Cloud Computing: Evolution,Technologies, Future

Dmitrii Ustiugov
PhD candidate, the University of Edinburgh

30.03.2021, Operating Systems course at the University of Edinburgh

In The Previous Episodes

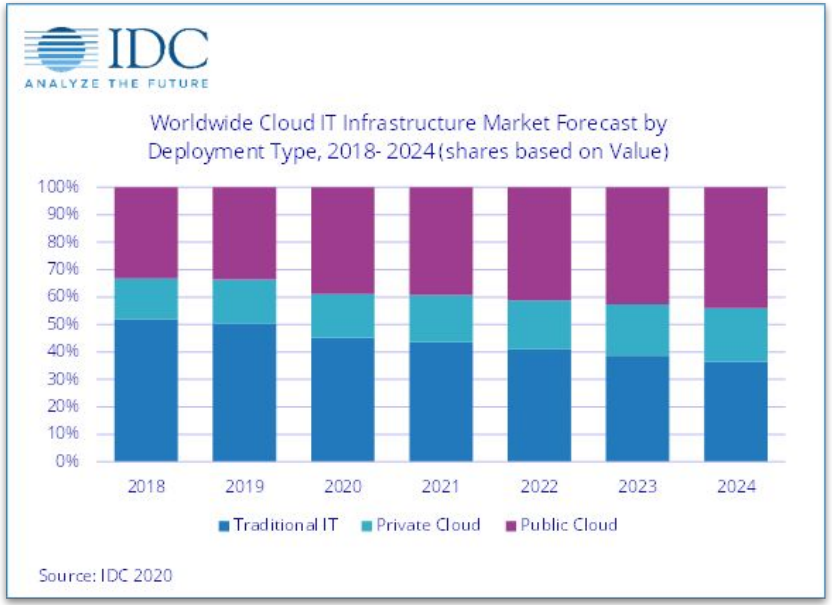
You've learned a lot about **individual** computers

- How applications work in the context of an OS
  - Key abstractions like processes and threads
  - Allocating and managing resources (e.g., memory, disk)
- Key technologies
  - Storage, synchronization, virtualization, etc.

This lecture: **Cloud computing**

- A gigantic computer rental for **clients** (businesses)
- Lots of challenges for **cloud providers**

Cloud as Today's Dominant Computing Platform



Cloud computing is prevalent

- Traditional IT decreases (-2% annually)
  - I.e., non-cloud, on premises
- Public cloud is to dominate (+10% annually)

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

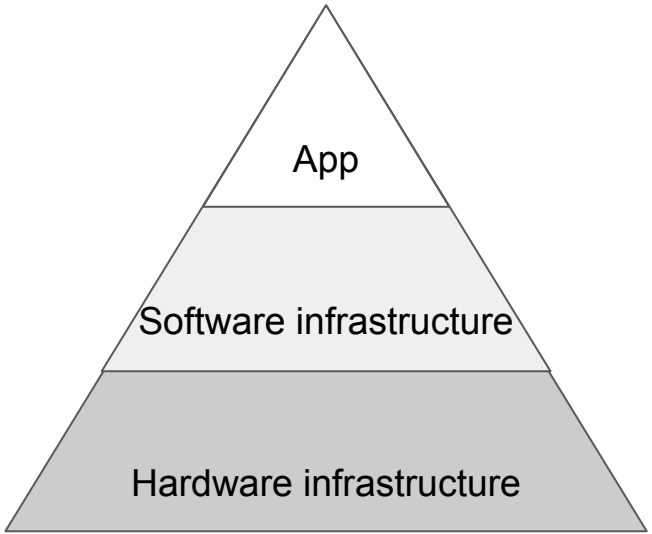
Users

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

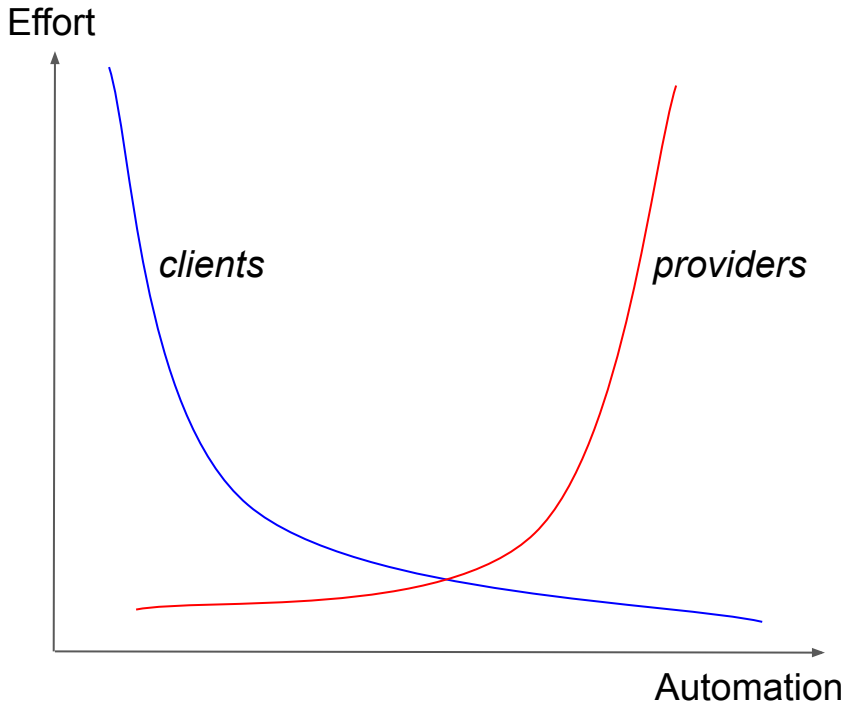
Computing Democratization: Provider vs. Client Efforts

