Virtual Machines and Containers

- What is a virtual machine?
- How to make a virtual machine fast
- Paravirtualization
- Containers

What is a Virtual Machine?

- An **emulator** replicates another system, entirely in software
- A **virtual machine** replicates another system, as much in hardware as possible
  - Some people might include emulators as a type of virtual machine
- A **container** divides processes into small sub-environments, all using a common kernel

Emulators

- **Emulators** replicate the entire system in software
  - Can support cross-platform VMs
  - Can implement legacy (or fictional) platforms
  - Allow for experimentation
  - But slow

Example: Bochs (x86 emulation)

JIT

- **JIT** (just-in-time compilation)
  - Decodes a program in software
  - Then writes a new program to do the same work
  - Compiles the program when needed
  - Runs at native speed (almost)

Example: Java Virtual Machine

JIT

Emulator / Interpreter	**JIT**
• Decode Instruction	• Decode Instruction
• Call:	• Print out:
– C code	– C code
	• Compile C code
• Run natively	• Run natively

JIT

- **TPS:**
  - A JIT compiler does the same work as the emulator, *plus* some new things. Why does this pay off?

JIT

- **TPS:**
  - A JIT compiler does the same work as the emulator, *plus* some new things. Why does this pay off?

- Loops!
  - Most code runs many times
  - But requires block-analysis in the JIT system
    - Where are the jumps?

Emulators+JIT

- **Emulators** can use **JIT**
  - JIT-compile code before it runs
    - Add checks for syscalls, interrupts
    - Kernel code may have different rules/powers than user code
  - Will run at (near) native speed

Example: QEMU

Emulators+JIT

- **Emulator + JIT** : is it a VM?
  - Fuzzy boundary
  - Most users don’t know, don’t care

  - I’ll draw a distinction in these slides, but just to point out alternate strategies

Virtual Machines

- **Virtual Machines**
  - Run code that was written for real machines
    - Including kernel code
  - Typically try to replicate real hardware
    - Actual disks, actual displays, actual network cards
  - Allow maximum flexibility

Reminder: A lot of people might say that emulators are just one type of virtual machine. I don’t disagree, but will make a distinction for just this slide deck.

Virtual Machines

Glossary

- A **hypervisor** is the code that manages one or more VMs
  - Compare to “supervisor” (a.k.a. kernel)
  - Sometimes called the “monitor” or “vmm”
- The **host** is the OS that runs natively on the hardware
  - Often, the hypervisor is a program in this OS
  - Sometimes, the hypervisor is a component of the host kernel

Virtual Machines

Glossary

- A **guest** is a VM running inside some hypervisor
- The **guest kernel** is software that must run as the kernel inside its VM, but ***must not*** be given real kernel permissions in the host.

Virtual Machines

- **Virtual Machines**
  - Allow multiple VMs on one physical computer
  - Can run any OS, entirely unmodified*
  - Allow you to simulate as many (or little) CPUs and memory as you like
    - Should we overcommit? Typically not
  - Are indistinguishable internally from a real machine
    - Great for security "honeypots"

* We are going to talk about **paravirtualization** soon.

Virtual Machines

- **Virtual Machines**
  - Run the same architecture* as the “host” machine
  - Typically safe to run user code natively
    - Need to set up virtual memory
    - Need to intercept system calls
    - But otherwise relatively easy

* Since we’re not including emulators as VMs in this deck

Virtual Machines

- **TPS:**
  - Why is running the **guest kernel** code so much harder than guest user code?

Virtual Machines

- **TPS:**
  - Why is running the **guest kernel** code so much harder than guest user code?

- Hardware access (devices)
- Page table config
- Processor status, protected registers
- Interrupts
- Context Switches → (more)

Virtual Machines

- **TPS:**
  - Why is running the **guest kernel** code so much harder than guest user code?

- Physical page allocation
- Swapdisks
- Networking (what is my IP address, how do I receive a packet?)
- HLT instruction (or equivalent)

Virtual Machines

- Guest kernel problems generally fall into three categories
  - Access to memory
    - Page tables
    - Devices
    - Other processes
  - **Privileged instructions**
    - Set page table, etc.
  - (coming soon)

Virtual Machines

- Memory-related issues can be caught with **page faults**
  - Guest kernel tries to touch a page
  - Fails, hypervisor gets involved
  - Either add access or *emulate* the access

- But what if you have many accesses in a row?
- What if the memory controls hardware, such as a page table?

Virtual Machines

- Intercepting memory accesses with page faults
  - Correct
  - Can support legacy kernels
  - But slow

Virtual Machines

- Similarly, privileged instructions cause Invalid Instruction exceptions
  - Catch the exception
  - Emulate the change

- Hopefully, these are rarer…
  - But it's still slow

Virtual Machines

What was the 3rd problem? (It's a nasty one!)

Virtual Machines

What was the 3rd problem? (It's a nasty one!)

"Support for full virtualization was never part of the x86 architectural design. Certain supervisor instructions must be handled by the VMM for correct virtualization, but executing these with insufficient privilege fails silently rather than causing a convenient trap [36]. Efficiently virtualizing the x86 MMU is also difficult."

