Cloud Resources Management

Dmitrii Ustiugov
PhD candidate, the University of Edinburgh

08.11.2021, Extreme Computing course at the University of Edinburgh

In The Previous Episodes

You've learned a lot about distributed systems and frameworks

- Large-scale parallel computations
- Big data processing & management

This lecture: **Cloud computing & cloud resources management**

- A gigantic computer rental for **clients** (businesses)
- Cloud-scale OS, key components, technologies, resources management

The Cloud Servers Era

Pay a cloud provider for Infrastructure-as-a-Service (IaaS)

- Rental instead of acquisition
- Examples: AWS EC2, Microsoft Azure

Provider is responsible for acquiring & maintaining the computers

Client manages the “cloud” infrastructure

- E.g., decides when to rent more/fewer computers
- Pay-for-what-you-use model (evolving all the time)

Requirements for Running Cloud Services (Client side)

High availability

- Low response time
- Data durability

Resources scaling

- Adjust to dynamic traffic changes

Security

- Isolation across services
- Isolation between services and the provider

Datacenters

Dmitrii Ustiugov - EASE Lab

Large scale

- 10s of thousands of compute nodes

Provider-managed

- Power supply & cooling
- Hardware and software upgrades

Geographically distributed

- Clients rent resources around the globe

Google's datacenter campus

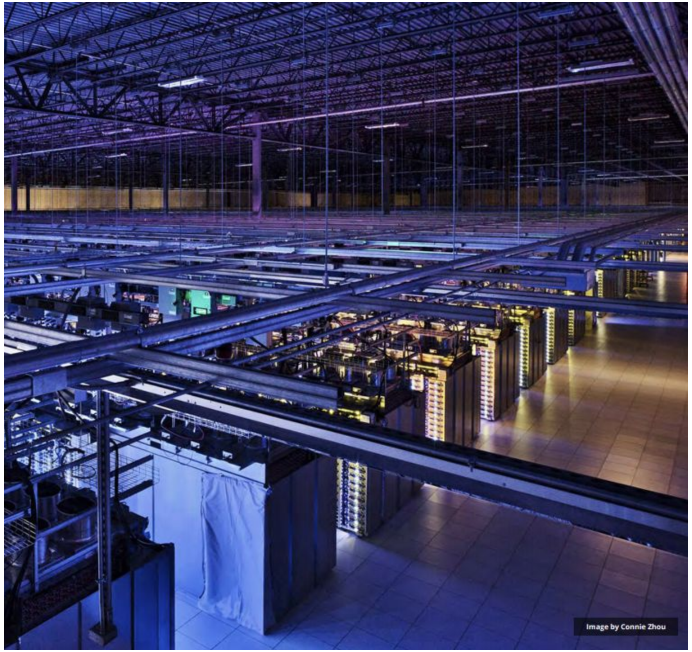
Inside a Datacenter

Collection of cheap and standard components

- Racks of compute and storage nodes
- Network

Providers manages the bare-metal infrastructure

- Power and cooling
- Software and hardware infrastructure upgrades
- Security



Adopted from John Wilkes, Google

Client View

Clients submit compute tasks

- Job
  - E.g., financial report generation
- Interactive service
  - E.g., a web form, social networks

Clients list resource requirements per task

- If interactive, for how long to run?
- **Software**: OS type, language runtime, …
- **Hardware**: CPU, memory, disk, network speed, …

```
job hello_world = {
  runtime = { cell = 'ic' }                 // Cell (cluster) to run in
  binary = '.../hello_world_webserver'      // Program to run
  args = { port = '%port%' }                // Command line parameters
  requirements = {        // Resource requirements (optional)
    ram = 100M
    disk = 100M
    cpu = 0.1
  }
  replicas = 10000      // Number of tasks
}
```

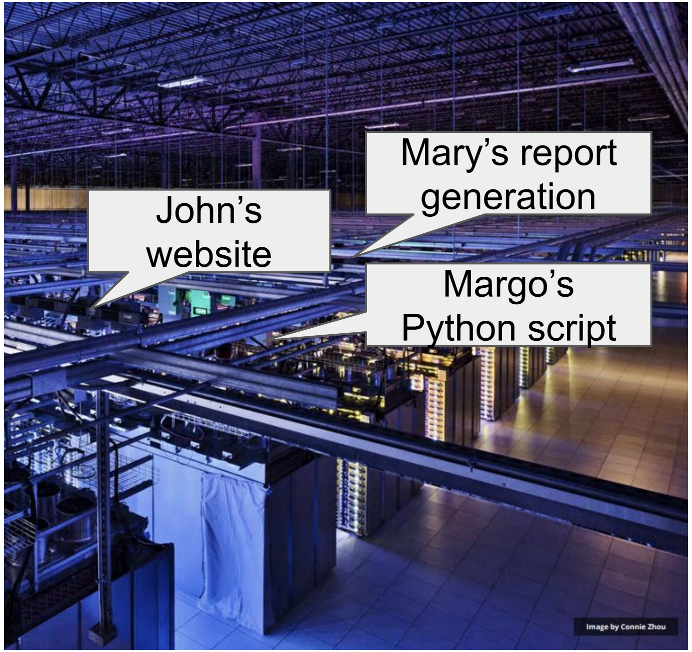
Adopted from John Wilkes, Google

Provider View

Millions of compute tasks to schedule per day

Challenges

- All client-side requirements:
  - High availability, fault tolerance, scaling, security, …
- How to minimize a provider's costs
  - Utilize all resources efficiently
  - Power off everything not in use



Adopted from John Wilkes, Google

Dmitrii Ustiugov - EASE Lab

Cluster Scheduling with Kubernetes (k8s)

Control plane (master node)

- Centralized scheduler
- Highly available

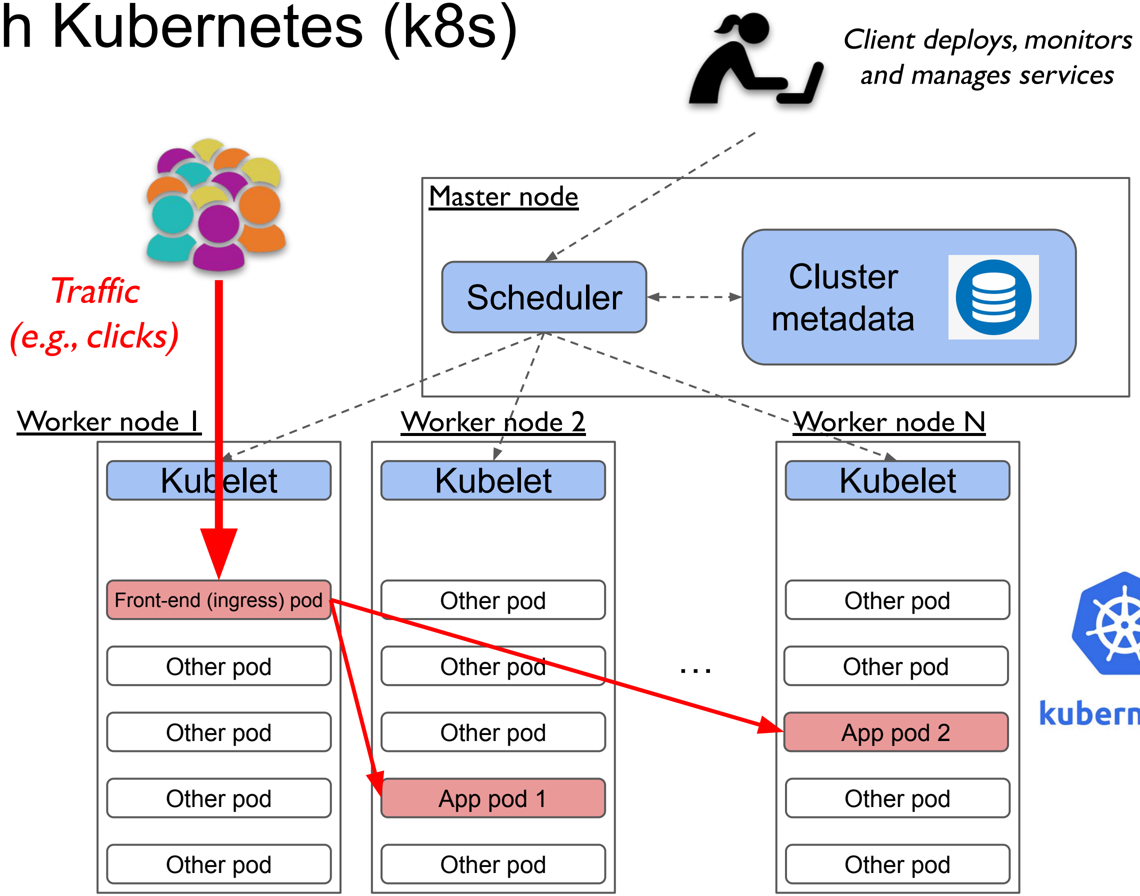
Services (i.e., the apps)

