From Chef to Docker

A Saner Management for a Multi-facet Development Environment

Whoami

Yoav Landman

JFrog CTO

What Frog?

What Frog?

What Frog?

artifactory

What Frog?

So…

Bintray

Bintray

Distribution as a service

Remember 2013

Developing for multi-component environments while keeping your sanity

A session at DevNexus 2013

Fred Simon

Monday 18th February, 2013 2:30pm to 3:45pm (EST)

Managing a modern multi-component application in a continuous integration/deployment environment can be very tricky.

In this session we will concentrate on two of the aspects - environment setup, showcasing tools like Vagrant, Chef and Puppet for creating and controlling development, testing, staging, production and other environments and pushing the deliverables through those environments, using tools like Gradle, Jenkins and Artifactory.

Join us and get all the little tips and tricks that will help you rule your development world.

About the speaker

Fred Simon

JFrog and AlphaCSP founder. bio from Twitter

Untrack

Don't meet

Coverage of this session

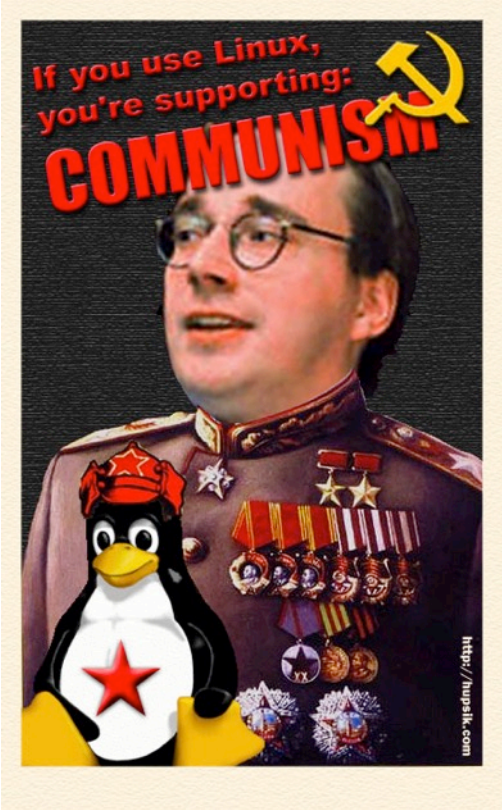
SLIDES

Developing for multi-component environments while keeping your sanity (speakerdeck.com)

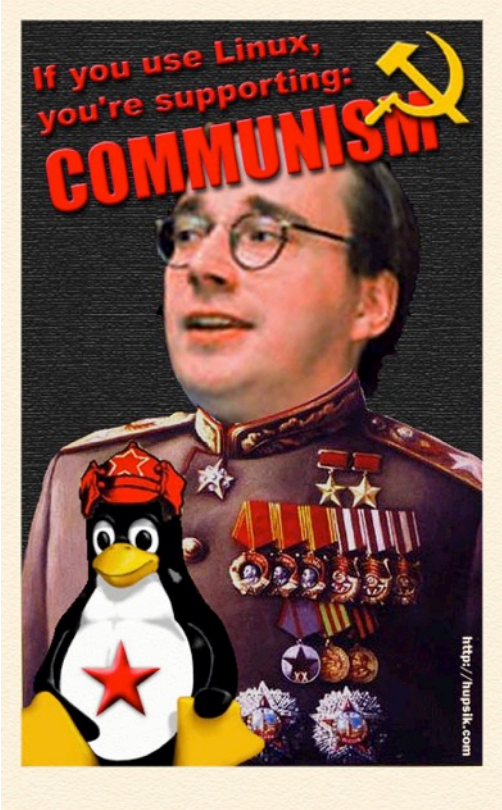
Favourite this? Added 2 years ago edit delete

Diverse technologies

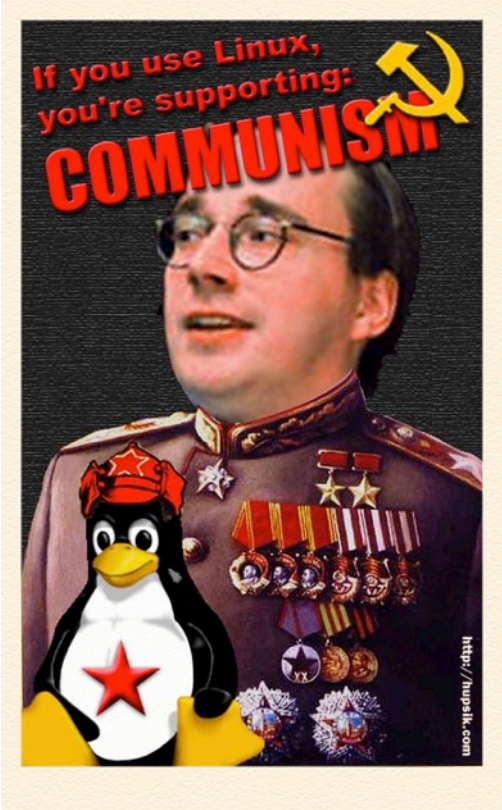
We are liberal



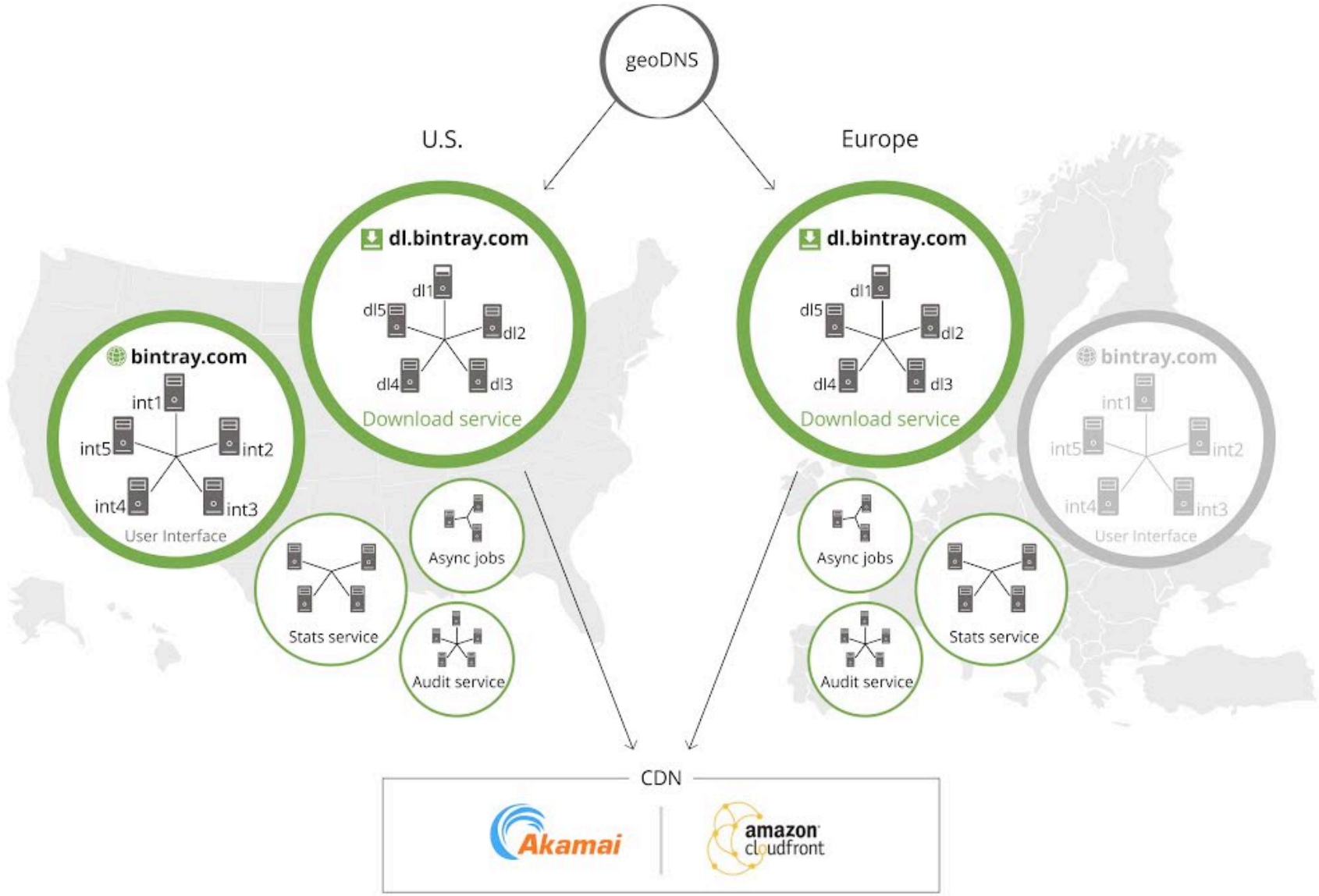
We are liberal



We are liberal




MICROSERVICE
ALL THE THINGS
memegenerator.net


geoDNS
U.S.
Europe
dl.bintray.com
dl1
dl2
dl3
dl4
dl5
Download service
dl.bintray.com
dl1
dl2
dl3
dl4
dl5
Download service
bintray.com
int1
int2
int3
int4
int5
User Interface
bintray.com
int1
int2
int3
int4
int5
User Interface
Async jobs
Stats service
Audit service
Async jobs
Stats service
Audit service
CDN
Akamai
amazon
cloudfront