"Xen and the Art of Virtualization,"
Barham et. al. SOSP '03.

Virtual Machines

- How to solve this?
  - Edit the code
    - Turn dangerous instructions into traps
  - JIT

- Or...

Paravirtualization

- **Paravirtualization** *almost* simulates the original hardware
  - But requires the guest kernel to make **hypervisor calls** for dangerous stuff
    - Page table access
    - Hardware config
    - Privileged instructions

- Needs a custom kernel build!

Paravirtualization

- The changes to the **guest kernel** are smaller than you'd think
  - Linux already supports many platforms
    - x86, ARM, RISC, PowerPC, IBM s370
  - All "dangerous" ops are already part of the "platform-specific" code. Just write new implementations

- Linux kernel source code: `arch/x86/xen/`

Paravirtualization

- **Paravirtualization** allows you to run a kernel which is 99% stock
  - Great for testing, debugging
  - Great for real-world applications
  - No need for user-app changes
  - But still pretty fast!

Paravirtualization

- **Paravirtualization** is the system of choice in the cloud
  - Mostly running Linux kernels
  - Wide variety of distros, configurations, apps
  - But Xen support has been in Linux for ages

Paravirtualization

```
> sudo dmidecode

# dmidecode 3.2
Getting SMBIOS data from sysfs.
SMBIOS 2.7 present.
11 structures occupying 378 bytes.
Table at 0x000EB01F.

Handle 0x0000, DMI type 0, 24 bytes
BIOS Information
        Vendor: Xen
        Version: 4.11.amazon
        Release Date: 08/24/2006
```

I ran this on an AWS EC2 server that I own.

Paravirtualization

```
> sudo dmidecode

# dmidecode 3.4
Getting SMBIOS data from sysfs.
SMBIOS 2.7 present.
69 structures occupying 3396 bytes.
Table at 0x000EC3C0.

...

Handle 0x0000, DMI type 0, 24 bytes
BIOS Information
        Vendor: Dell Inc.
```

The same command, on a Linux box at my house

Downsides of VMs

- **TPS:**
  - What are the downsides of running a VM?
  - How expensive is it to create a VM, can I run 100s on the same machine?

Downsides of VMs

- **TPS:**
  - What are the downsides of running a VM?
  - How expensive is it to create a VM, can I run 100s on the same machine?

- Emulated Hardware
  - Disk, network, keyboard, timer, CPUs
- Guest kernel
  - Paging, hardware access → (more)

Downsides of VMs

- **TPS:**
  - What are the downsides of running a VM?

- CPU management & balancing
- Memory load balancing & swap

- Kernel daemons (many misc tasks)
- ssh server (for remote access)

Downsides of VMs

- **Reality:**
  - Hard to run more than “a few” VMs on one host
    - Typically need ~1 CPU + >1 GB RAM each
  - Takes a while to spin up a new VM
    - Allocate space
    - Boot kernel, run init scripts
    - Start up processes

Intro to Containers

- Idea:
  - Manage several small groups of processes, inside a single kernel
  - Currently, we use **linux namespaces** to segregate our processes
  - Docker is a tool which simply *automates the steps* of creating a namespace

Intro to Containers

- A **linux namespace** allows you to give processes a "private space"
  - Processes inside a namespace see **only their local context**
  - Host can see the world
  - Lots of namespace types, a container usually defines a private namespace of *each type*

Great summary pages:
https://www.baeldung.com/linux/docker-container-process-host-pid
https://www.linkedin.com/pulse/running-containers-without-docker-ramesh-kumar

