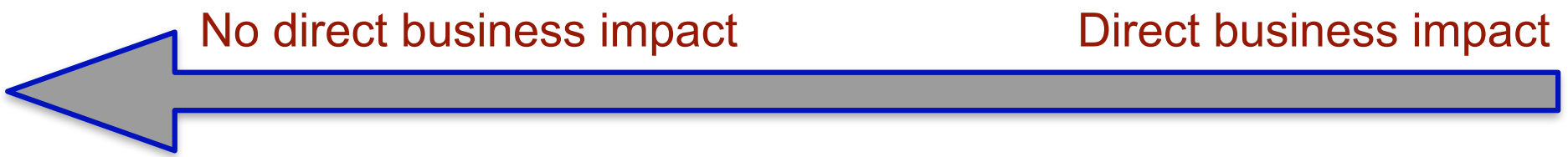
System Metrics vs. **Application Metrics** vs. **Business Metrics**

Types of Metrics

System Metrics vs. **Application Metrics** vs. **Business Metrics**

Types of Metrics

System Metrics vs. **Application Metrics** vs. **Business Metrics**



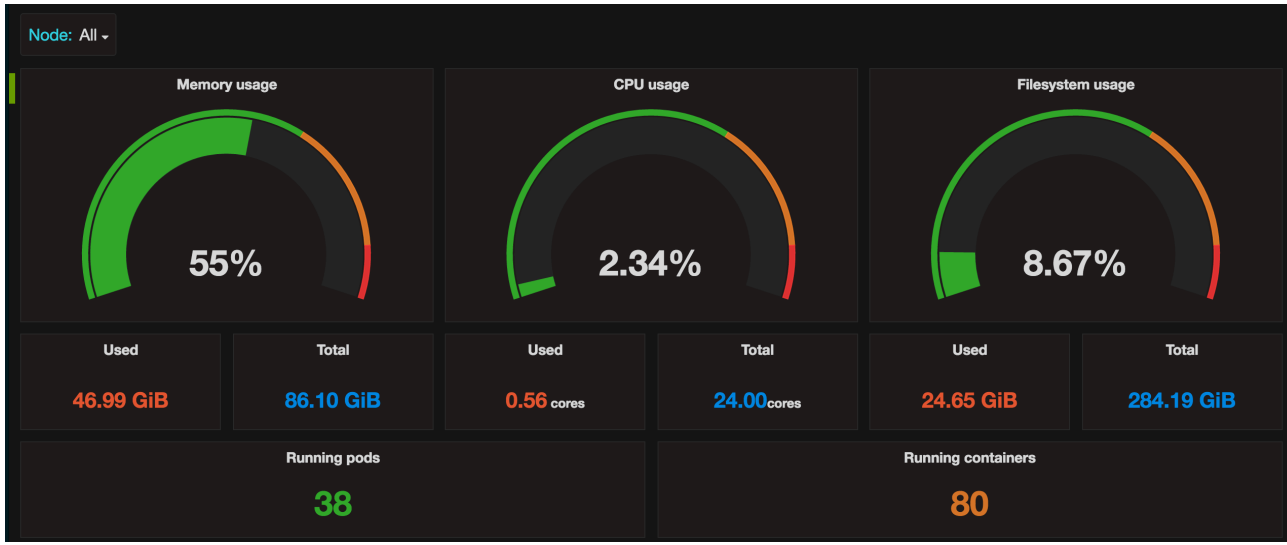
System Metrics

Metrics measuring the health of your **infrastructure**

Easy to monitor using standard tools

Useful as a proxy for application or business performance problems

Best way to identify Ops issues



Examples:

- Available space for additional Kubernetes pods
- Average Kubernetes node CPU load
- Free cluster memory

Application Metrics

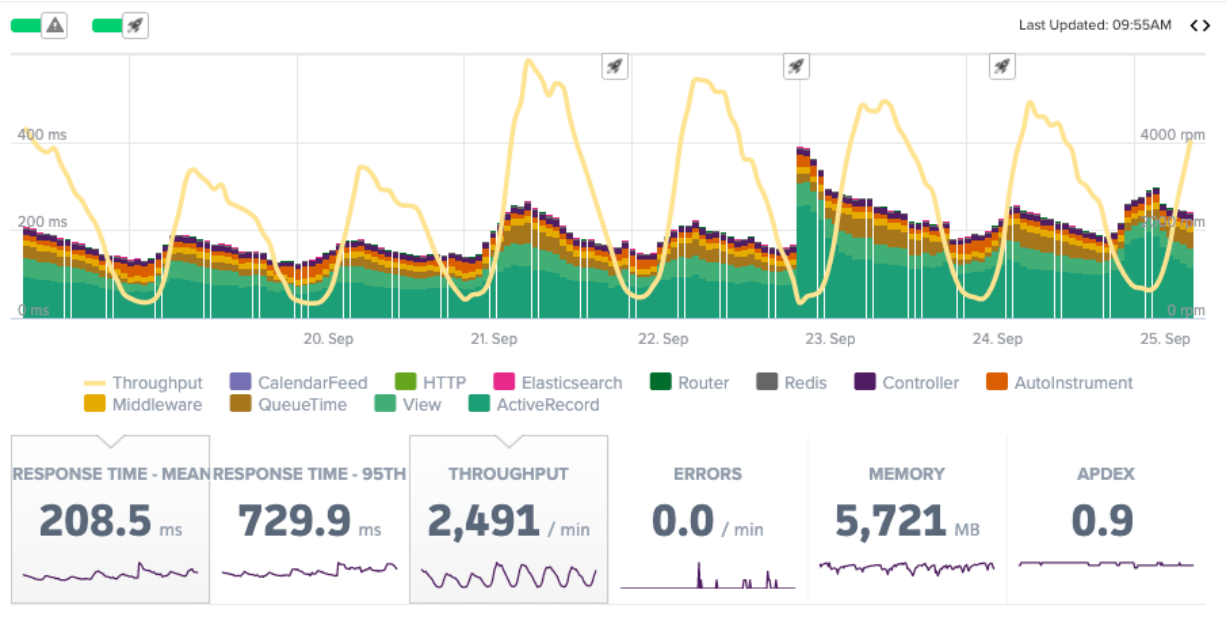
Metrics measuring the health of your **applications**