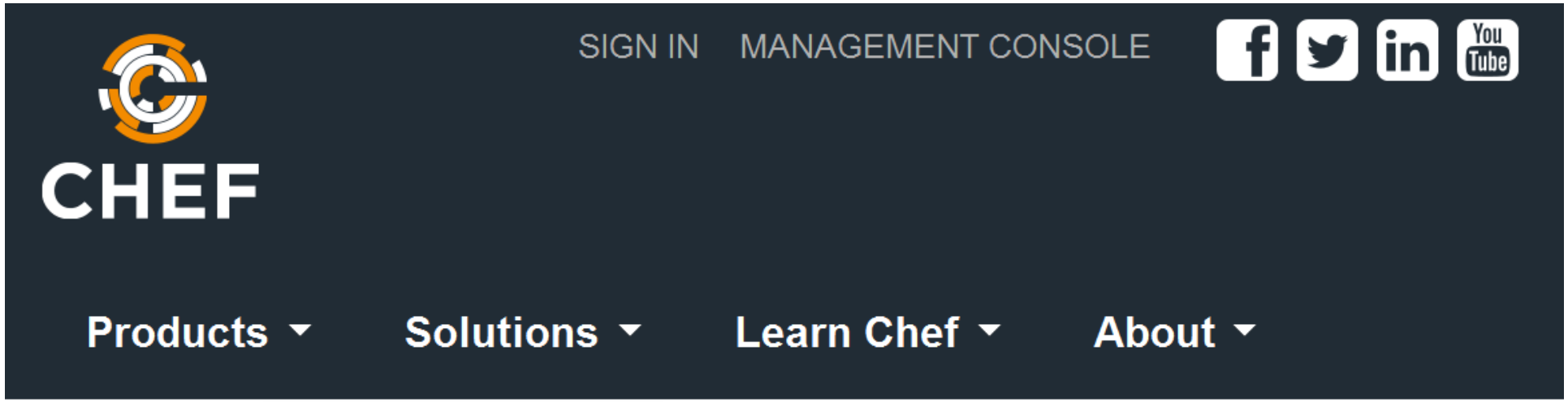

ONE DOES NOT SIMPLY
SETUP ALL THAT ON ALL THIS
quickmeme.com

OK, we have an idea

The Solution

+

Opscode Chef



Chef works with cookbooks

Cookbooks

A cookbook is the fundamental unit of configuration and policy distribution. Each cookbook defines a scenario, such as everything needed to install and configure MySQL, and then it contains all of the components that are required to support that scenario, including:

- Attribute values that are set on nodes
- Definitions that allow the creation of reusable collections of resources
- File distributions

Chef works with cookbooks

Cookbooks

A cookbook is the fundamental unit of configuration and policy distribution. Each cookbook defines a scenario, such as everything needed to install and configure MySQL, and then it contains all of the components that are required to support that scenario, including:

- Attribute values that are set on nodes
- Definitions that allow the creation of reusable collections of resources
- File distributions

Example: Install MySQL in Ruby

```
115
116   if platform_family? 'windows'
117     require 'win32/service'
118
119     windows_path node['mysql']['bin_dir'] do
120       action :add
121     end
122
123     windows_batch "install mysql service" do
124       command "\"#{node['mysql']['bin_dir']}\\mysqld.exe\" --install #{node['mysql']['service_name']}"
125       not_if { Win32::Service.exists?(node['mysql']['service_name']) }
126     end
127   end
128
```

Vagrant


DOCS
VAGRANT
HOW VAGRANT BENEFITS YOU
If you're a developer, Vagrant will isolate dependencies and their configuration within a single disposable, consistent environment, without sacrificing any of the tools you're used to working with (editors, browsers, debuggers, etc.). Once you or someone else creates a single Vagrantfile, you just need to vagrant up and everything is installed

Setup Vagrant with Chef

Setup Vagrant with Chef

```
    config.vm.box = "centos-6"

    # Enable and configure the chef solo provisioner
    config.vm.provision :chef_solo do |chef|
      chef.cookbooks_path = "#{checkout}"
      #chef.data_bags_path = "install/chef/data_bags"
      chef.roles_path = "install/chef/roles"

      chef.add_role("bintray_common_vagrant_server")
      chef.add_role("bintray_interaction_vagrant_server")
      chef.add_role("bintray_download_vagrant_server")
    end
end
```

Setup Vagrant with Chef

```
config.vm.box = "centos-6"

  # Enable and configure the chef solo provisioner
  config.vm.provision :chef_solo do |chef|
    chef.cookbooks_path = "#{checkout}"
    #chef.data_bags_path = "install/chef/data_bags"
    chef.roles_path = "install/chef/roles"

    chef.add_role("bintray_common_vagrant_server")
    chef.add_role("bintray_interaction_vagrant_server")
    chef.add_role("bintray_download_vagrant_server")
  end
end
```

Setup Vagrant with Chef

```
config.vm.box = "centos-6"

# Enable and configure the chef solo provisioner
config.vm.provision :chef_solo do |chef|
  chef.cookbooks_path = "#{checkout}"
  #chef.data_bags_path = "install/chef/data_bags"
  chef.roles_path = "install/chef/roles"

  chef.add_role("bintray_common_vagrant_server")
  chef.add_role("bintray_interaction_vagrant_server")
  chef.add_role("bintray_download_vagrant_server")
end
end
```

Setup Vagrant with Chef

```
config.vm.box = "centos-6"

# Enable and configure the chef solo provisioner
config.vm.provision :chef_solo do |chef|
  chef.cookbooks_path = "#{checkout}"
  #chef.data_bags_path = "install/chef/data_bags"
  chef.roles_path = "install/chef/roles"

  chef.add_role("bintray_common_vagrant_server")
  chef.add_role("bintray_interaction_vagrant_server")
  chef.add_role("bintray_download_vagrant_server")
end
end
```

Setup Vagrant with Chef

```
config.vm.box = "centos-6"

# Enable and configure the chef solo provisioner
config.vm.provision :chef_solo do |chef|
  chef.cookbooks_path = "#{checkout}"
  #chef.data_bags_path = "install/chef/data_bags"
  chef.roles_path = "install/chef/roles"

  chef.add_role("bintray_common_vagrant_server")
  chef.add_role("bintray_interaction_vagrant_server")
  chef.add_role("bintray_download_vagrant_server")
end
end
```

