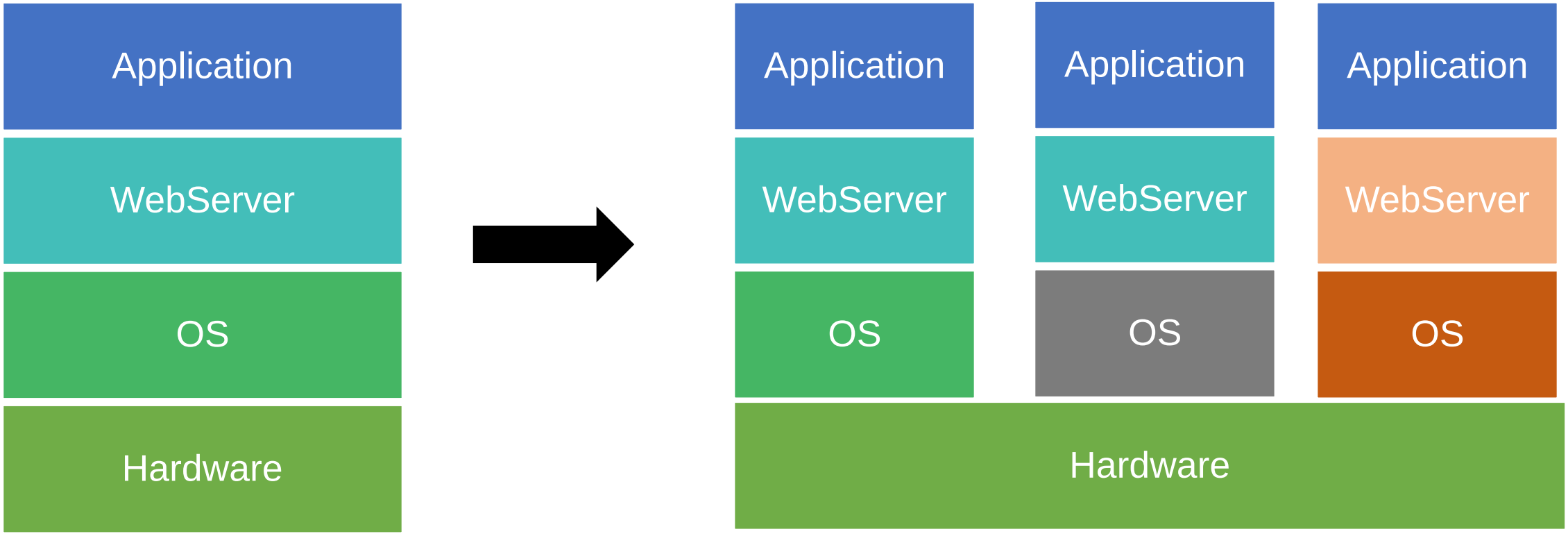
Fundamentals of Cloud Computing

Lecture 2

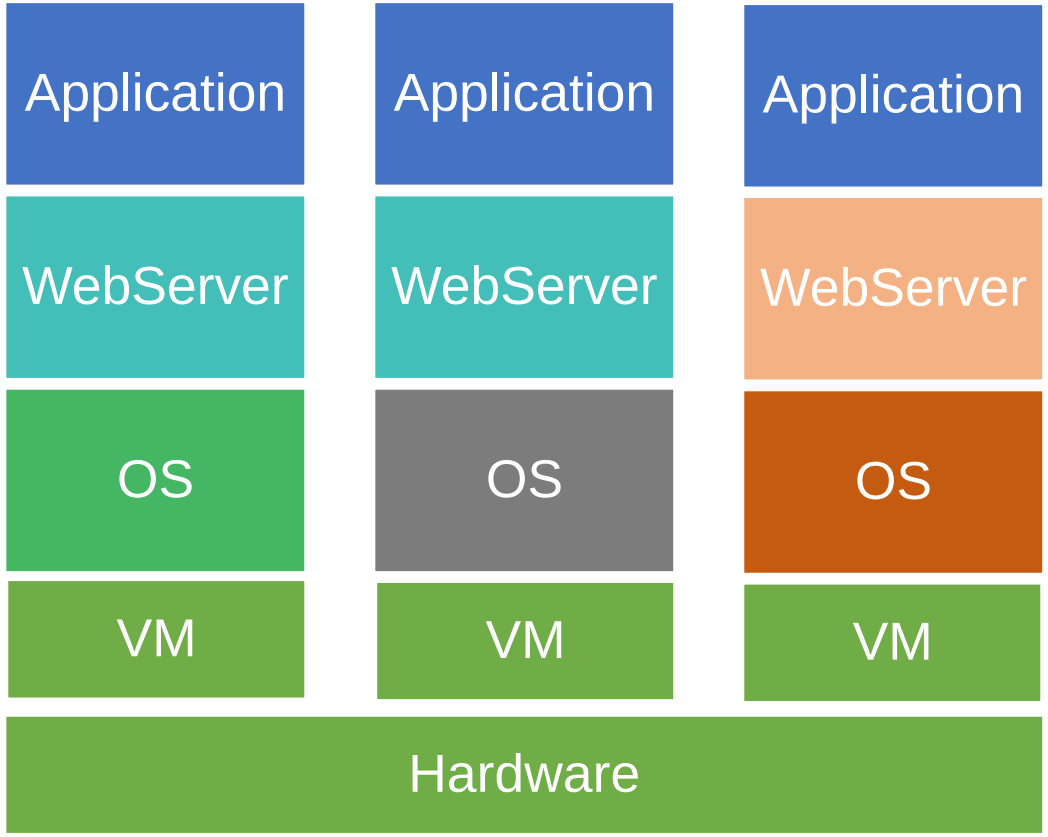
Sharing Resources

- 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

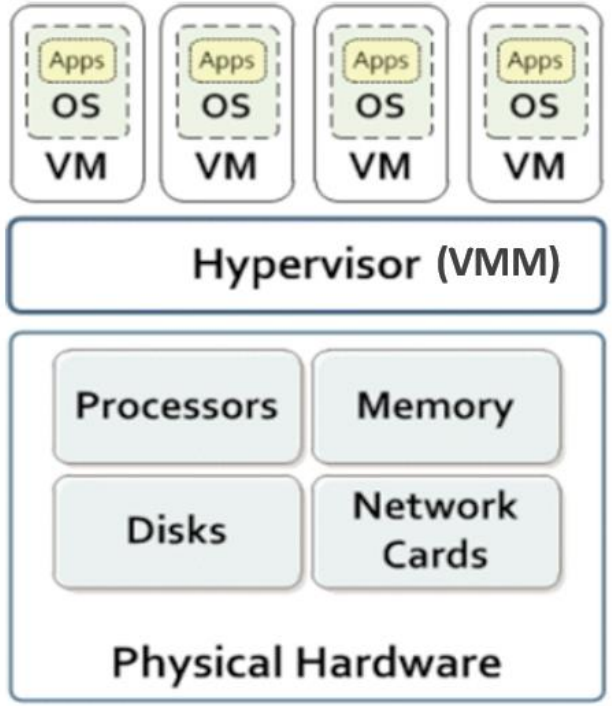


Virtual machine and Hypervisor

- 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

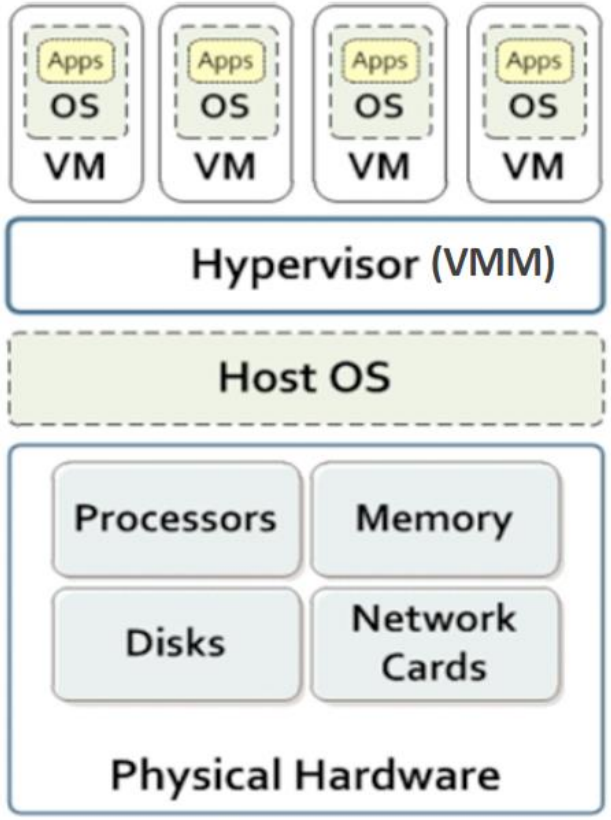
Type 1 Hypervisor

- 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



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!