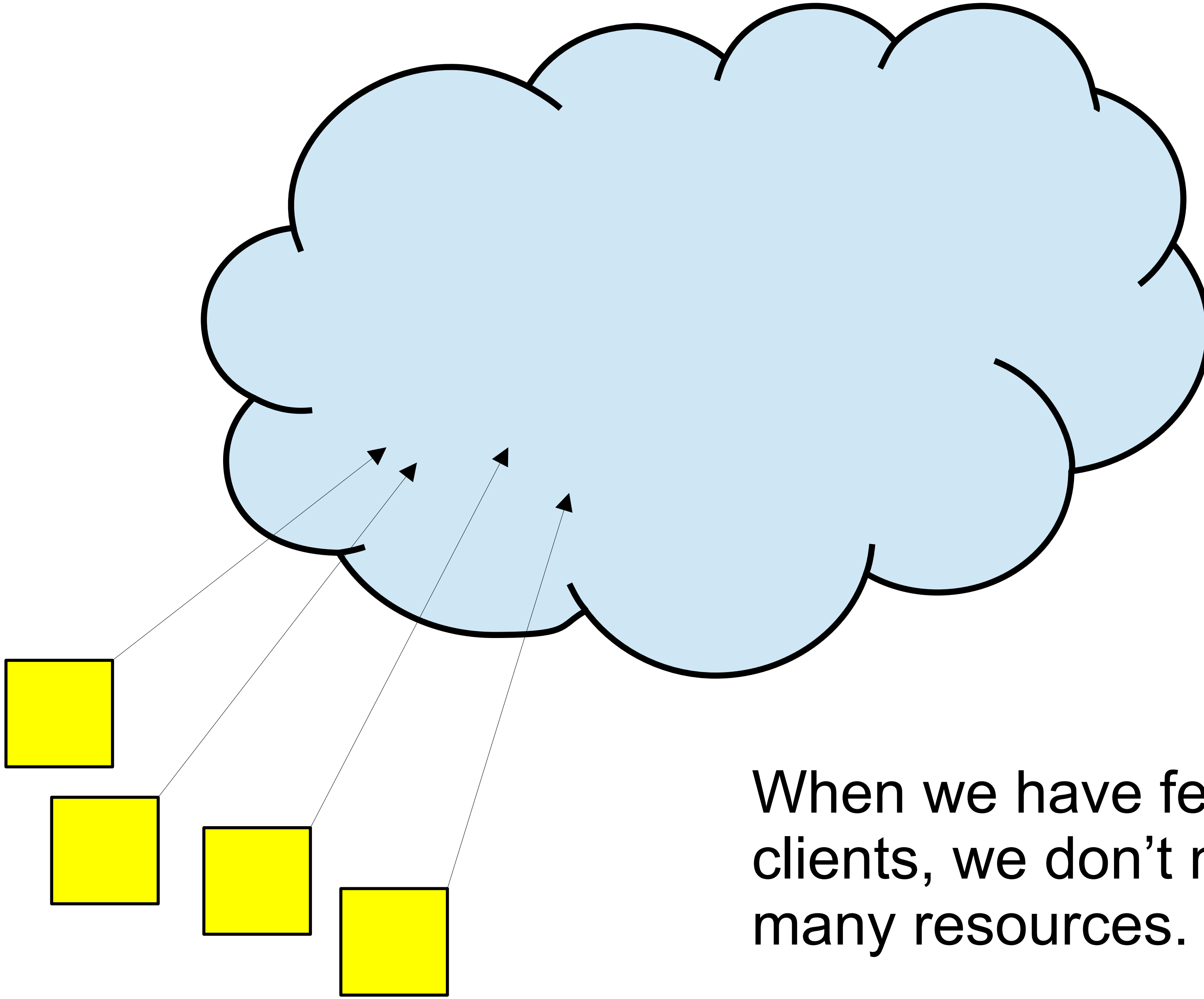
What is `docker` ?

- A tool for running **containers**

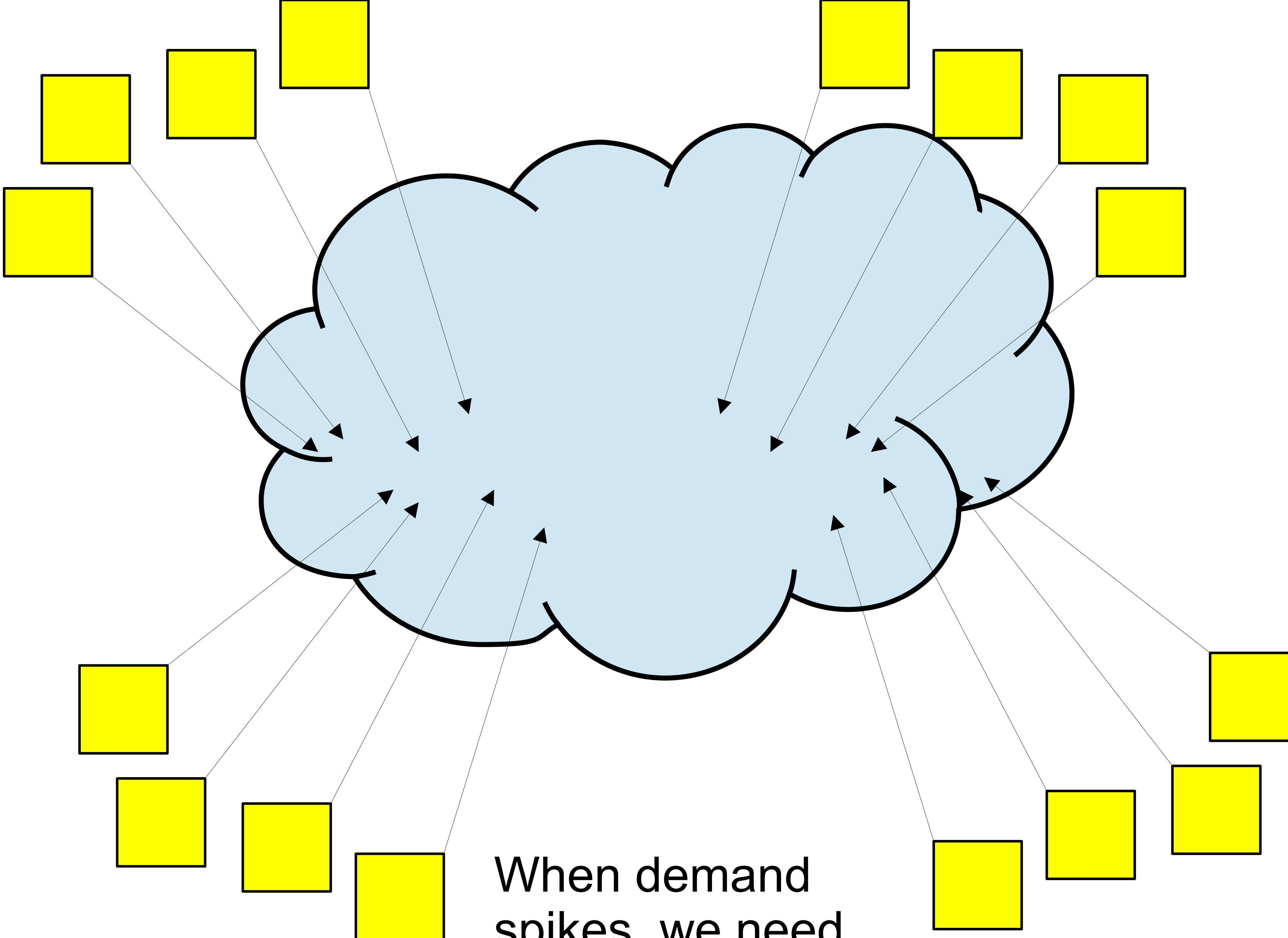
- A container is:
  - A private environment for running programs
  - Has a complete filesystem
  - Has private networking
  - It's ***almost*** like a Virtual Machine
- But **why** are they useful???