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

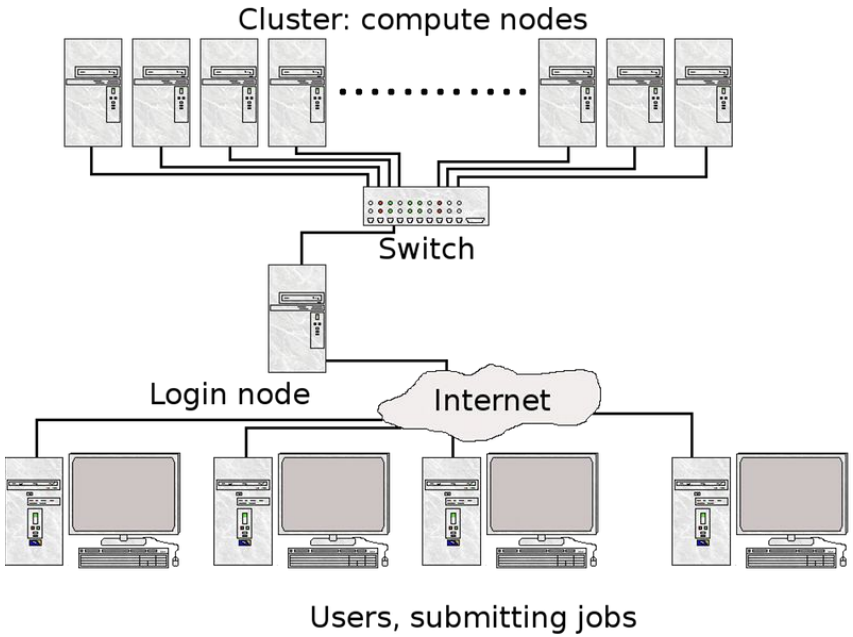
Pre-Cloud Era

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

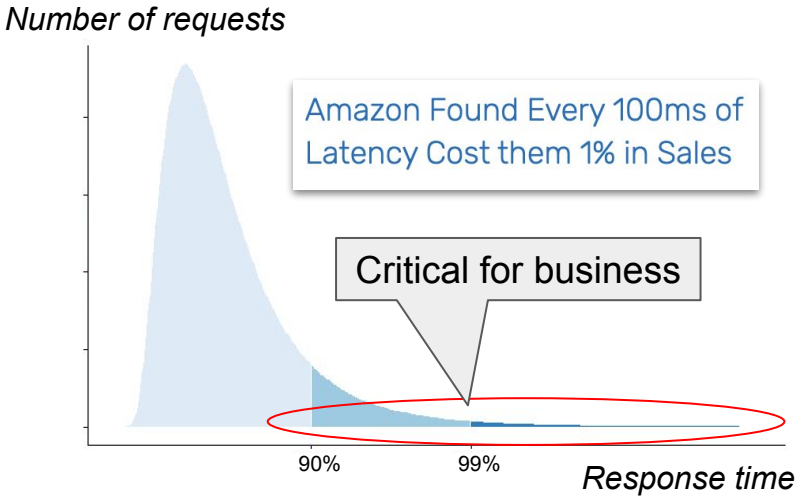
Security: Across applications, applications vs. infrastructure

And more

High Availability

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

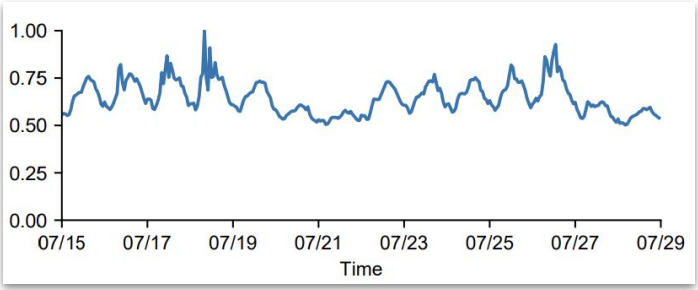
Guaranteeing high availability is challenging but important

Resources Scaling

Traffic continuously changes

- Day/night, workday/weekend, celebrity posts

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



Resources must be provisioned for the worst case

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

Timely scaling of a service's resources is key

Security

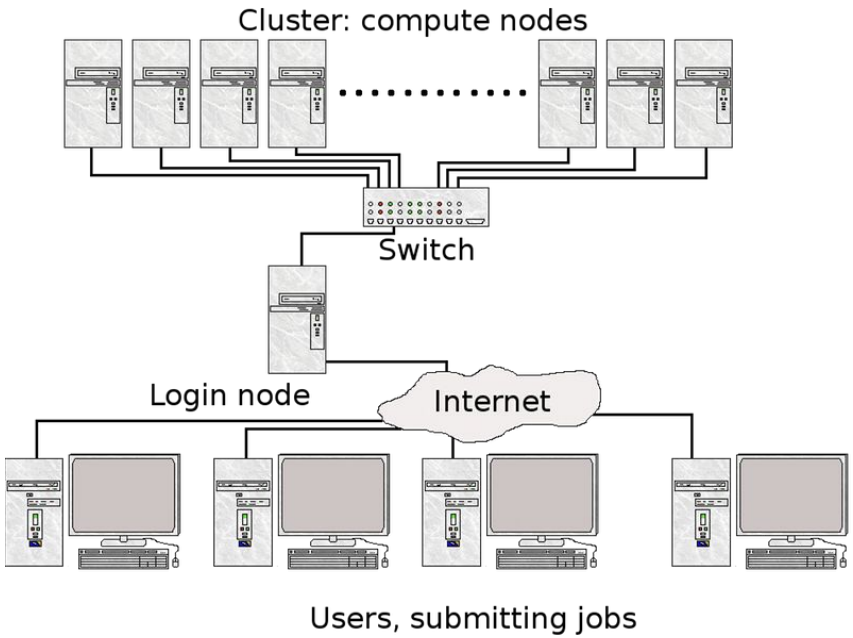
Security is a killer for business

- 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



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 stills needs to manage the “cloud” infrastructure

- E.g., decides when to rent more/fewer computers

Datacenters

Large scale

- 10s of thousands of compute nodes

Provider-managed

- Power supply
- Hardware and software upgrades

Geographically distributed

- Clients rent resources around the globe
- Recall the high-availability requirement