How it works

How it works

1. Check out cookbook and vagrant file

How it works

1. Check out cookbook and vagrant file
2. Vagrant downloads box

How it works

1. Check out cookbook and vagrant file
2. Vagrant downloads box
3. Vagrant starts the box

How it works

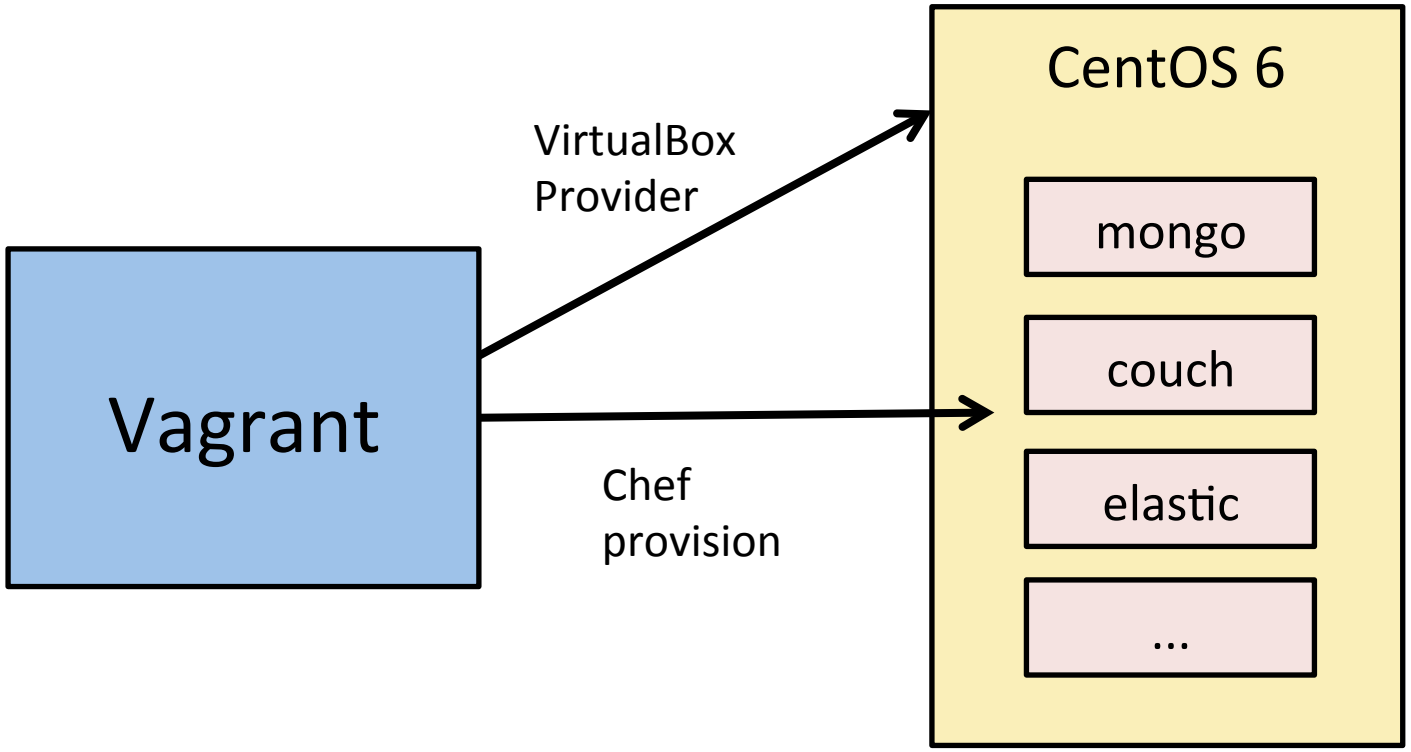
1. Check out cookbook and vagrant file
2. Vagrant downloads box
3. Vagrant starts the box
4. Vagrant triggers Chef

How it works

1. Check out cookbook and vagrant file
2. Vagrant downloads box
3. Vagrant starts the box
4. Vagrant triggers Chef
5. Chef plays recipes to set up everything

How did we like it?

Solves the problem!

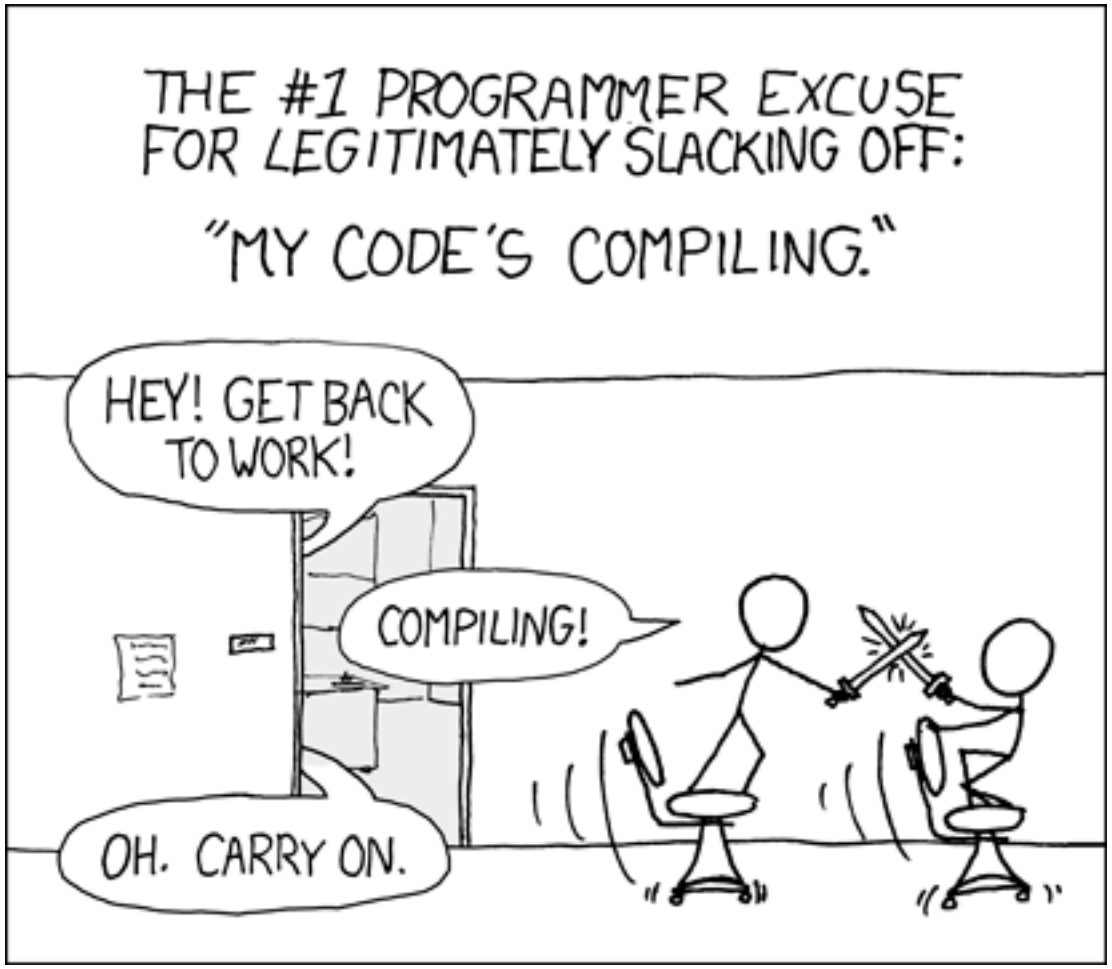

Vagrant
VirtualBox
Provider
Chef
provision
CentOS 6
mongo
couch
elastic
...

But…

Developers break things


ONE DOES NOT SIMPLY
REPLAY ALL CHEF RECIPES

I am not slacking off, my environment is provisioned



Solution I:

Pre-build instead of build

Not really new idea

How it works

How it works

1. Vagrant downloads box

How it works

1. Vagrant downloads box
2. Vagrant starts the box

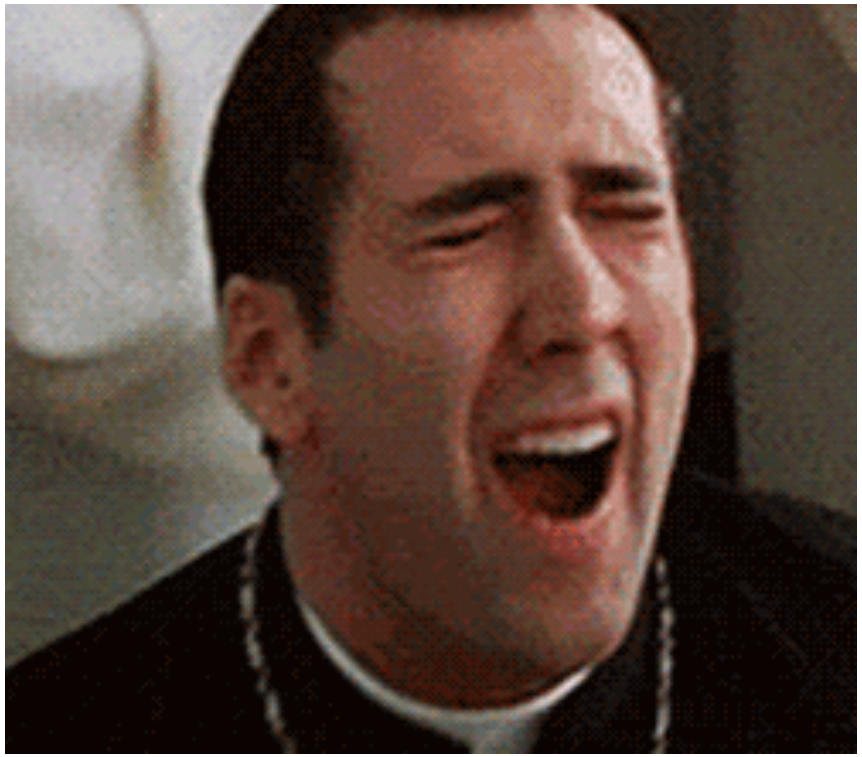
How it works

1. Vagrant downloads box
2. Vagrant starts the box
3. Everything is up and running!

How did we like it?