“The Cloud”

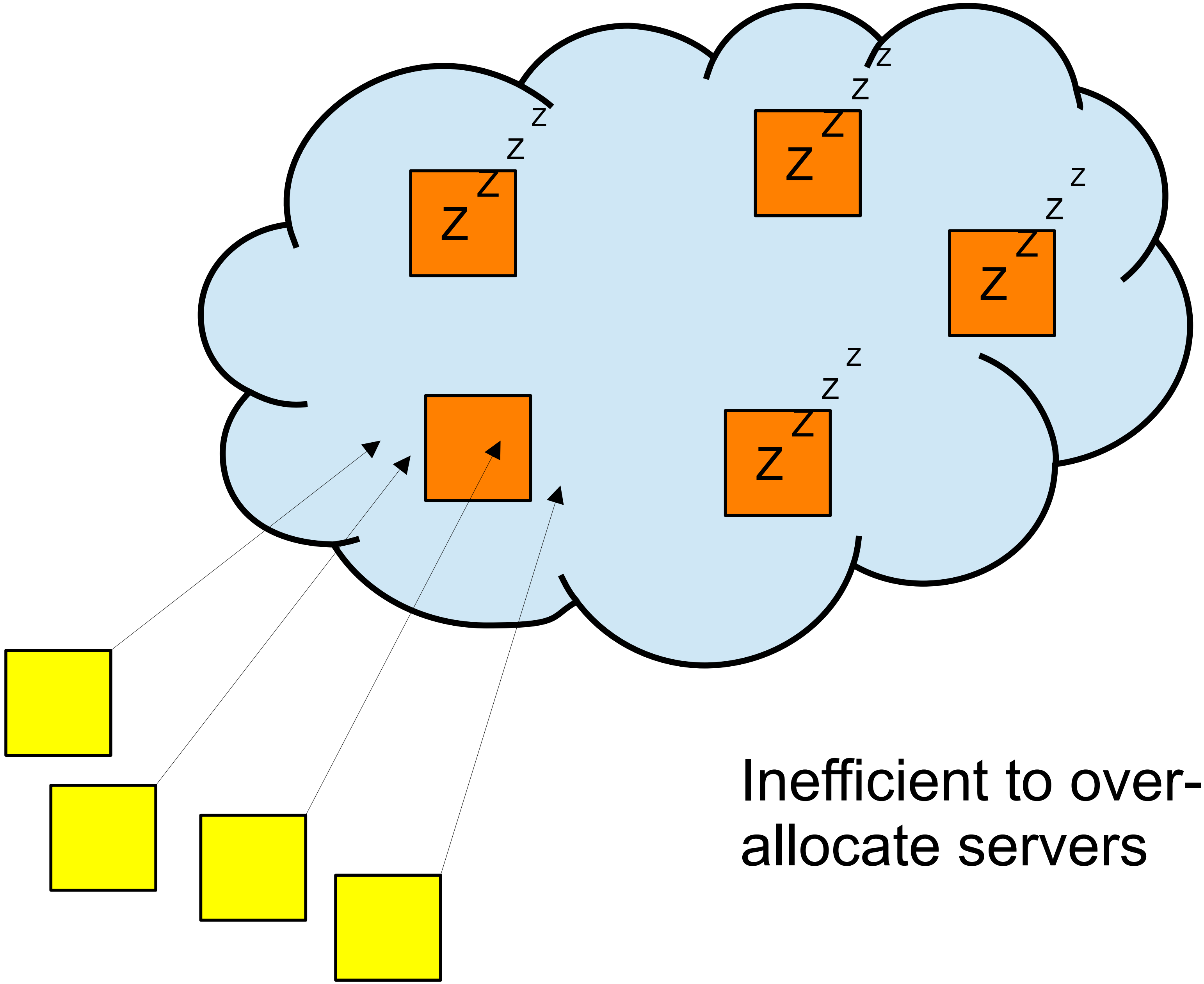
- Many computers
- Scale up/down



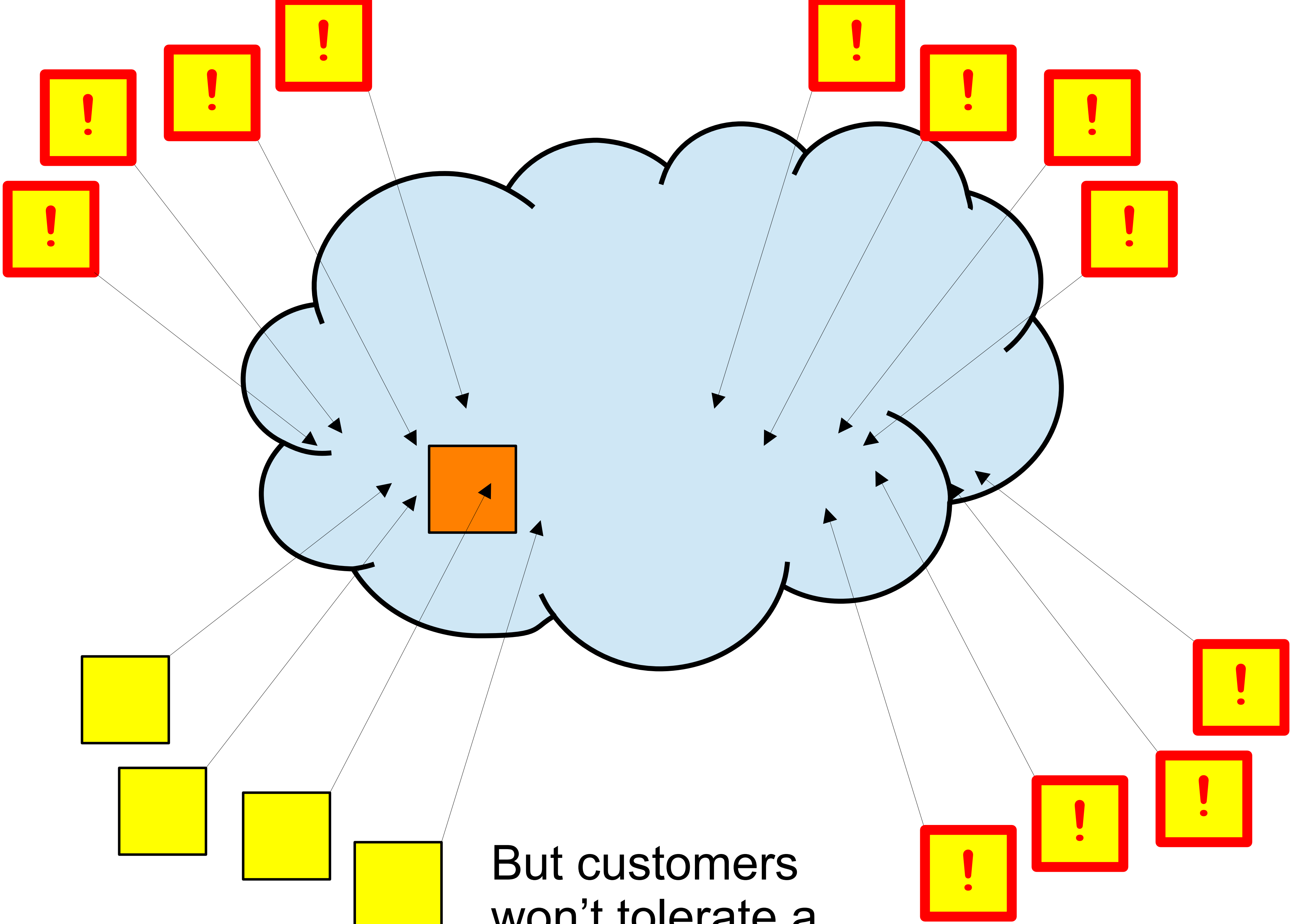
When we have few clients, we don't need many resources.



When demand spikes, we need more.



Inefficient to over-allocate servers



But customers won’t tolerate a slow service

Why Containers

- A common problem in the cloud:
  - How to start up new machines ***quickly*** and ***cheaply?***
  - (And how to take them down without losing anything?)

- Need reliable software
  - Including standard libraries
- Need to be able to build new versions quickly
- Need to be able to test efficiently

