FACULTY OF ENGINEERING AND ARCHITECTURE

MICROSERVICES AND CONTAINERS

Chapter 10: Parallel and Distributed Software Systems

GHENT UNIVERSITY

OUTLINE

1. Application building: from then to now
2. The microservice approach
3. Microservice frameworks and developer tools
4. Containers as a microservice enabler
5. Conclusions

MICROSERVICE APPROACH

GHENT UNIVERSITY

DEFINITION

Microservice architectural style:

- Approach to developing a single application as a suite of small services
- Each running in its own process
- Communicating with lightweight mechanisms (e.g. REST or message queues)

Microservices:

- Built around business capabilities
- Independently deployable by fully automated deployment machinery
- Bare minimum of centralized management
- Writ
- Using different data storage technologies

Microservice core principle:
loose coupling, strong functional cohesion

SOME EXAMPLES OF COMPANIES EMBRACING MICROSERVICES

NETFLIX

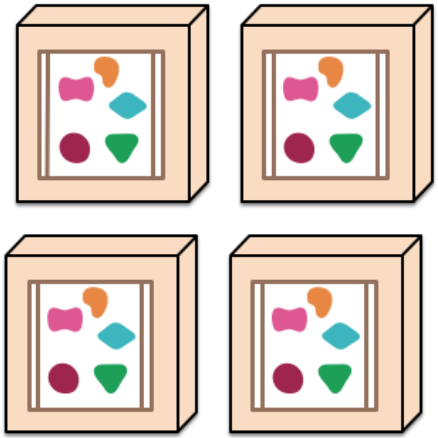
airbnb

Source: *Mastering Chaos – A Netflix guide to Microservices*
https://dev.tube/video/CZ3wIuvmHeM

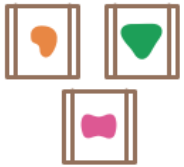
FROM MONOLITHIC APPLICATIONS TO SUITES OF SERVICES

A monolithic application puts all its functionality into a single process...

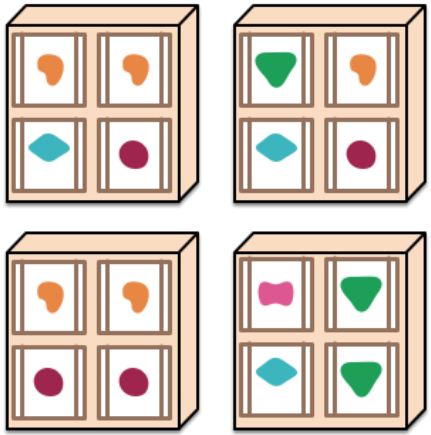
... and scales by replicating the monolith on multiple servers



A microservices architecture puts each element of functionality into a separate service...



... and scales by distributing these services across servers, replicating as needed.



GHENT UNIVERSITY

SO HOW 'MICRO' IS A MICROSERVICE?

Cannot be expressed in lines of code

- Some languages way more expressive than others
- Potentially lots of dependencies

Keep service focussed on cohesive functionality / single business capability

Rule of thumb

- Microservice is something that could be rewritten in two weeks
- Your service is small enough once it no longer feels "too big"
- Manageable by a small team

The smaller you go, the more complexity increases of having more and mo

4 important properties of Microservices

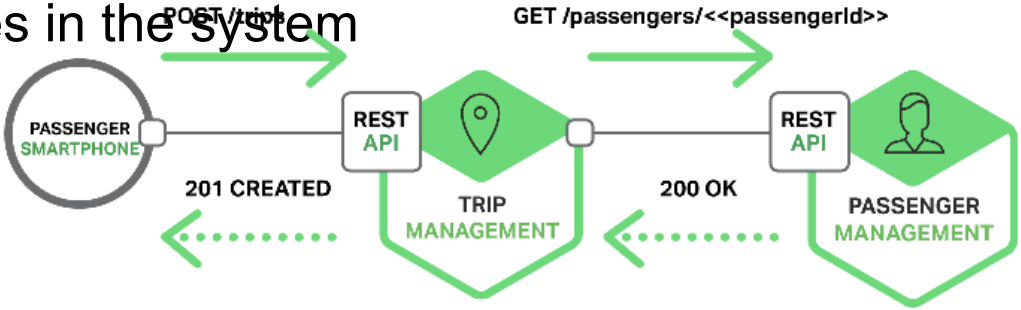
MESSAGE DRIVEN

Asynchronous message-passing between components

- Addressable recipients await the arrival of messages and react to them
- Establishes a boundary that enables
  - Loose coupling
  - Isolation