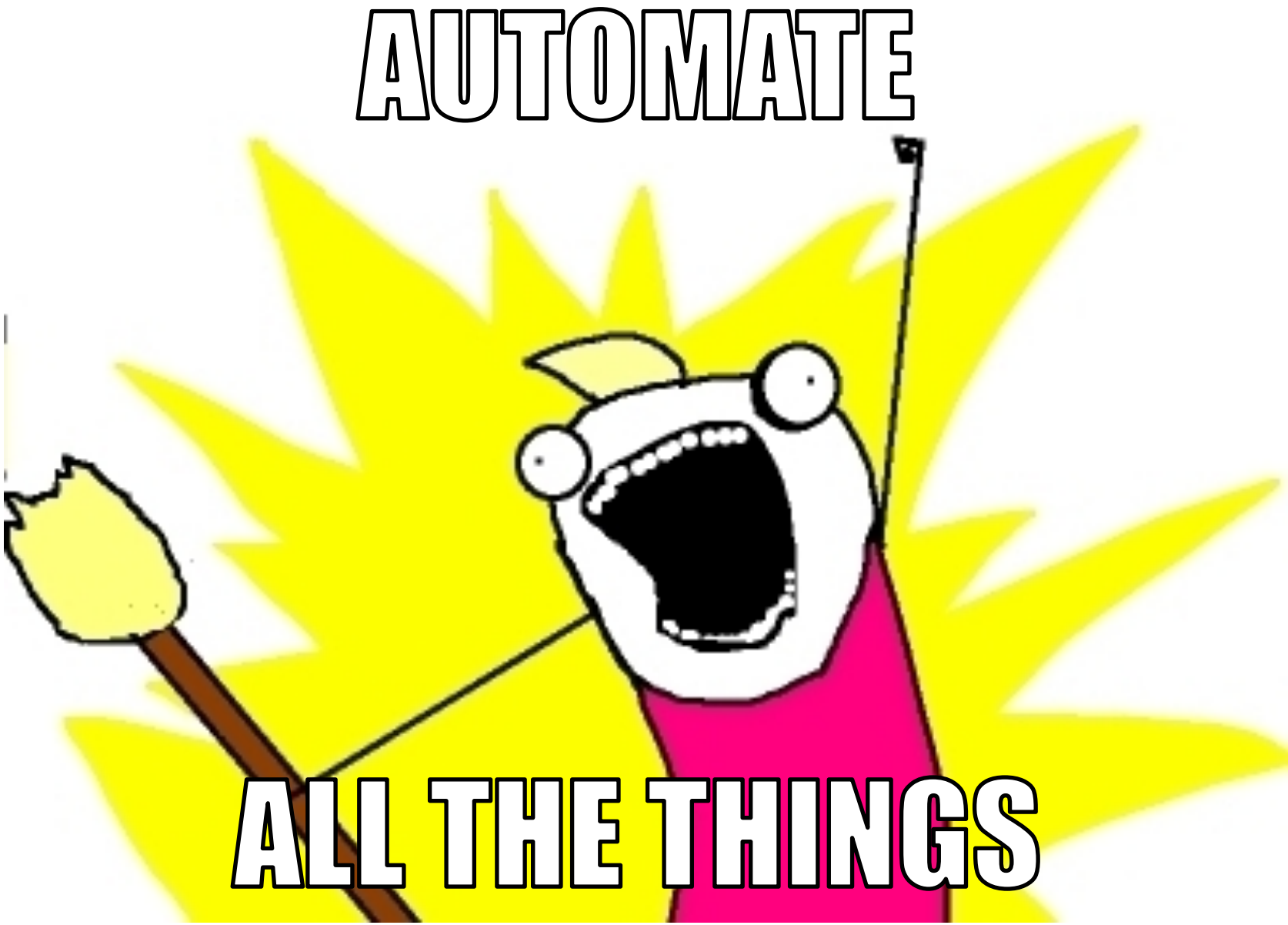
Solves the problem!

But every time something changing in the environment…




ONE DOES NOT SIMPLY
REBUILD VM ON EVERY CHANGE

OK, we have an idea


AUTOMATE
ALL THE THINGS

Automating vm creation

Automating vm creation

Chef?

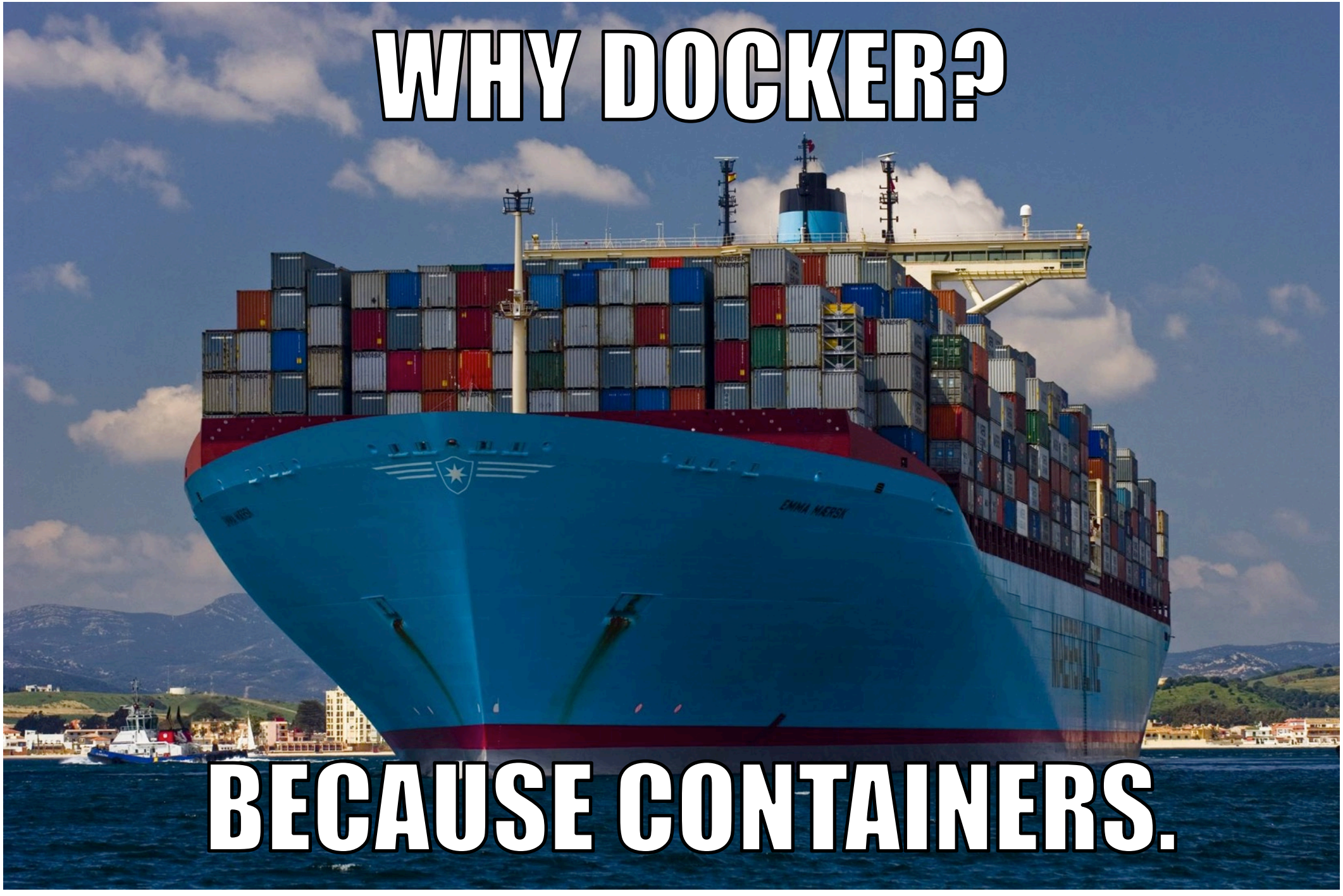
Automating vm creation

Packer?