Still (reasonably) easy to monitor using (different) standard tools

Best way to identify Dev (software) issues

Examples:

- Service response time
- Service error rate
- Page load time



Business Metrics

Metrics measuring the **business value** of your application

Requires custom probes and monitoring

Ultimately what “counts”, but difficult to define in a generic manner

Examples:

- Conversion Rate
- Clickthrough Rate
- “Relevant Search Results on First Page” (Google)
- “Orders Confirmed per Visitor” (Amazon)
- “Hours Played Per User” (some online game)
- … … …

Monitoring Tools

Significant growth market

System metrics:

- For hardware / VMs: standard system monitoring tools
- For Kubernetes: **Prometheus**

Application metrics:

- “Application Performance Monitoring” (APM)
- NewRelic, Dynatrace, Datadog, …

Business metrics:

- Mostly custom dashboards
- Built on top of standard monitoring and dashboarding solutions (Prometheus, **Grafana**)

Prometheus

Standard open source monitoring system for Kubernetes

Basic concepts:

- Vanilla Prometheus is really just a *Time Series Database*
- *Metrics* are defined as (named) *time series* of *key/value pairs*
- *Alerts* are defined as time series queries
  - Using custom query language PromQL
- Most installations will come with some standard metrics predefined
  - E.g., Kubernetes system metrics
  - But the real power comes from being able to push your own data and defining your own metrics

Grafana

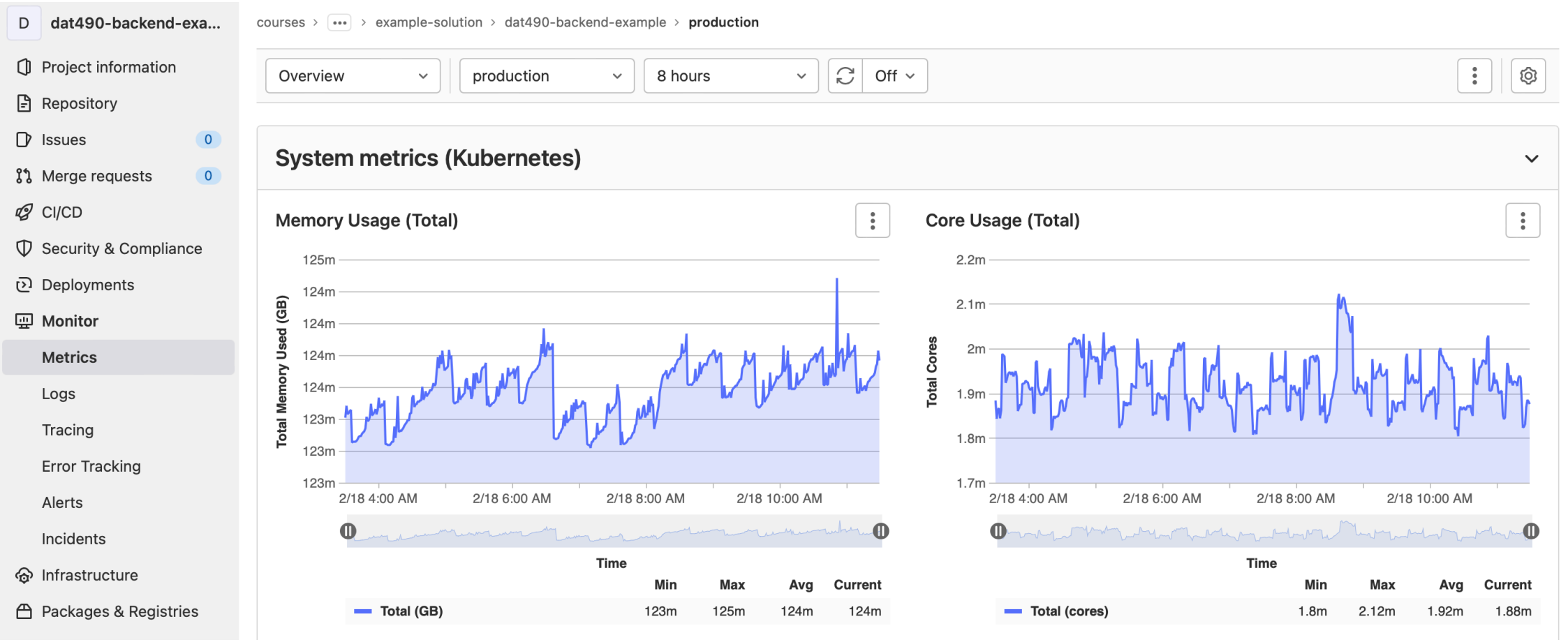
Visualization / dashboarding system

Standard “frontend” for Prometheus

Example Video:

https://player.vimeo.com/video/561723556?badge=0&autopause=0&player_id=0&app_id=58479&loop=true

Prometheus and Grafana in GitLab



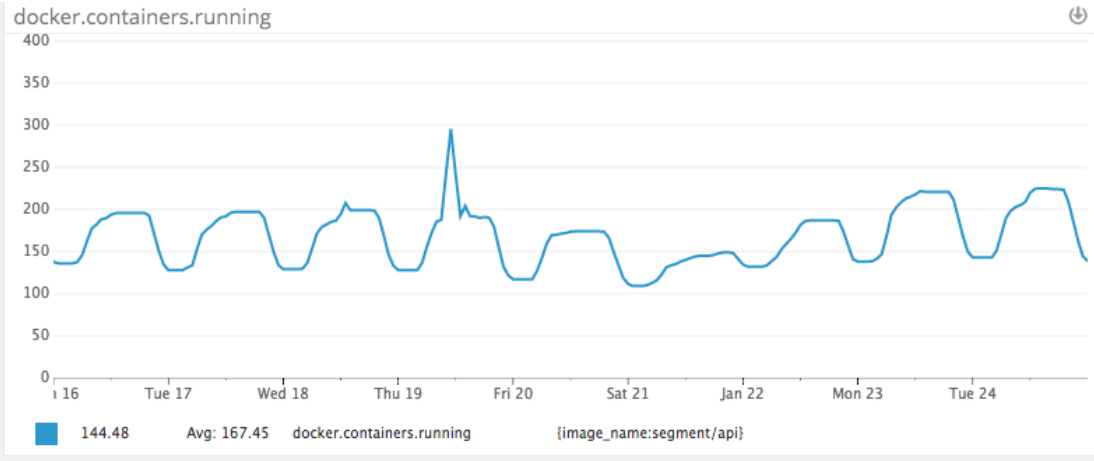
Identifying Faults in Time Series

Two basic problems of monitoring:

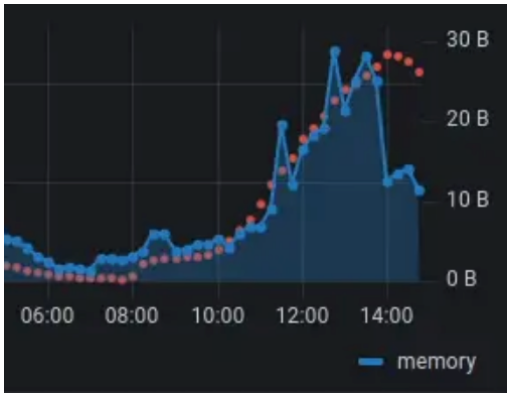
When to decide that a certain behaviour of our system is abnormal (faulty)? **Fault identification.**

How to reason about the original cause (root cause) of abnormal behaviour? **Root cause analysis.**

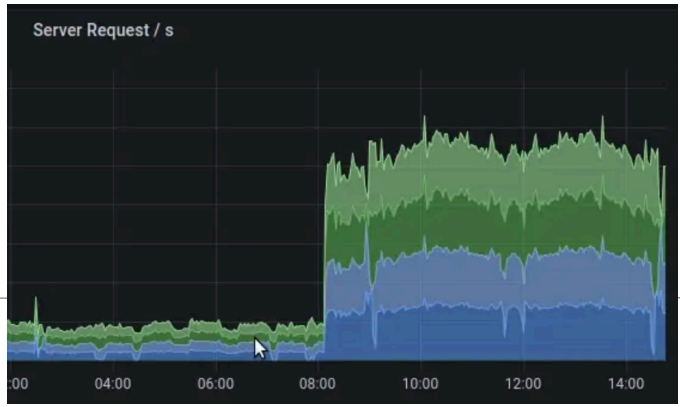
Fault Identification



Is this normal behavior?