IBM System/370

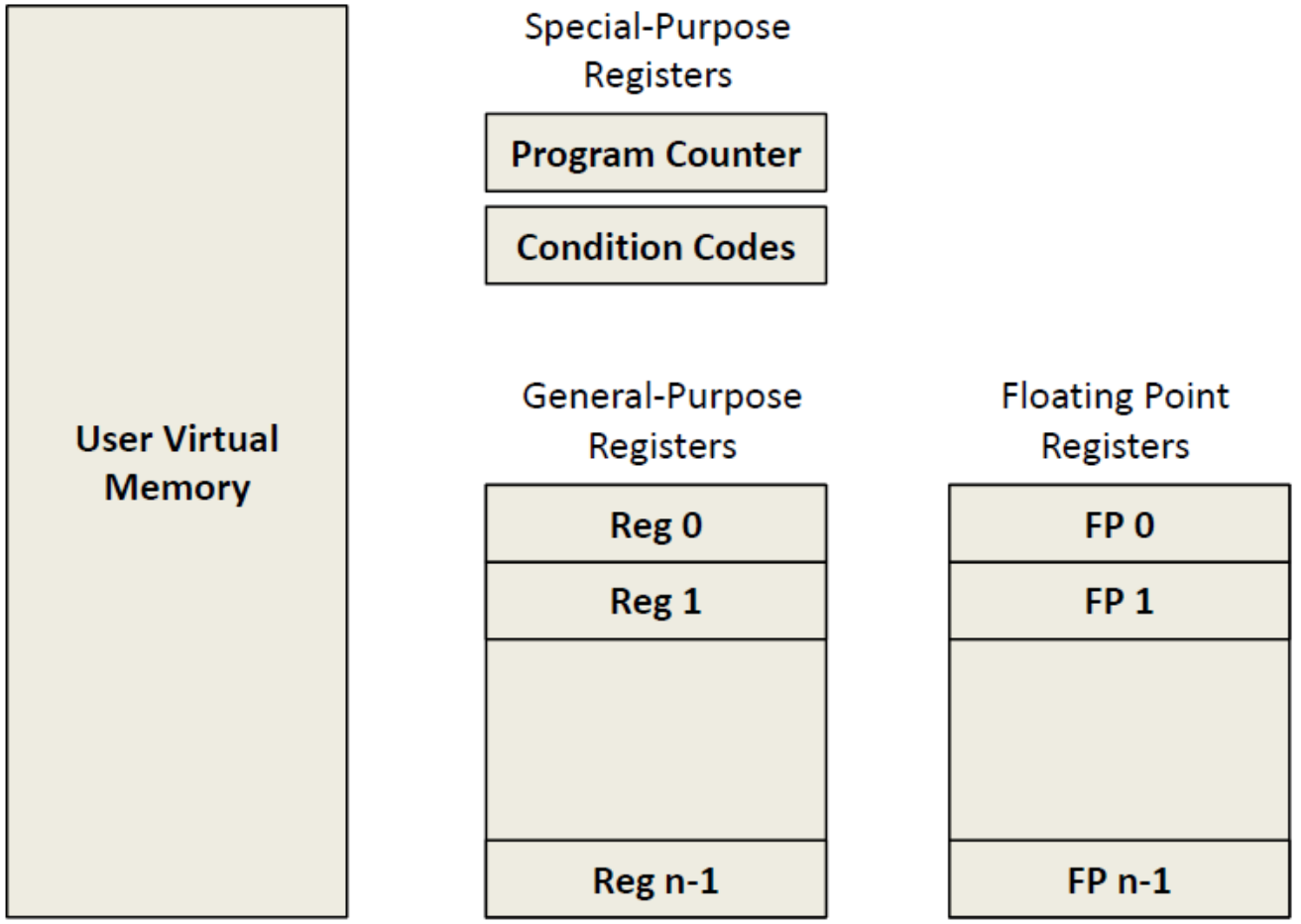
- CP/CMS (Control Program/Cambridge Monitor System), 1968
  - Time sharing providing each user with a single-user OS
  - Allow users to concurrently share a computer
- Hot research topic in 60s-70s; entire conferences devoted to VMs

CPU Organization Basics

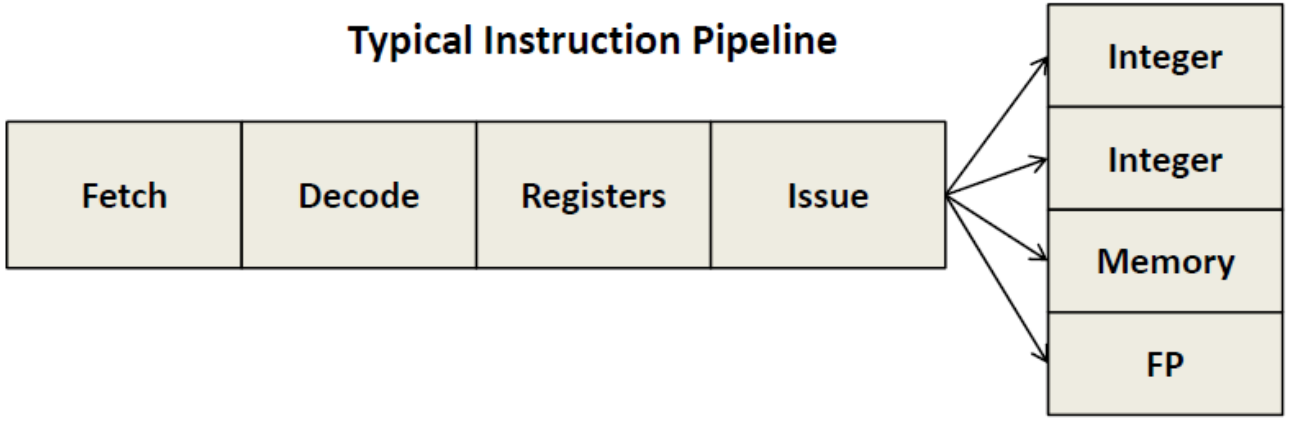
- Instruction Set Architecture (ISA)
- Defines:
  - the state visible to the programmer
    - registers and memory
  - the instruction that operate on the state

- ISA typically divided into 2 parts
  - User ISA: Primarily for computation
  - System ISA: Primarily for system resource management

User ISA: State



User ISA: Instructions

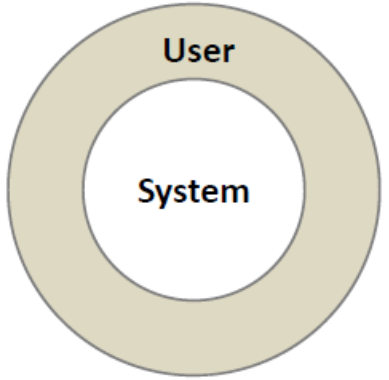


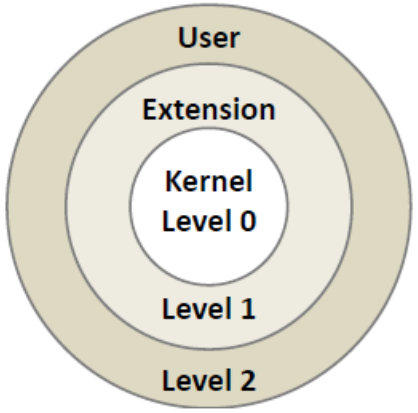
Integer	Memory	Control Flow	Floating Point
Add Sub And Compare ...	Load byte Load Word Store Multiple Push ...	Jump Jump equal Call Return ...	Add single Mult. double Sqrt double ...

Instruction Groupings

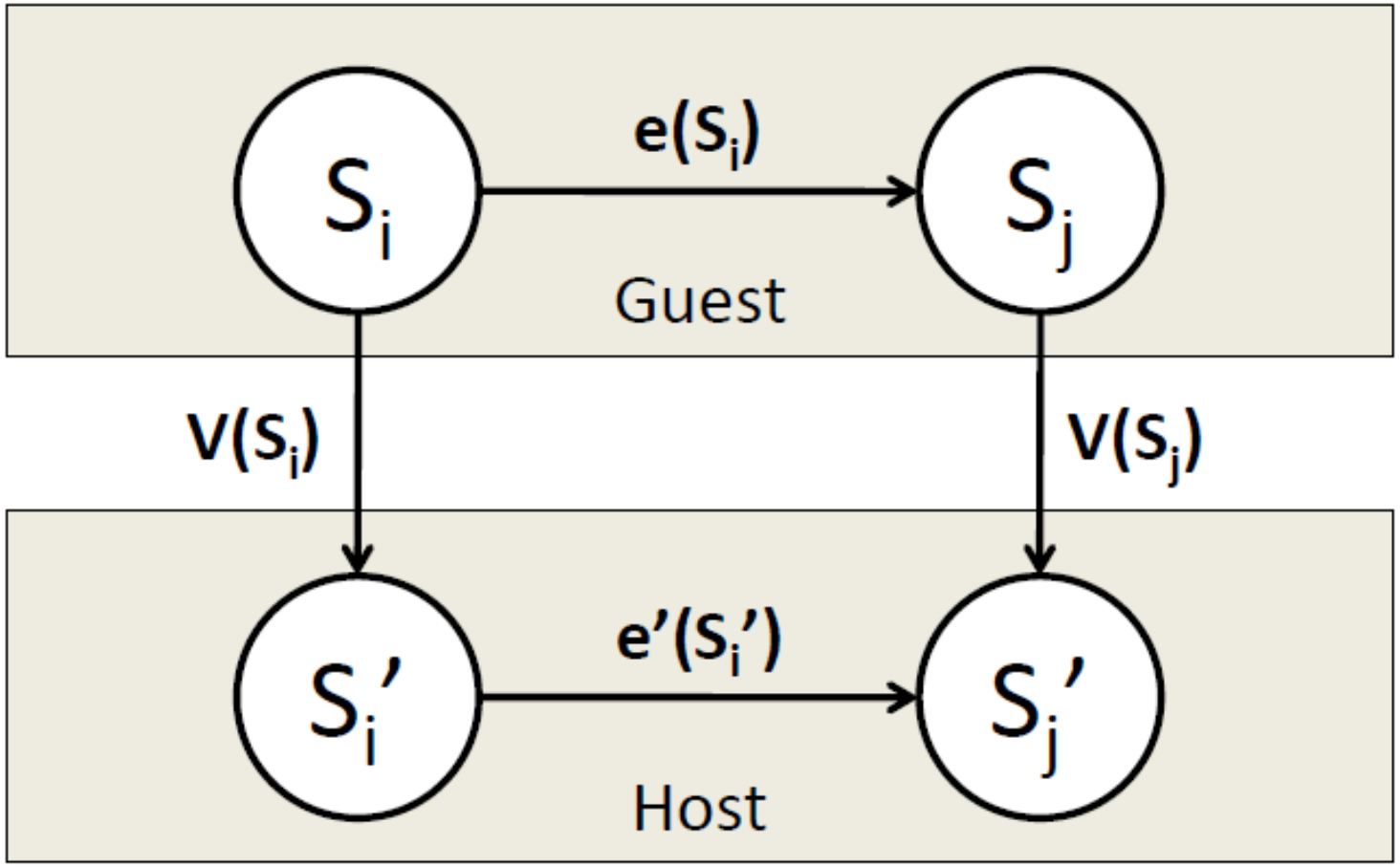
System ISA

- Privilege levels
- Control registers
- Traps and Interrupts
  - Hardcoded vectors
  - Dispatch table
- MMU
  - Page tables
  - Translation Lookaside Buffer
- I/O device access