Pods (instances of a service)

- Units of scheduling
- Units of scaling

Worker nodes

- Hosts for pods
- Per-worker kubelet manages pods



kubernetes

Dmitrii Ustiugov - EASE Lab

Co-location at Google with Borg (k8s predecessor)

Tight **packing** of jobs on each node

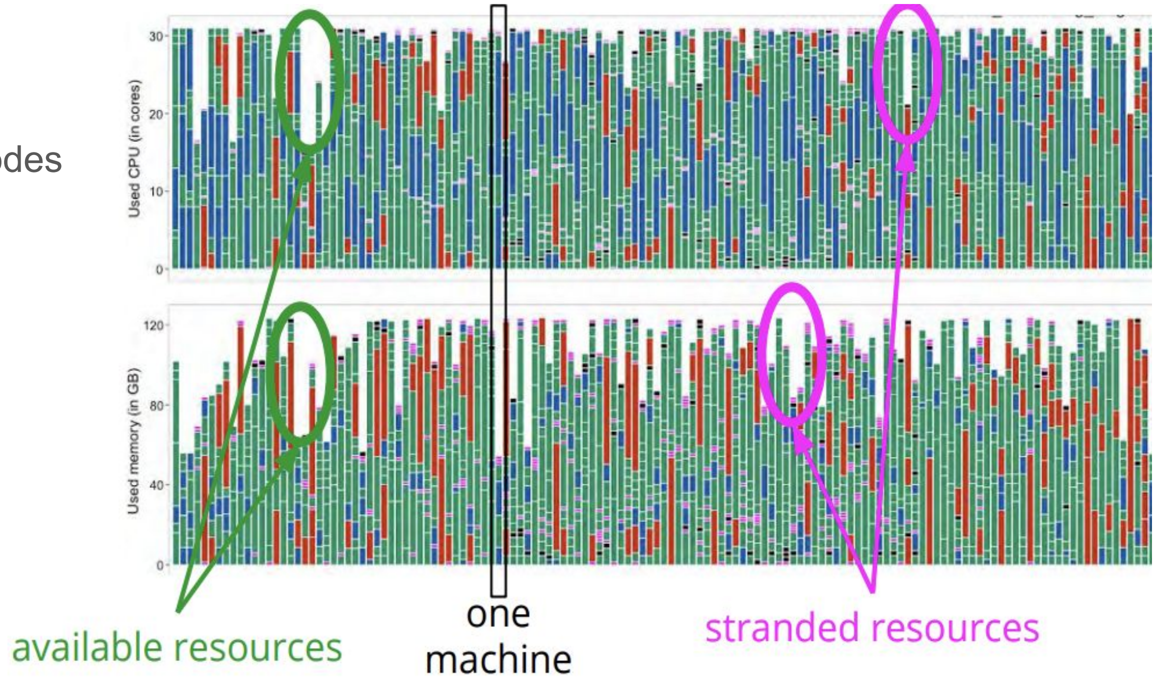
Minimize the number of **underutilized** nodes

Minimize **stranded** resources

- Nodes with free memory but no free cores

Continuous cluster nodes monitoring

- Resource reclamation
- Health checks per node & pod



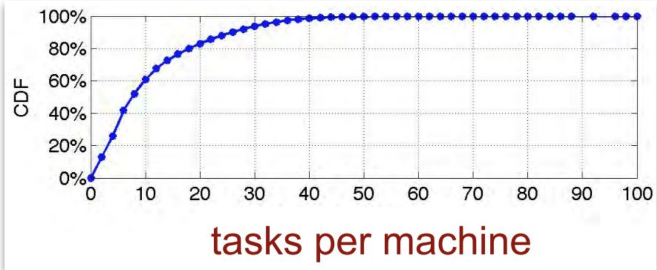
Dmitrii Ustiugov - EASE Lab

Q&A: Datacenter

Datacenter Resources Rental Challenges

What is the **rental granularity** and for how long?

- Renting for days is wasteful
  - E.g., need more resources during the day, less at night
- Renting entire bare-metal nodes is too expensive
  - Many compute tasks are too small or short



CPI^2 (Google, 2013)

Co-locating compute tasks seems natural, but:

- How? Is it secure? Is it possible to satisfy all client requirements?

"Careful" co-location of compute tasks is necessary

Requirements

Client side

High availability

- Low response time
- Data durability

Resources scaling

- Adjust to dynamic traffic changes

Security

- Isolation across jobs
- Isolation between a job and the provider

Provider side

High resource utilization

kubernetes

- Nodes either in use or powered off
- Aggressive co-location of jobs

Minimal infrastructure overhead

- Performance & memory

Security

- Isolation across clients, clients and provider

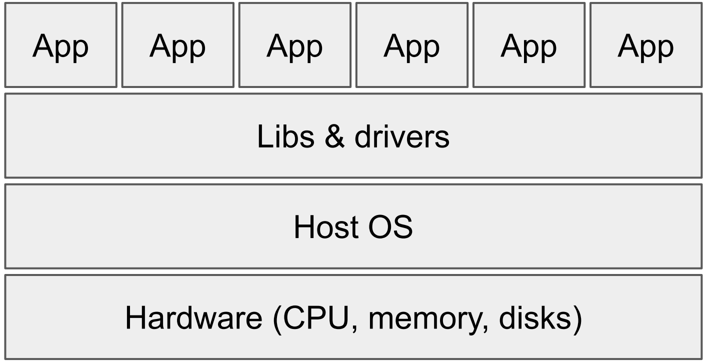
Isolation Technologies

Is there a solution that combines both?

Isolation Spectrum Extremes

Dmitrii Ustiugov - EASE Lab

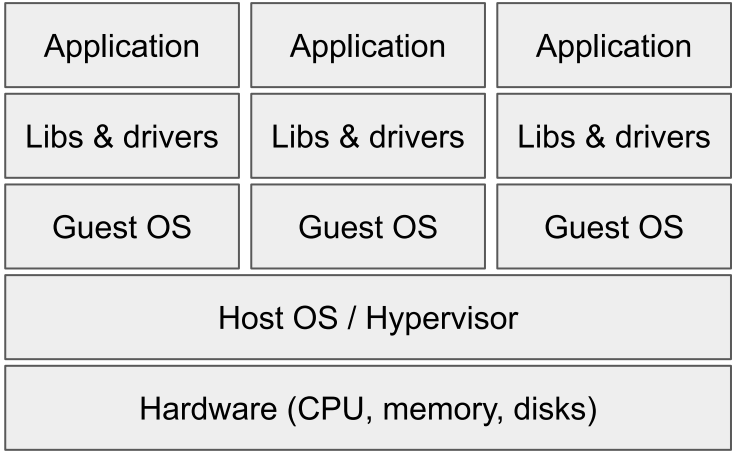
Processes



Low overhead 👍
Vast attack surface 👎

- Shared host OS kernel, CPU, NICs, disks
- May crash the host OS (the blue screen of death)

Virtual machines



High degree of isolation 👍
High overhead 👎

None of the two extremes suffice

The Two Roads towards a “Perfect” Isolation Technology

Dmitrii Ustiugov - EASE Lab

Make process isolation stronger

Namespace abstractions

- Virtualize the process tree
- Virtualize the network
- Virtualize the filesystem (“chroot”)

Filter system calls to the host kernel

- Which syscalls? With which arguments?