Hard to say without knowing more …



Basic Fault Identification Strategies

Ultimately a statistical problem:

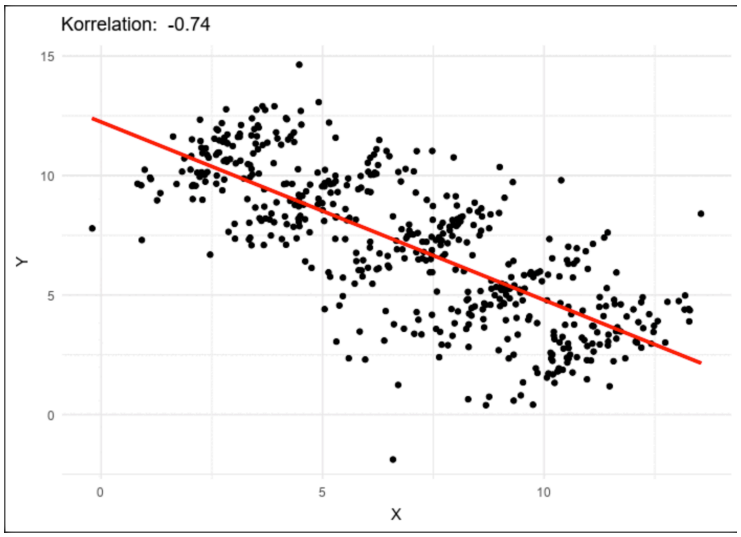
Given a time series *t*, identify all time points t_n where the distribution statistically significantly changed

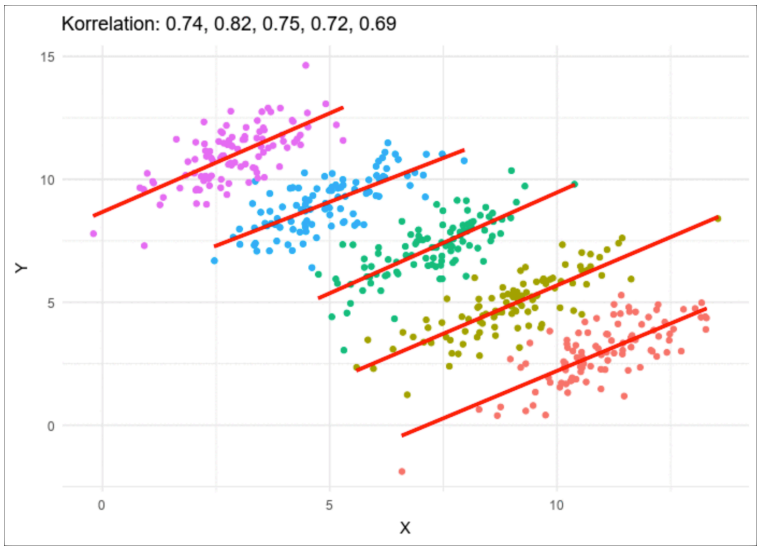
Strategies:

- Use threshold values (recall: that's what most load balancers do)
- Use (human) visual analysis (challenge: needs human attention)
- Use statistical changepoint analysis (challenge: delayed, false positives)
  - ARIMA, SARIMA, various ML approaches

The Simpson's Paradox

Aggregating multiple distributions can show a different (even reversed) trend





The Simpson's Paradox

When reasoning about monitoring data we frequently aggregate different sources of data:

- Metrics from different pods
- Metrics from different nodes
- Metrics produced by different types of users
- …

Can lead to (for instance) correctly functioning pods cloaking faults in a small number of malfunctioning ones

Important to **stratify** data

- But: increases number of metrics, complexity of analysis

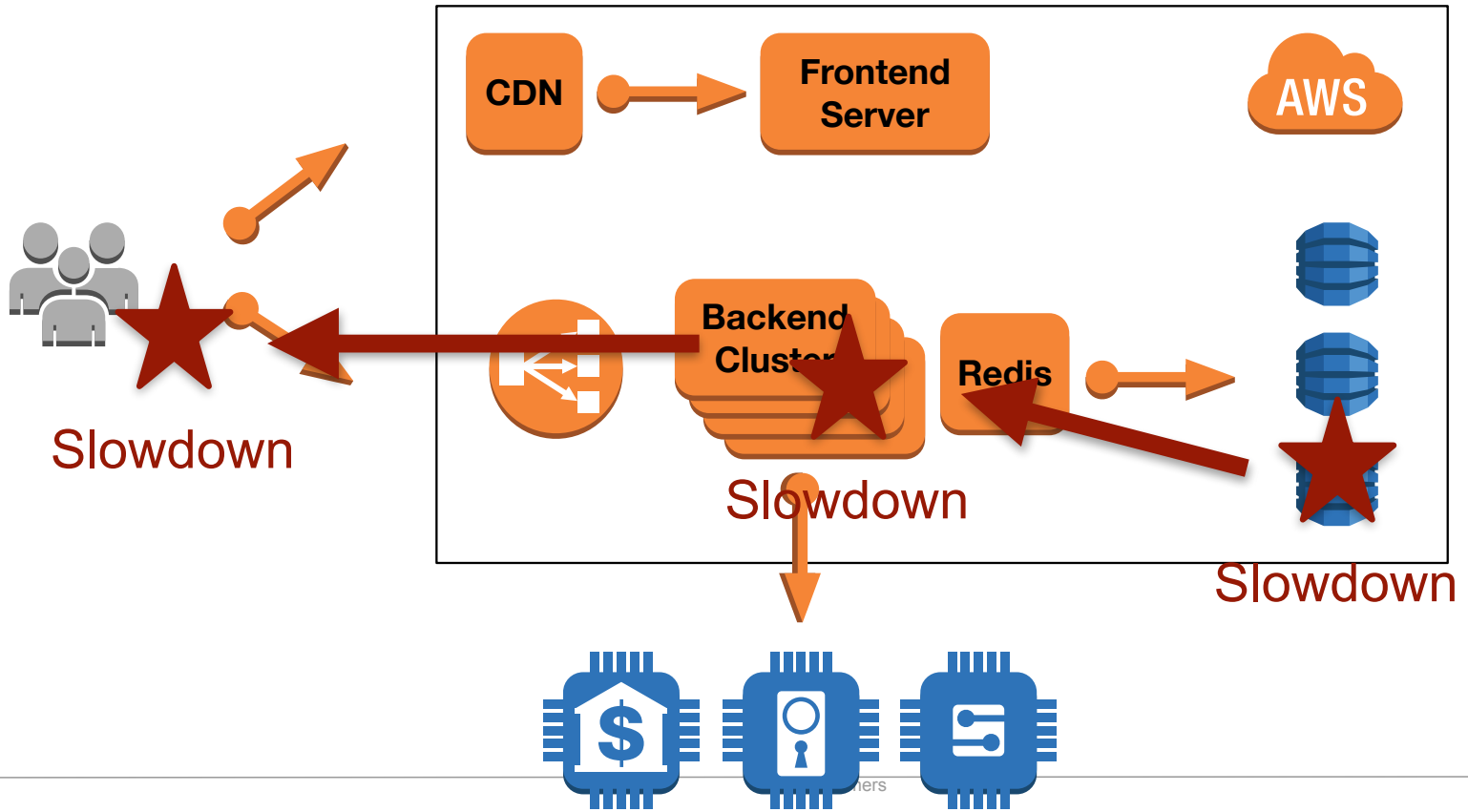
Root Cause Analysis (RCA)

