Fundamentals of Cloud Computing

Lecture 2

Economic Fundamentals

Cloudonomics:Quantitative analysis of clouds

- **Common Infrastructure**
  - Benefits of pooling resources and performing statistical multiplexing of workloads
- **Location independent**
  - Relationship between #nodes and latency
- **Online connectivity**
  - Capital and latency tradeoff between P2P vs hub networks
  - Improvement that network optimizations can bring
- **Utility Pricing (pay per use)**
  - When is the cloud cheap compared to owning?
  - Is it beneficial to use cloud if it is more expensive than owning?
- **on-Demand provisioning**
  - If your service is growing/declining exponentially, what is the overhead of fixed capacity?

* Joe Weinman. Cloudonomics: The Business Value of Cloud Computing, Wiley, 2012

Utility Pricing

- Q: Should you go for the public cloud if the unit price is higher than a home-grown solution?
- Consider a car
  - Buy (lease) for EUR 10 per day vs. Rent a car for EUR 30 a day
  - If you need a car for 2 days in a month, buying would be much more costly than renting
  - It depends on the load/demand

Utility Pricing: Calculation

- ***L(t)*: load (demand for resources) *0<t<T***
- ***P = max(L(t))* : Peak Load**
- ***A = Avg(L(t))* : Average Load**
- ***B = Baseline (owned) unit cost ; B_T = Total Baseline Cost***
- ***C = Cloud unit cost; C_T = Total Cloud Cost***
- ***U = C / B* : Utility Premium** *For the rental car example, U=3*

- **B_T =** $P \times B \times T$ (since Baseline should handle peak load)
- **C_T =** $\int C \times L(t) dt = A \times U \times B \times T$
- ***When is cloud cheaper than owning?***
  - $C_T < B_T$
  - $A \times U \times B \times T < P \times B \times T \rightarrow U < P/A$
- **When Utility premium is less than Peak to Average load ratio**

Utility Pricing: Summary

- Utility Pricing is good when demand varies over time
  - Usually the case of a start-up or a seasonal business

- When Utility Premium is less than ratio of Peak Demand to Average Demand, Cloud computing is beneficial

- Next, we look at the possible savings that Cloud providers can create using statistical multiplexing

Common Infrastructure: Multiplexing

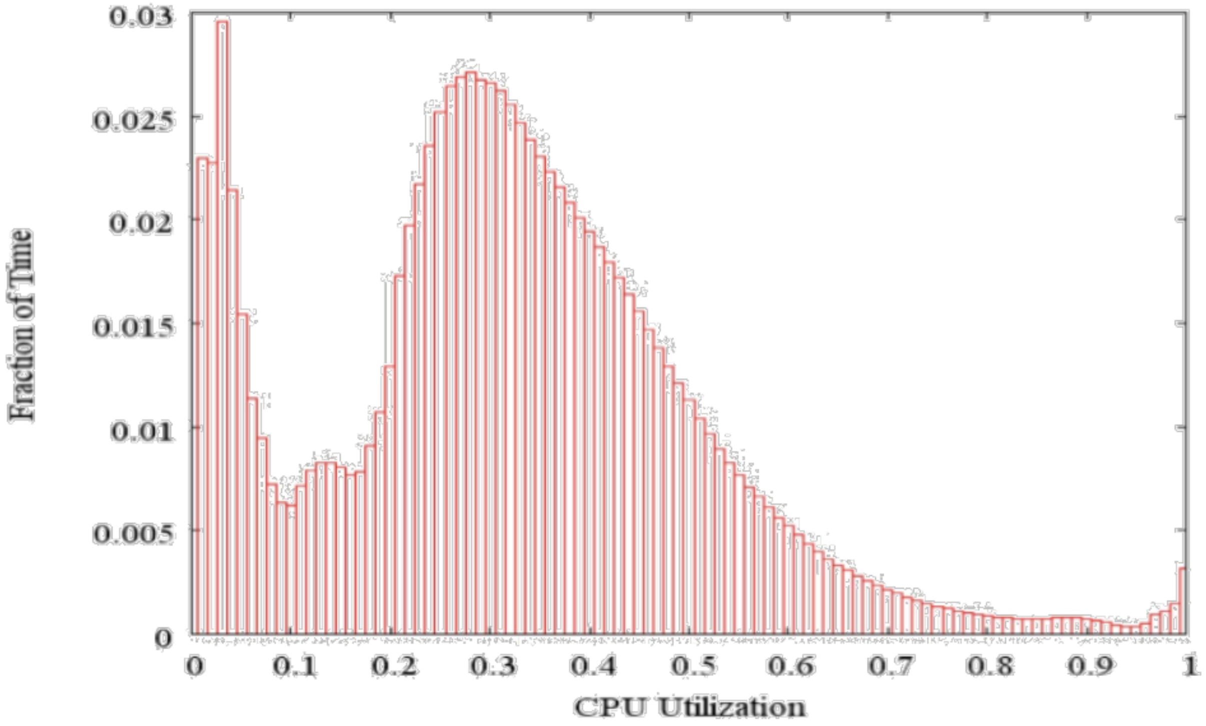
- **Resource pooling**
  - Allows economies of scale
  - Reduces overhead cost
  - Allows cloud provider more negotiating power when buying infrastructure (volume purchasing)

- **Multiplexing (multi-tenancy)**
  - Allows statistics of scale

Common Infrastructure: Multiplexing

- Assume you combine 2 infrastructures into a bigger one
  - One is built to peak requirements
  - The other is built to less than peak

Activity profile of a sample of 5,000 Google Servers over a period of 6 months



Benefits of multiplexing

- Benefits for the infrastructure built to peak
  - Load multiplexing yields higher utilization and lower cost per delivered resource w.r.t. unconsolidated workloads

- Benefits for part of the system built to less than peak
  - Load multiplexing can reduce the unserved requests
  - Reduces a penalty function associated with such requests (e.g., a loss of revenue or a Service-Level agreement SLA violation payout).

A Measure of Smoothness

- **Lets define coefficient of (load) variation Cv**
  - Cv=σ/|μ|
  - non-negative ratio of the standard deviation σ to the absolute value of the mean |μ|.
  - The larger the mean for a given standard deviation, or the smaller the standard deviation for a given mean, the “smoother” the load curve is
- **Importance of *smoothness***
  - An infrastructure with fixed assets servicing highly variable load will achieve lower utilization than a similar one servicing relatively smooth demand.
  - So for workloads with high Cv, or low smoothness, fixed asset servicing is bad
- Let’s see what happens to Cv when we multiplex workloads

Case study: Independent jobs

- Let X1, X2…Xn be n independent jobs (random variables) with identical standard deviation σ and positive mean μ
  - Hence, $Cv(X1)=Cv(X2)=\sigma/\mu$

- Consider the random variable $X=X1+X2+\ldots+Xn$ (multiplexing)

- Statistics 101
  - $\text{mean}(X)=\text{mean}(X1)+\text{mean}(X2)+\ldots+\text{mean}(Xn)=n\mu$
  - $\text{var}(X)=\text{var}(X1)+\text{var}(X2)+\ldots+\text{var}(Xn)=n\sigma^2$

Smoothing under multiplexing

- Hence standard deviation of X is
  - stdev(X) = $\sqrt{Var(X)} = \sqrt{n}\ .\ \sigma$

- Finally Cv(X)= $\sqrt{\boldsymbol{n}}\ .\ \sigma\ /\ n\mu = \sigma\ /\ \sqrt{\boldsymbol{n}}\mu$
  - *i.e., Cv(X) = Cv(Xi) /* $\sqrt{\boldsymbol{n}}$
  - We obtain "smoother" aggregate load with multiplexing!
  - Hence, we have benefits from statistics of scale in addition to those from economies of scale

- Best Case: Negative correlation
  - Consider jobs Xi and 1-Xi
  - Multiplexing: X = Xi + 1 – Xi = 1
  - For random variable X = 1, σ = 0, Cv = 0
  - Optimally smooth, best CPU utilization!