Make VMs leaner

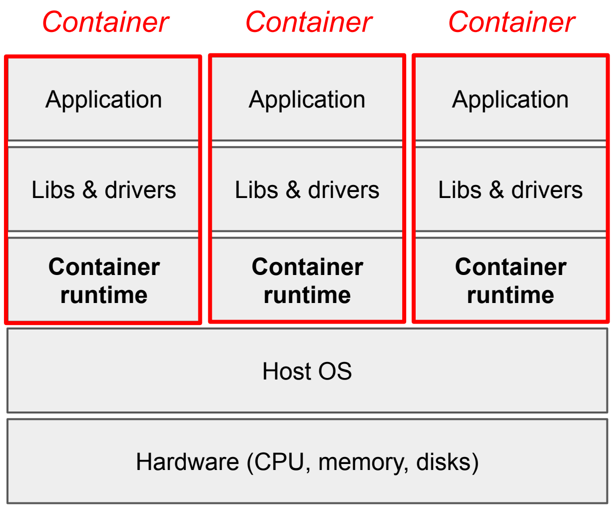
Is **guest OS** necessary?

- “Double” memory allocation in host & guest
- “Double” scheduling in host & guest
- …

Need to emulate **all possible** devices?

- E.g., is a 10-years old NIC still relevant?

Containers: Towards Secure Processes



Originated from Linux cgroups & namespaces, zones in Solaris OS, etc.

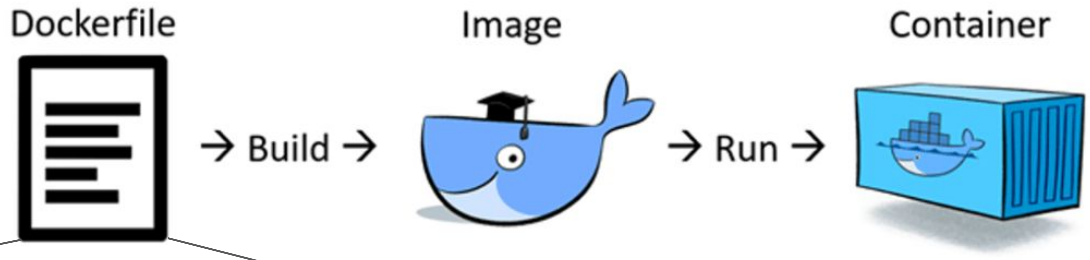
A container is a **combination of technologies**:

- Namespaces:
  - Isolated **PID tree**: All processes forked from container's private PID 1
  - Virtual **network**: Each container has its own IP address
  - Isolated **root filesystem**
- Resource groups (e.g., Linux cgroups)
  - E.g., limiting CPU quota and physical memory allocation

Docker revolution through **automation**

- Easy building & deploying using existing technologies
- AppArmor for syscall filtering ("jailing")

Docker as a Deployment-Native Solution




```
FROM ubuntu:20.04

RUN apt update && \
    apt install python3-pip <...> && \
    pip3 install <...>

COPY my_python_code /path

CMD [“python”, “/path/main.py”]
```

Clients specify their jobs with a dockerfile

- “A recipe” for constructing a Linux container

Docker images are built using a union filesystem (Linux unionFS)

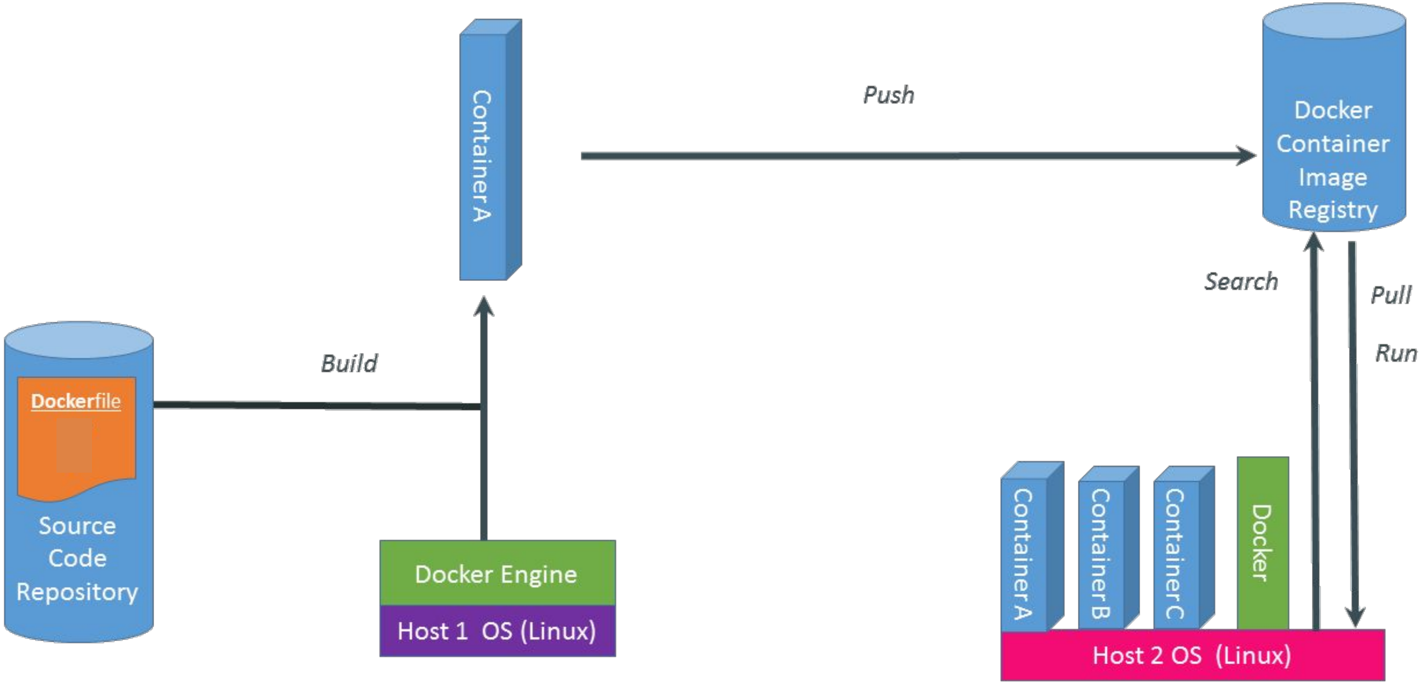
- Each line in a dockerfile is a read-only filesystem layer

Images are ready-to-run on any host with Linux and compatible kernel

Dmitrii Ustiugov - EASE Lab

Docker Workflow

Dmitrii Ustiugov - EASE Lab



Lightweight Virtualization with AWS Firecracker Hypervisor

Support stock Linux guest OS

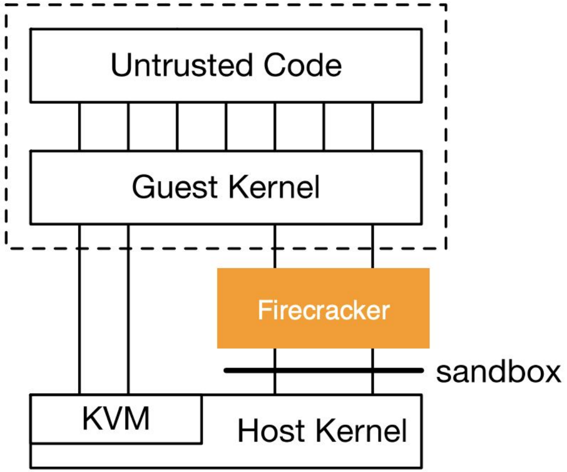
- No compromises in security and compatibility

Offload duplicate functionality to host OS & CPU hardware

- Kernel-based Virtual Machine (Linux KVM)
  - Virtual CPU is a host thread
  - Guest-physical memory is host virtual memory
- Hardware extensions for virtualization
  - E.g., nested page tables: one for host, one for guest

Minimize the emulation layer

- Minimal set of emulated devices: one NIC type, one disk type



State-of-the-Art Isolation

Low overhead

Processes

High security

Virtual machines

Lightweight virtualization
+
Docker container deployment

Firecracker MicroVM:

- ➡️ No compromises in VM isolation
- ➡️ 125ms VM startup time
- ➡️ <5MB memory overhead