Once we have identified a problem, how do we know what is causing it?

Basic problem:

Faults **propagate** in a distributed system

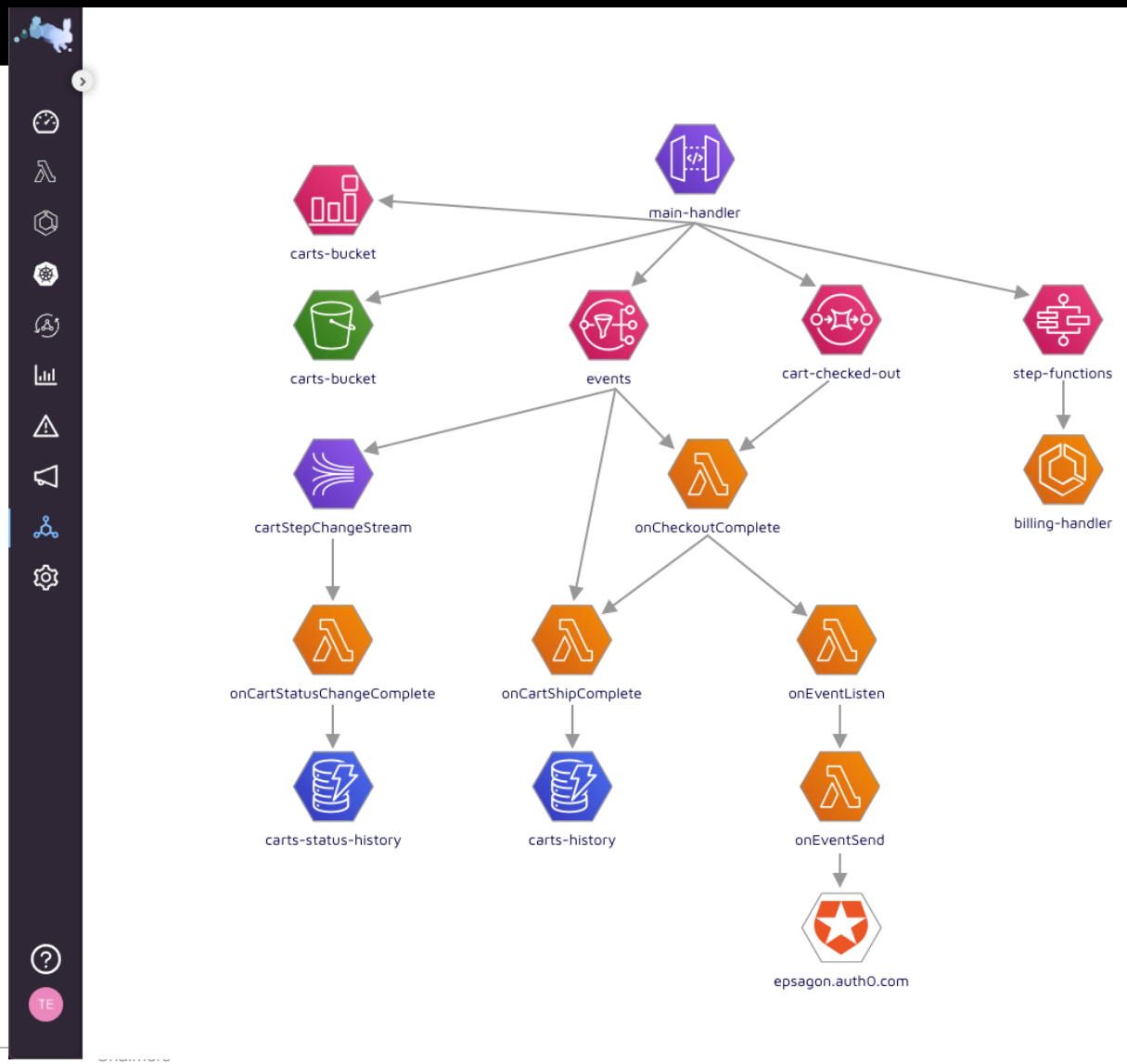
Fault Propagation



Service Dependency Graph

For each (type of) request, we can draw a graph representing service dependencies in our system

Directed Acyclic Graph (DAG)



Distributed Tracing

In large systems, service dependency graphs are generated via **distributed tracing**

RCA is only feasible if we can **trace** (follow) individual requests through the components / services of our system

Trace IDs

Service dependency graphs can be generated through **tracing**

Basic mechanism:

- Insert (unique) **trace id** into each request (e.g., request header)