Enables load management and elasticity by monitoring and shaping the message queues in the system

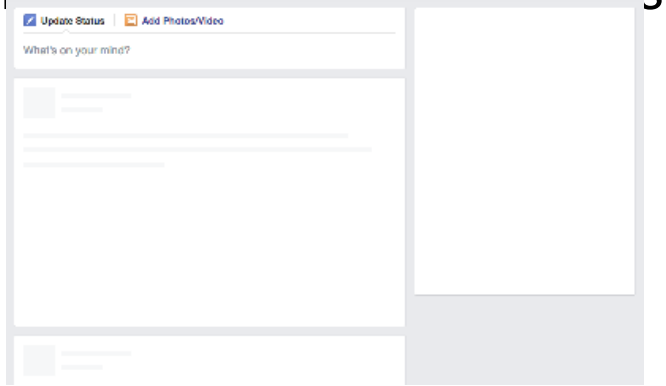
Non-blocking



RESPONSIVE

The system responds in a timely manner (aim: 0.1 seconds)

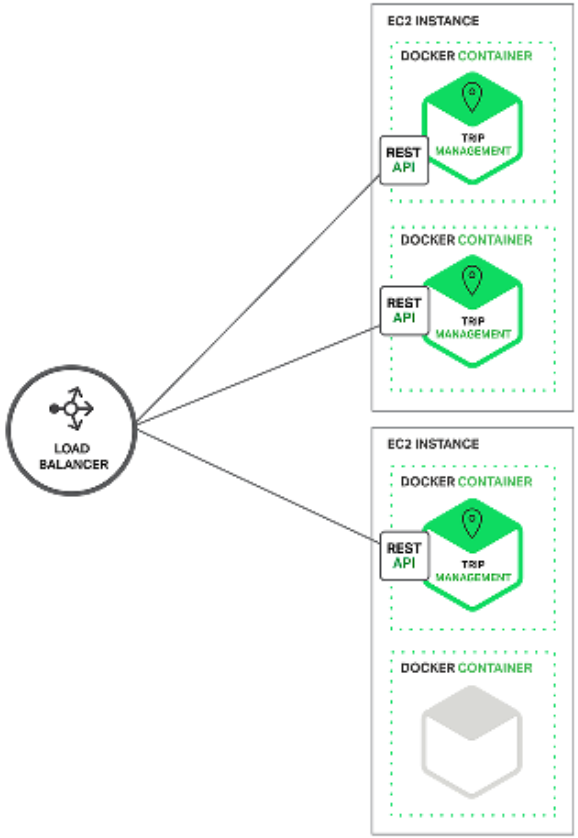
- Client side lazy loading: first load important stuff and show it ASAP
- Show progress
- Individual slow performing service should not slow down others



As far as users know, when the response time exceeds their expectation, the system

ELASTIC

- System stays responsive under varying workload
  - Changes in input rate lead to increased or decreased resource allocations
  - No contention points or central bottlenecks
  - Distribution of input amongst components
- An elastic system can allocate / deallocate resources for every individual component dynamically to match demand
- Both predictive and reactive elastic scaling



RESILIENT

Any service call can fail

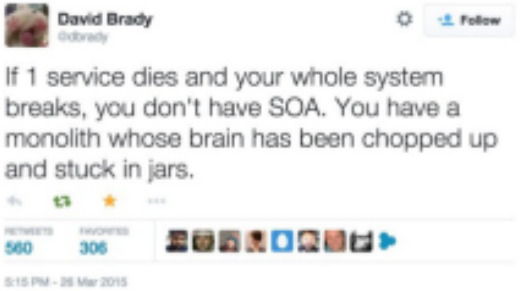
Detect failures quickly by monitoring

- Service metrics (e.g. requests per second)
- Business metrics (e.g. orders per minute received)

and automatically restore services when issues are detected

Provide fallback services

- E.g. Netflix graceful degradation: If recommendation service is down revert to most popular movies instead of personalized