Virtualization: An Isomorphism



Formally, virtualization involves the construction of an isomorphism from guest state to host state

Virtualizing System ISA

- Hardware needed by monitor
  - Ex: monitor must control real hardware interrupts
- Access to hardware would allow VM to compromise isolation boundaries
  - Ex: access to MMU would allow VM to write any page
- All access to the virtual System ISA by the guest must be emulated by the monitor in software.
- System state kept in memory.
- System instructions are implemented as functions in the
- monitor.

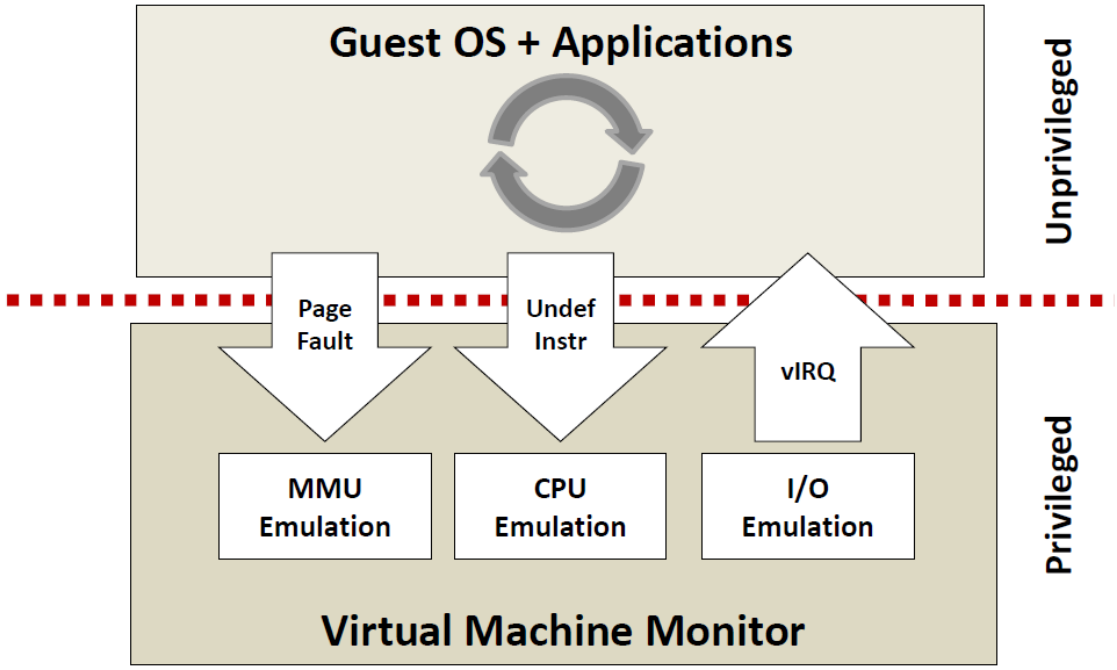
Virtualization with interpretation

```
static struct {
   uint32  GPR[16];
   uint32  LR;
   uint32  PC;
   int     IE;
   int     IRQ;
} CPUState;
```

- Fetch/Decode/Execute pipeline in software
- CISC instruction: incl(%eax)
  - int *ip; …. ++(*ip);
- Interpreting it as a C function
  - r = GPR[EAX];
  - tmp= read_mem(r);
  - tmp++;
  - write_mem(r, tmp);
- Postives: Easy to implement, done by emulators (Bochs)
- Negatives: Super slow compared to direct execution (~50x)!

Trap & emulate approach

- “A processor or mode of a processor is strictly virtualizable if, when executed in a lesser privileged mode:
  - all sensitive instructions that modify system state must be privileged
  - all instructions that access privileged state trap
  - Popek & Goldberg, 1974



Virtualization fades away (1980s)

- Interest died out in 80s
  - More powerful, cheaper machines
- 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
- Central problem: Not all architectures are strictly virtualizable
  - X86 has several sensitive instructions that do not trap

Virtualization with binary translation

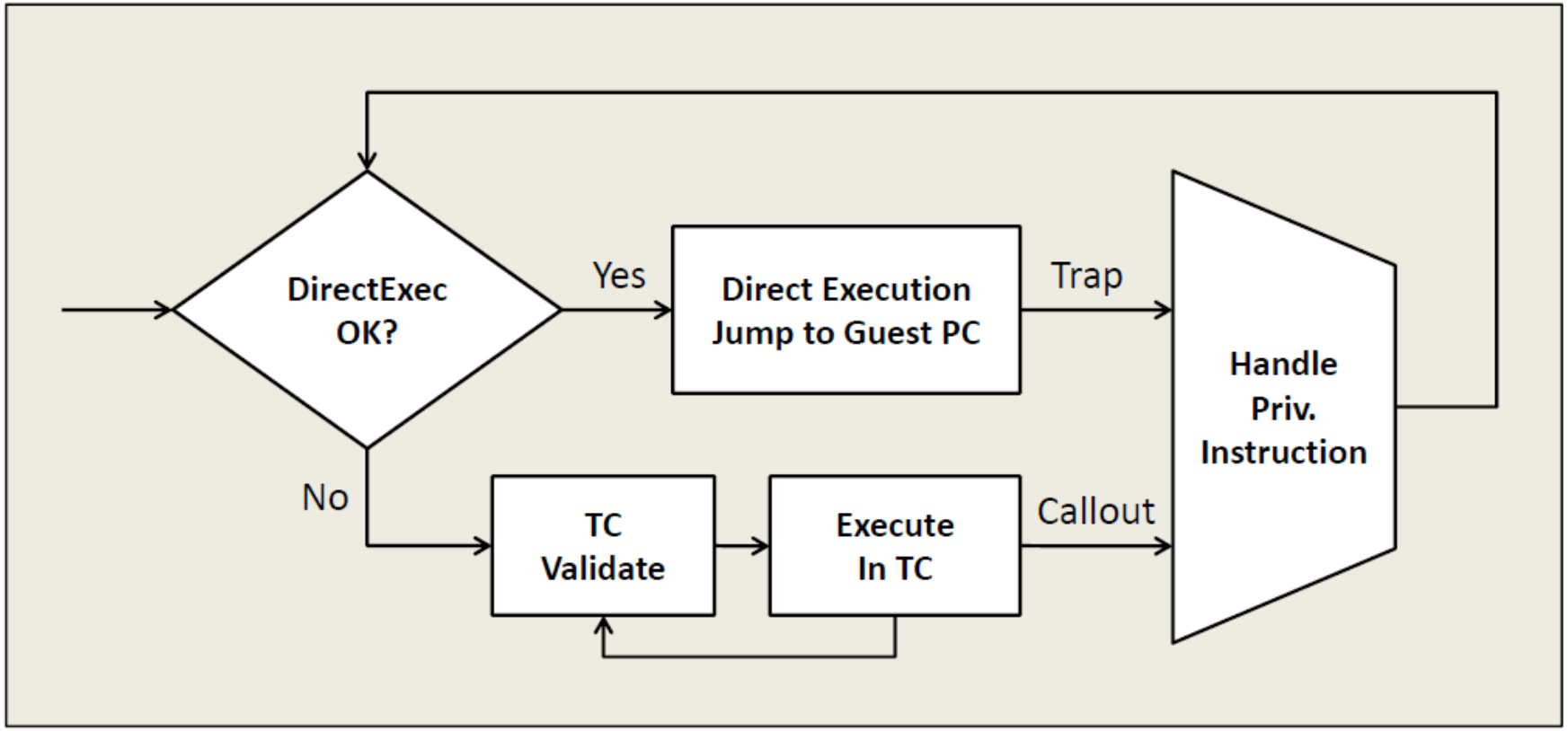
- Translate each guest instruction to the minimal set of host instructions required to emulate it

- Ex: incl (%eax)
  - leal mem0(%eax), %esi
  - incl(%esi)

- Advantages
  - Avoid function-call overhead of interpreter-based approach
  - Can re-use translations by maintaining a translation cache