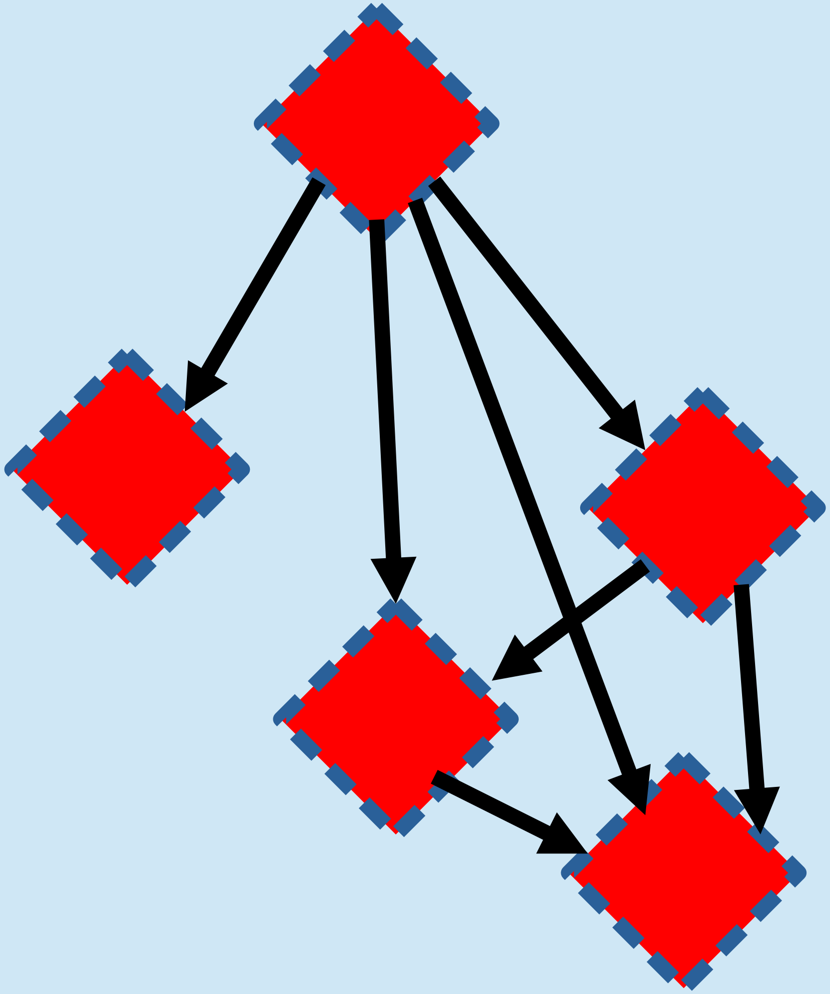
Computer

How do we install software on it,
cheaply and quickly?

How do we upload a configuration,
so it knows what other computers
it should talk to?

Older Solution: Packages

- In the past (and still now) software organized into packages
  - Download a single file, hit “install”
- Had to worry about dependencies
  - Can I update my OS, or my libc, without breaking things?
- Hard to run ancient code, because of ancient dependencies