Understand the impact of each service outage and work out how to properly degrade functionality

GHENT UNIVERSITY

RESILIENT TO FAILURE: NETFLIX'S SIMIAN ARMY

Chaos Monkey

- Introducing random failures in their production AWS servic
- Team of engineers ready to intervene

Chaos Gorilla: disables an entire availability centre

Latency Monkey

- Introduces artificial delays

Janitor Monkey

- Seeks unused resources and disposes of them

Security Mo

- Se

The best defense against failures is to fail often, forcing your services to be built in a resilient way

MICROSERVICE FRAMEWORKS / DEV TOOLS

Dropwizard (http://www.dropwizard.io)

- Java framework for developing RESTful web services

Vert.X (http://vertx.io/)

- Toolkit for building reactive applications on JVM

Spring Boot (https://projects.spring.io/spring-boot/)

- Eases development of Spring applications

Restlet (https://restlet.com/)

- Microservice-oriented API creation / testing / execution

Spark (http://sparkjava.com/)

- Micro-framework for creating web applications in Java 8

Lagom (http://www.lightbend.com/lagom)

- Microservice framework built on Akka and Play

Note: a lot of JVM-based microservice frameworks. Note that JVM is no longer Java-language only and can run Ruby, Python, Scala, Groovy, Clojure, JavaScript, etc.

And many more: WSO2, Jolie, Baratine, KumuluzEE, Jlupin, etc.

CONTAINERS

GHENT
UNIVERSITY

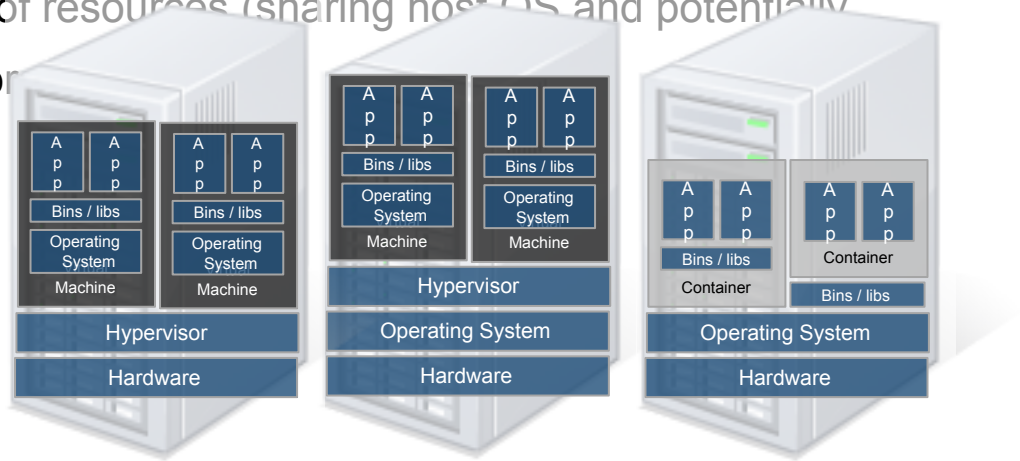
CONTAINERS

VMs

- Virtualization of hardware
- Flexible, robust and safe, but fairly big performance hit

Containers

- Lightweight
- Better use of resources (sharing host OS and potentially binaries/ libr



Type 1 Hypervisor | Type 2 Hypervisor | OS-level virtualization

PROS AND CONS OF CONTAINERS

No performance hit due to emulation of instructions

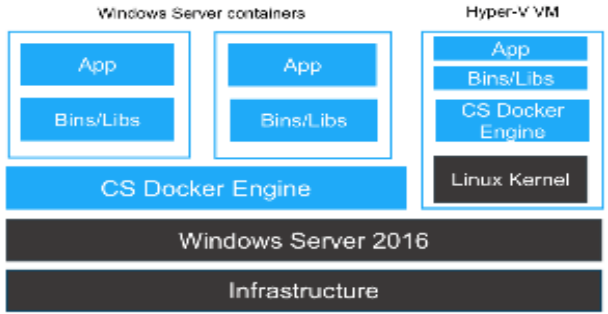
Flexibility
- Containerize applications or full systems
- If it works on your machine, it will work on another machine