- Disadvantage: Still slow than direct execution (~5x)
  - Done by QEMU

Disco's approach

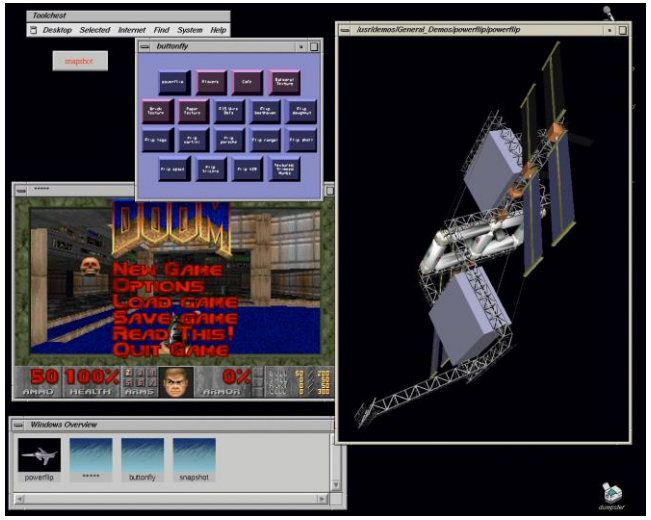


- Binary Translation for the Kernel
- Direct Execution (Trap-and-emulate) for the User
- U.S. Patent 6,397,242

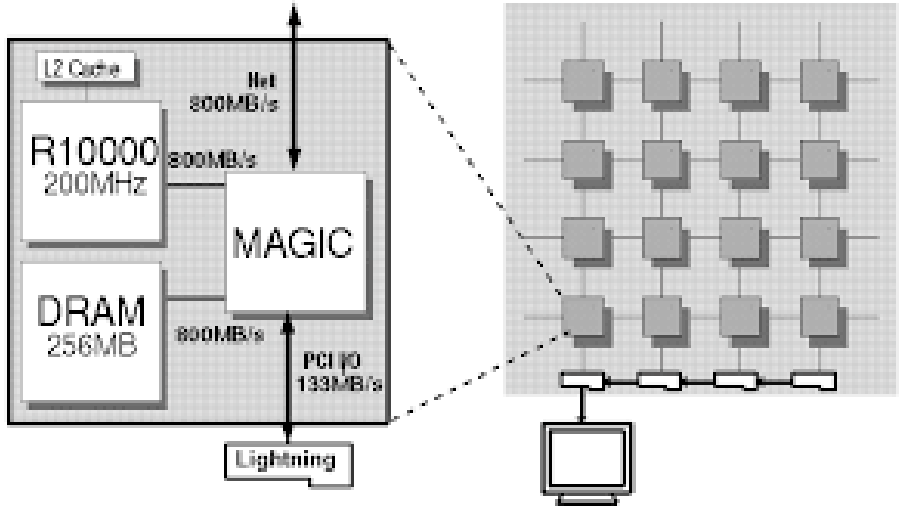
Disco

- 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 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

Creating VMs with vagrant: Demo

- Vagrant is a tool for building and managing virtual machine environments in a single workflow.
  - Machines are provisioned on top of VirtualBox, VMware, AWS, or any other provider.
  - Provisioning tools such as shell scripts, Chef, or Puppet, can automatically install and configure software on the virtual machine.
- Creating VM demo
  - `vagrant init hashicorp/bionic64`
  - `vagrant up`
  - `vagrant ssh`
    - `sudo apt install stress; stress -c 1 -t 120;`
  - `Vagrant suspend; vagrant resume;`
  - `Reprovision`
    - `Uncomment config.vm.network "forwarded_port", guest: 80, host: 8080`
    - `Uncomment vbox provisioning section and add vb.cpus = 3`
    - `Add config.vm.provision :shell, path: "bootstrap.sh" to provisioning section`
    - `Vagrant reload --provision`
    - `wget http://localhost:8080/index.html`

Virtualization today

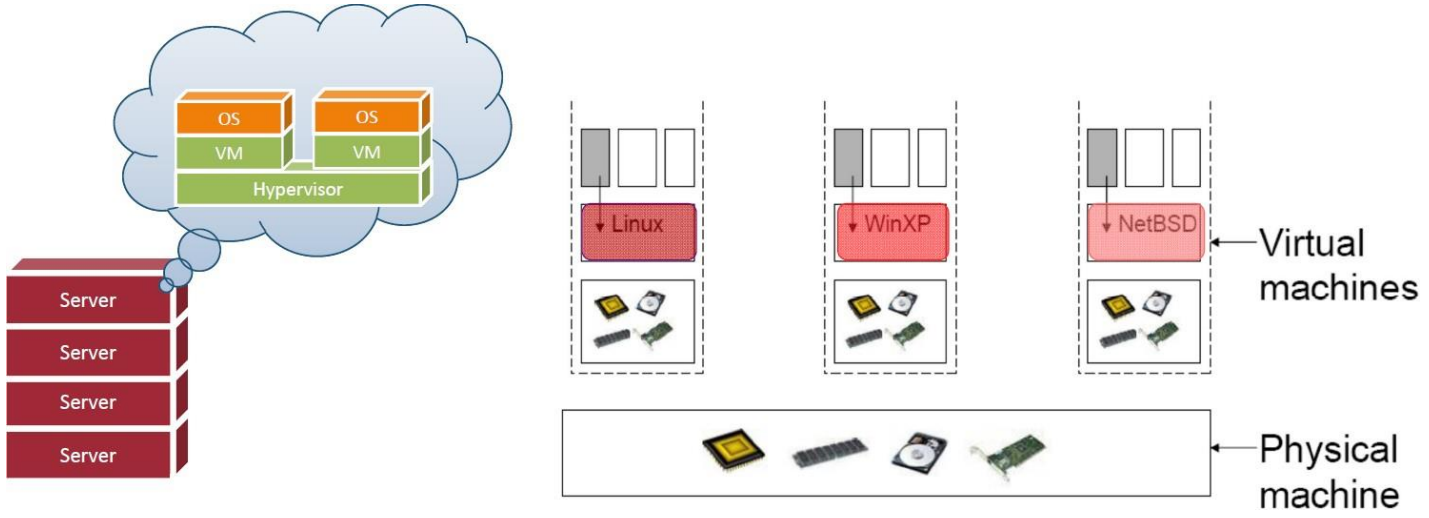
- Support for virtualization in hardware for x86
  - Intel VT and AMD-V (2008)
  - Optimization: VMCBs (virtual machine control blocks)
    - Shadow state for guest maintained by hardware
    - Many privileged instructions update/read only VMCB rather than reading the actual hardware
    - VMCB maintained and read by VMM.

- We did not cover
  - Memory, I/O, storage virtualization, Para-virtualization, nested virt., …
  - Check book “Hardware and Software Support for Virtualization”