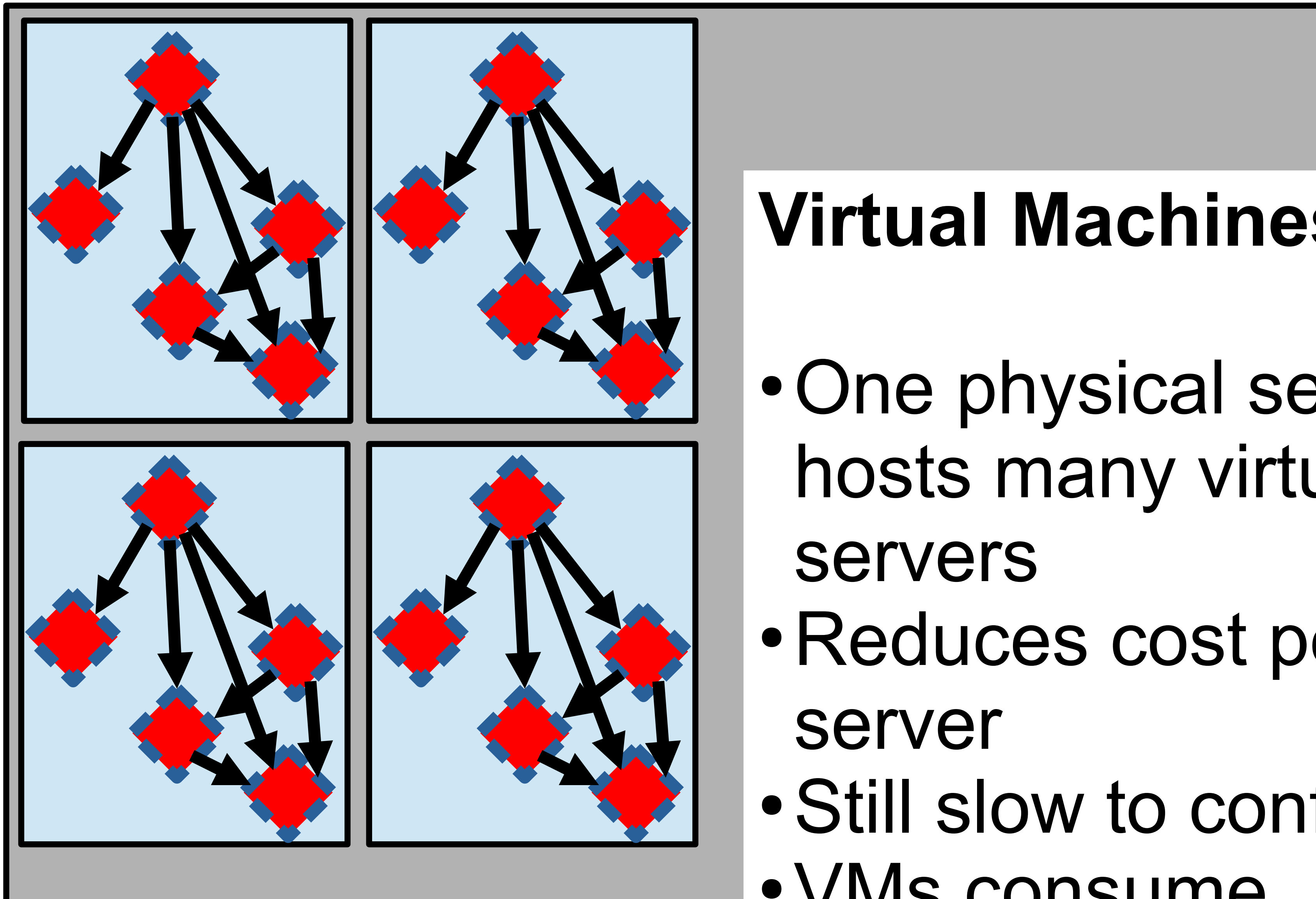


Software Packaging

- Individual programs, installed one at a time
- Track dependencies
- Lots of steps to configure a server

Older Solutions: VMs

- Useful: **virtual machines**
  - One physical machine
  - Several virtual machines internally
  - Each VM looks like a physical machine to the software

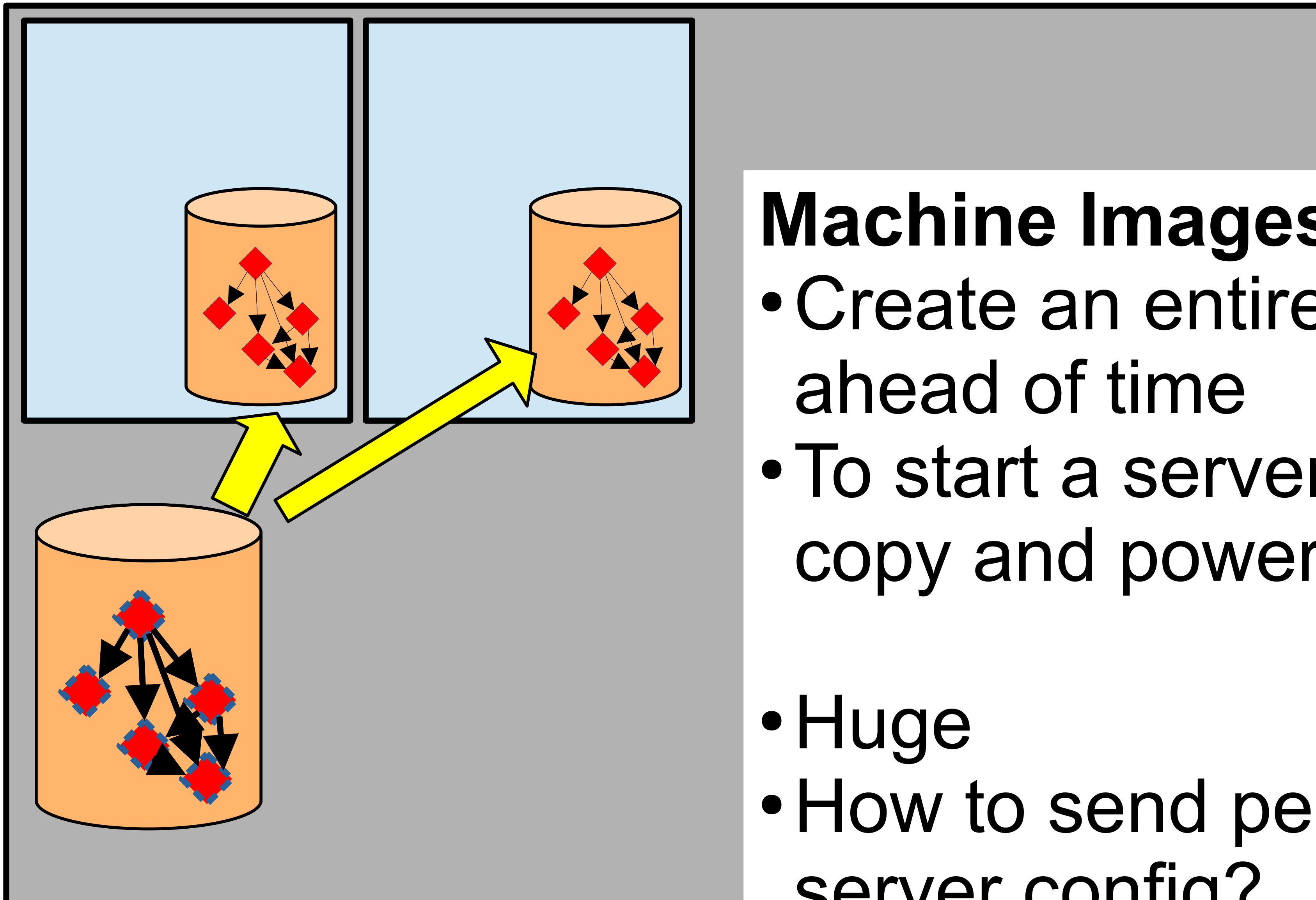


Virtual Machines

- One physical server hosts many virtual servers
- Reduces cost per server
- Still slow to configure
- VMs consume resources when idle