Automating vm creation

docker?


DOCKER
DOCKER EVERYWHERE


WHY DOCKER?
EMMA MAERSK
BECAUSE CONTAINERS.

Operating-system-level virtualization

From Wikipedia, the free encyclopedia
(Redirected from Software container)

Operating-system-level virtualization is a server virtualization method where the kernel of an operating system allows for multiple isolated user space instances, instead of just one. Such instances (often called **containers**, virtualization engines (VE), virtual private servers (VPS), or jails) may look and feel like a real server from the point of view of its owners and users.

On Unix-like operating systems, this technology can be seen as an advanced implementation of the standard chroot mechanism. In addition to isolation mechanisms, the kernel often provides resource management features to limit the impact of one container's activities on the other containers.

Contents [show]

Uses [edit]

Operating-system-level virtualization is commonly used in virtual hosting environments, where it is useful for securely allocating finite hardware resources amongst a large number of mutually-distrusting users. System administrators may also use it, to a lesser extent, for consolidating server hardware by moving services on separate hosts into containers on the one server.

Other typical scenarios include separating several applications to separate containers for improved security, hardware independence, and added resource management features. The improved security provided by the use of a chroot mechanism, however, is nowhere near ironclad.[1] Operating-system-level virtualization implementations capable of live migration can also be used for dynamic load balancing of containers between nodes in a cluster.

Overhead [edit]

Operating-system-level virtualization usually imposes little to no overhead, because programs in virtual partitions use the operating system's normal system call interface and do not need to be subjected to emulation or be run in an intermediate virtual machine, as is the case with whole-system virtualizers (such as VMware ESXi, QEMU or Hyper-V) and paravirtualizers (such as Xen or UML). This form of virtualization also does not require support in hardware to perform efficiently.

Operating-system-level virtualization

From Wikipedia, the free encyclopedia
(Redirected from Software container)

Operating-system-level virtualization is a server virtualization method where the kernel of an operating system allows for multiple isolated user space instances, instead of just one. Such instances (often called **containers**, virtualization engines (VE), virtual private servers (VPS), or jails) may look and feel like a real server from the point of view of its owners and users.

On Unix-like operating systems, this technology can be seen as an advanced implementation of the standard chroot mechanism. In addition to isolation mechanisms, the kernel often provides resource management features to limit the impact of one container's activities on the other containers.

Contents [show]

Uses [edit]

Operating-system-level virtualization is commonly used in virtual hosting environments, where it is useful for securely allocating finite hardware resources amongst a large number of mutually-distrusting users. System administrators may also use it, to a lesser extent, for consolidating server hardware by moving services on separate hosts into containers on the one server.

Other typical scenarios include separating several applications to separate containers for improved security, hardware independence, and added resource management features. The improved security provided by the use of a chroot mechanism, however, is nowhere near ironclad.[1] Operating-system-level virtualization implementations capable of live migration can also be used for dynamic load balancing of containers between nodes in a cluster.

Overhead [edit]

Operating-system-level virtualization usually imposes little to no overhead, because programs in virtual partitions use the operating system's normal system call interface and do not need to be subjected to emulation or be run in an intermediate virtual machine, as is the case with whole-system virtualizers (such as VMware ESXi, QEMU or Hyper-V) and paravirtualizers (such as Xen or UML). This form of virtualization also does not require support in hardware to perform efficiently.

Operating-system-level virtualization

From Wikipedia, the free encyclopedia
(Redirected from Software container)

Operating-system-level virtualization is a server virtualization method where the kernel of an operating system allows for multiple isolated user space instances, instead of just one. Such instances (often called **containers**, virtualization engines (VE), virtual private servers (VPS), or jails) may look and feel like a real server from the point of view of its owners and users.

On Unix-like operating systems, this technology can be seen as an advanced implementation of the standard chroot mechanism. In addition to isolation mechanisms, the kernel often provides resource management features to limit the impact of one container's activities on the other containers.

Contents [show]

Uses [edit]

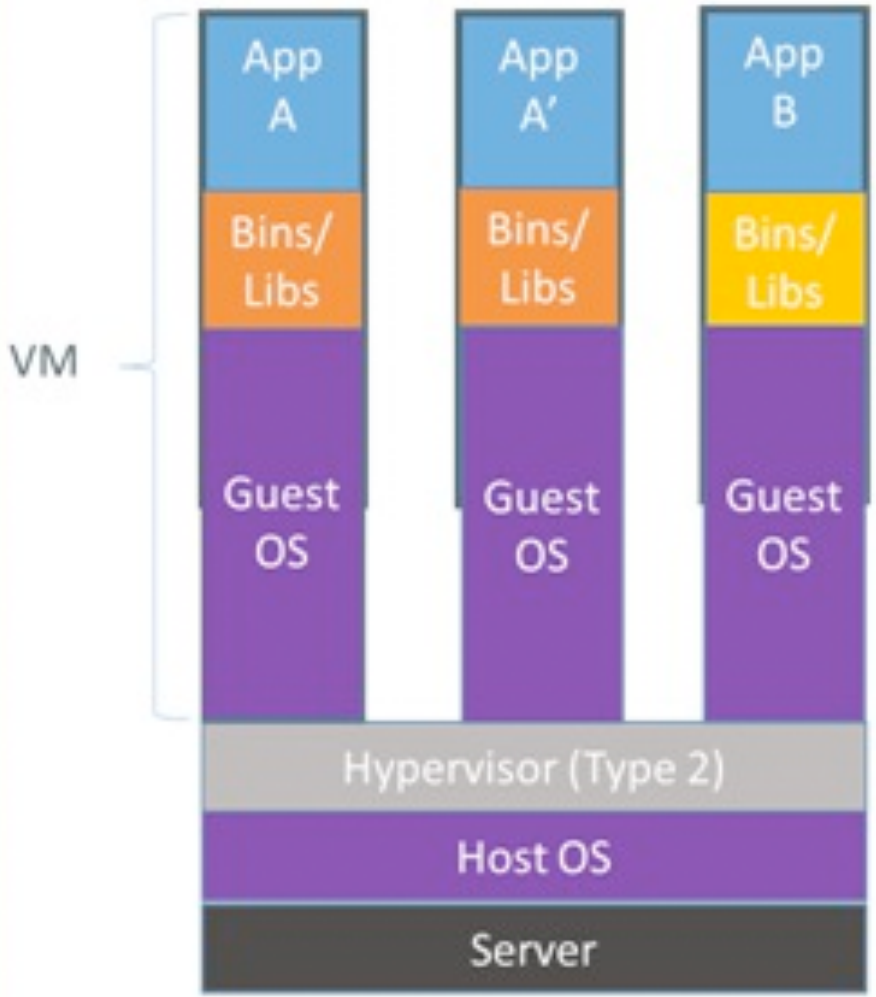
Operating-system-level virtualization is commonly used in virtual hosting environments, where it is useful for securely allocating finite hardware resources amongst a large number of mutually-distrusting users. System administrators may also use it, to a lesser extent, for consolidating server hardware by moving services on separate hosts into containers on the one server.

Other typical scenarios include separating several applications to separate containers for improved security, hardware independence, and added resource management features. The improved security provided by the use of a chroot mechanism, however, is nowhere near ironclad.[1] Operating-system-level virtualization implementations capable of live migration can also be used for dynamic load balancing of containers between nodes in a cluster.