Virtualization and the cloud: IaaS recap

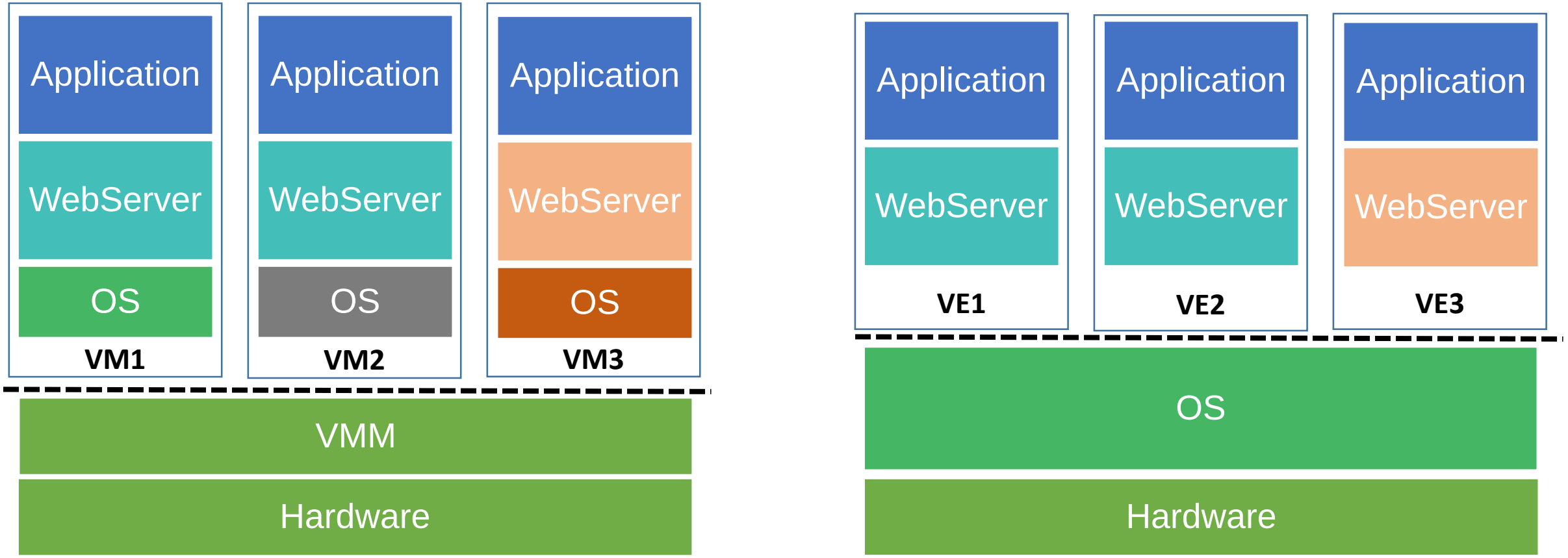
Virtualization is the enabler of IaaS

- 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

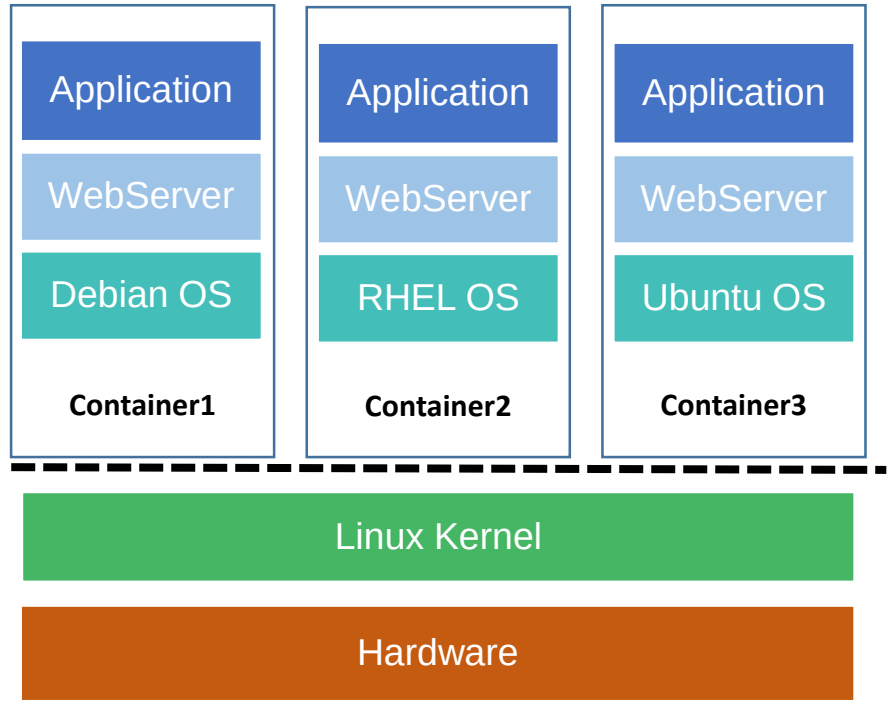
- Hosting providers & PaaS providers
  - Need to host multiple applications/tenants on a single server
- Full hardware virtualization still expensive
  - Full OS + libraries + application per tenant => reduced multitenancy
  - Use of VMMs means license fee
- OSes have always supported multitenancy
  - 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

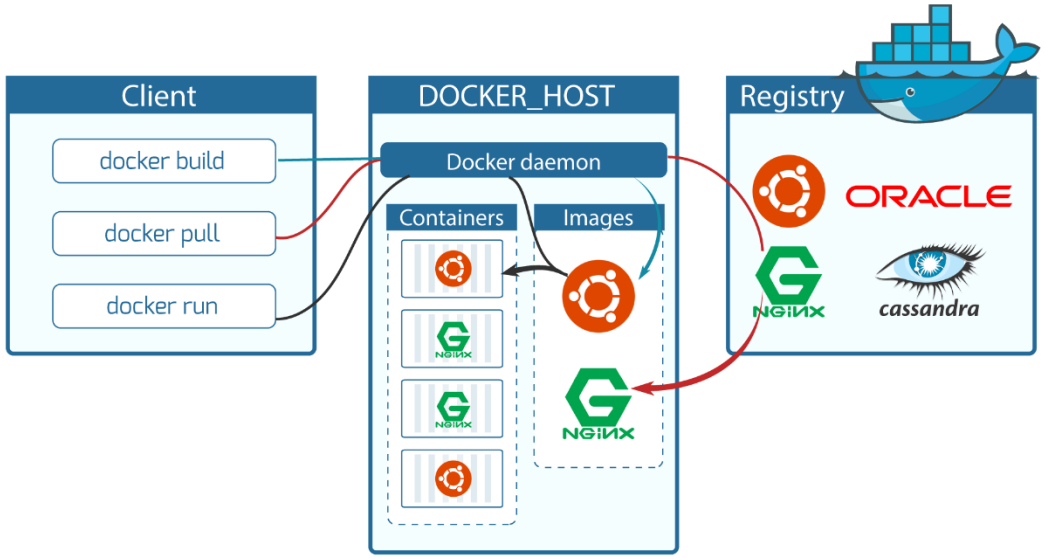


From LXC to Docker

- 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)

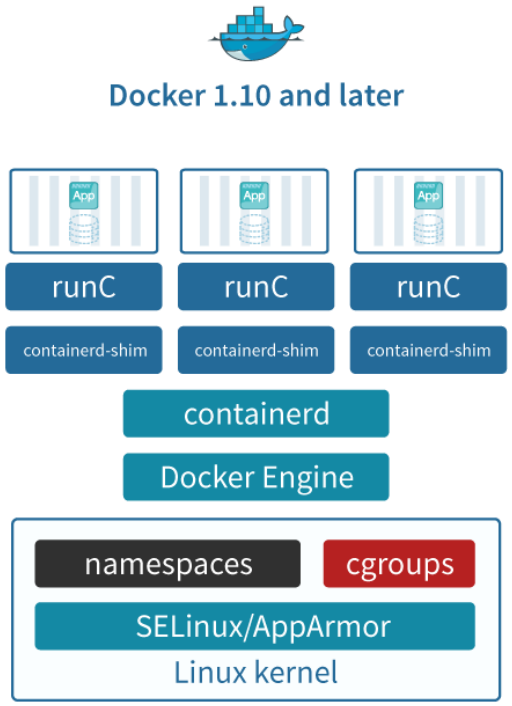


Creating containers with Docker: Demo

- Reprovisioning the VM to install docker
  - Can use the vagrant **provisioner** to install docker
    - Can automatically build image, pull image, run container, …
    - Note: Vagrant can also use Docker as a **Provider**. NOT THE SAME!
    - Add line config.vm.provision "docker" to Vagrant file
    - `vagrant reload –provision`
- Running a packaged application
  - `Wget https://github.com/raja-appuswamy/accel-align-release/archive/refs/tags/v1.0.0.tar.gz`
    - `Application does not run due to uninstalled dependency`
  - `Build and run docker image`
    - `Update Dockerfile to use libtbb2 rather than libtbb-dev`
    - `docker build --tag accalignlocal:1.0 .`
    - `docker image ls`
    - `docker run –it –name ac1 accalignlocal:1.0`
  - `Or pull from dockerhub directly`
    - `docker run -it rajaappuswamy/accel-align`
- Destory VM
  - `vagrant destroy`