Google's datacenter campus

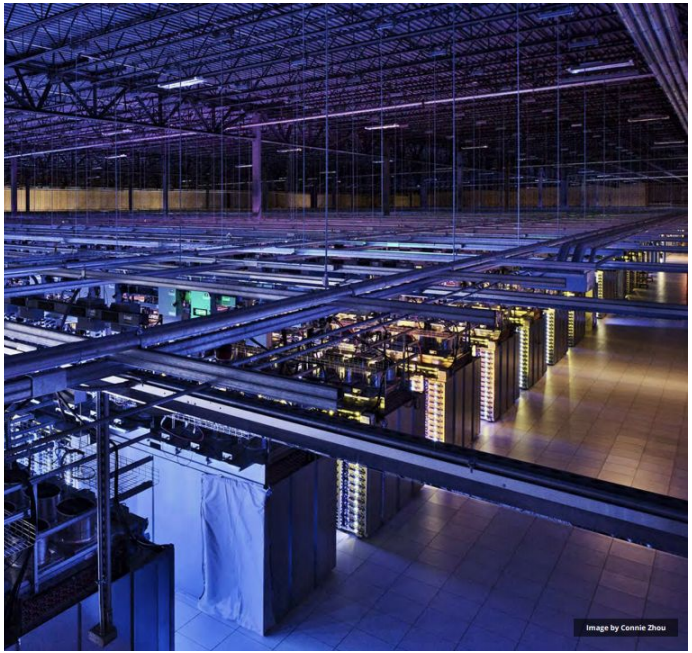
Inside a Datacenter

Collection of cheap and standard components

- Racks of compute and storage nodes

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

```
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

Clients list resource requirements per task

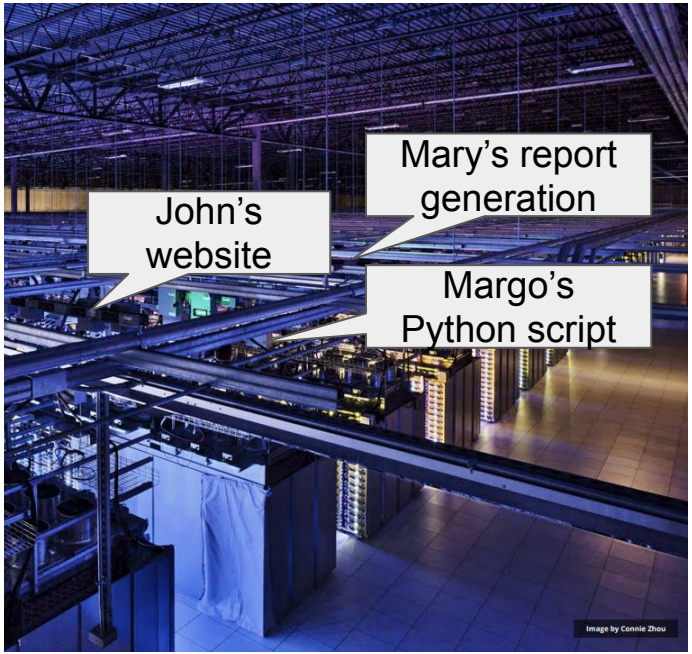
- If interactive, for how long to run?
- **Software**: OS type, language runtime, …
- **Hardware**: CPU, memory, disk, network speed, …

Provider View

Millions of compute tasks to schedule

Challenges

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



Adopted from John Wilkes, Google

Cluster Scheduling with Kubernetes (k8s)

Control plane (master node)

- Centralized scheduler
- Highly available

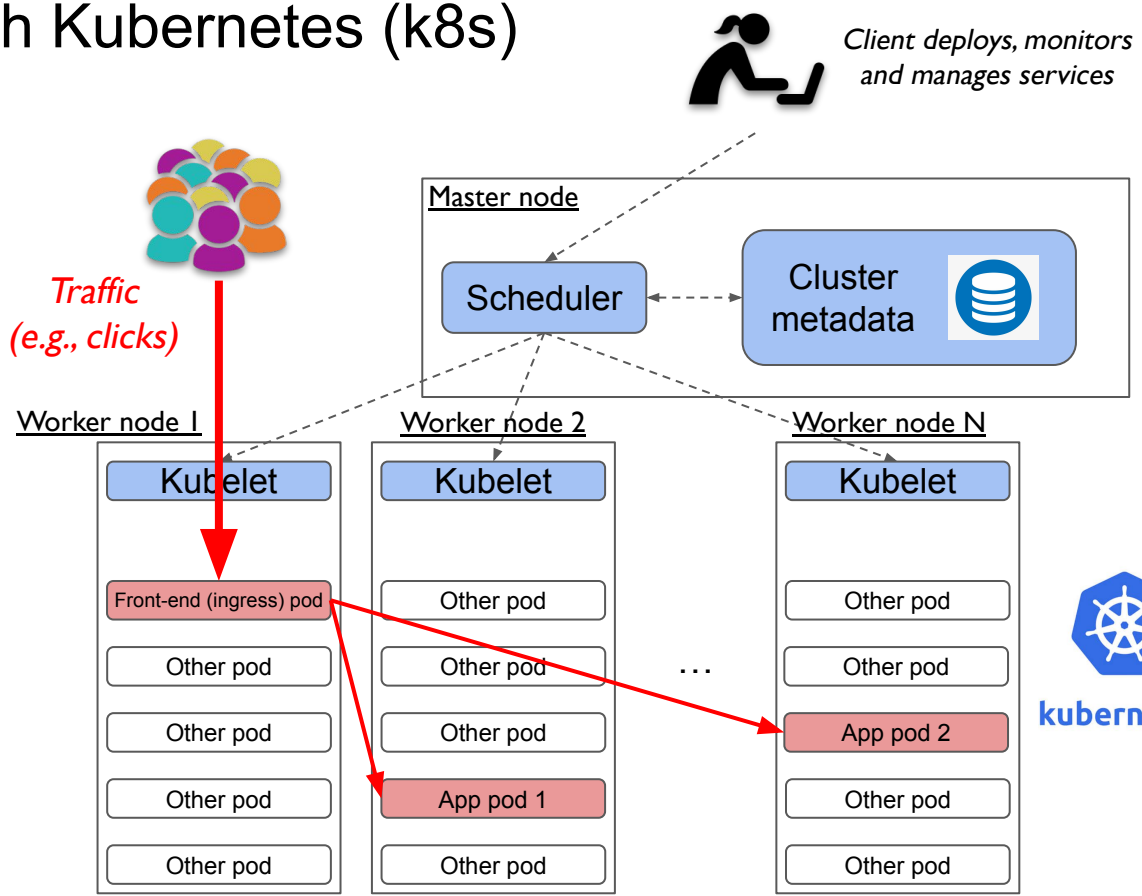
Services (i.e., the apps)

Pods (instances of a service)

- Units of scheduling