Q&A: Containers & VMs

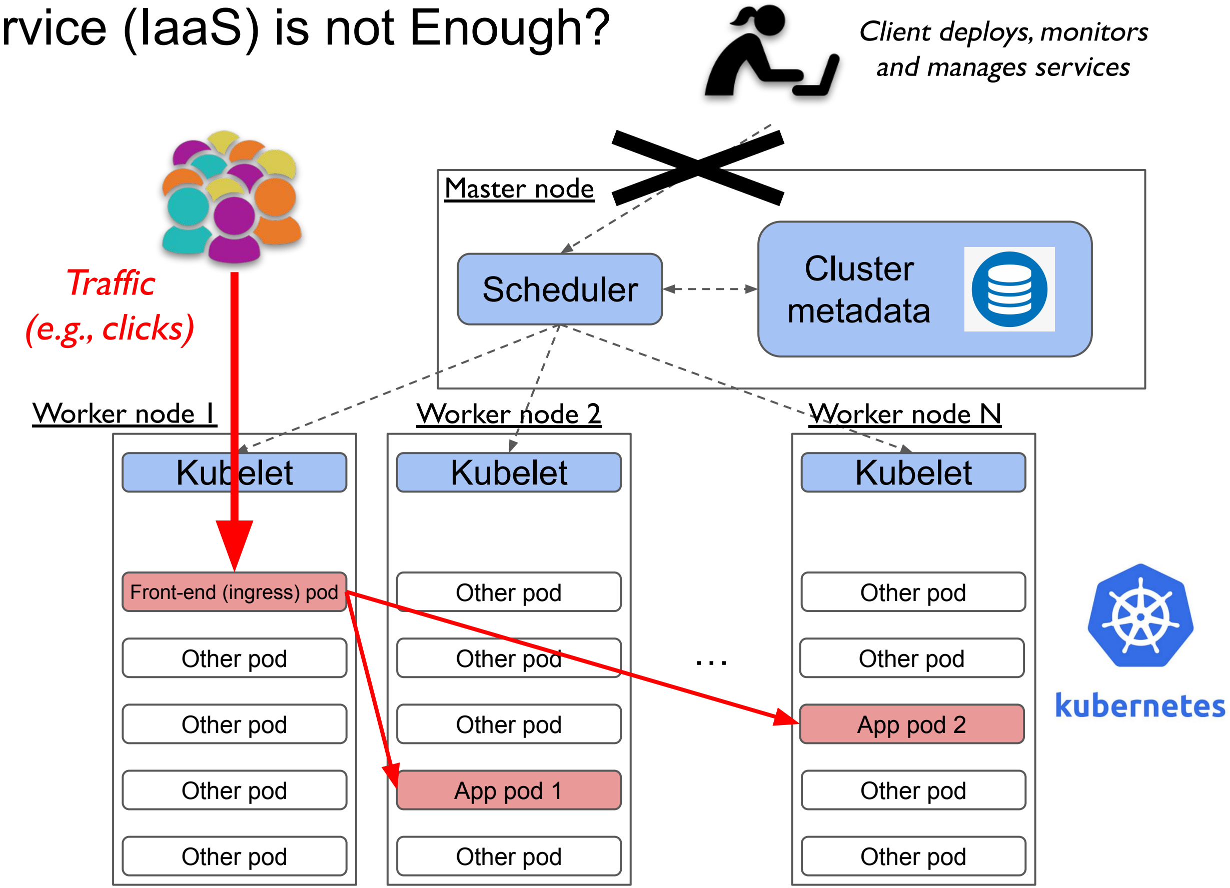
Why Infrastructure-as-a-Service (IaaS) is not Enough?

Provider maintains the datacenter

- Acquisition & operation
- Power
- Hardware & software upgrades

Client still manages the infrastructure

- Scale per-application resources
- Rent VMs
- Request CPU, memory, disk, etc.



Infrastructure management puts a significant burden on a client

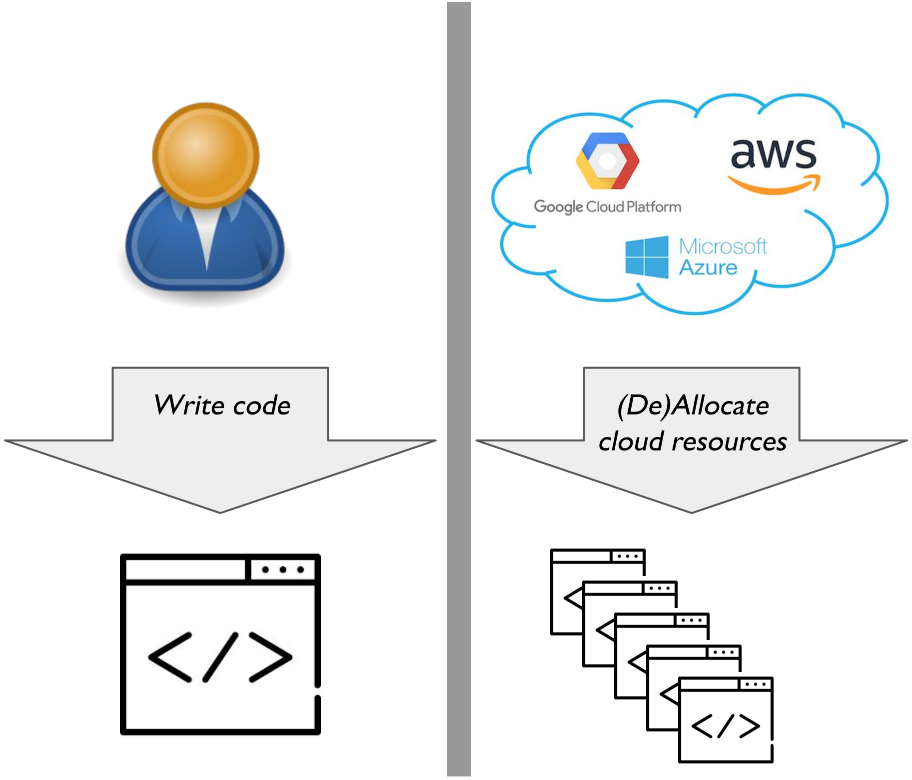
The Future of Cloud Computing is *Serverless*

Serverless programming via labor division

- Clients write code
- Providers adjust cloud resources

Pay-as-you-go pricing model

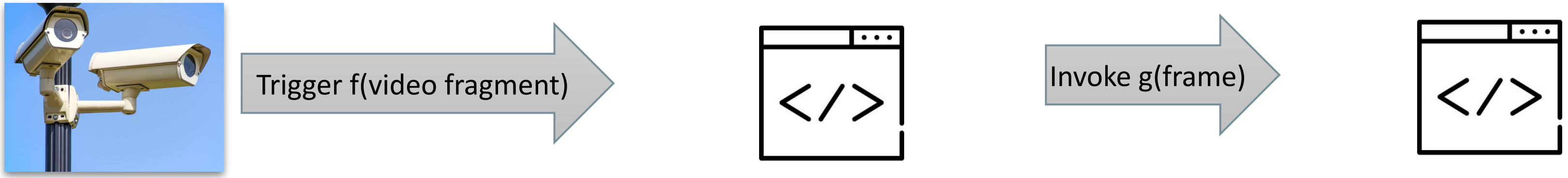
- Per {1-millisecond x 1-megabyte} billing
- Free of charge when not in use



Service Developer's Perspective

<u>**f()**:Video frame decode</u>

<u>**g()**: Object recognition</u>

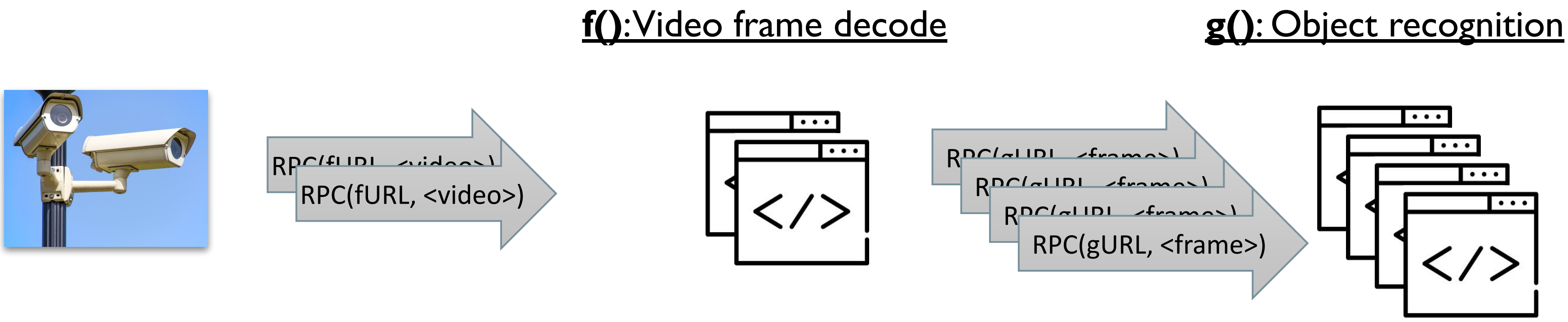


Write each function's business logic

Compose functions via event triggers and RPC calls

Serverless premise: "No need to think about servers"