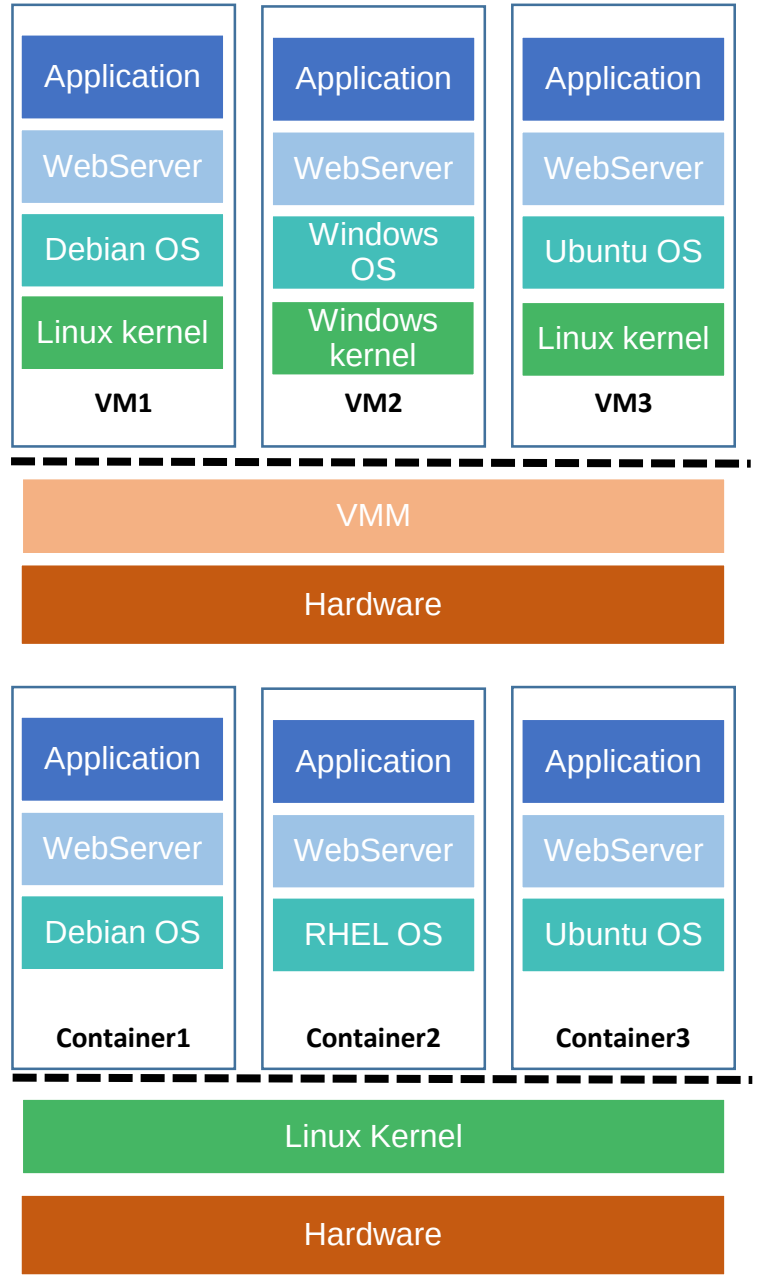
Container ecosystem today

- 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

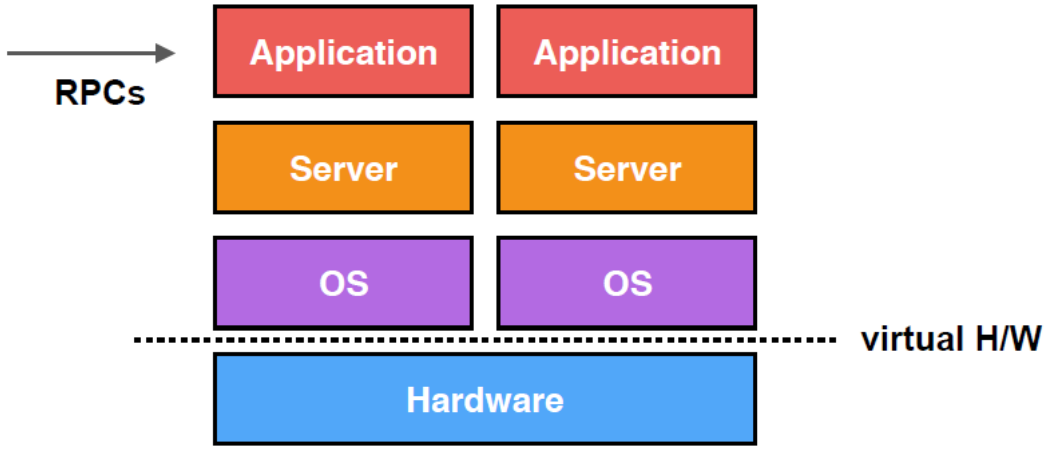
Container Case Study

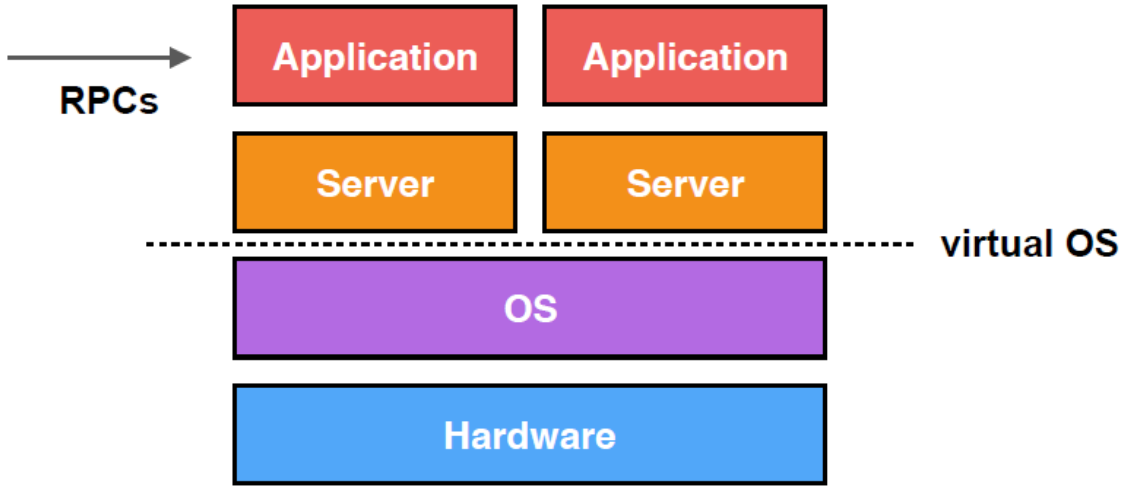
- Google Borg
  - Internal container platform at Google (http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/43438.pdf)
  - 25 second median startup!
    - 80% of time spent on package installation

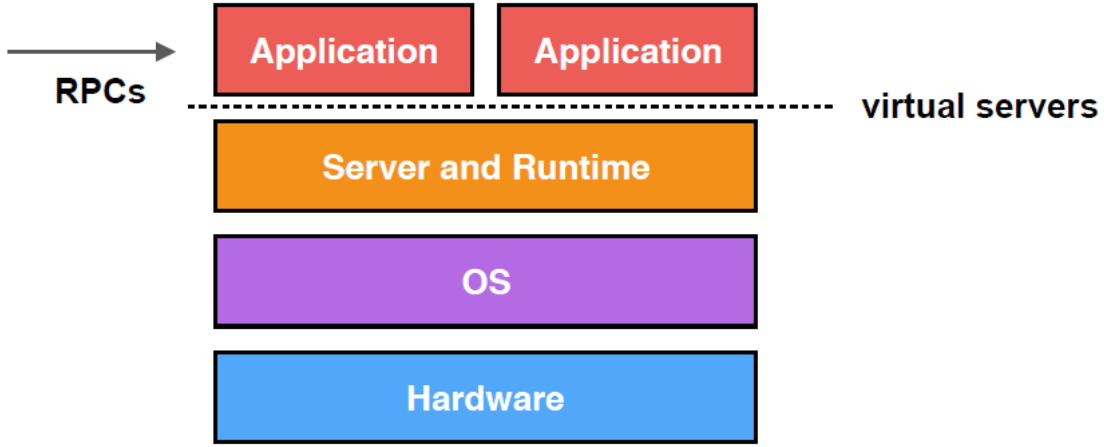
- What if we have an light-weight HTTP app?
  - Say < 1,000 LoC that runs for 200 msecs on each HTTP request?
  - In a container, the app would take too long to start

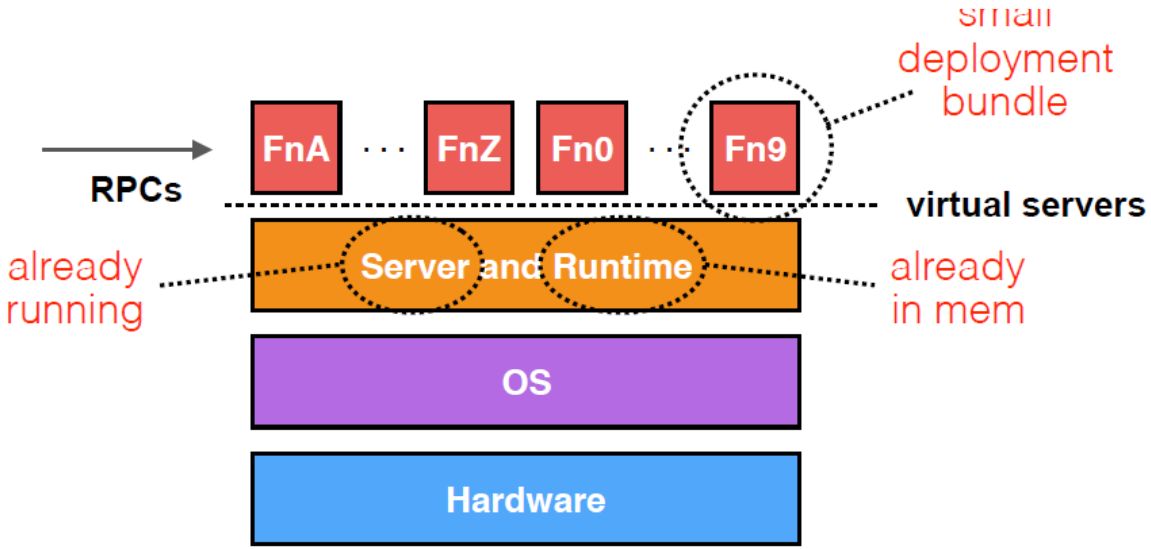
- Containers cannot deal with flash crowds, load balancing, interactive development, etc