Older Solutions: VM Images

- In the past (rarely now) build complete machine images
  - Full hard drive image, including OS
- Run on VMs
- Still common as the **start** point for virtual computing
  - Add packages (or containers) to build a system



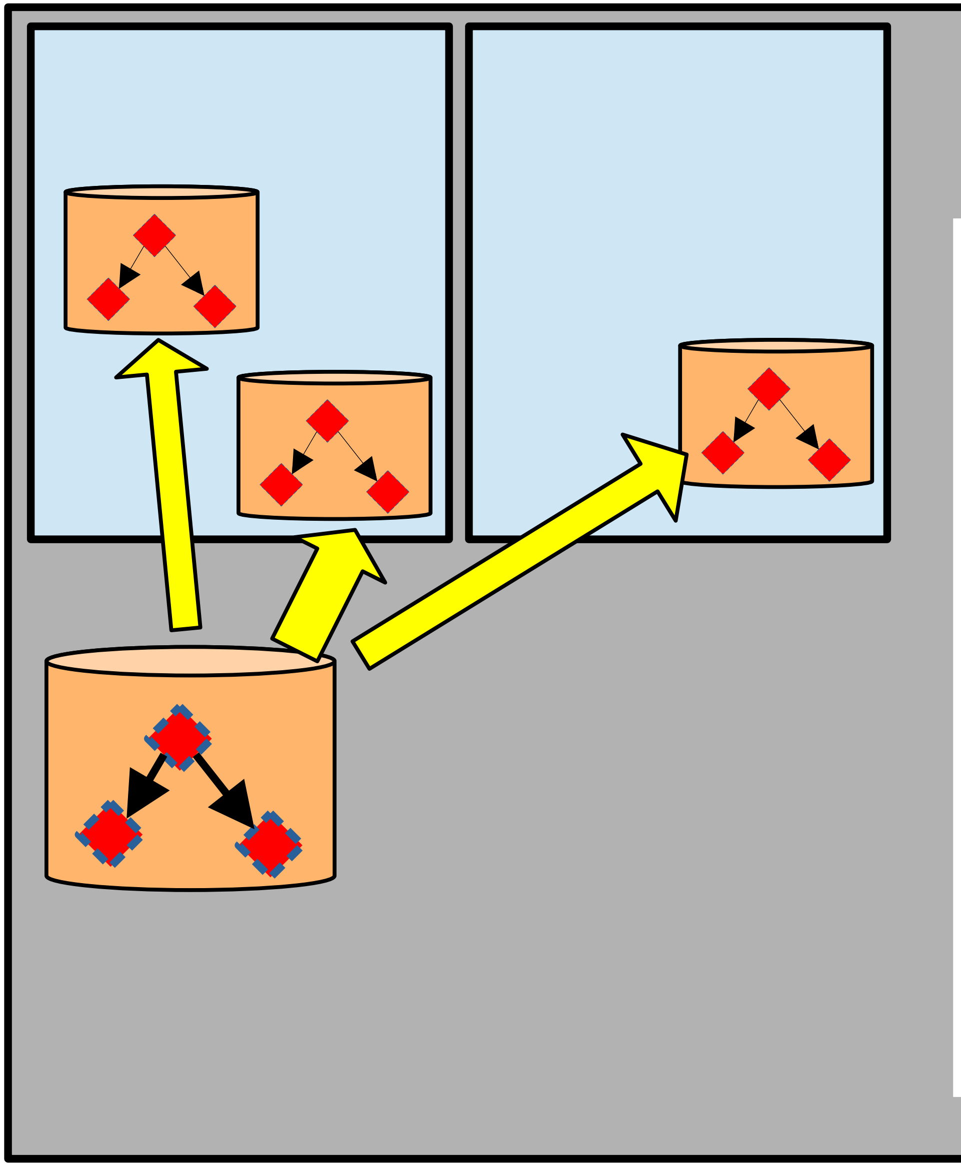
Machine Images

- Create an entire disk ahead of time
- To start a server, just copy and power on

- Huge
- How to send per-server config?

Containers

- Containers try to be the best of both worlds
  - Complete disk image, including dependencies
  - But no OS, no virtual hardware
  - Usually stripped to **bare bones** to make image small
- Download an image, run anywhere
- Lots of features to make images smaller
  - Immutable images + deltas



Containers

- Like a VM image, but stripped to the **bare bones**
- Designed to run only **one application**
- No simulated HW

Containers

- Because we don’t simulate HW, idle containers consume ***very*** little CPU
- Images often composed as “delta” off a standard image
  - Makes it cheap to transfer over the network and/or store locally
- Because OS doesn’t have to boot, we can start a new container ***lightning fast*** (if we already have the image)

Containers

- Summary:
  - A container is like a VM
  - But even cheaper
  - Designed to “create once, run everywhere”
    - All containers based off of **immutable images**

Containers

WARNING!

- Therefore….
  - The data in a given container is ***not saved*** when the container stops
  - All we remember is the ***starting image***

WARNING!