Provider’s Perspective



Function instances are *ephemeral*, spawned on demand

- 0 to ∞ instances of each function
- Provider to balance load and spawn / tear down instances

Serverless reality: Great for users, challenging for providers

Serverless behind the Scenes (Amazon Lambda)

Functions are deployed as lightweight VMs (MicroVMs)

- Packaged as **Docker images**
- Function code
  - Provider's runtime: **HTTP-level** server
  - Client-defined handle in a **high-level** language (Python, NodeJS, Java, etc.)

Enable elastic scaling via compute/state separation

- Functions are **stateless:** Any instance can handle **any** invocation of the function
- Must be composed with **conventional storage services** and **databases**

Knative: Serverless Under the Hood

Client deploys a function to FaaS

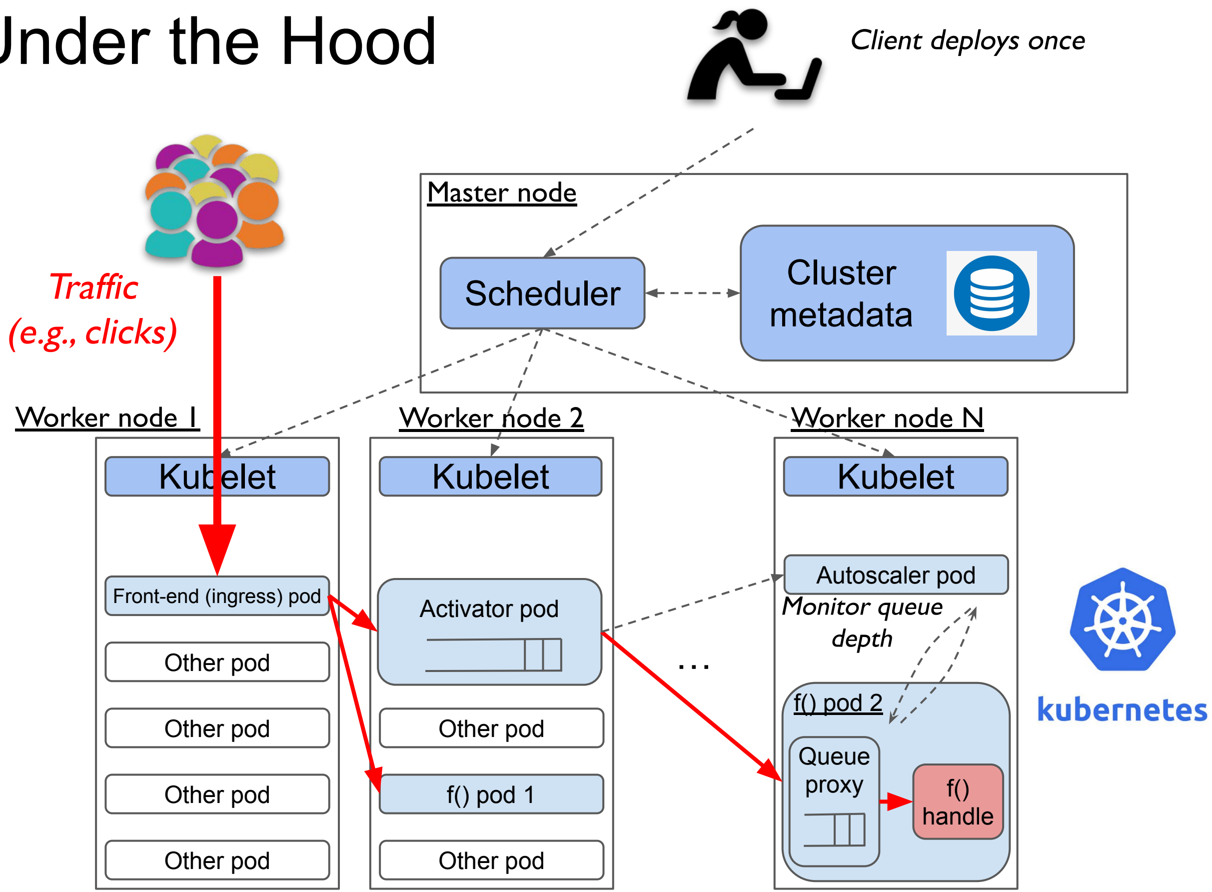
- Provides the code
- Defines the triggers

Provider scales a functions based on

- Invocation traffic to ingress
- Adjusting the instance number to the arrival rate

A function instance is a pod, containing

- A function handle (VM or container)
- Queue-proxy monitors the load and reports to the autoscaler service



The client is only responsible for the function handle, rest is by the provider

Recap: The Evolution of Cloud

	Pre-cloud	Cloud (IaaS)	FaaS (serverless)
App	Monolith	(Micro)services	Functions
Runtime & Guest OS			
Scaling			
Host OS			
Bare metal compute nodes			
Networking			
Storage			

Client's responsibility

Provider's responsibility

Takeaways

Cloud is a huge computer **rental** system

Datacenter managed as a **pool of resources**

- **User requirements**: High availability, scalability, security
- **Provider goals**: High utilization, minimal infrastructure overheads

The future of cloud computing is **serverless**

- **Labor division**: Users write code, providers scale the resources
- **Function-as-a-Service** programming model
- **Autoscaling** of function instances & **pay-for-what-you-use** billing

Q & A

Backup

Cluster Scheduling with Kubernetes (k8s)

Dmitrii Ustiugov - EASE Lab

Control plane (master node)

- Centralized scheduler
- Highly available

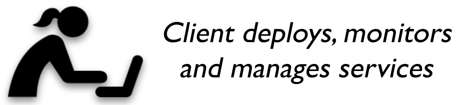
Services (i.e., the apps)

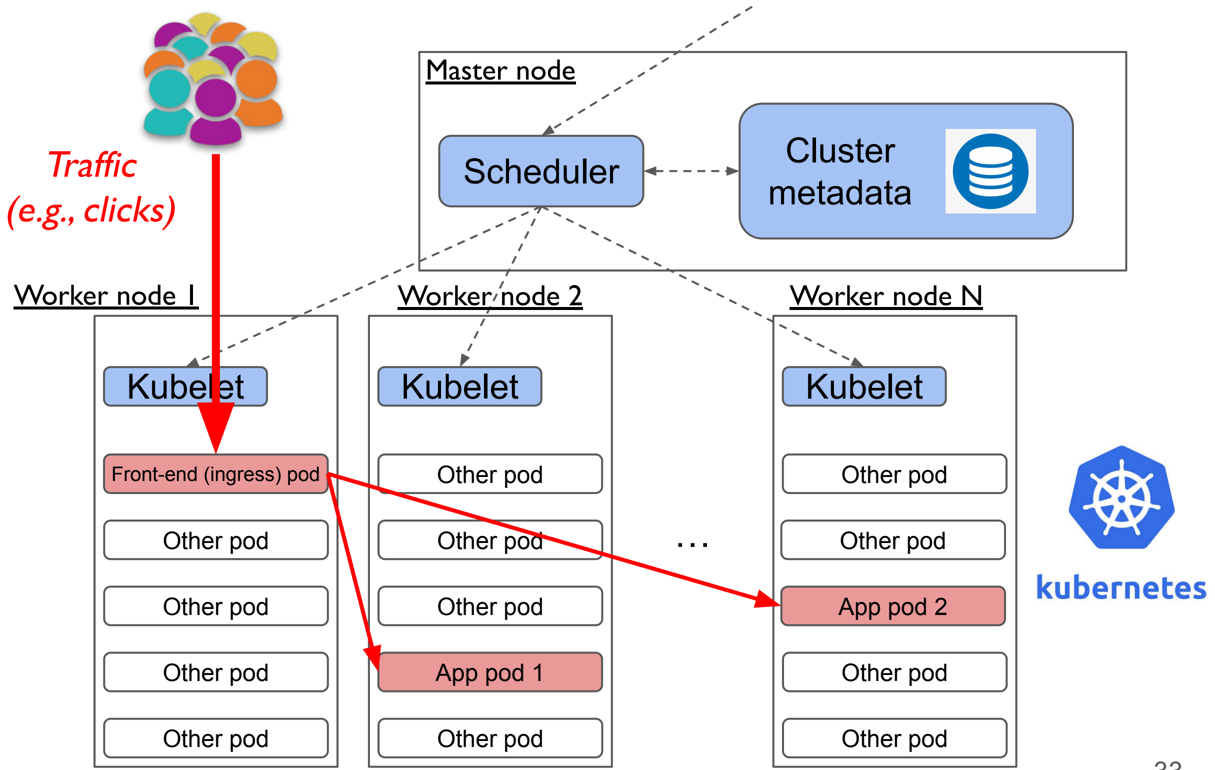
Pods (instances of a service)