Lightweight
- No entire OS in each container
- Sharing of bins and libraries
- Provisoned / instantiated in a few seconds
- Minimal per-container penalty
- “Just Enough OS” on the server (e.g. CoreOS)

Cannot host a guest OS different from the host one*

Weaker isolation and thus security

*actually Docker on Windows also supports Hyper-V containers where Docker runs in a small Linux VM (Alpine Linux-based)



GHENT UNIVERSITY

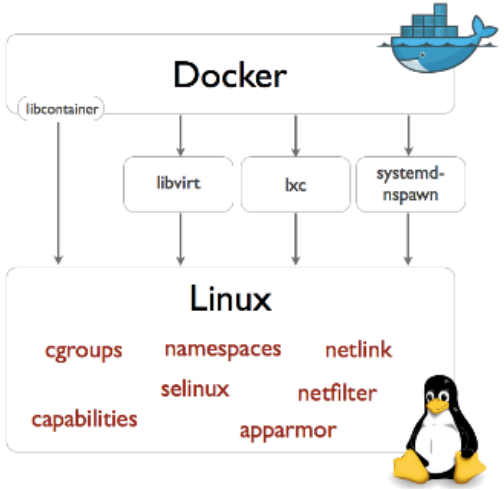
DOCKER

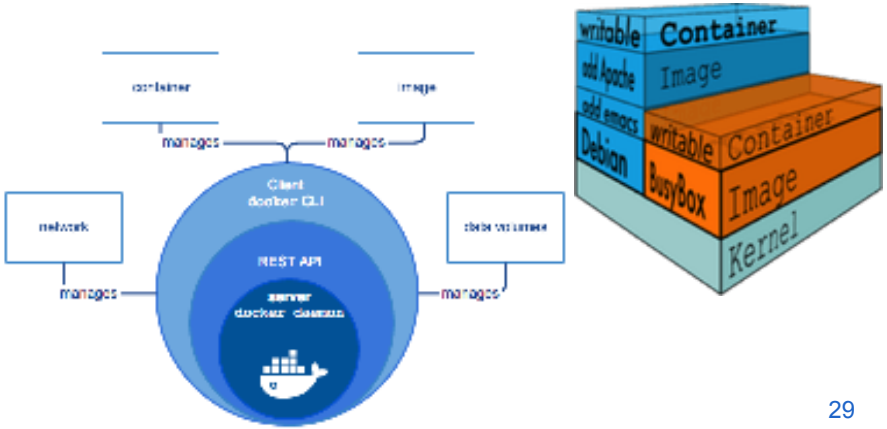
Most popular open source container technology, conceived in 2013

- Extended LXC, later moved to libcontainer (cross-system abstraction)
- Now available on Mac OSX, Windows 10/Server2016, etc.

Consists of Docker Engine (Daemon + CLI) and Docker Hub

- Daemon
  - Builds images, runs and manages containers + REST API
  - CopyOnWrite functionality
- Hub
  - Provides Docker library of images
  - Images: onion layered filesystem
  - Linked to automated builds (Github/Bitbucket)





GHENT UNIVERSITY

LIGHTWEIGHT OS

Lightweight OSs being developed focused on container-usage:

- CoreOS (focused on server containers)
- Red Hat Project Atomic (focused on server containers)
- Ubuntu Core / Snappy (focused on IoT)
- Microsoft Nano Server (focused on server containers)