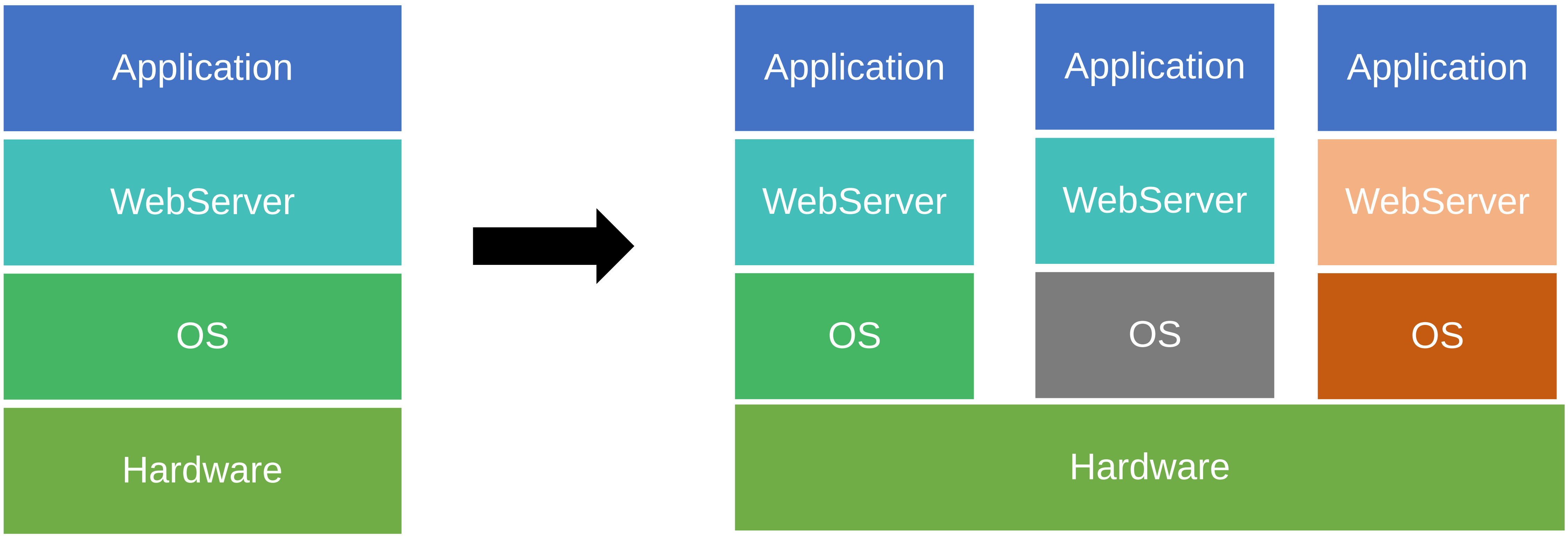
Common Infrastructure: Summary

- Negative-correlated jobs
  - Private, mid-size, and large-size providers can experience similar statistics of scale
- Independent jobs
  - Mid-size providers can achieve similar statistical economies to an infinitely large provider
- Takeaway lesson: cloud provider of any size can be profitable!
  - At least according to “Value of Common Infrastructure”

Infrastructure Fundamentals
Virtualization

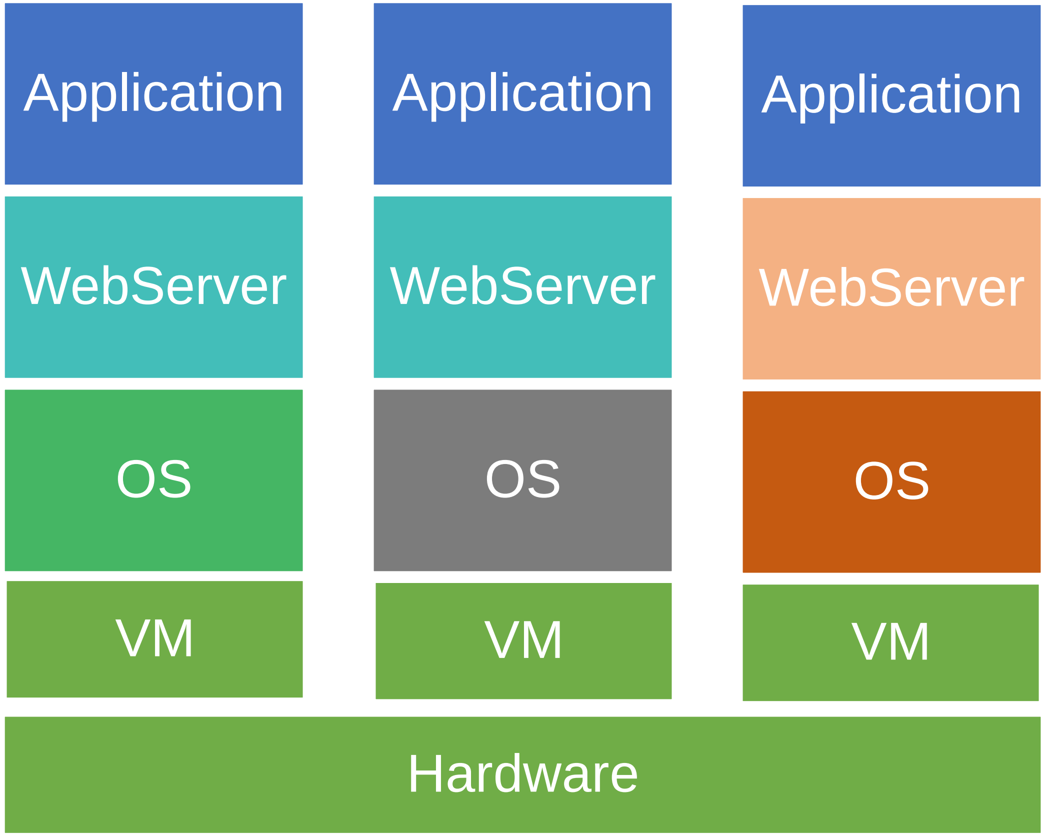
Sharing Resources

- Economics of cloud computing requires resource sharing
- How do we share a physical computer among multiple applications?



Virtualization: An Abstraction

- Introduce an abstract model of what a generic computing resource should look like – a “virtual machine” (VM)
  - CPU -> virtual CPU
  - Disk -> virtual Disk
  - NIC -> virtual NIC
- Provide one such “virtual machine” per tenant & host multiple virtual machines on the same physical machine



Virtual machine and Hypervisor

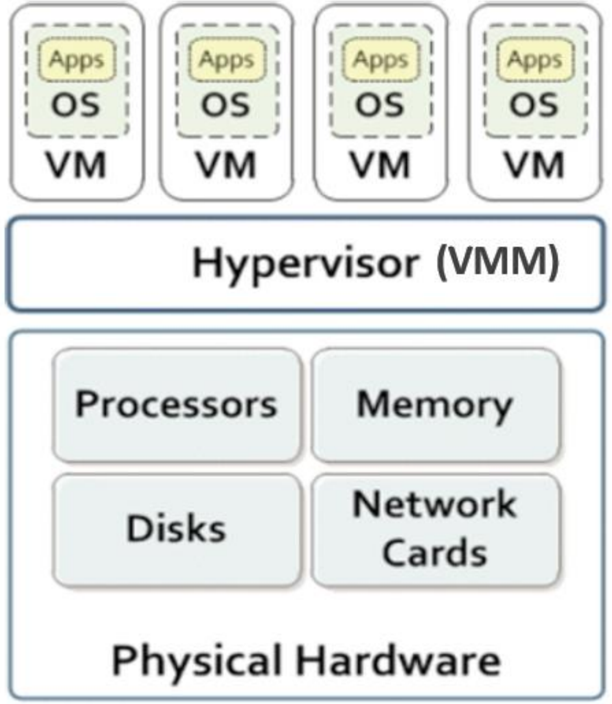
- Virtual Machine:
  - “A fully protected and isolated copy of the underlying physical machine’s hardware.” -- definition by IBM

- Virtual Machine Monitor (aka VMM, aka Hypervisor):
  - “A thin layer of software that's between the hardware and the Operating system, virtualizing and managing all hardware resources”

- Two types of hypervisors
  - Type 1: VMM runs directly on physical hardware
  - Type 2: VMM built on top of a host OS

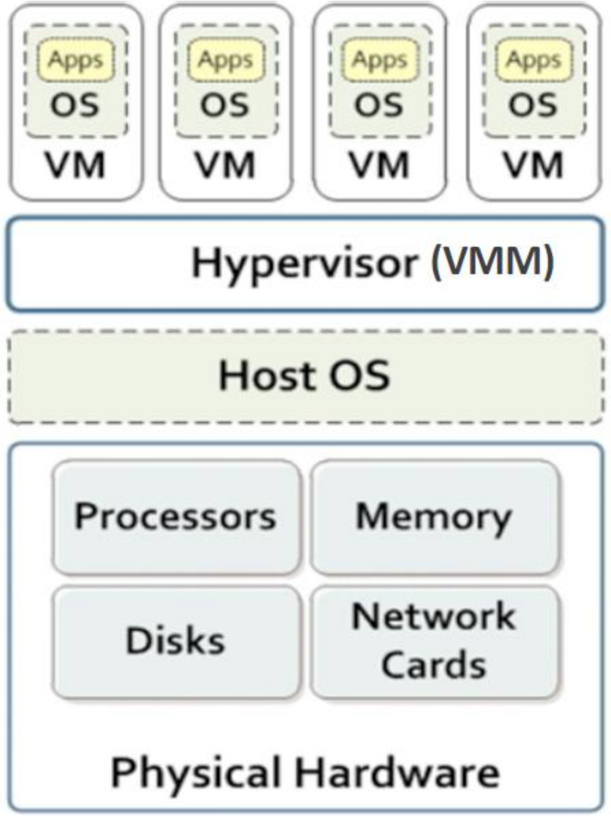
Type 1 Hypervisor

- VMM directly implemented on physical hardware
- VMM performs scheduling and allocation of resources
- Eg: VMWare ESX Server