- Units of scaling

Worker nodes

- Hosts for pods
- Per-worker kubelet manages pods



Co-location at Google with Borg (k8s predecessor)

Tight packing of jobs on each node

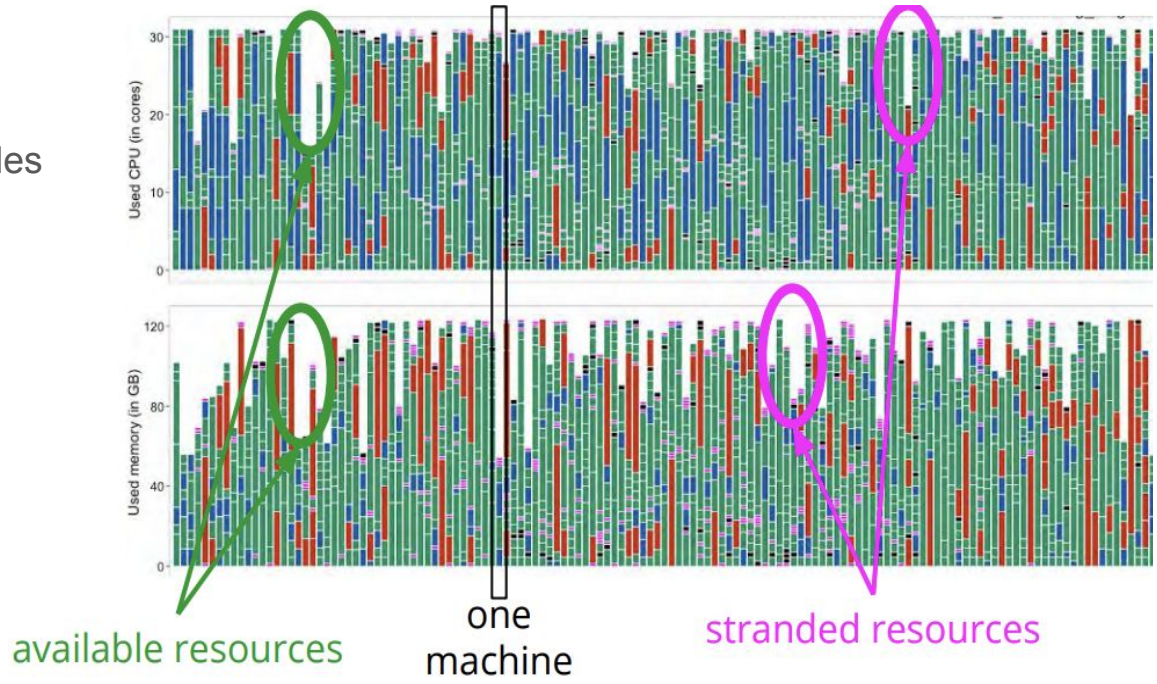
Minimize the number of underutilized nodes

Minimize stranded resources

- Nodes with free memory but no free cores

Continuous cluster nodes monitoring

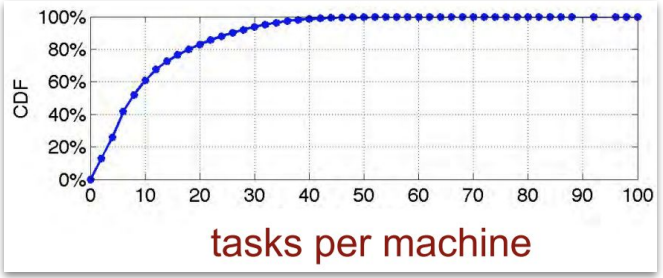
- Resource reclamation
- Health checks per node & pod



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 

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

Minimal infrastructure overhead

- Performance & memory

Security 

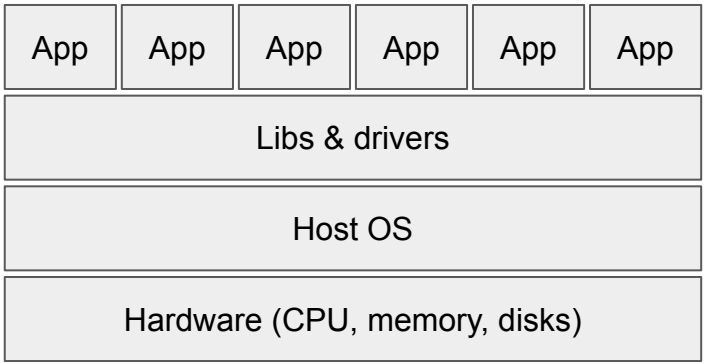
- Isolation across clients, clients and provider

Isolation Technologies

Is there a solution that combines both?

Isolation Spectrum Extremes

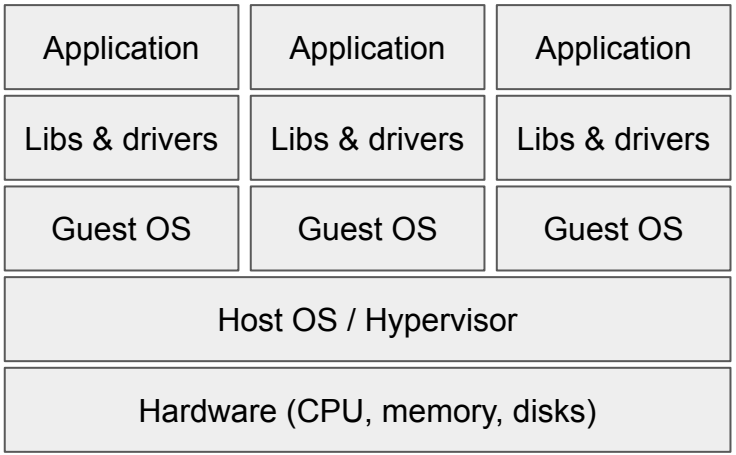
Processes