CONTAINER MANAGEMENT SYSTEMS

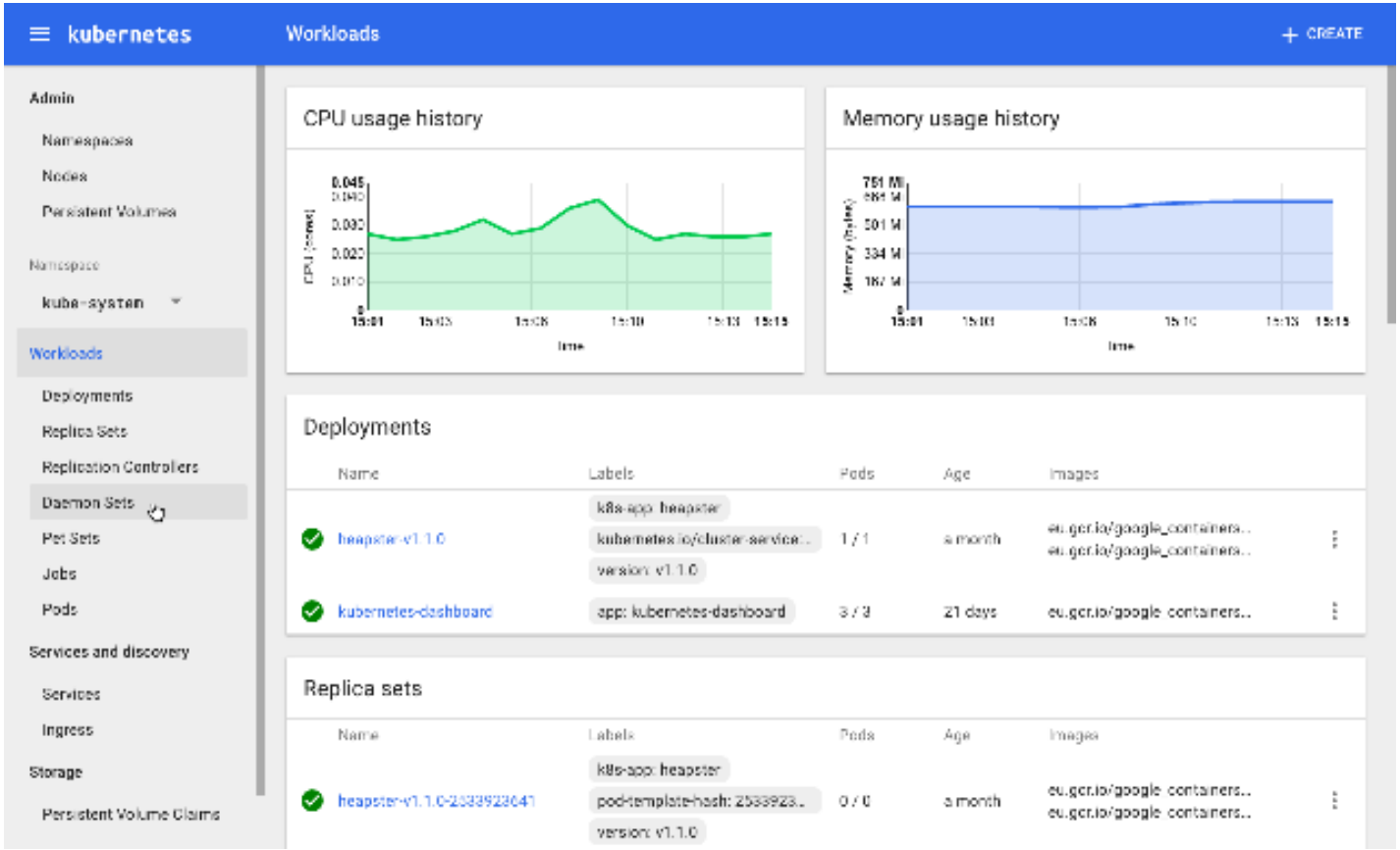
Automate deployment, scaling and management of containerized applications, including

- Automated resource provisioning (critical / best effort)
- Self-healing containers that fail
- Automated roll-out and rollbacks
- Storage orchestration
- Application configuration management
- Service discovery and load balancing
- etc.