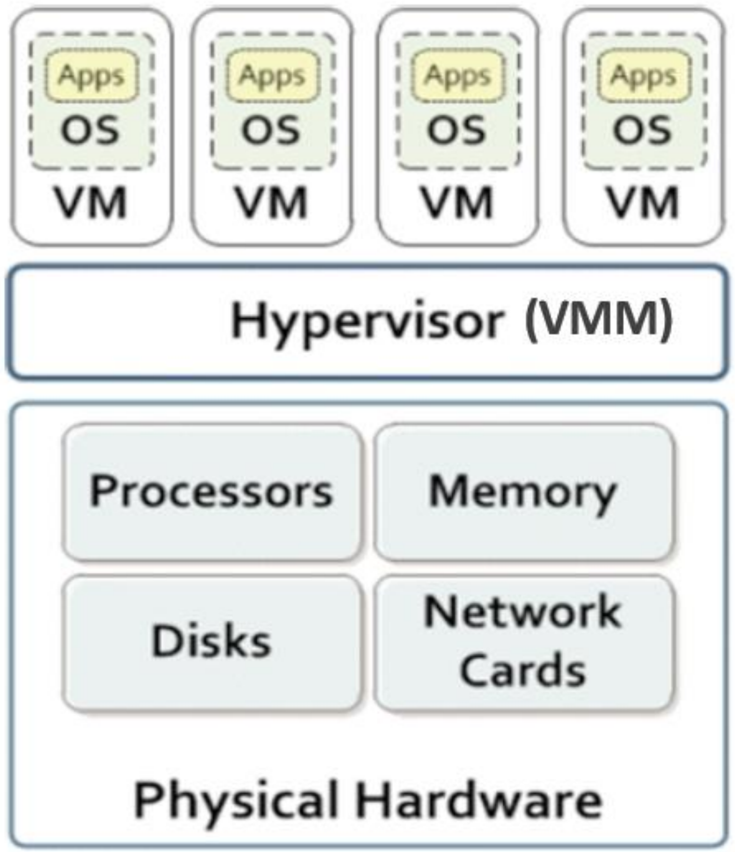


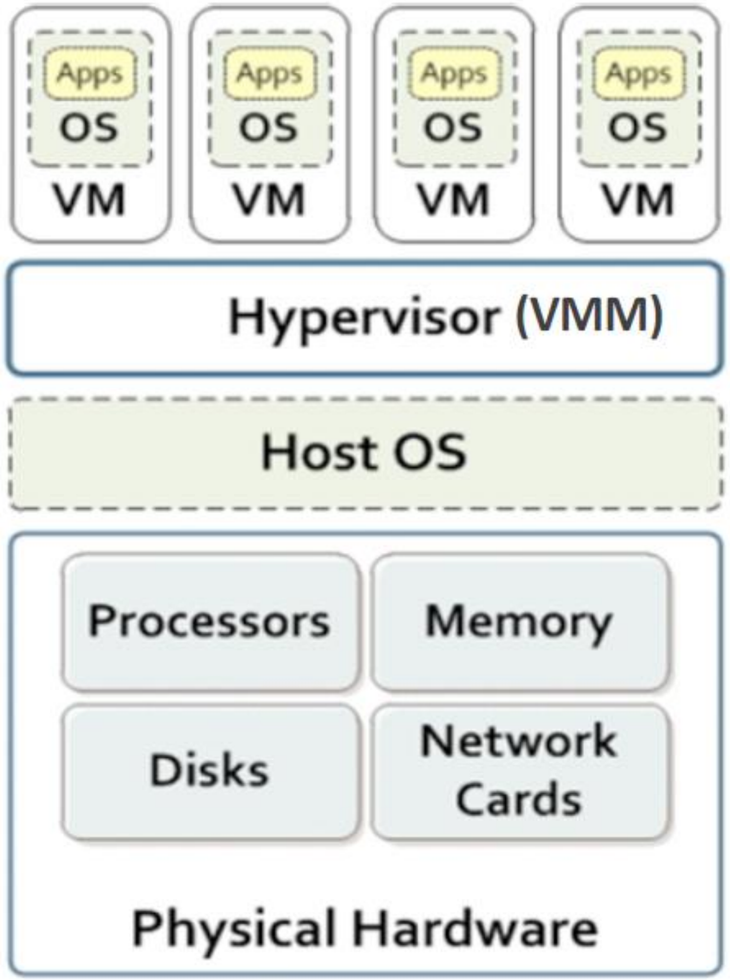
Type 2 Hypervisor

- VMMs built completely on top of a host OS
- Host OS provides resource allocation and standard execution environment to each “guest OS”
- Example: User-mode Linux (UML), QEMU



Type 1 vs type 2





Virtualization: A bit of history (1960s, 70s)

- IBM sold big mainframe computers
  - Companies could afford one
  - Wanted to run apps designed for different OSes.
- Idea: add a level of indirection!
- CP/CMS (Control Program/Cambridge Monitor System), 1968
  - Time sharing providing each user with a single-user OS
  - Allow users to concurrently share a computer
- Hot research topic in 60s-70s; entire conferences devoted to VMs

IBM System/370

Virtualization fades away (1980s)

- Interest died out in 80s
  - More powerful, cheaper machines
  - Ex: sun workstation
- Could deploy new OS on different machine
  - More powerful OSes (UNIX, BSD, MINIX, Linux)
- No need to use VM to provide multi-user support


Ken Thompson, Dennis Richie


Andy Tanenbaum

New Beginnings (1990s)

- Multiprocessor in the market
  - Innovative Hardware
- Hardware development faster than system software
  - Customized OS are late, incompatible, and possibly buggy
- Commodity OSes not suited for multiprocessors
  - Do not scale due to lock contention, memory architecture
  - Do not isolate/contain faults; more processors, more failures

It’s Disco time

Disco: Running Commodity Operating Systems on Scalable Multiprocessors, Edouard Bugnion, Scott Devine, and Mendel Rosenblum, SOSP’97

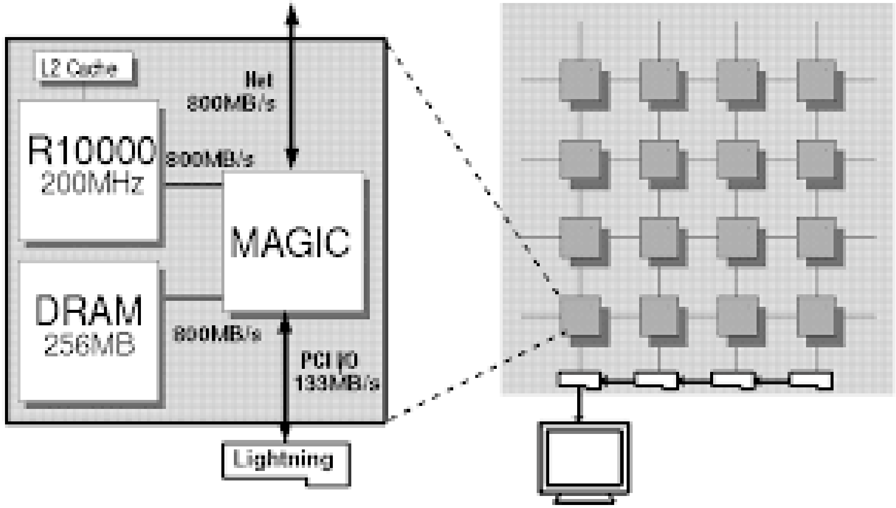
- Idea: Insert a software layer -- Virtual Machine Monitor -- between hardware and OS running commercial OS
- Virtualization:
  - Used to be: make a single resource appear as multiple resources
  - Disco: make multiple resources appear like a single resource
- Trading off between performance and development costs

Disco

- Extend modern OS to run efficiently on shared memory multiprocessors with minimal OS changes

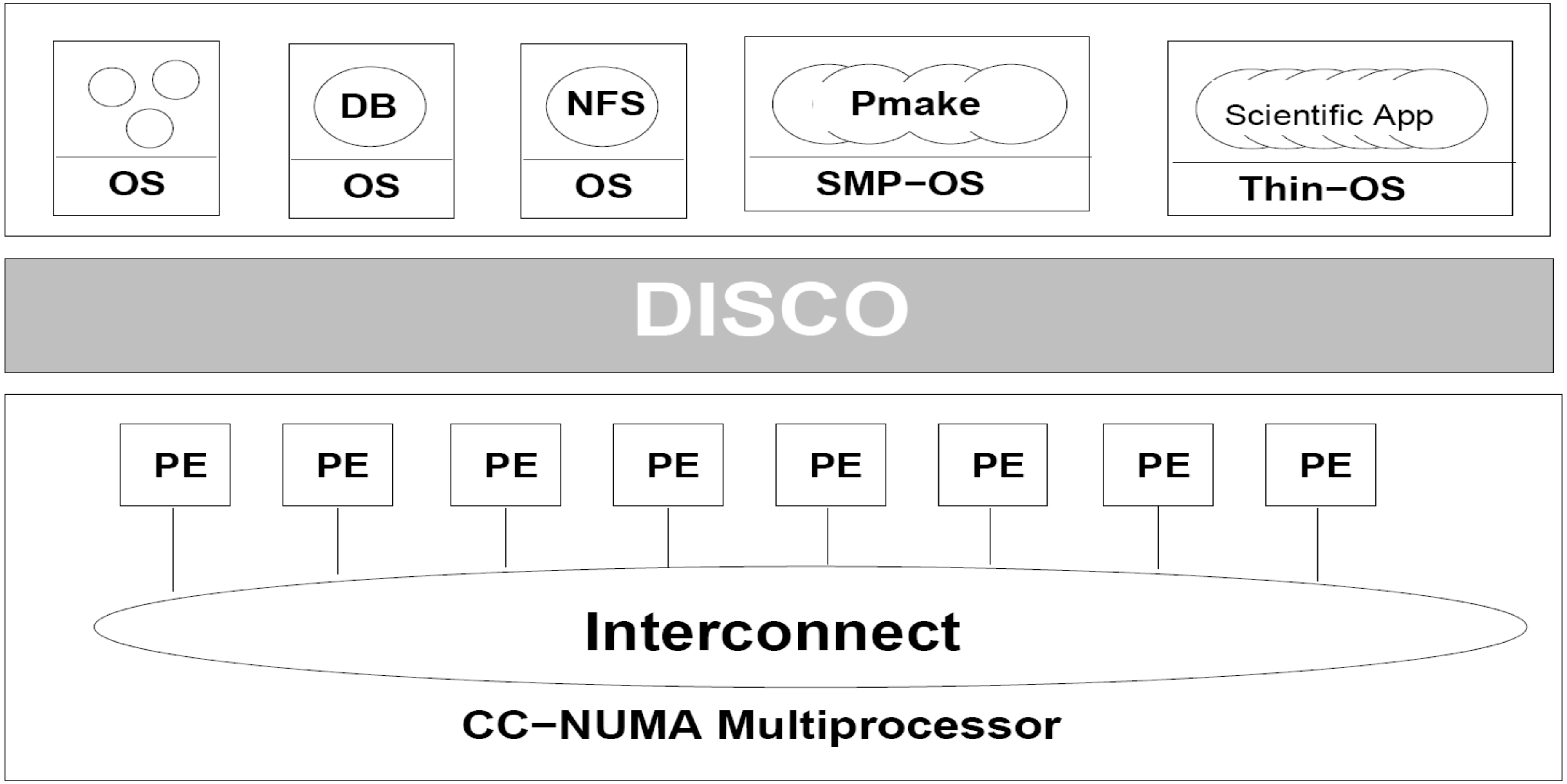
- A VMM built to run multiple copies of Silicon Graphics IRIX operating system on Stanford Flash multiprocessor