Three generations of virtualization





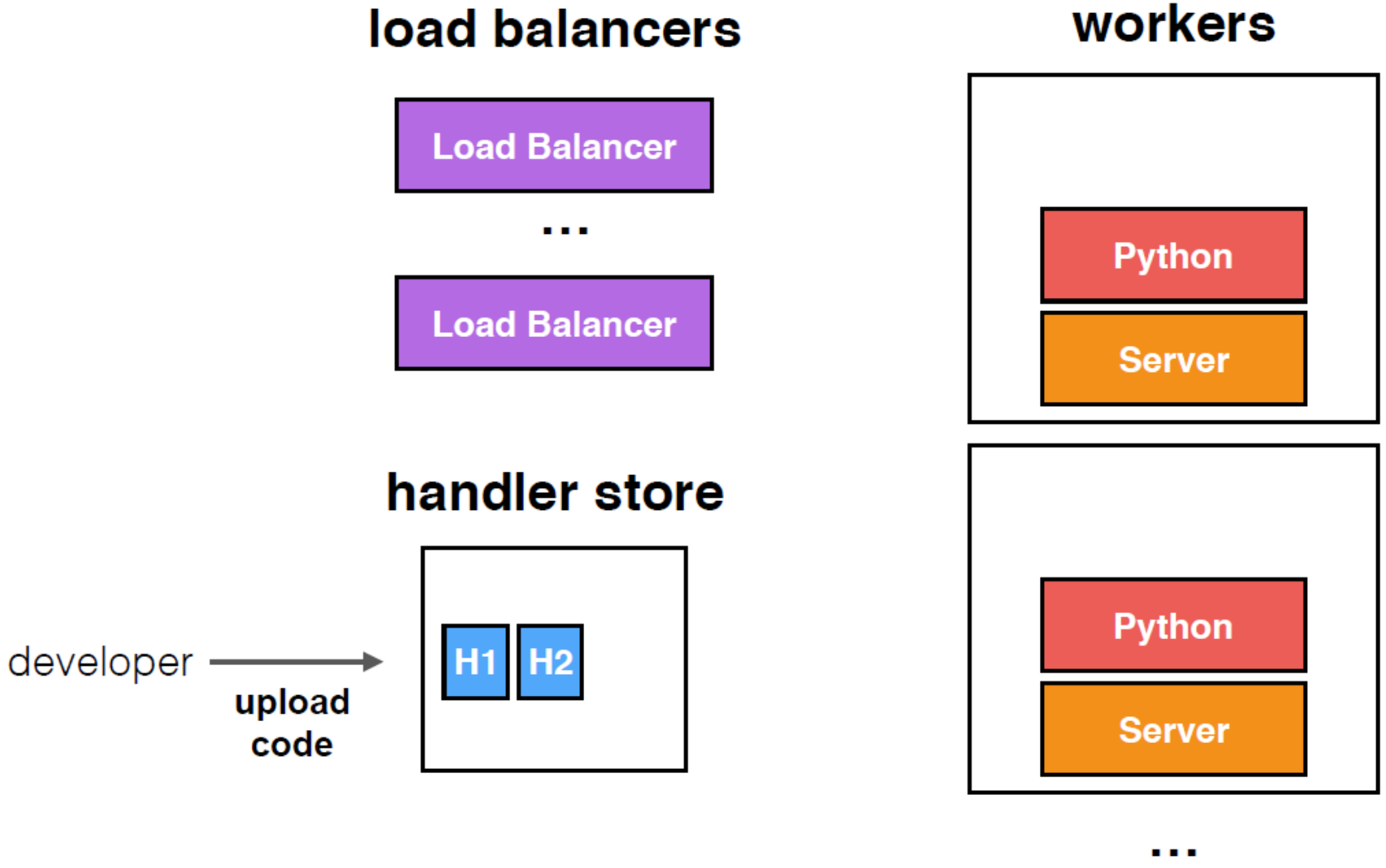




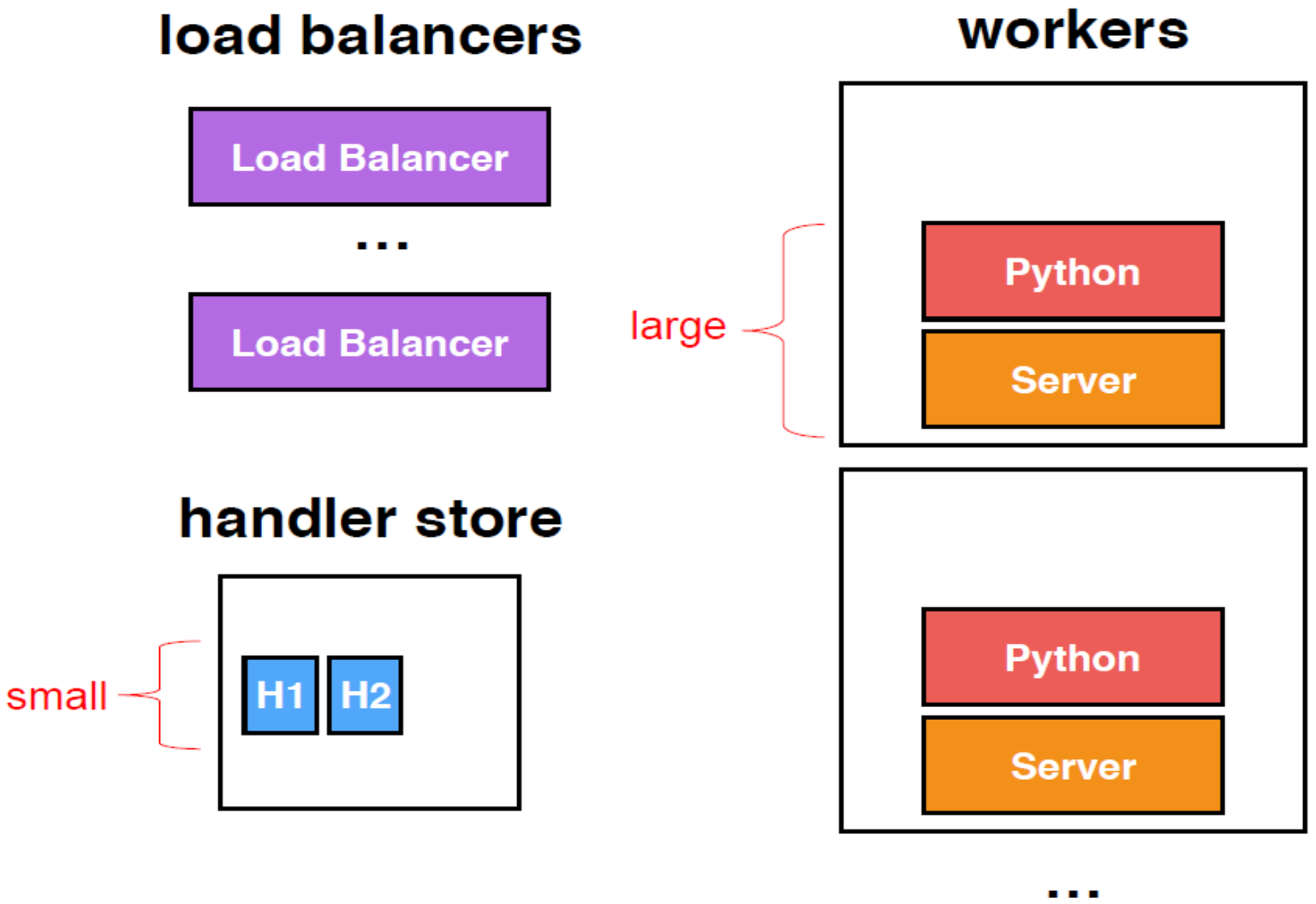
Serverless functions: Model

- 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

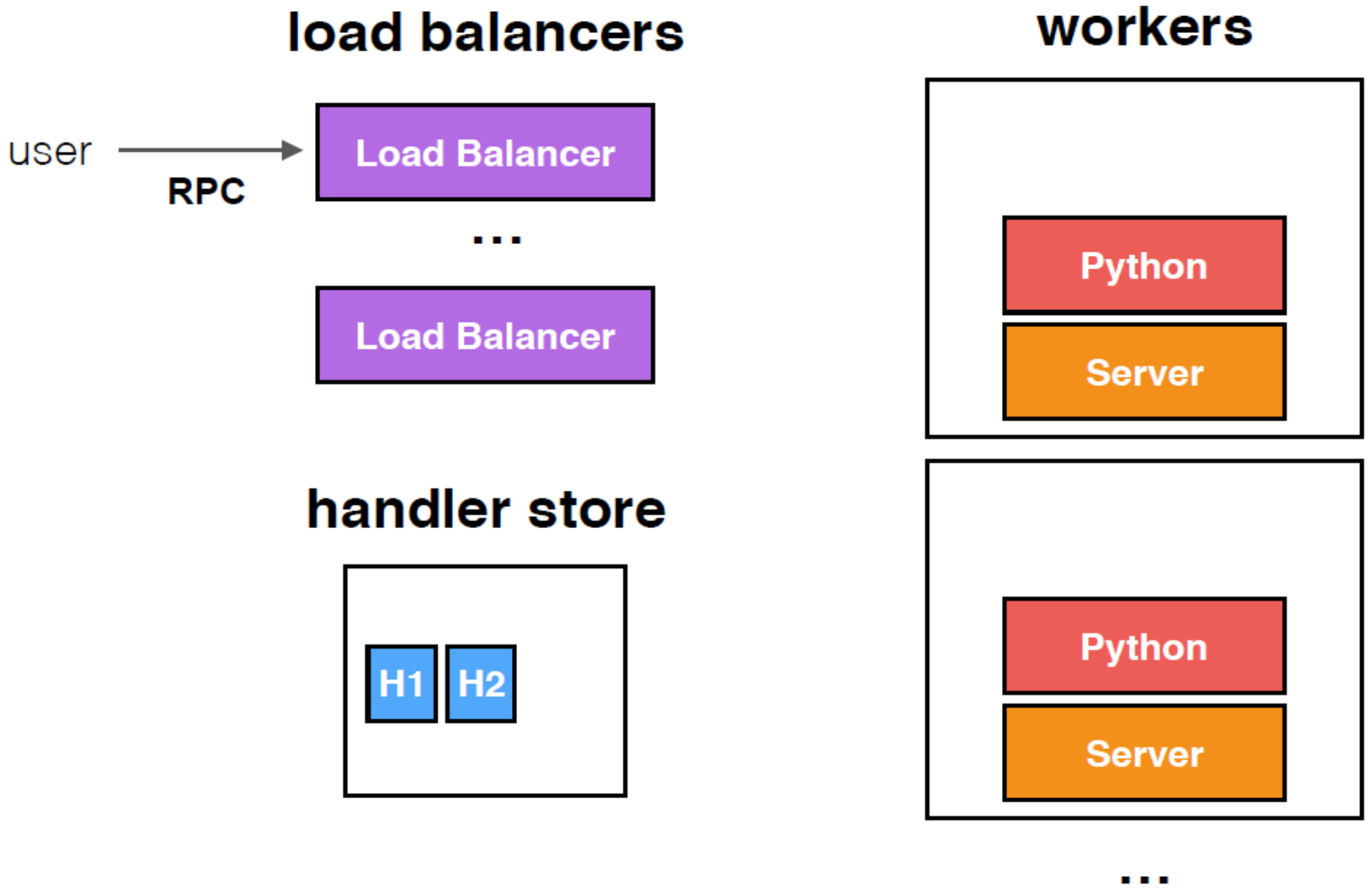
Architecture