- Ensure that trace id is included in invocations of each dependent service
- Allows to reconstruct call graph from individual invocation logs
- Service dependency graphs can then be derived from individual traces

Practical Usage

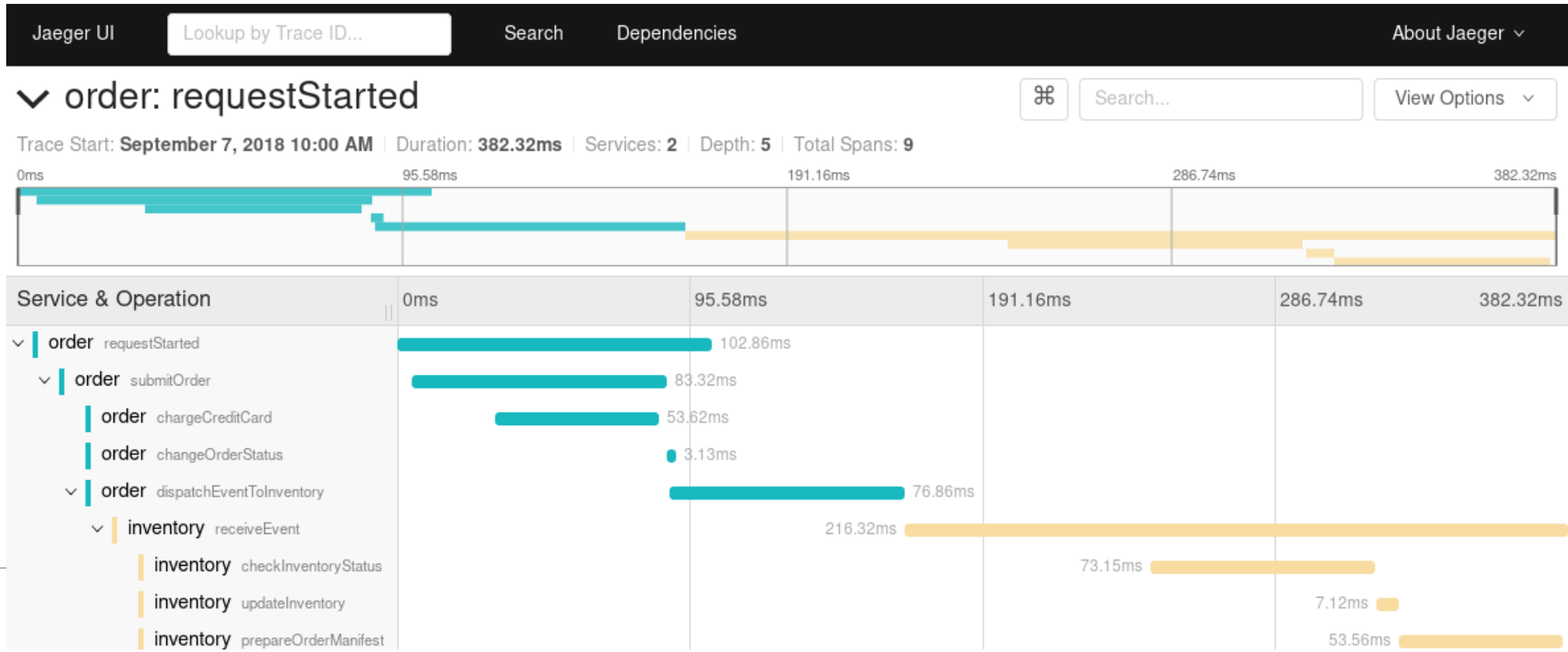
Service dependency graphs have many uses in practice:

- Live architecture documentation
- Performance profiling
- Root cause analysis of performance problems

  The root cause is the service whose slowdown cannot be explained by the slowdown of a dependency

Profiling Request Performance using Tracing

Example: Jaeger (open source APM solution)



So far …

We have learned how monitoring can help us **identify** faults

And how tracing can help narrow down **where** the fault is

What can we **do** about faults now?