IRIX Unix based OS



Stanford FLASH: cache coherent NUMA

Disco Architecture



Disco to VMWare

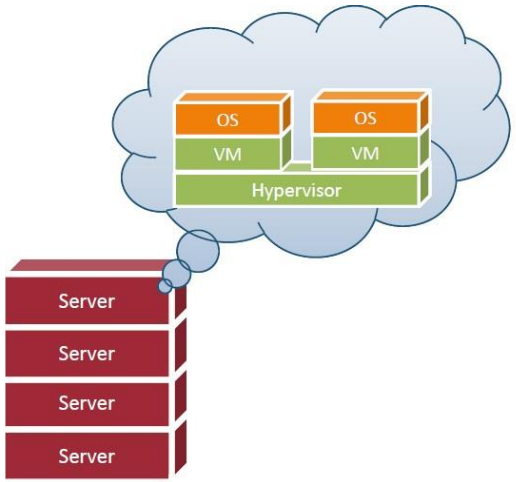
Mendel
Rosemblum
(Stanford University)

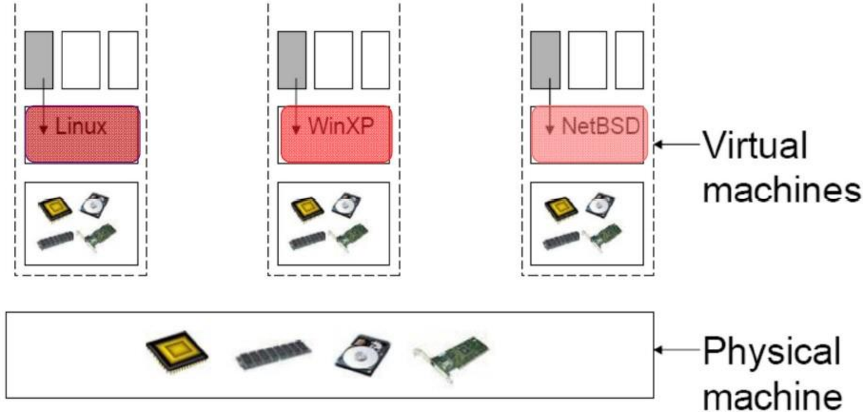
- Started by creators of Disco
- Initial product: provide VMs for developers to aid with development and testing
  - Can develop & test for multiple OSes on the same box
- Actual, killer product: **server consolidation**
  - Enable enterprises to consolidate many lightly used services/systems
  - Cost reduction, easier to manage
  - Eventually over 90% of VMWare’s revenue

Virtualization and the cloud: IaaS recap

Virtualization is the enabler of IaaS

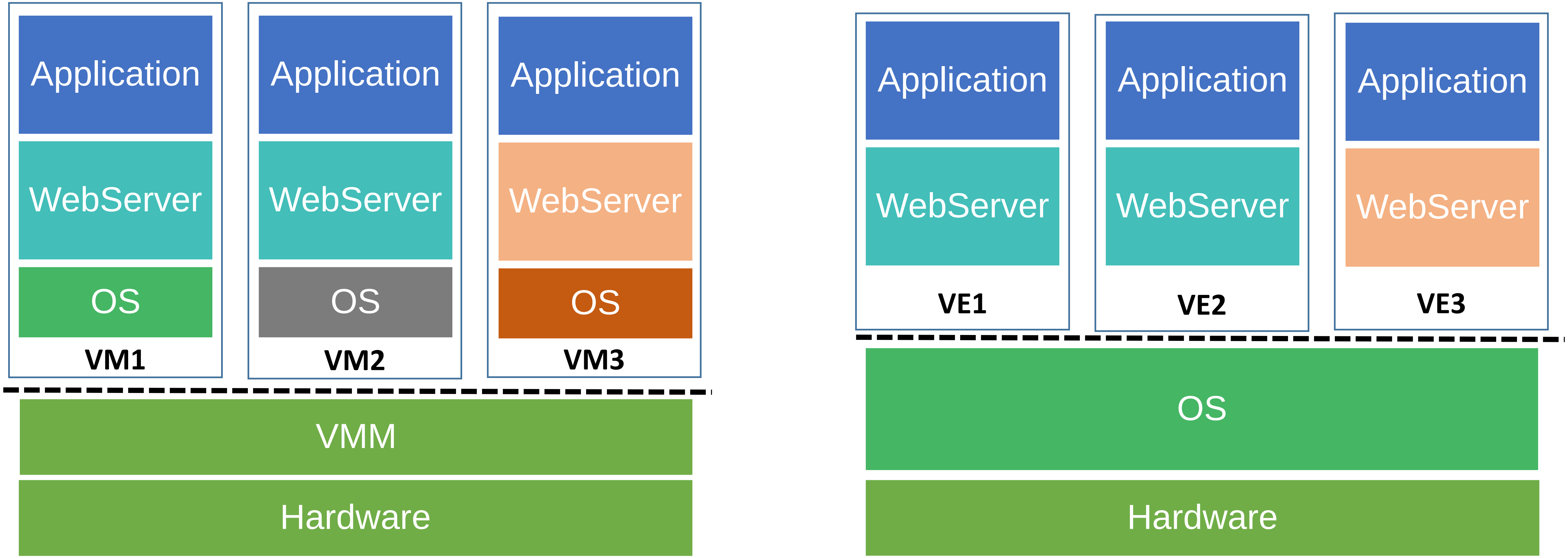
- The cloud provider leases to users Virtual Machine Instances (i.e., computer infrastructure) using the virtualization technology
- The user has access to a standard Operating System environment and can install and configure all the layers above it





From VMs to Containers

Is full hardware emulation the only option for virtualization?



Can we raise the abstraction one level?
Can we support OS-level virtualization to create “Virtual Environments”?

Use case for OS-level virtualization

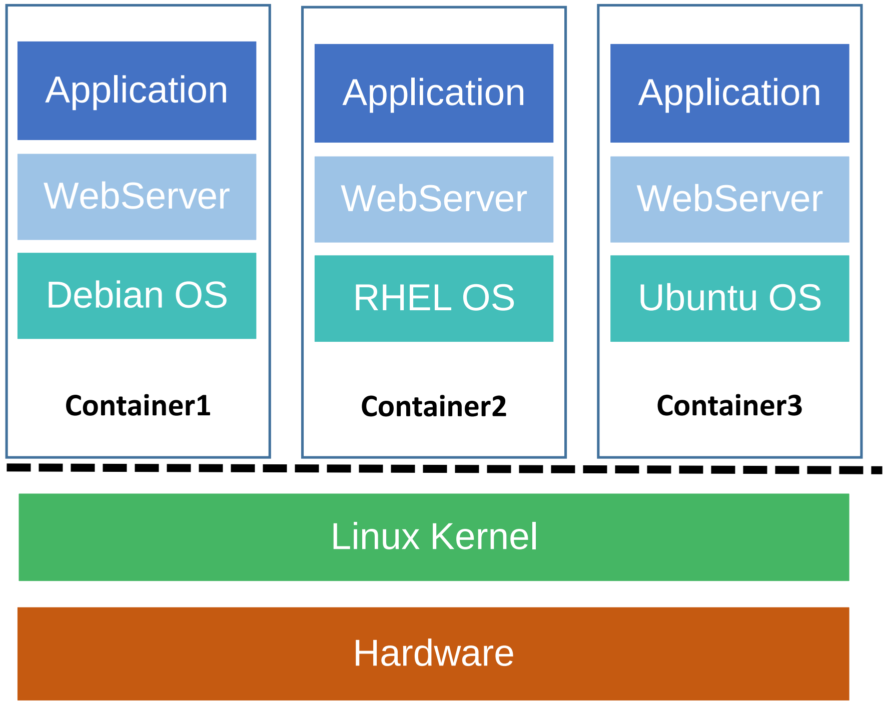
- Hosting providers & PaaS providers
  - Need to host multiple applications/tenants on a single server
- Full hardware virtualization still expensive
  - Full OS + libraries + application per tenant => reduced multitenancy
  - Use of VMMs means license fee
- OSes have always supported multitenancy
  - Multiple processes share same OS
  - Virtual memory & file systems for sharing memory and storage
- Can we extend OS to securely isolate multiple applications?
  - Observe and control resource allocation across groups of processes
  - Limit visibility and communication across groups of processes