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

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

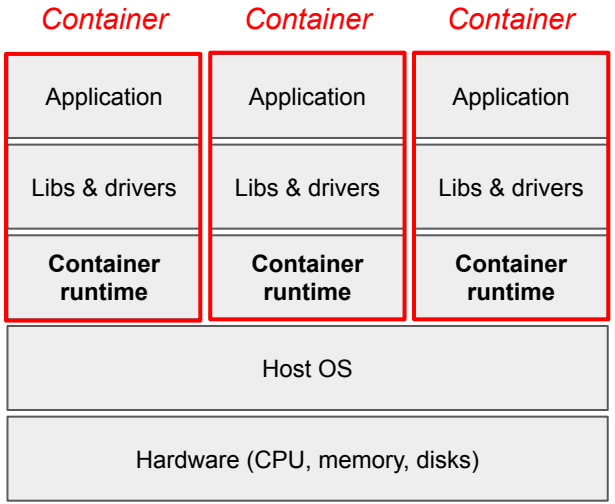
Is **guest OS** necessary?

- “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.

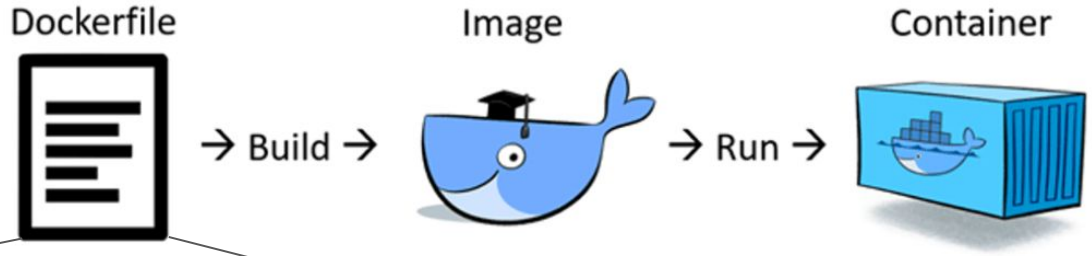
A container is a **combination of technologies**:

- 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

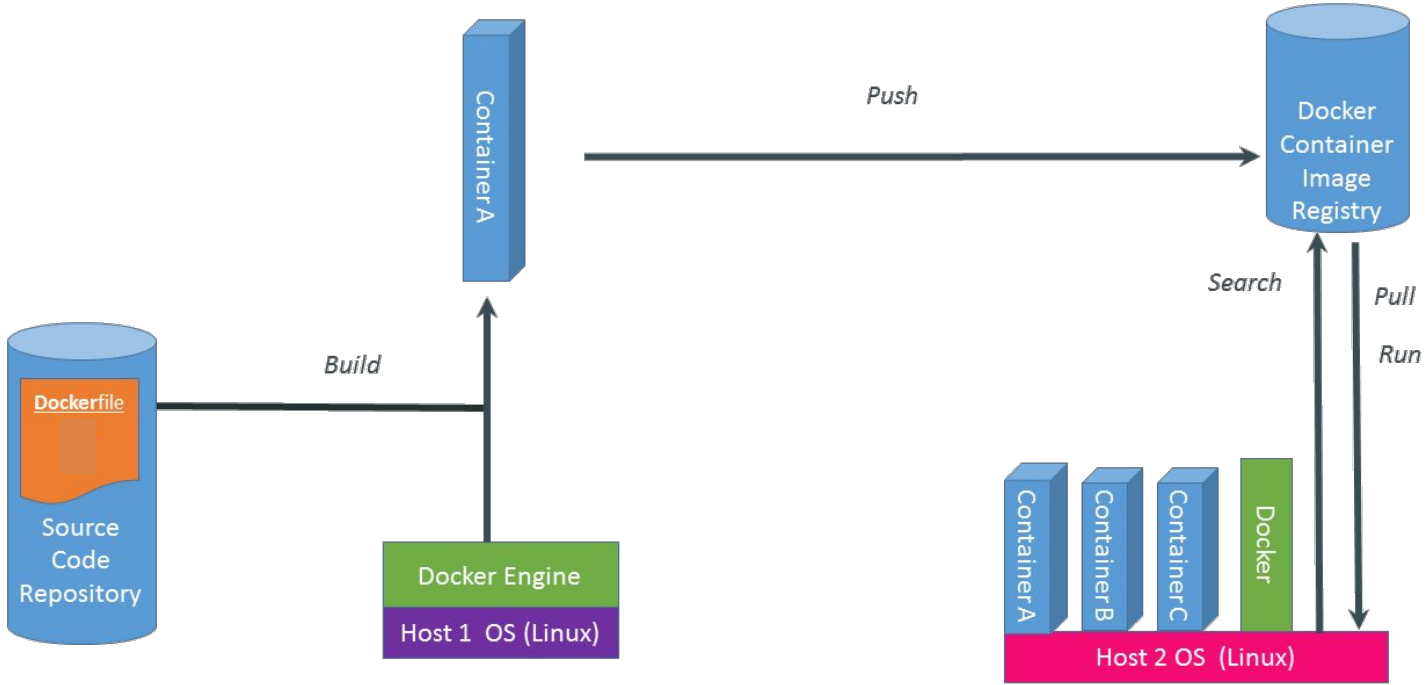
- "A recipe" for constructing a Linux container

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

Docker Workflow



Lightweight Virtualization with AWS Firecracker Hypervisor

Support stock Linux guest OS

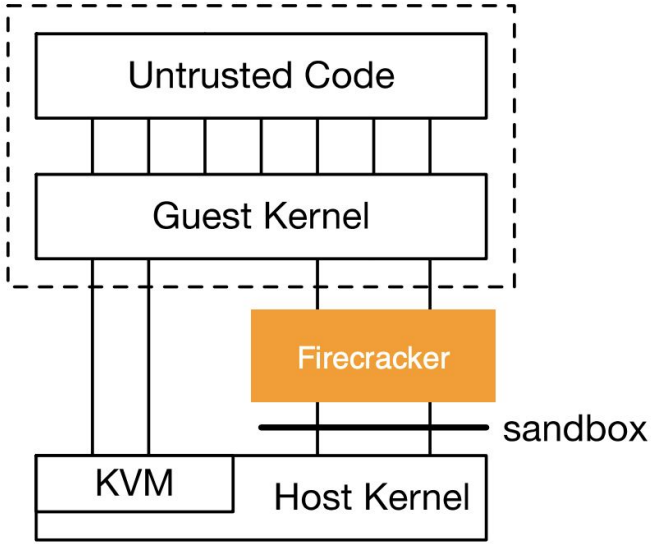
- No compromises in security and compatibility

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

Lightweight virtualization
+
Docker container deployment

Firecracker MicroVM:

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