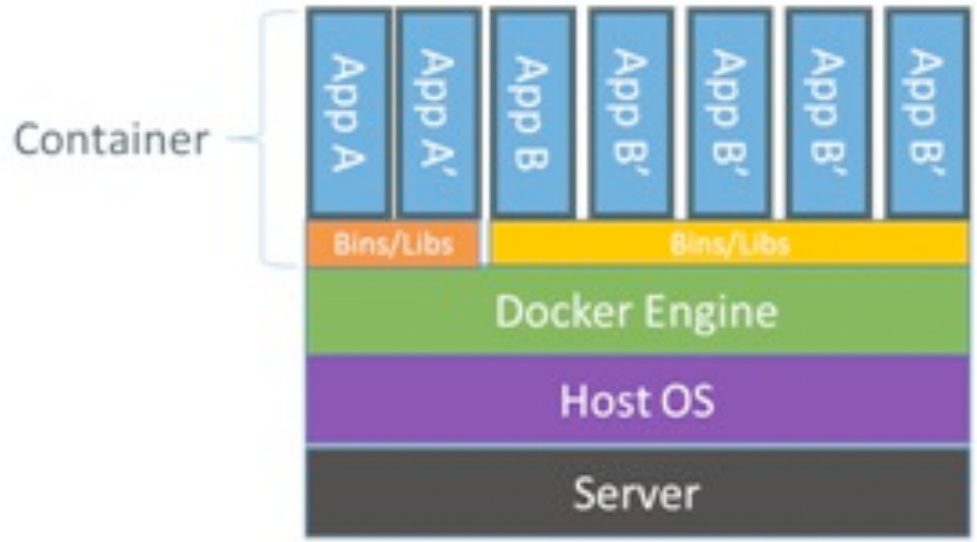
Overhead [edit]

Operating-system-level virtualization usually imposes little to no overhead, because programs in virtual partitions use the operating system's normal system call interface and do not need to be subjected to emulation or be run in an intermediate virtual machine, as is the case with whole-system virtualizers (such as VMware ESXi, QEMU or Hyper-V) and paravirtualizers (such as Xen or UML). This form of virtualization also does not require support in hardware to perform efficiently.

Containers vs. VMs



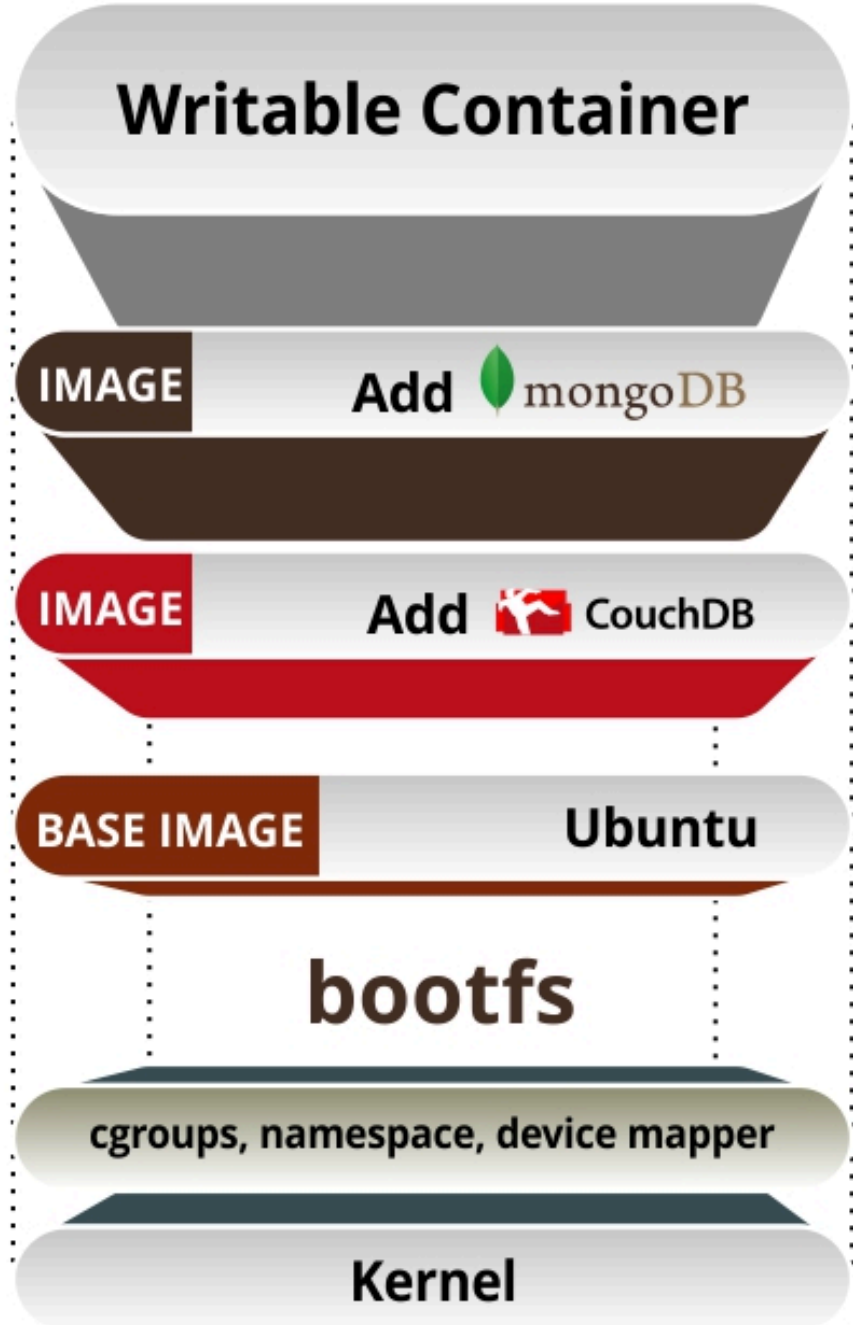
Containers are isolated, but share OS and, where appropriate, bins/libraries



http://www.programering.com/a/MDMzAjMwATk.html

Layered approach

- Along other benefits, efficient:
- Docker pull brings deltas only


Writable Container
IMAGE
Add mongoDB
IMAGE
Add CouchDB
BASE IMAGE
Ubuntu
bootfs
cgroups, namespace, device mapper
Kernel

Docker 💙 microservices

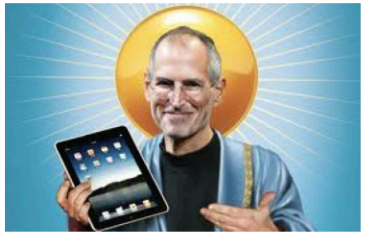
Docker fits much the micro-service arch of bintray

(+ we do not need all services always)