Linux kernel functionality to support virtualization

- Cgroups (control groups)
  - Metering and limiting resources used by a group of processes
  - Ex: Partitioning resources in a server so that
    - Memory: Researchers: 40%, Professors: 40%, Students: 20%
    - CPU: Researchers: 50%, Professors: 30%, Students: 20%
    - Network: WWW browsing (20%), NFS (60%), Others (10%)
- Namespaces: Limiting what processes can view
  - Abstract a global system resource and make it appear as a separated instance to processes within a namespace.
    - Just like what a process within a VM can “see”
  - Example namespaces
    - Net: Each process group gets its own net interfaces, routing tables,
    - Pid: Processes can only see other processes in same PID namespace
    - Mount, IPC, …

From virtual machines to Linux containers

- Cgroups + namespaces provide (for most part) everything needed to create “containerized” processes.
  - SELinux and a few other pieces for security
- LXC (Linux Containers)
  - User-land tools that allows creation and running of multiple isolated Linux virtual environments (VE) on a single host
  - Apart from linux kernel, everything can be isolated--userland, libraries, runtimes, and applications
  - Ex: can be used to run multiple LINUX distros as containers

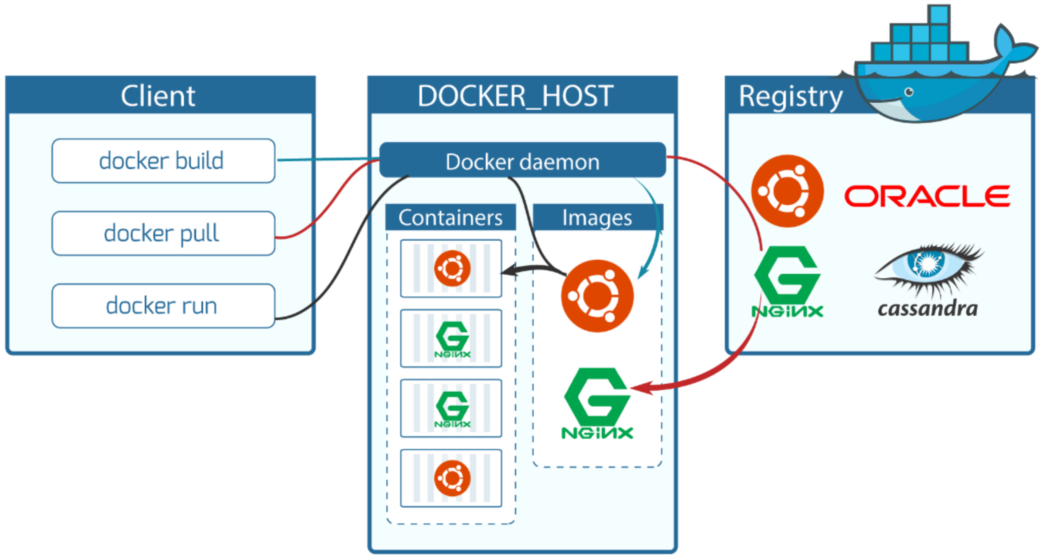


From LXC to Docker

- Early LXC was built for sysadmins
  - No support for moving images, copy-on-write, sharing previously created images
- But developers have different needs
  - Application developed on dev servers, tested in build servers, and deployed in across production servers.
  - How do we make sure configuration is the same? Dependencies are met?
- Docker was developed by Solomon Hykes and others at dotCloud in 2013 to solve this problem with containers
  - **Package system** that can pack an application and all dependencies as a container image after development
  - **Transport system** that ensures that the application image runs exactly similar on test and production systems

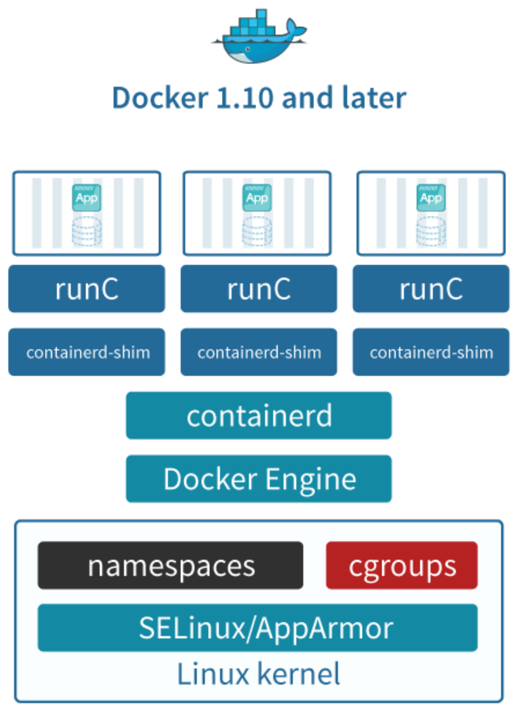
More on Docker

- Docker was originally built on LXC and provided
  - A container image format
  - A method for building container images (Dockerfile/docker build)
  - A way to manage container images (docker images, docker rm , etc.)
  - A way to manage instances of containers (docker ps, docker rm , etc.)
  - A way to share container images (docker push/pull)
  - A way to run containers (docker run)



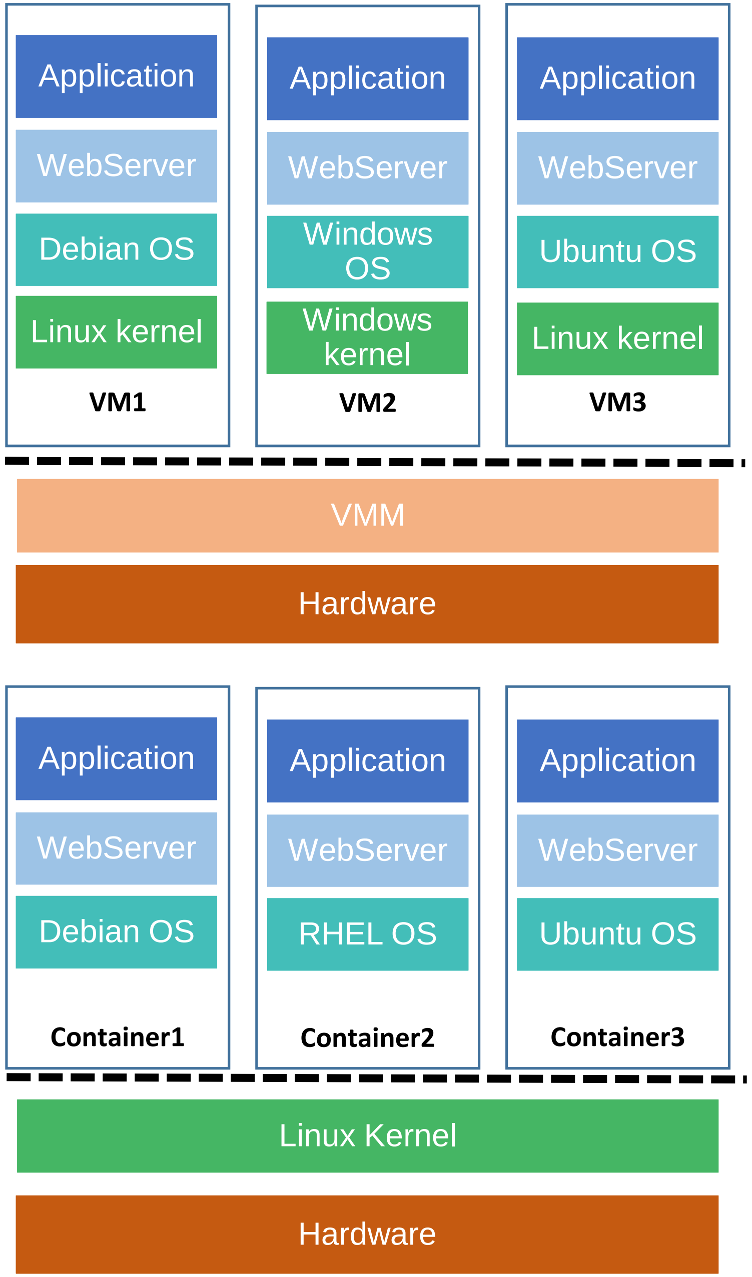
Container ecosystem today

- Low-level Container runtimes
  - Set up and manage namespaces, cgroups and container execution
  - Ex: runC (open container initiative), lxc, rkt
- High-level container runtimes
  - Image format, image management, sharing
  - Ex: cri-o, containerd from Docker (builds on runc)
- Docker today is a collection of components
  - Docker engine: user-facing daemon, REST API, CLI
  - “Runtime”: Containerd, container-shim, runC