- Units of scheduling
- Units of scaling

Worker nodes

- Hosts pods
- Kubelet manages pods per request from the scheduler





Knative: Serverless Under the Hood

Client deploys once

Client deploys a function to FaaS

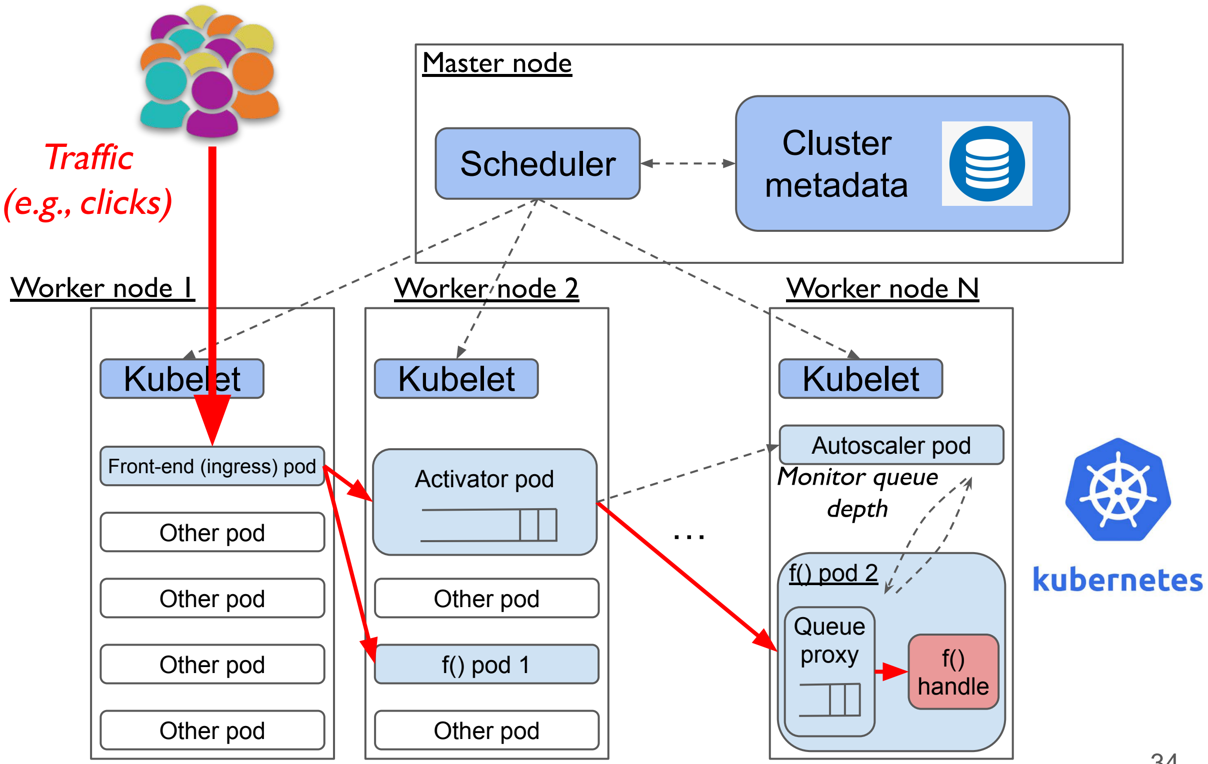
- Provides the code
- Defines the triggers

Provider scales a functions based on

- Invocation traffic to ingress
- Adjusts the instance number to the arrival rate

A function instance is a pod, containing

- An isolated function (VM or container)
- Queue-proxy that monitors the load and reports to the autoscaler service



Cluster Scheduling with Kubernetes (k8s)

Control plane (master node)

- Centralized
- Highly available

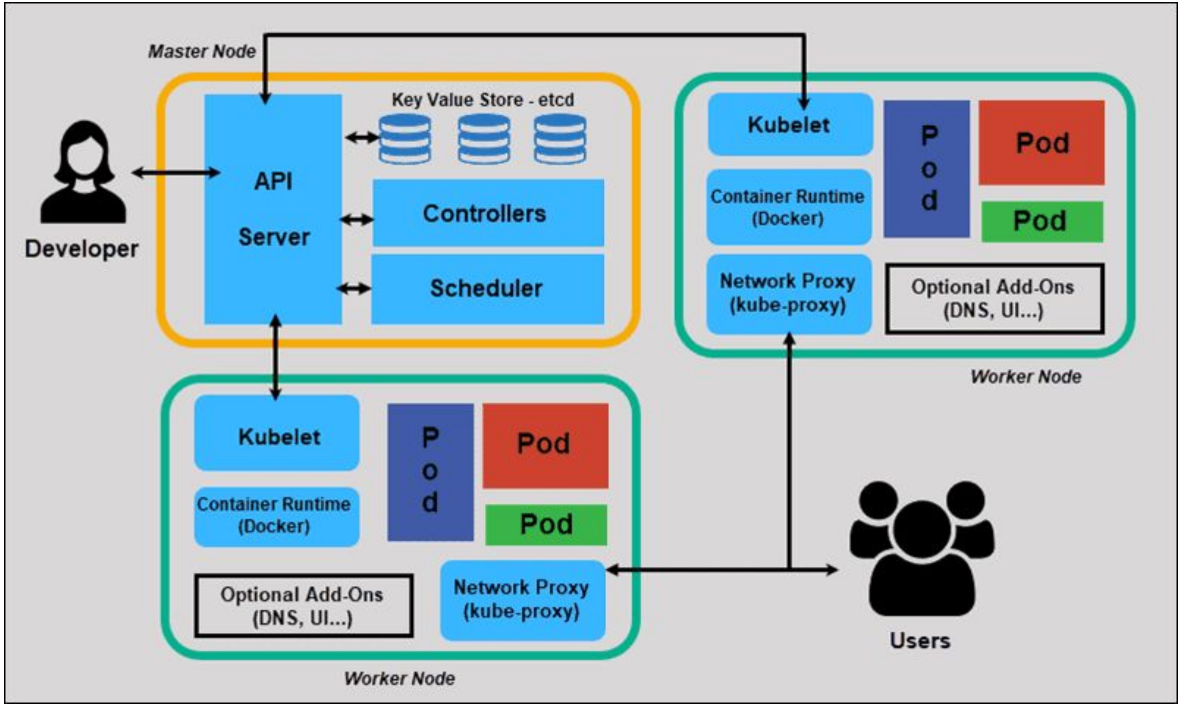
Services (app building blocks)

Pods (instances of a service)

- Units of scheduling
- Units of scaling

Worker nodes

- Hosts pods



Dmitrii Ustiugov - EASE Lab

Cloud Evolution: Recap

2. What is IaaS – PaaS – FaaS - SaaS?

Mobiliya

Private Cloud	Infrastructure (as a service)	Platform (as a service)	Function (as a service) (serverless arch)	Software (as a service)
Functions	Functions	Function	Functions	Functions
Data	Data	Data	Data	Data
Application	Data	Application	Application	Application
Runtime	Runtime	Runtime	Runtime	Runtime
Backend Code	Backend Code	Backend Code	Backend Code	Backend Code
OS	OS	OS	OS	OS
Virtualization	Virtualization	Virtualization	Virtualization	Virtualization
Server Machines	Server Machines	Server Machines	Server Machines	Server Machines
Storage	Storage	Storage	Storage	Storage
Networking	Networking	Networking	Networking	Networking