- First decide if it's a dev or ops issue
- If dev issue: decide if caused by recent release
- If yes: trigger procedure for handling faulty releases

How to handle a problematic release

Three basic ways to react to a fault:

1. Roll back the change (roll back)
2. Fix the problem with a hotfix (roll forward)
3. Disable the problematic feature (feature toggles)

Rolling Back

Default way to deal with a problematic code release is to revert it

Assumption:

- Individual changes are easy to isolate in Git
- No mixing of unrelated fixes in commits / pull requests

Potential problem: **state corruption**

State Corruption

Version **n**

```
var orderSchema = new Schema({
    orderRef : { type: String },
    totalPrice : { type: Number },
    productsList : [
        {
        type: String,
        ref: "Product"
        }
    ],
    orderStatus : { type: String }
});
```

Version **n+1**

```
var orderSchema = new Schema({
    orderRef : { type: String },
    totalPrice : { type: Number },
    productsList : [
        {
        type: String,
        ref: "Product"
        }
    ],
    orderStatus : { type: String },
    customer : { type: String },
});
```

State Corruption

After rollback of a stateful service, the state (e.g., database) may contain invalid data entries

- Attributes that have been added / removed
- Corrupted entries resulting from faulty release

Possible fixes:

- Rollback of the database to a previous snapshot (**rarely desirable**)
- Triggering compensation actions:
  - Triggering “inverse” business transactions (e.g., refunding money)
  - Repairing corrupted database entries
  - Individually rolling back corrupted transactions

Rolling Forward

If a faulty change can't easily be rolled back (or rolling back is undesirable for other reasons) an alternative may be to rush a fix into production

Called (tongue-in-cheek) "rolling forward"

Downsides:

- Takes time
- How much of the normal quality control are you willing to skip on a hotfix?

Many mature CI/CD teams **disallow** rolling forward entirely

If rolling forward is allowed, teams should specify a **clear process** / pipeline for emergency fixes

Feature Toggles

The “ideal” way to handle faulty features would probably be if we could just disable them at runtime