Advantages of containers

- Abstraction levels
  - Hypervisors work at hardware abstraction level
  - Containers work at OS abstraction level
- Containers offer higher density
  - VMs need O(GB) vs containers that need O(MB)
  - Can pack many more containers per server
- Containers improve elasticity
  - Easy to “scale up” container than a VM
  - Reason for container adoption in hosting and PaaS environments
  - Example: Everything in Google from gmail to search is containerized
- Native CPU performance
  - No virtualization overhead
- Dramatically improves software development lifecycle
  - Easy to build, test, deploy software without worrying about portability



From Containers to Serverless Functions

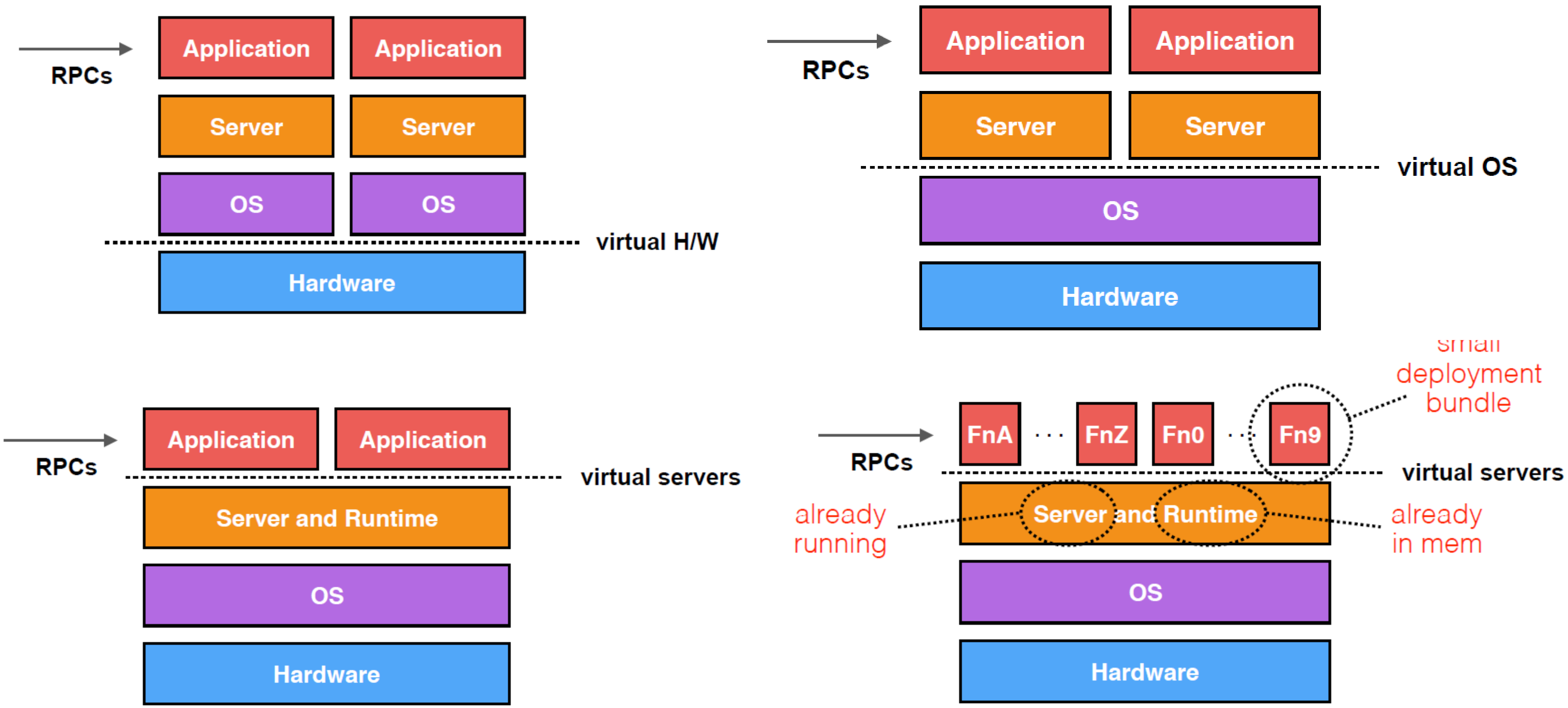
Container Case Study

- Google Borg
  - Internal container platform at Google (http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/43438.pdf)
  - 25 second median startup!
    - 80% of time spent on package installation

- What if we have an light-weight HTTP app?
  - Say < 1,000 LoC that runs for 200 msecs on each HTTP request?
  - In a container, the app would take too long to start

- Containers cannot deal with flash crowds, load balancing, interactive development, etc