Public Cloud Provider - responsibility

Application Writer - responsibility

Awesome Vizualisation picked from : Ref : http://www.slideshare.net/manuel_silveyra/austin-cf-meetup-20150224/3

PS: We expect Container as a Service term in 2017-18 too, there is a separate section on it later

23

Business and Computing

Today, business is digital: IT as a service, marketing campaigns, social nets, …

Say, you are going to open a new bakery

Bakery goods

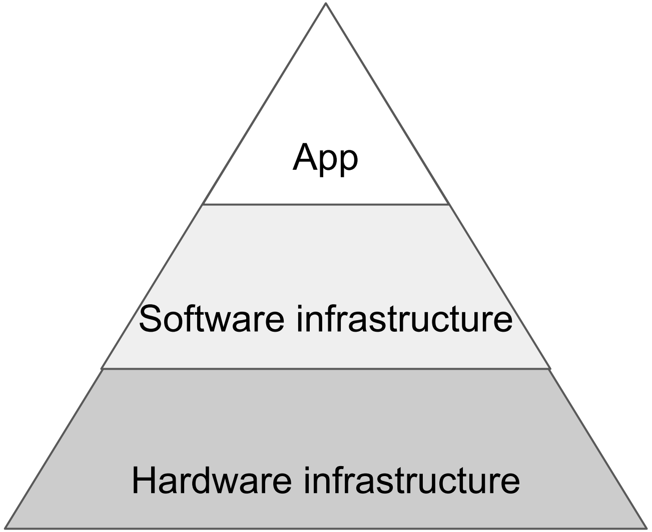
Users

Food delivery app

How easy is to build an online application?

How to Build & Maintain Your Online Service?

The Cost Pyramide



Building an online application is hard

How to Deploy Your Digital Infrastructure?

How do you build an online service in

- Pre-cloud era (buy computers)
- Cloud servers era (rent computers from cloud providers)
- Serverless computing (never think of computers)

Main trend: Democratization of computing

Computing Democratization: Provider vs. Client Efforts

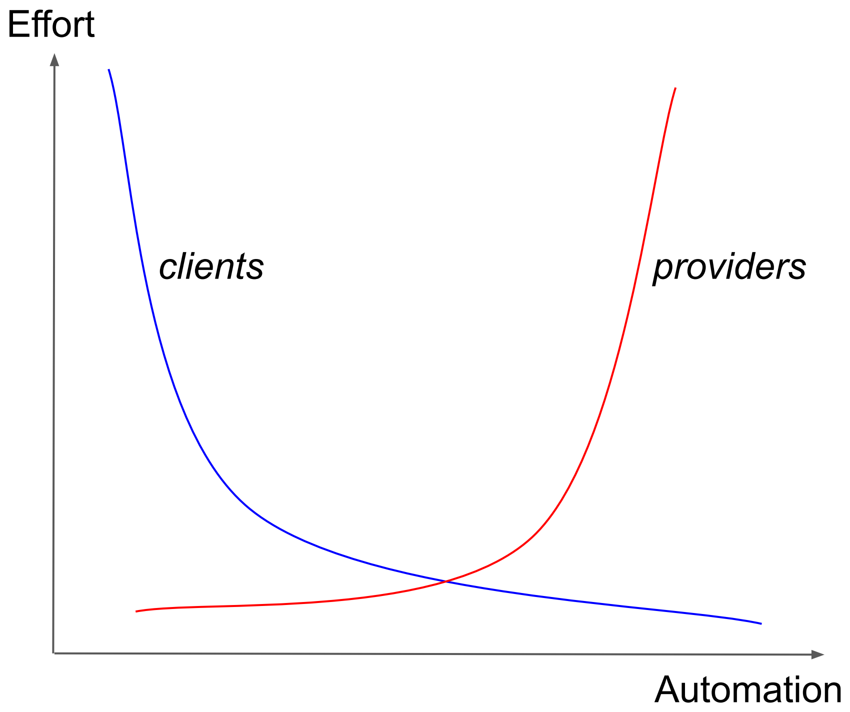
Dmitrii Ustiugov - EASE Lab

Clients demand

- Low time-to-market is king
- Choose cheap & easy infrastructure

Providers deliver

- High degree of automation
- Gradually takes over client responsibilities
  - Infrastructure acquisition & upgrades
  - Resources allocation (rental)
  - And more!



Cloud democratization demands more from the cloud providers

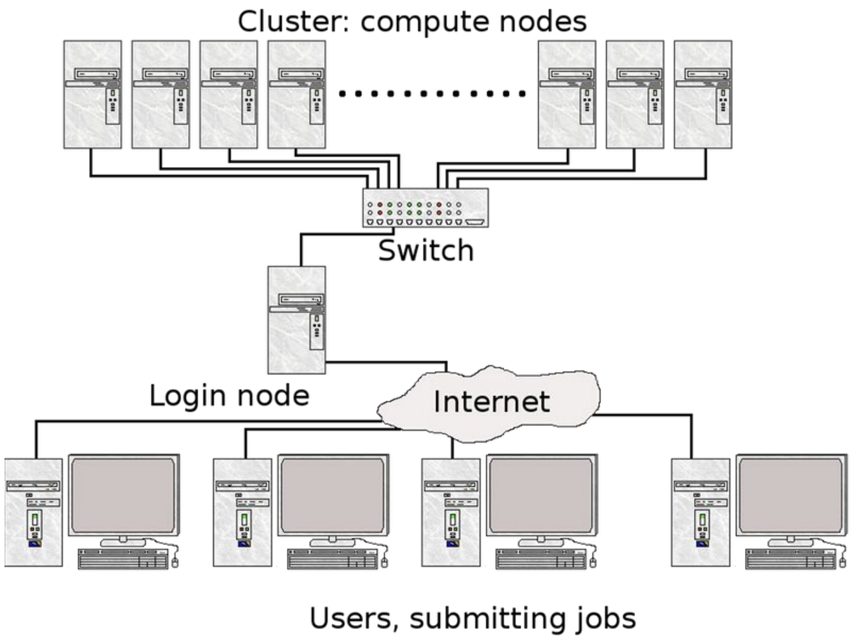
Pre-Cloud Era

Buy a compute cluster on premises

- How to assemble, connect, maintain?
- How to power up?
- …

Hire IT department that manages everything

- How to ensure low response time?
- How to fix a security breach?
- …



With on-premises infrastructure, clients are responsible for everything

Client Requirements for Computing (in Any Era)

High availability: Users always get a consistent response in time

Resources scaling: Always enough computers to handle the user traffic

Security: Across applications, applications vs. infrastructure

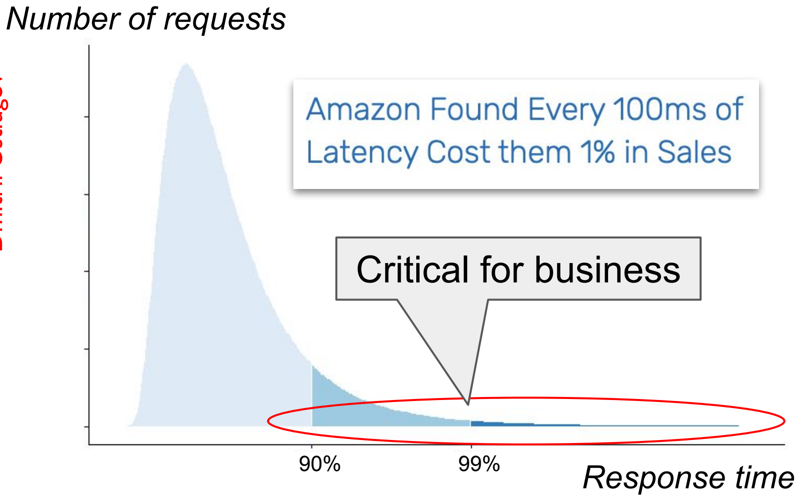
And more

High Availability

Dmitrii Ustiugov - EASE Lab

Low response time

- Low mean time is not good enough
- The goal is to satisfy 99.9..9% of customers