—> **Feature Toggles / Feature Flags**

Similar to rolling back, but without having to actually do a redeployment

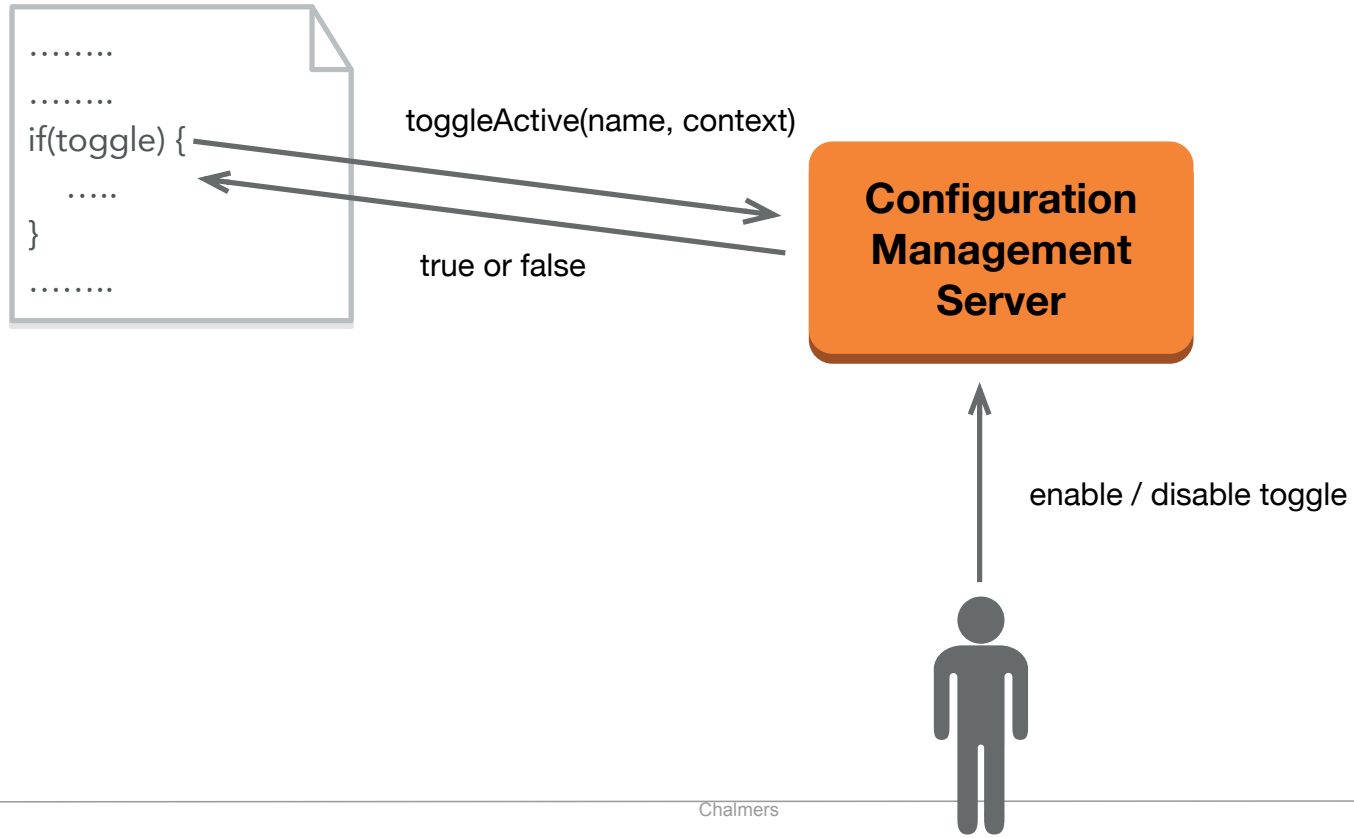
Feature Toggles

Basic concept simple to understand:

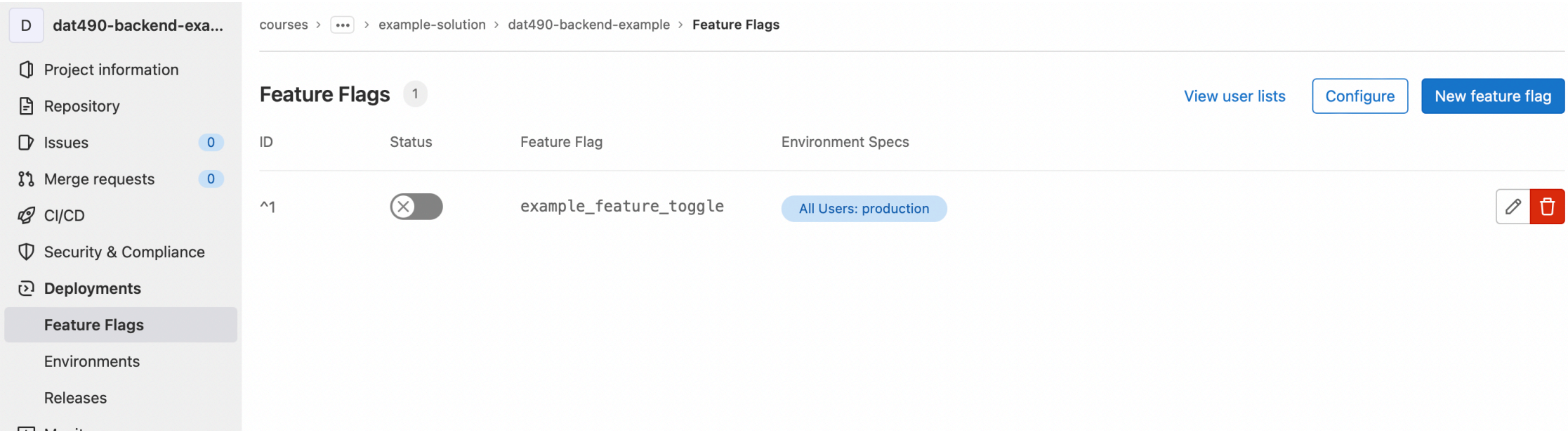
Imagine each new feature being wrapped in an **if-statement**

```
if(featureEnabled(newRecommendationSystem)) {
    // call and use the new recommendation system
} else {
    // do what you did before this change
}
```

Architecture of a Feature Toggle System



Feature Toggles in GitLab



Basic Concepts

Toggle (or Toggle Point):

Place in the service source code where toggle is used

Toggle Name:

Unique name of a feature toggle (most teams use some naming pattern to ensure uniqueness)

Toggle Context:

Additional info to decide if a toggle should be enabled or disabled

Examples: user identifier, experiment identifier, etc.

Configuration Management Server (or Toggle Router):

Central database of toggle states

Decides based on toggle name and context the state of any given toggle (on / off)

Use Cases for Feature Toggles

- **Disabling faulty features**

  Quicker and less intrusive than a rollback
- **Enabling early access to test users**
- **A/B Testing**

  (More on the latter two in next lecture)

Disadvantages

- **Feature toggles require upfront planning**
  - When you have a fault it is too late to decide that this feature needed to be behind a toggle
  - Some teams launch *every* change behind a toggle (but then toggle maintenance becomes tedious)
- **Feature toggles need continuous maintenance**
  - Toggles that are not eventually removed become technical debt
  - Excessive toggles can lead to confusion and errors
    - Ops and DevOps staff will only be able to wrap their heads around a finite number of toggles

Best Practices
(somewhat subjective guidance ;))

- Use feature toggles **generously**, but:
- Also have a clear process and **time frame** for when a toggle is to be removed again
- See it as a **bad smell** if a toggle is intended to stay in the code “indefinitely"

- Use feature toggles for **release engineering**
- Do not **abuse** them, for example for authorization or system configuration

Feature Toggles in ScalyShop

GitLab has a configuration management server **built-in**

Compatible with **Unleash** (an open source feature toggling system)

- https://github.com/Unleash/unleash

Can be used in conjunction with any client library that supports the Unleash API

- Client libraries for many programming languages available
- Suggestion for assignment: https://docs.getunleash.io/sdks/node_sdk