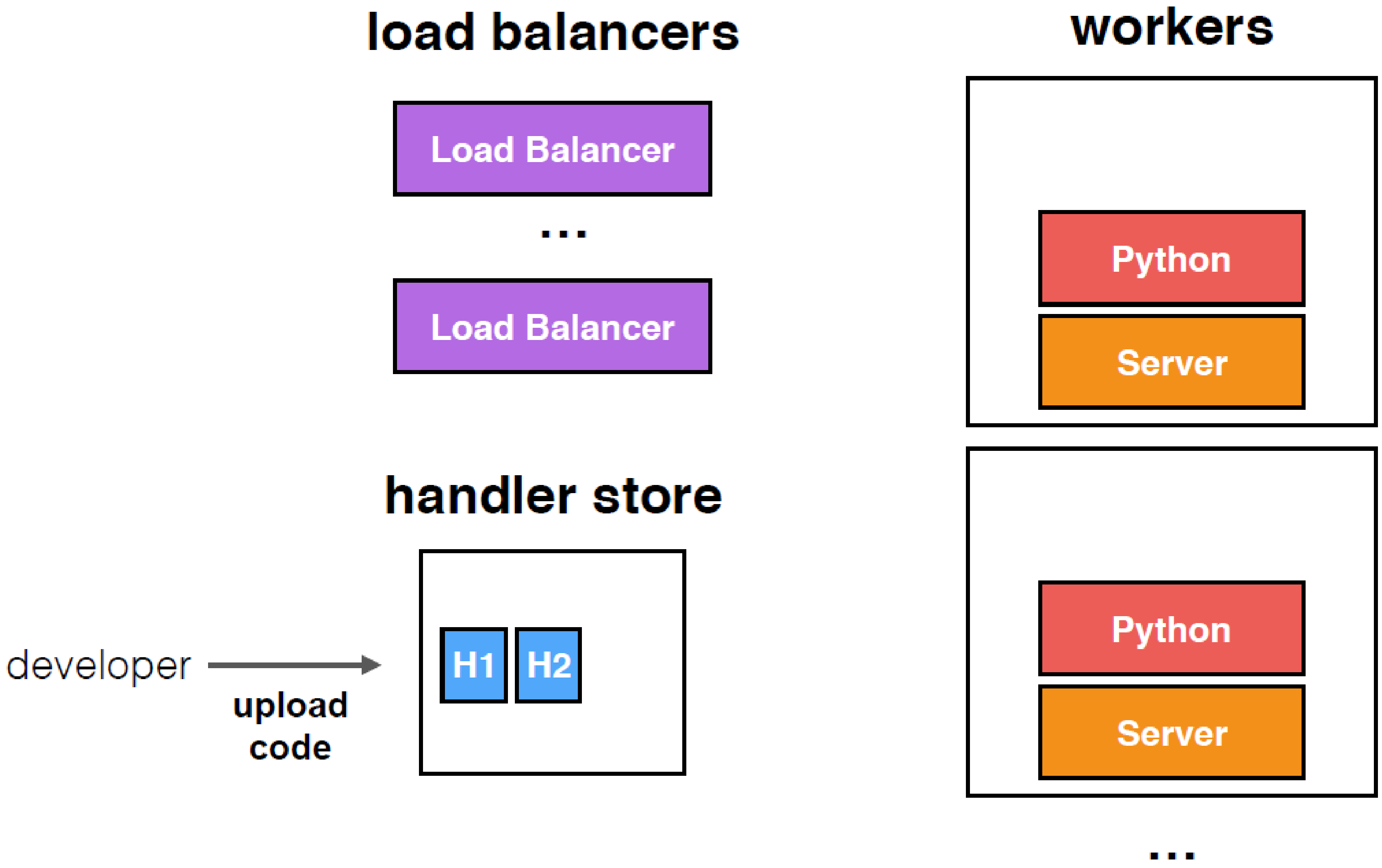
Three generations of virtualization



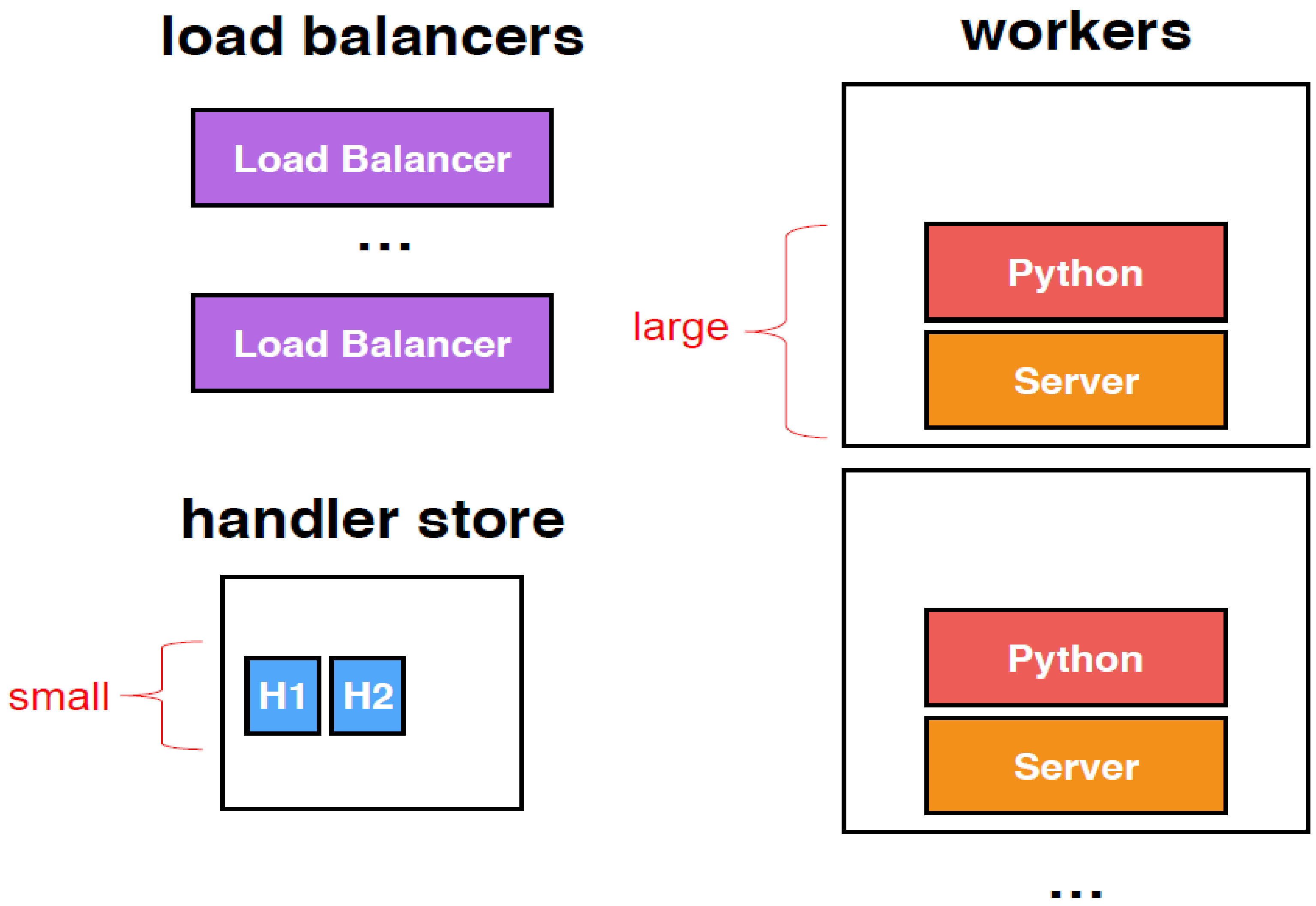
Serverless functions: Model

- Run user handlers in response to events
  - web requests (RPC handlers)
  - database updates (triggers)
  - scheduled events (cron jobs)
- Pay per function invocation
  - actually pay-as-you-go
  - no charge for idle time between calls
  - e.g., charge actual_time * memory_cap
- Share server pool between customers
  - Any worker can execute any handler
  - No spinup time
  - Less switching
- Encourage specific runtime (C#, Node.JS, Python)
  - Minimize network copying
  - Code will be in resident in memory