Data consistency and durability

- Concurrent updates
  - E.g., people write comments on Facebook
- Durable updates
  - E.g., never lose one's Instagram followers

Valid even in the presence of disasters

Fire in 500m^2 OVH datacenter, France, March 2021

Guaranteeing high availability is challenging but important

Resources Scaling

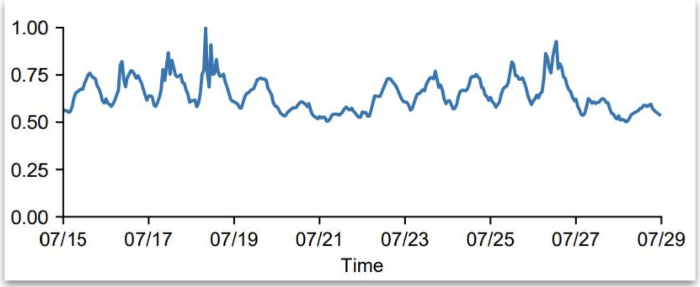
Traffic continuously changes

- Day/night, workday/weekend, celebrity posts

Resources must be provisioned for the worst case

- What is the worst case? An earthquake or a celebrity scandal?

Traffic to Microsoft Azure infrastructure [Shahrad et al, ATC'20]



Timely scaling of a service's resources is key

Security

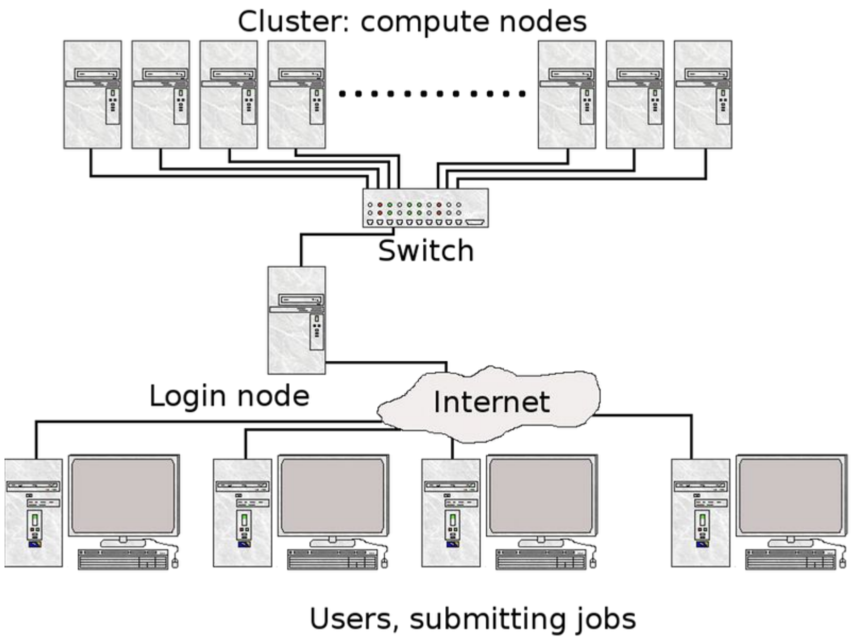
Security is a killer for business

- Compromises are usually unacceptable

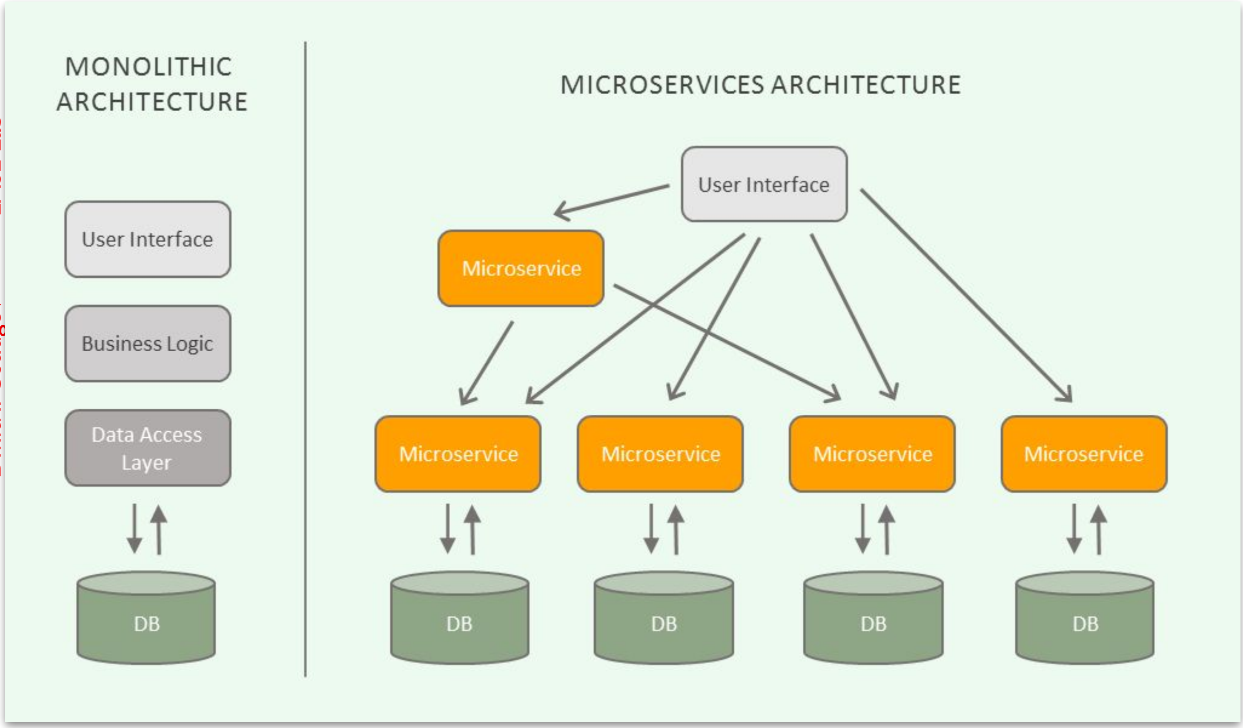
Security breaches happen regularly

- Malicious users, libraries, OS bugs, etc.
- How to avoid? Mitigate?

Security by obscurity is not the answer



How to Make Jobs Easy to Develop & Scale?



Source: https://hackernoon.com/how-microservices-saved-the-internet-30cd4b9c6230

Split services into *microservices*

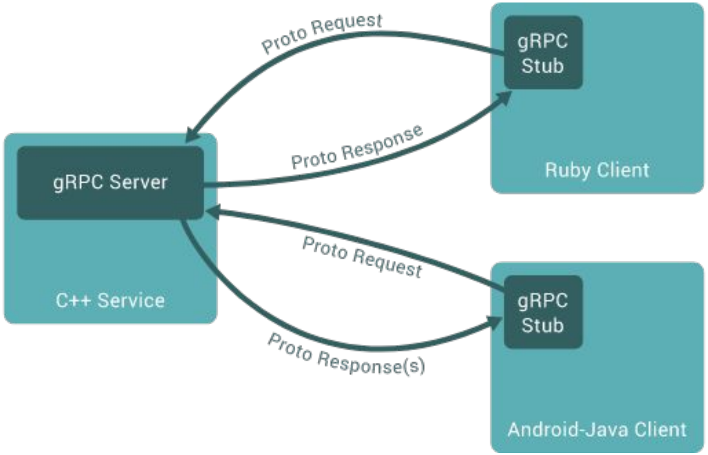
- Easy to develop & maintain
- Easy to scale
- Easy to make fast

Separate business logic & data

- **User-specific** stateless logic
- **Generic** scalable databases (provider-managed)

Microservice Architecture

[gRPC architecture, Google](#)



How to split an application into microservices?

- A microservice serves **one purpose**
- Communicate over **lingua franca** RPC fabric
  - Language-agnostic protobuf file + code generation
  - Support wide ranges of programming languages
  - Examples: gRPC (Google), Apache Thrift (Facebook)

Agile development model

- **Independent updates** of each microservice
  - A microservice's update **does not** bring entire service down
- Each microservice managed by a specific developers' team

Developing and scaling of microservices is easier than monolith apps