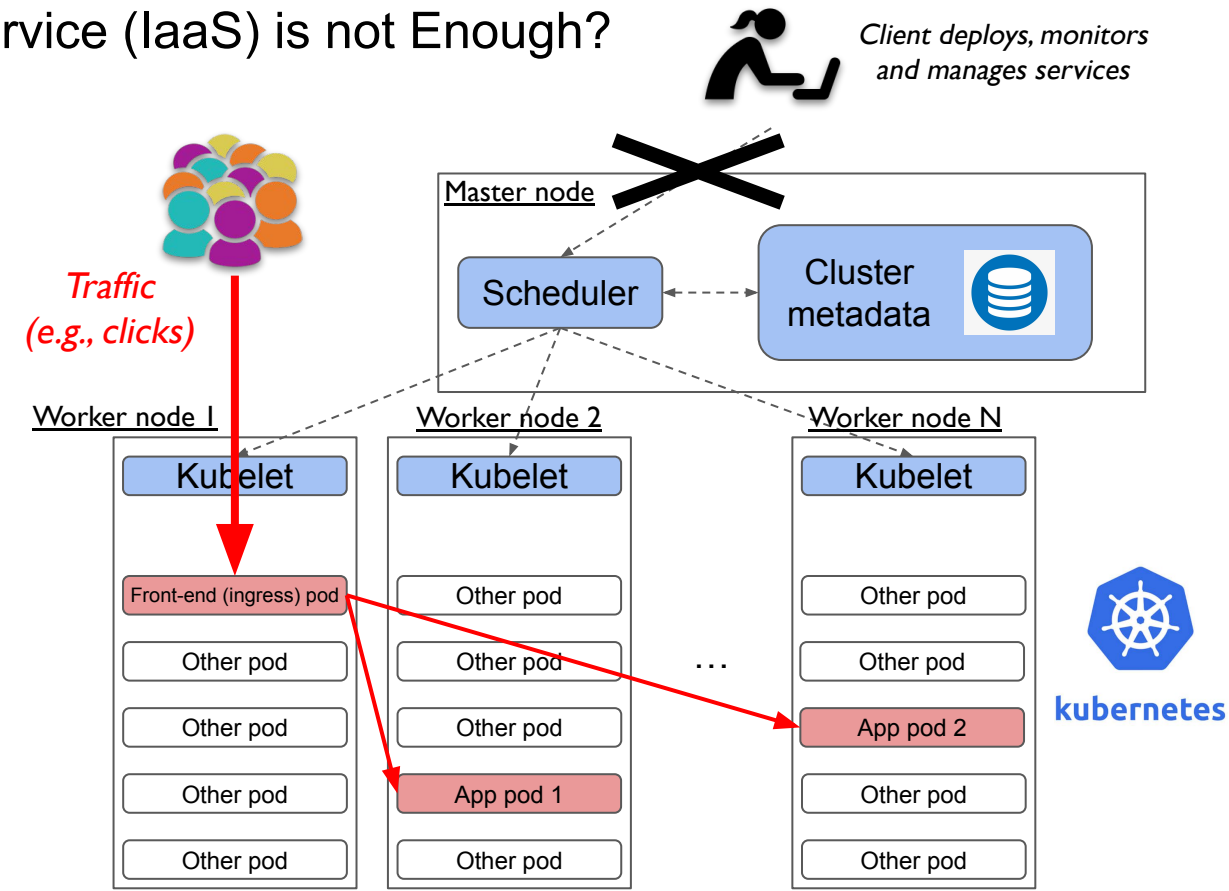
Why Infrastructure-as-a-Service (IaaS) is not Enough?

Provider maintains the datacenter

- Acquisition & operation
- Power
- Hardware & software upgrades

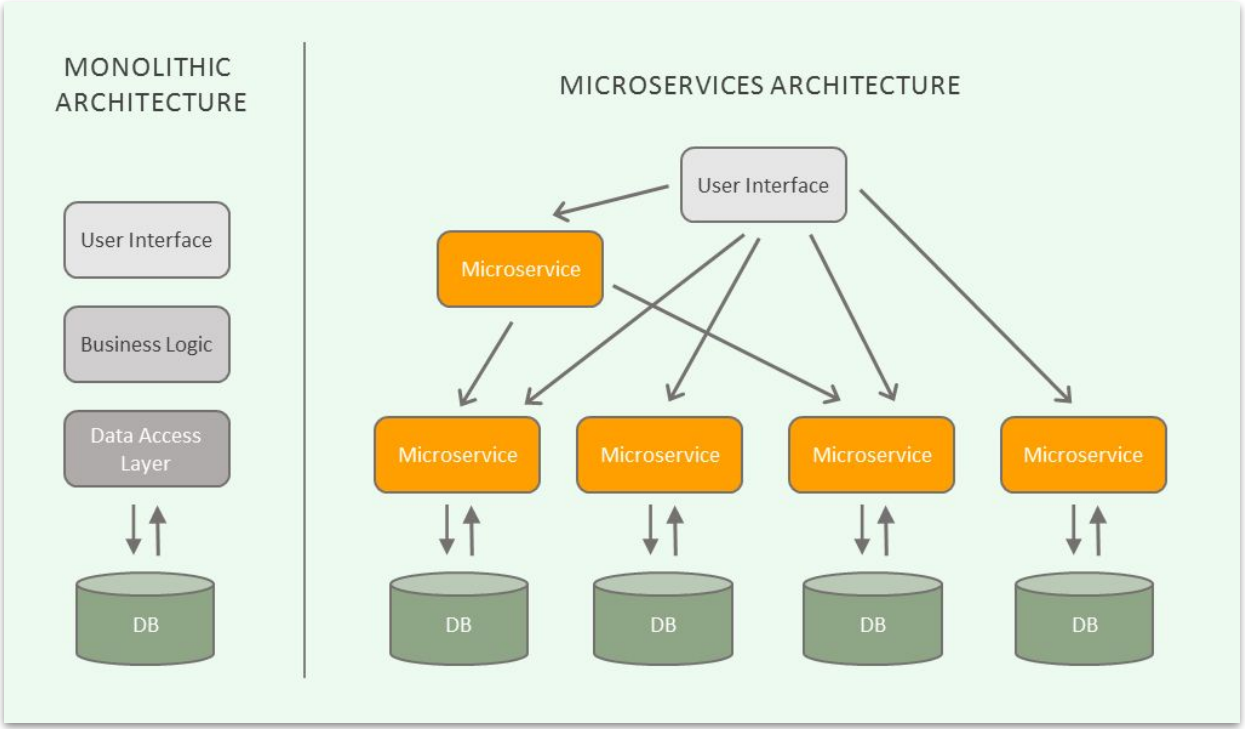
Client still manages the infrastructure

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



Infrastructure management puts a significant burden on a client

How to Make Jobs Easy to Develop & Scale?



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)

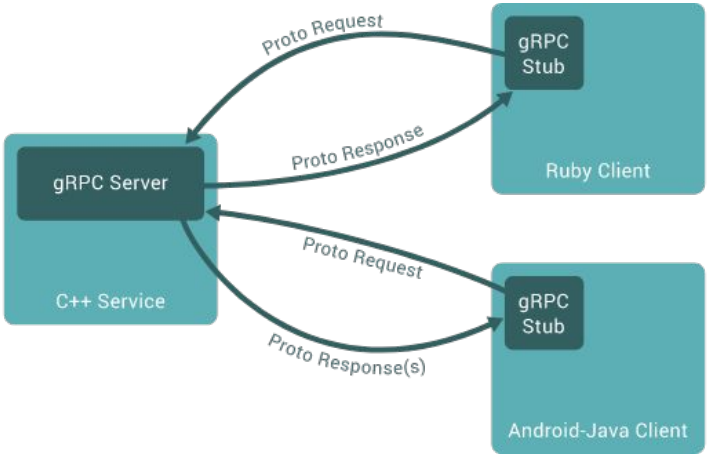
Source: https://hackernoon.com/how-microservices-saved-the-internet-30cd4b9c6230

Microservice Architecture

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)

gRPC architecture, Google



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

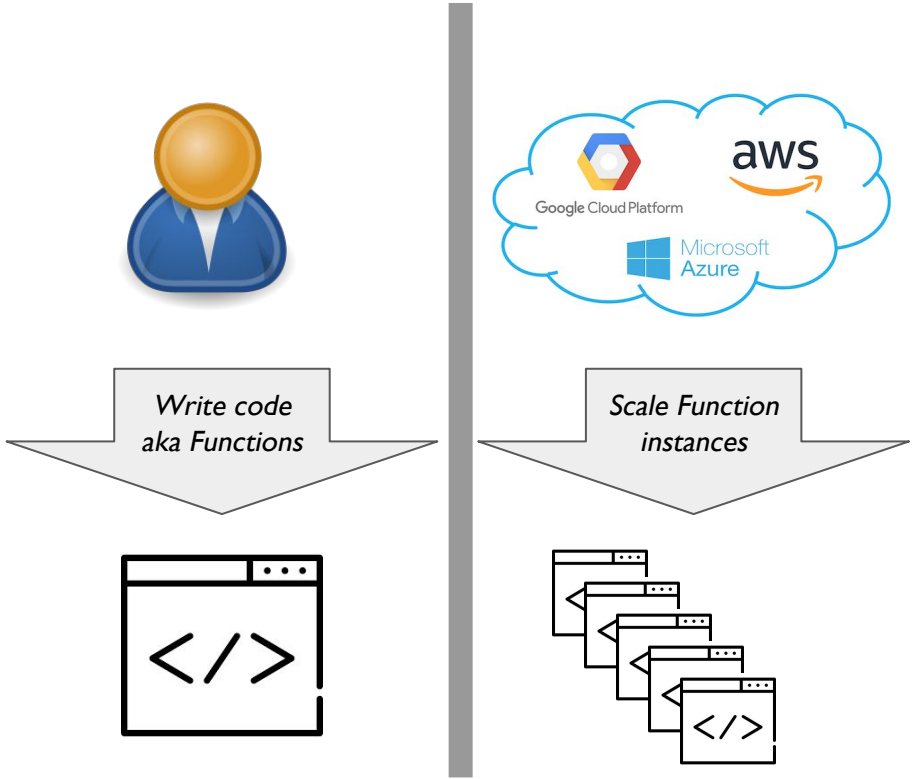
The Future of Cloud Computing is *Serverless*