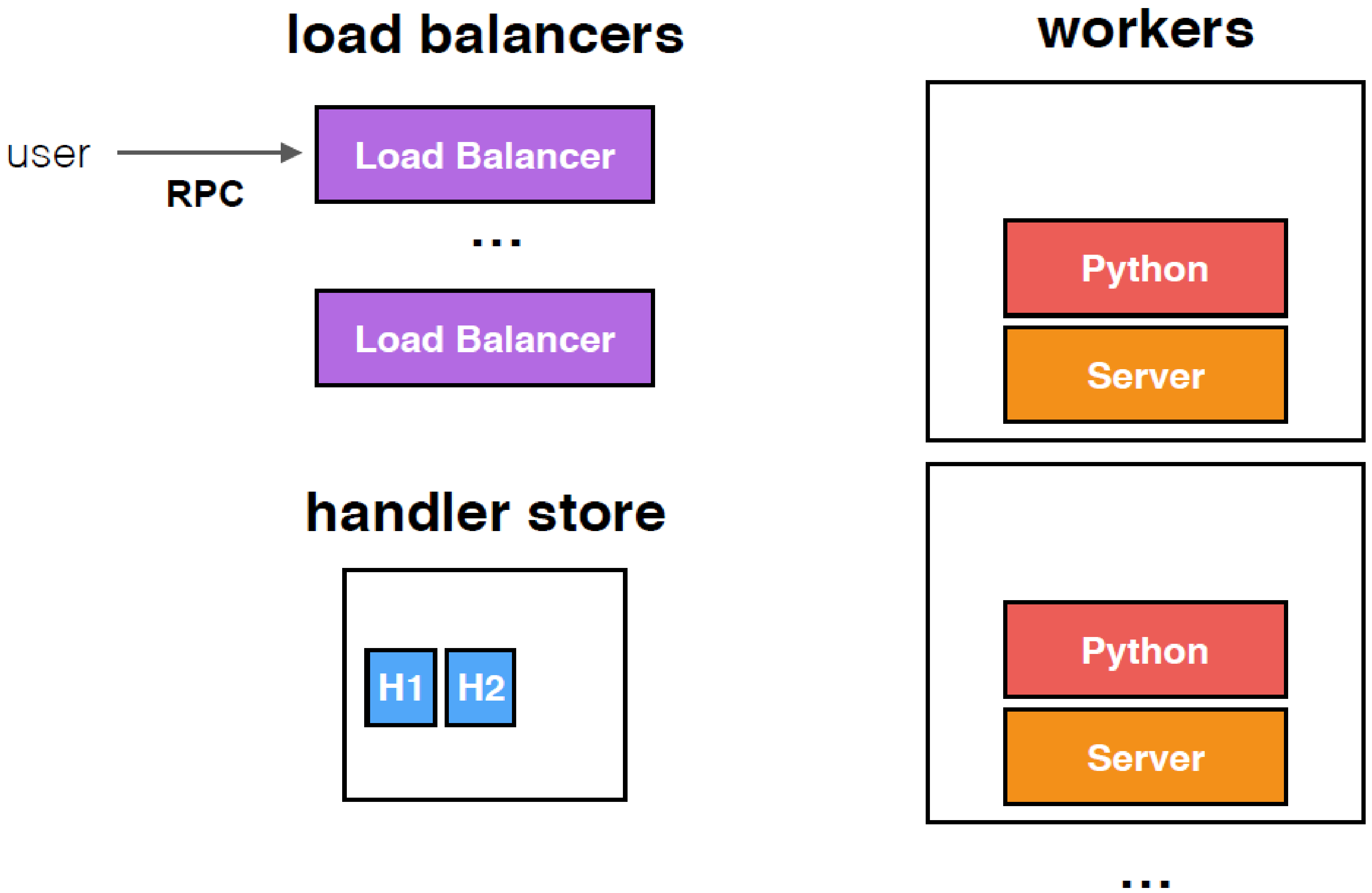
Architecture



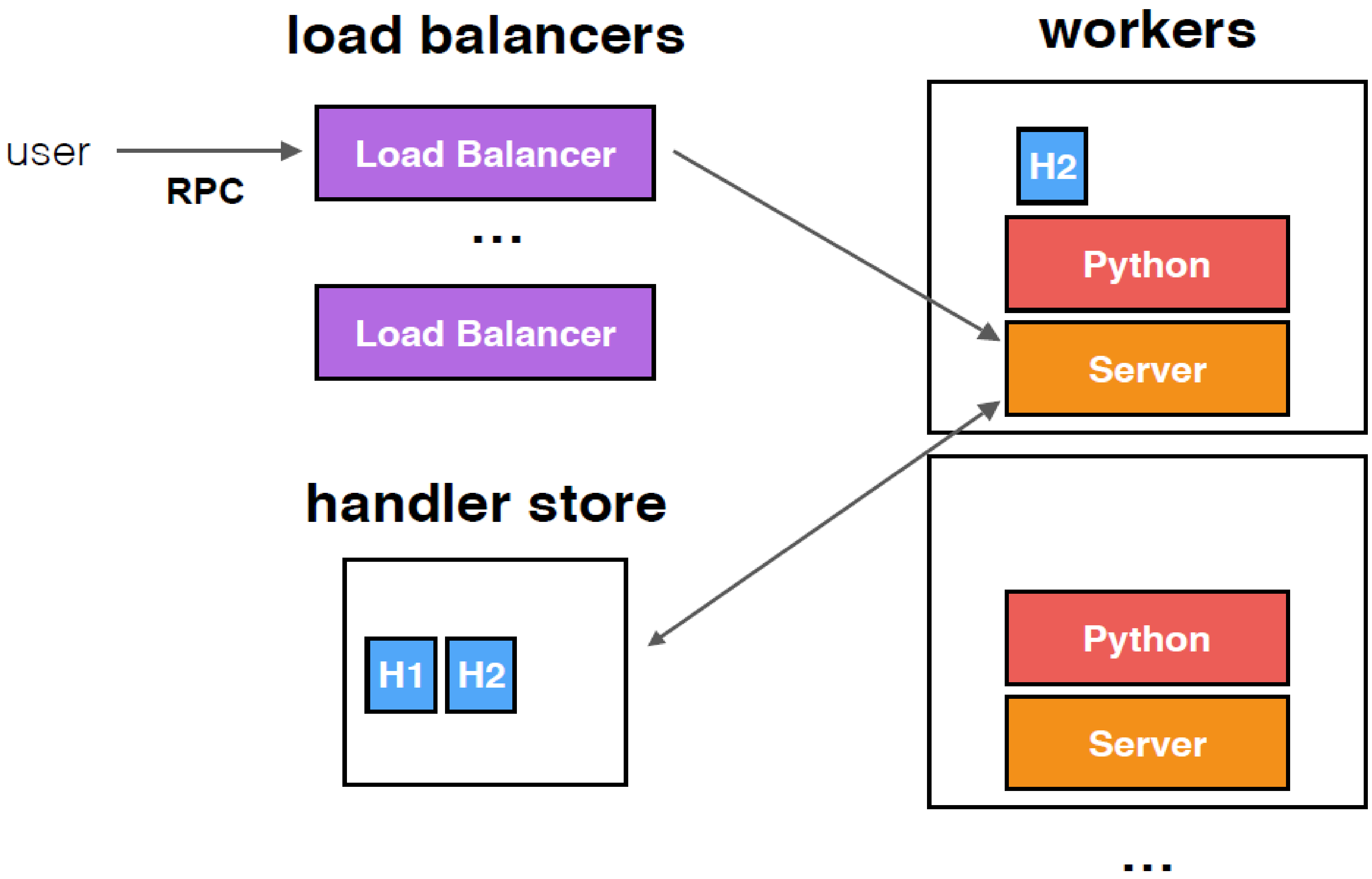
Architecture



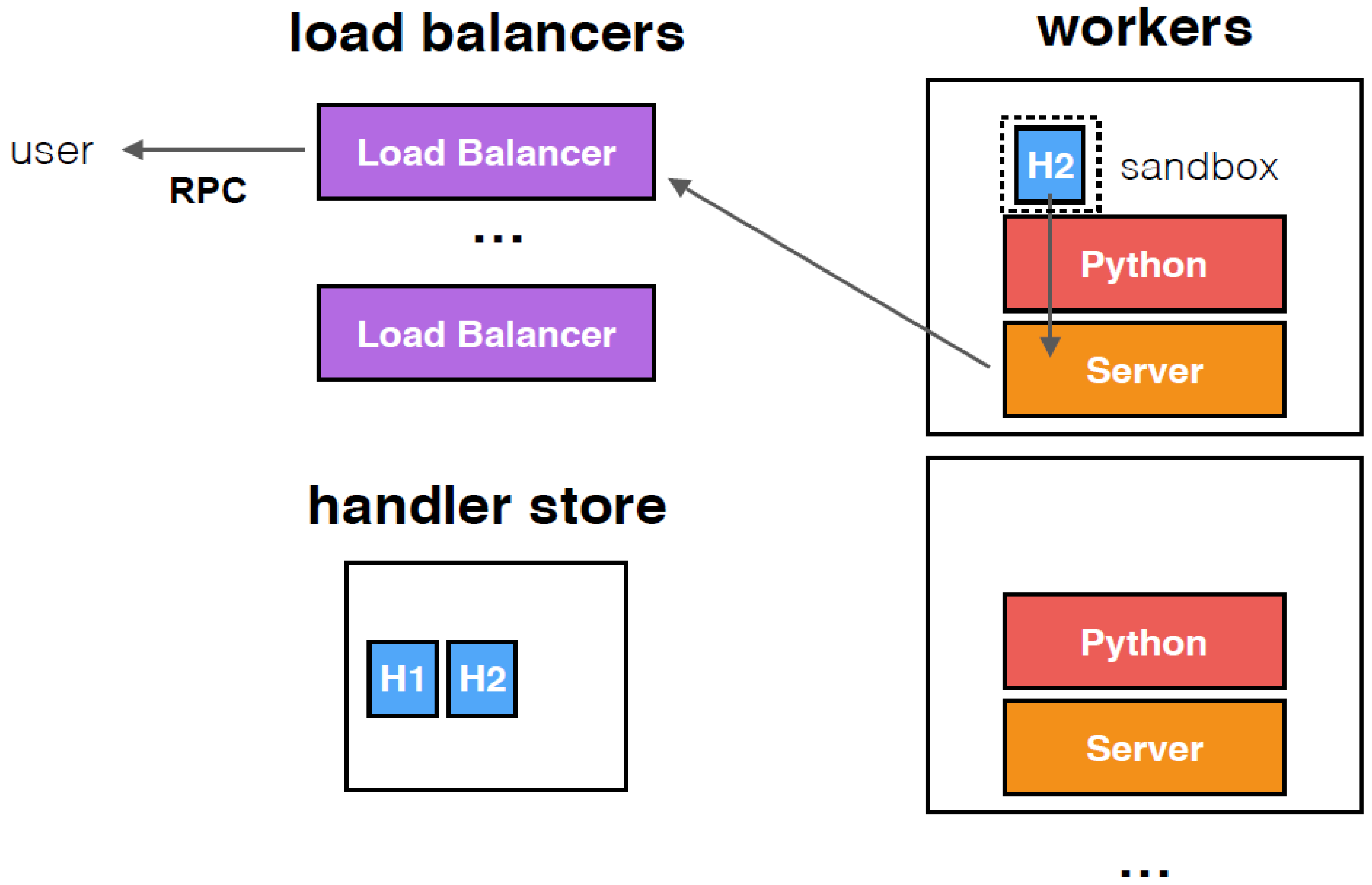
Architecture



Architecture



Architecture



Functions vs containers

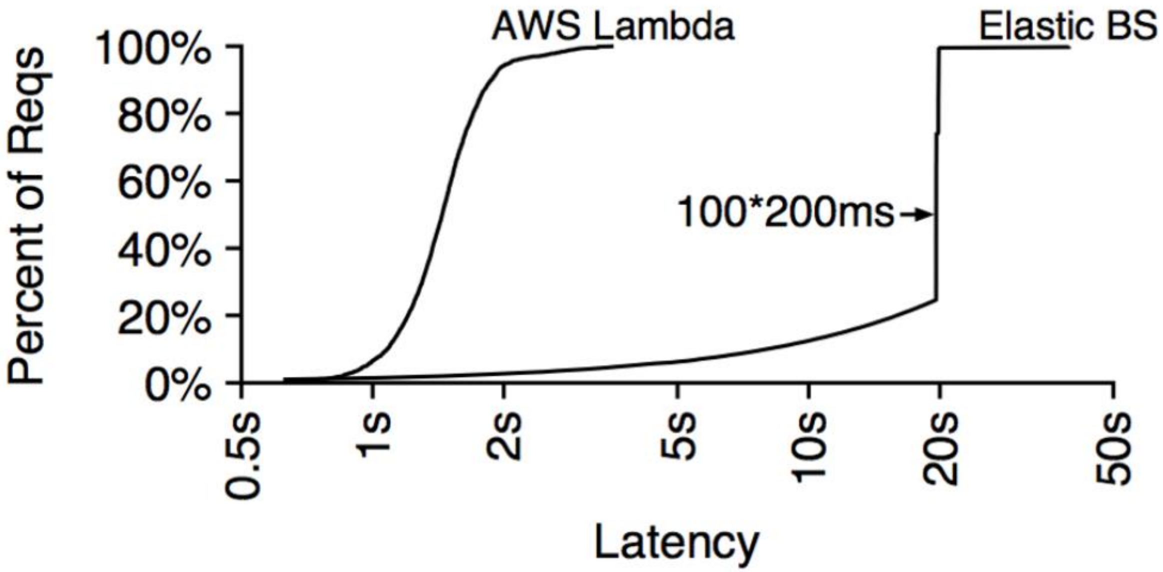
Serverless Computation with OpenLambda,

Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani†, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

- Experimental setup:
  - Amazon Elastic Beanstalk
    - Autoscaling cloud service
    - Build applications as containerized servers, service RPCs
    - Rules dictate when to start/stop (various factors)
  - AWS Lambda serverless functions
- Workload
  - Simulate a small short burst
  - Maintain **100 concurrent requests**
  - Use **200 ms** of compute per request
  - Run for **1 minute**

Scalability result

- AWS Lambda RPC has a median response time of only 1.6s
  - Lambda was able to start 100 unique worker instances within 1.6s
- An RPC in Elastic BS often takes 20s.
  - All Elastic BS requests were served by the same instance; as a result, each request had to wait behind 99 other 200ms requests.



Functions vs explicit provisioning

- With VMs or containers, we need to decide
  - What type of instances?
  - How many to spin up?
  - What base image?
  - What price spot?
  - And then wait to start…..
- Functions truly delivery the promise of the cloud
  - finally pay-as-you-go
  - finally elastic
  - will fundamentally change how people build scalable applications