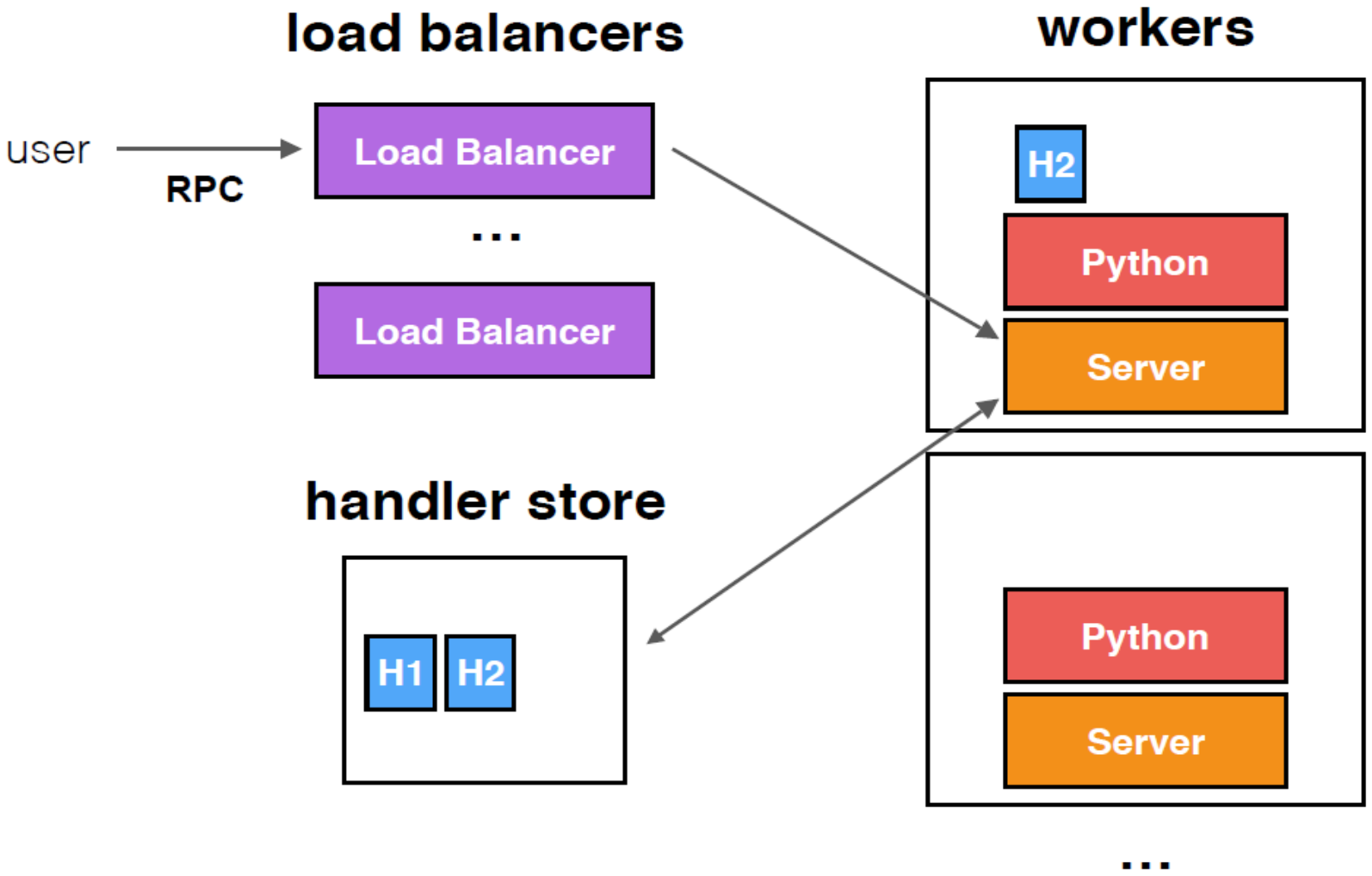
Architecture



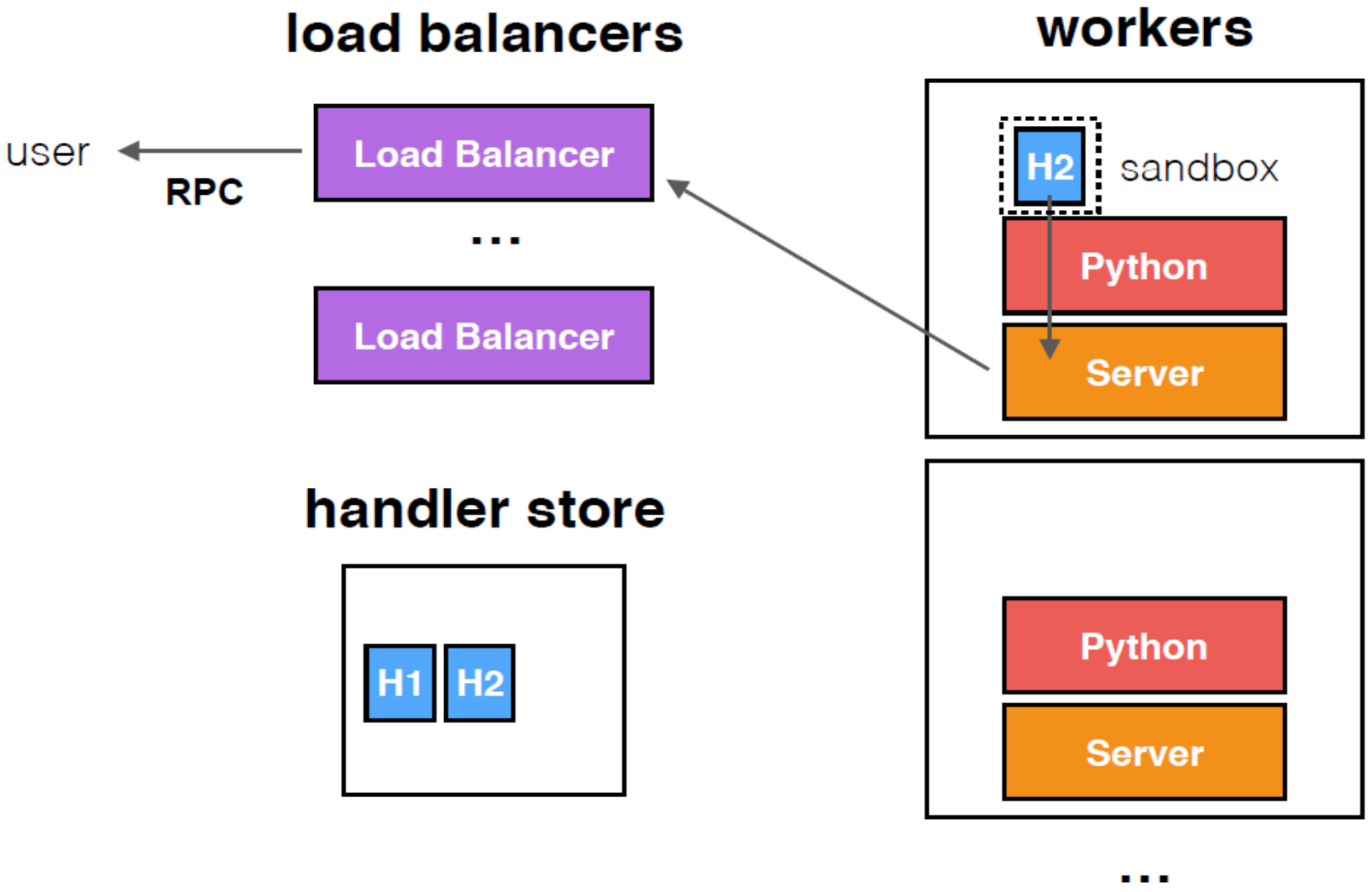
Architecture



Architecture



Architecture



Functions vs containers

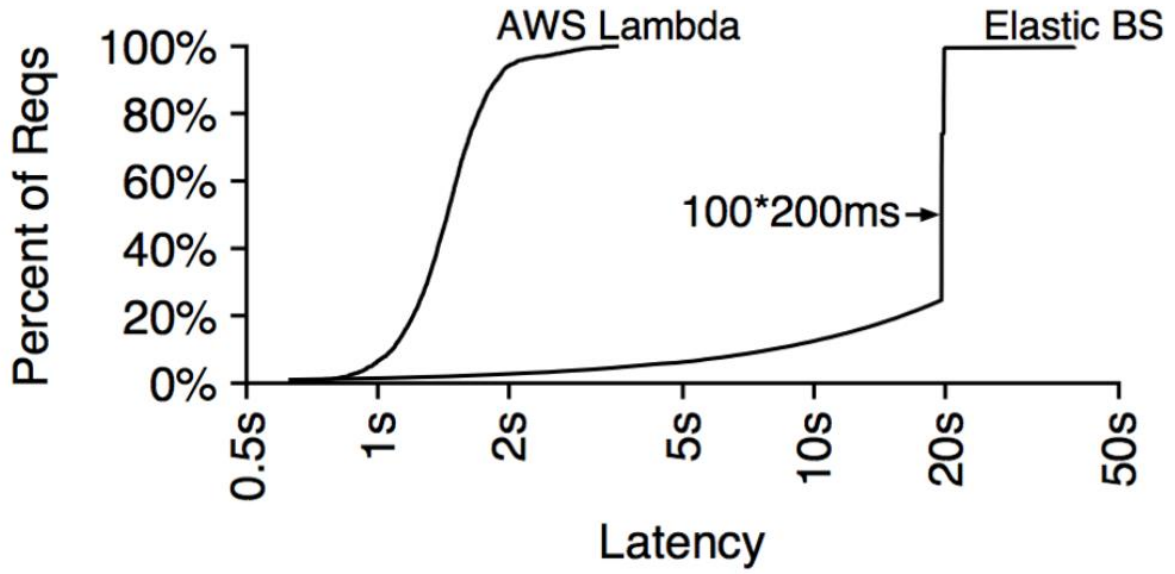
Serverless Computation with OpenLambda,

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