Functions-as-a-Service (FaaS) paradigm shift

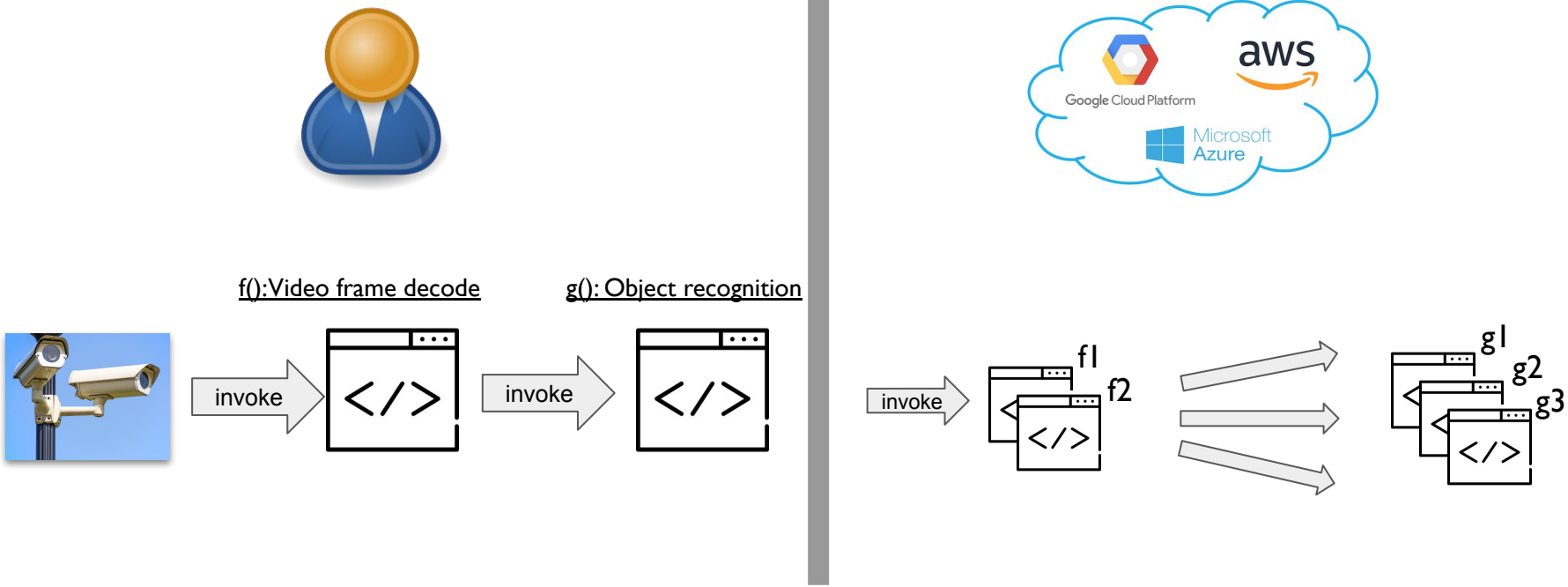
- Clients write code as **functions**
  - Specify when to invoke (e.g., on clicks)
- Providers adjust **per-function** resources
  - Scale instances of functions **with traffic** from **0** to virtually **infinity**

Pay-as-you-go pricing model

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



Serverless App: The Client and Provider Perspectives



Clients provides app's functions, providers scale instances of functions

Serverless behind the Scenes (Amazon Lambda)

Functions are deployed as lightweight VMs

- Packaged as **Docker images**
- Function invocation is connected to **triggers** (e.g., clicks, image uploads)
- Function code
  - Provider's runtime: **HTTP-level** server
  - Client-defined handle in a **high-level** language (Python, NodeJS, Java, etc.)

Limitation: Functions are **stateless**