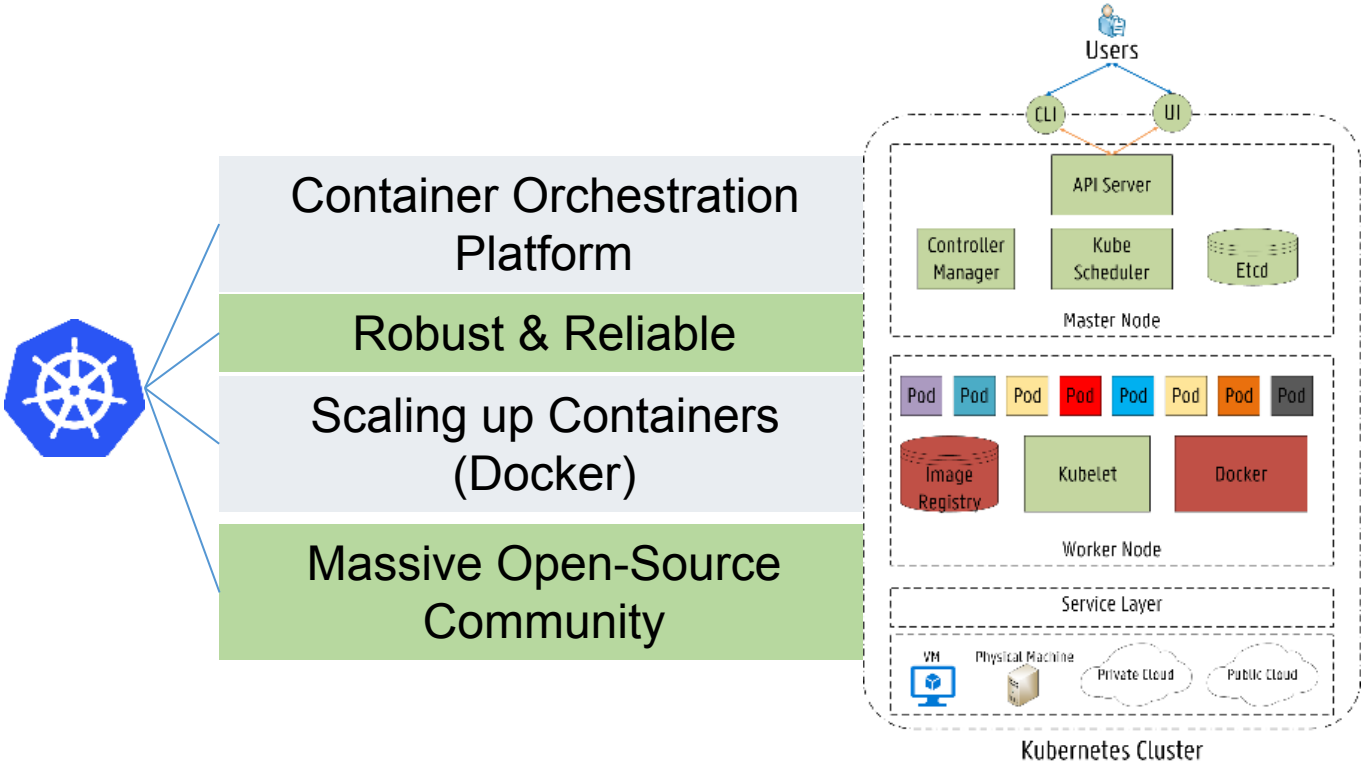
OPEN-SOURCE KUBERNETES (RELEASED 2015)

Designed on same principles (Borg) that allow Google to run billions of containers per week

KUBERNETES


kubernetes
Workloads
+ CREATE
Admin
Namespaces
Nodes
Persistent Volumes
Namespace
kube-system
Workloads
Deployments
Replica Sets
Replication Controllers
Daemon Sets
Pet Sets
Jobs
Pods
Services and discovery
Services
Ingress
Storage
Persistent Volume Claims
CPU usage history
Memory usage history
Deployments
Name
Labels
Pods
Age
Images
heapster-v1.1.0
1 / 1
a month
Replica sets
Name
Labels
Pods
Age
Images
heapster-v1.1.0-2333923641
0 / 0
a month

KUBERNETES (K8s)



OPTIMIZED SCHEDULING

Towards network-aware scheduling in K8s. Why?