- Any function instance can handle **any** invocation of that function
- Must be composed with **conventional storage services** and **databases**

Knative: Serverless Under the Hood

Client deploys a function to FaaS

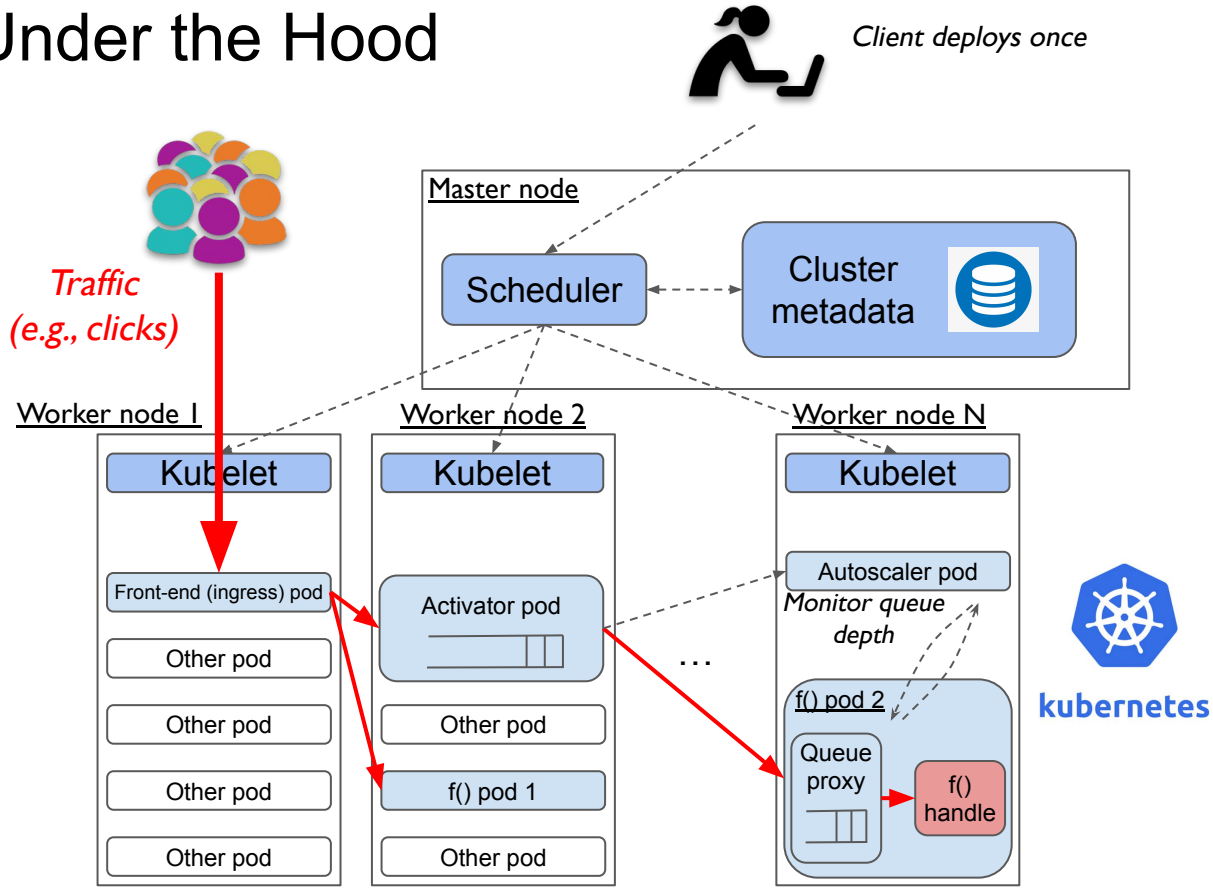
- Provides the code
- Defines the triggers

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	Microservices	Functions
Runtime & Guest OS			
Scaling			
Host OS			
Bare metal compute nodes			
Networking			
Storage			

Client's responsibility

Provider's responsibility

Takeaways

Low **time-to-market** is key for clients business

Democratization of computing

- Providers gradually take over many of their clients' responsibilities
  - Providers manage the infrastructure, clients focus on the business logic
- The future of computing is **serverless**

The **pay-as-you-go** pricing model

- Cloud resources rental with **fine-grain autoscaling**