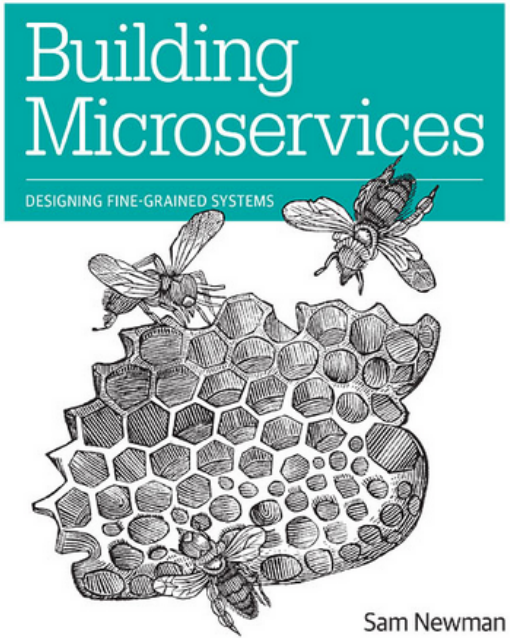
- The K8s scheduler typically focuses on Resource Efficiency (e.g., CPU and Memory).
- No contextual awareness about application dependencies or infrastructure topology.
- Low latency plays a major role for several applications (e.g., IoT, video streaming).

How can we improve the scheduling?

- Consider latency and network bandwidth in the scheduling process.
- Diktyo framework – open source

GHENT UNIVERSITY

ADDITIONAL REFERENCES



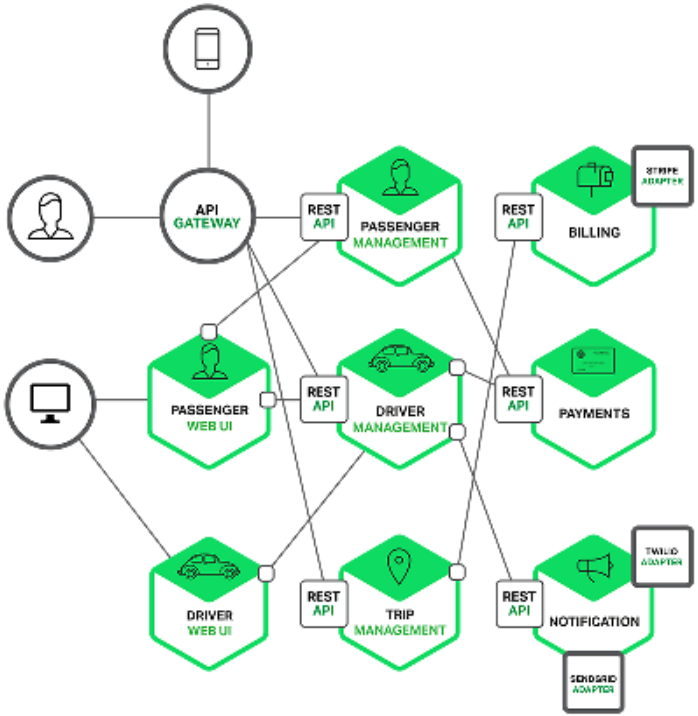
GHENT UNIVERSITY

CONCLUSIONS

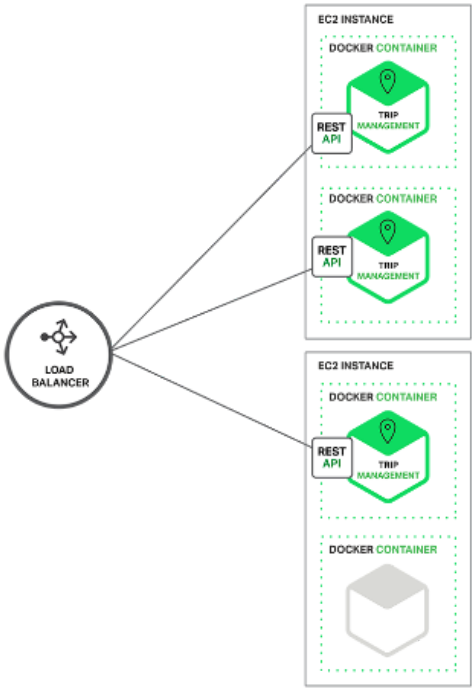
GHENT UNIVERSITY

OVERVIEW

Microservices



Containers



GHENT UNIVERSITY

CONCLUDING

Evolution to smaller micro-services

- Functional decomposition of existing application
  - Loose coupling / (business) functional cohesion
- Elastic: Scaling rapidly
- “We will fail” instead of “make me as reliable as possible”
- You build it, you run it (DevOps approach)

Containers = go-to enabler of micro-services

- Rapid scaling
- Lightweight
- Designed to run anywhere

GHENT UNIVERSITY