Docker is not enough

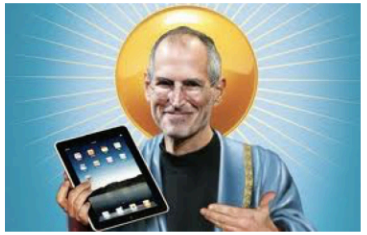
Docker is not enough

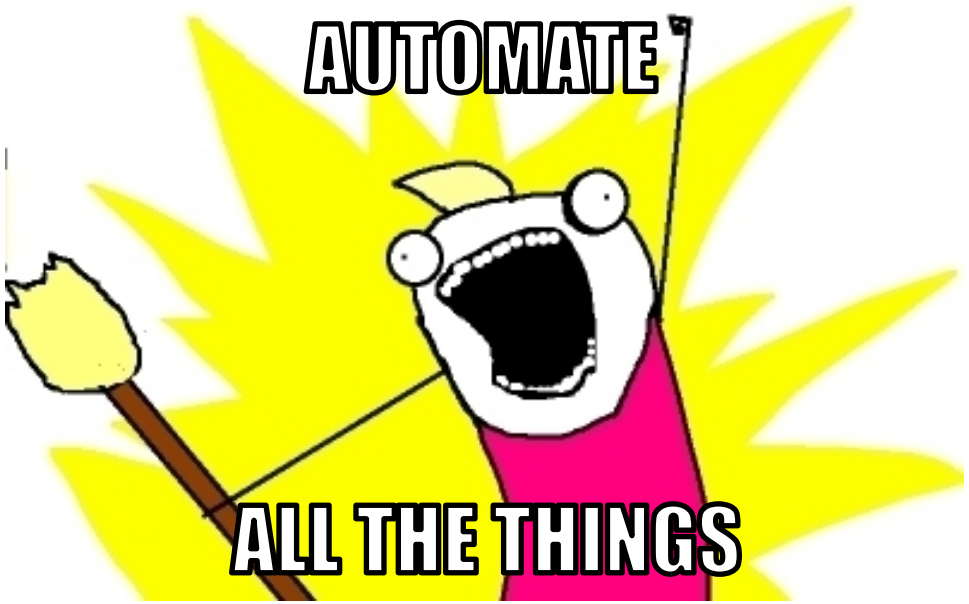
We are liberal



Docker is not enough

We are liberal

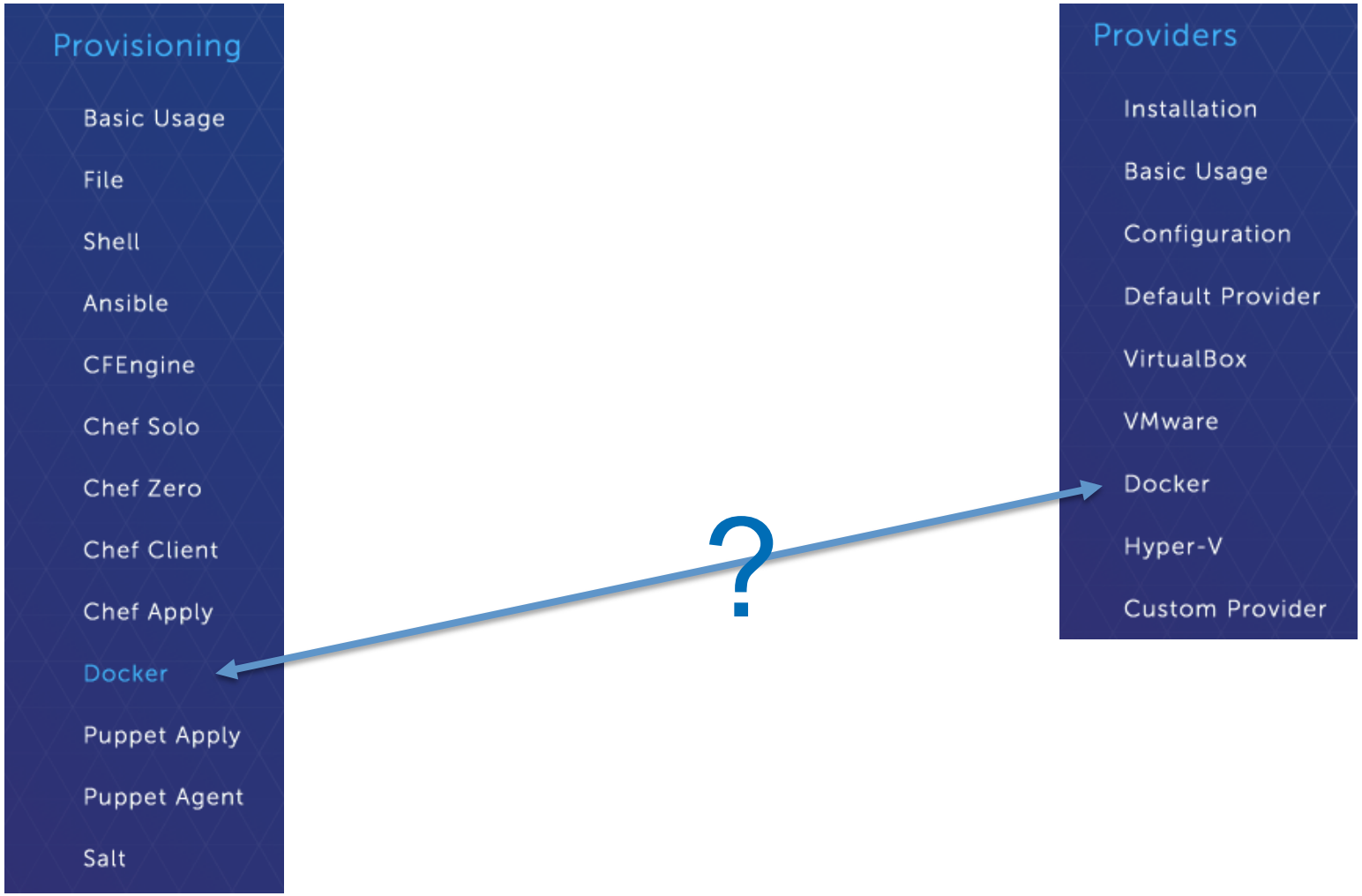




Why Vagrant and Docker

- Configure host vm to run containers
  - Plugable Docker hosts
- Install Docker and run Docker
- Vagrant can control Docker containers – run, destroy, link

Docker support in Vagrant



Provider? Provisioner?

Provider vs Provisioner

- Provider: defines the type of machines to manage, affects vm creation
- Provisioner: configures an existing machine on first vagrant up
- Vagrant destroy - undo machine creation

Solution II:

Vagrant + Docker
All the services in one container

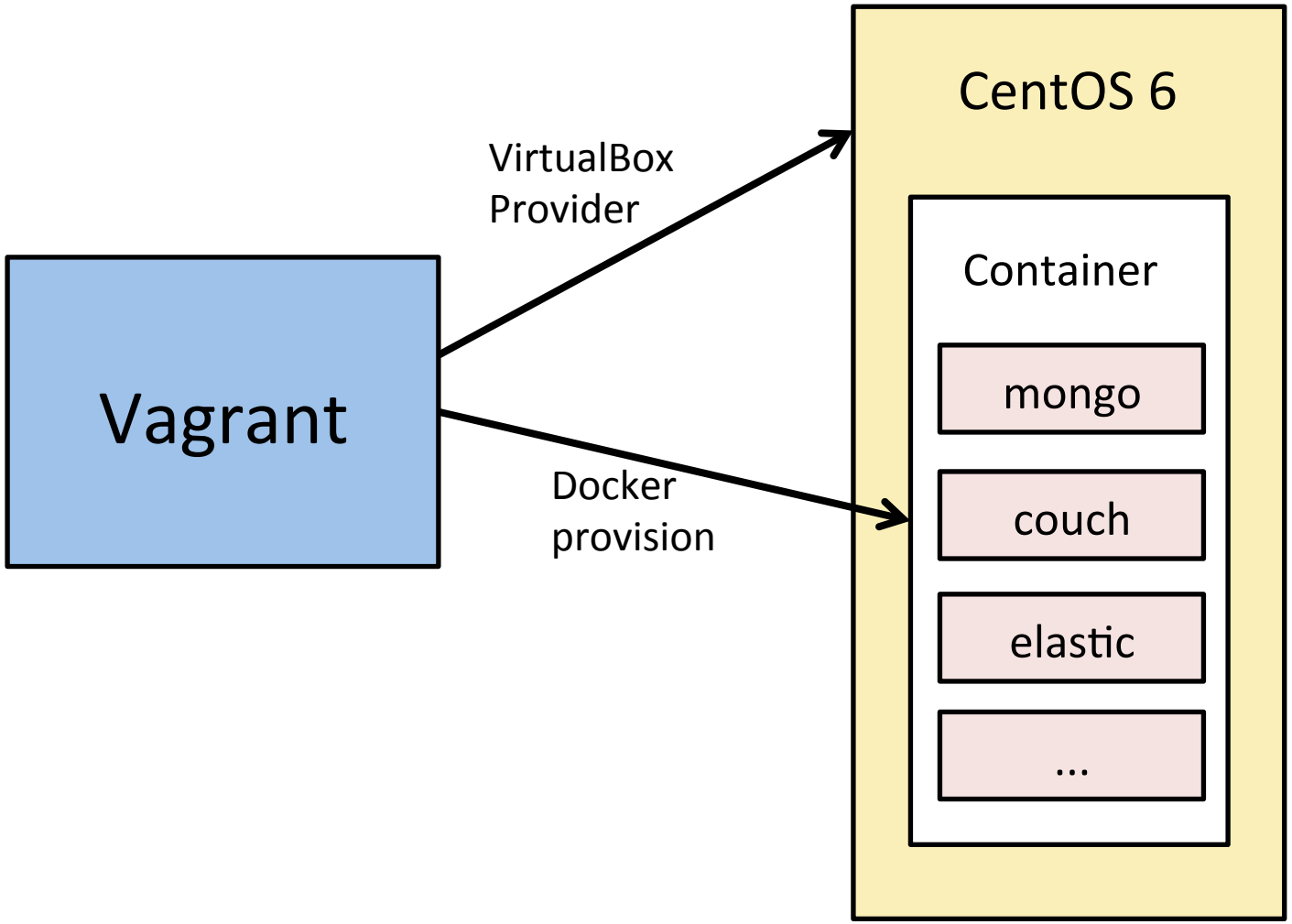
DOCKER PROVISIONER

Provisioner name: `"docker"`

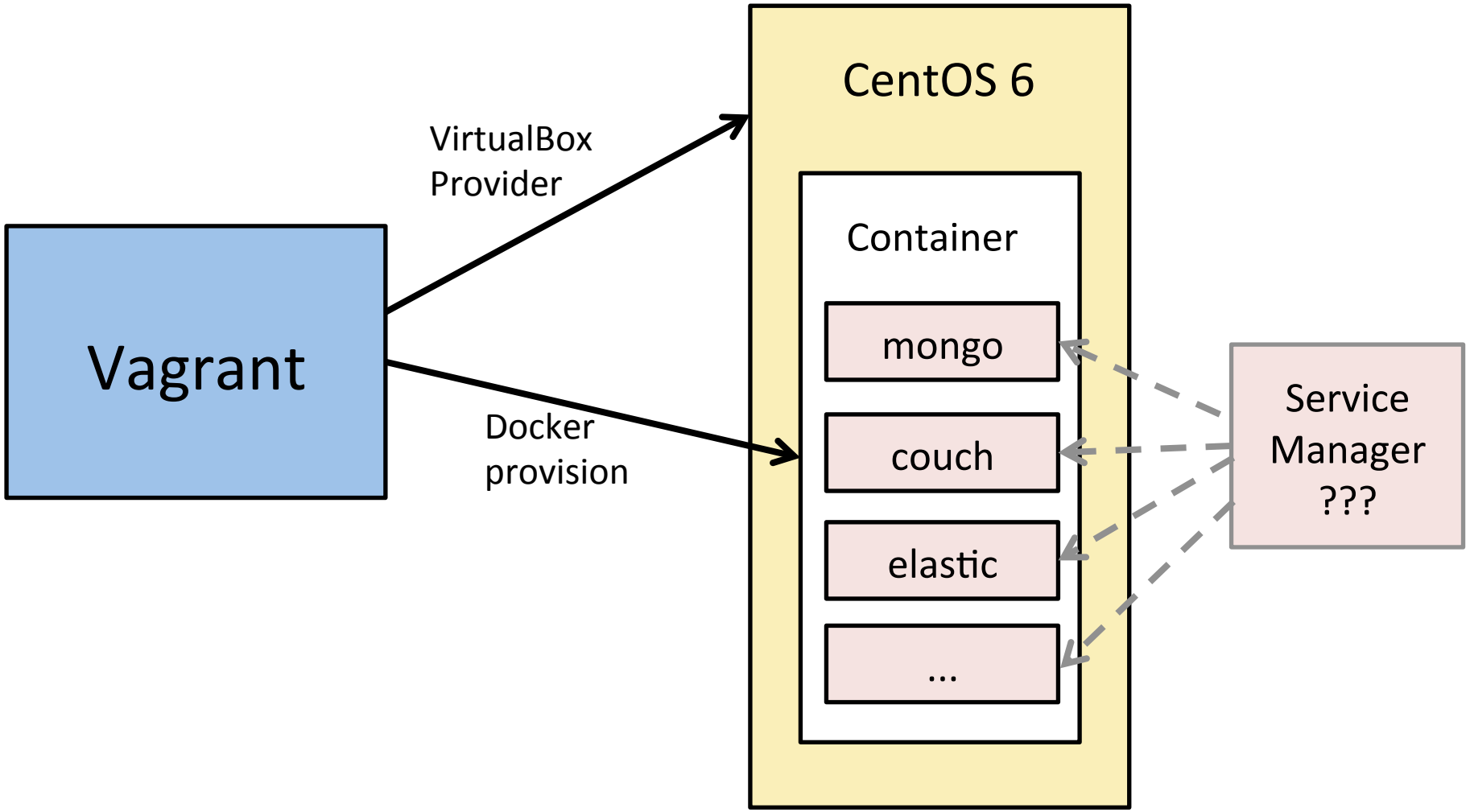
The docker provisioner can automatically install Docker, pull Docker containers, and configure certain containers to run on boot.

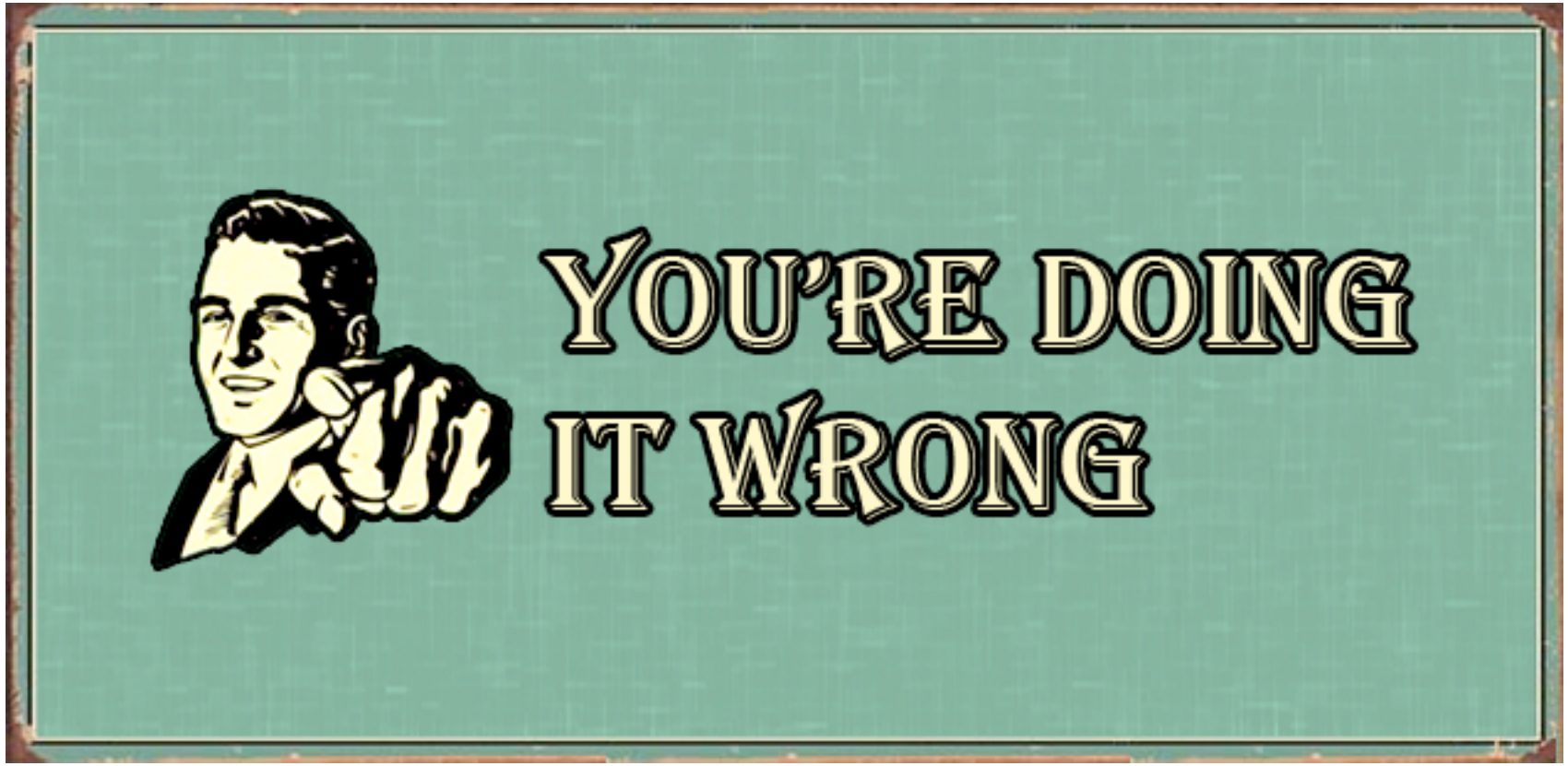
The docker provisioner is ideal for organizations that are using Docker as a means to distribute things like their application or services. Or, if you're just getting started with Docke

the Docker provisioner provides the easiest possible way to begin using Docker since the


Vagrant
VirtualBox
Provider
Docker
provision
CentOS 6
Container
mongo
couch
elastic
...

```
Vagrant.configure("2") do |config|
  config.vm.box = "centos6”

  config.vm.provider :virtualbox do |vb|
     vb.gui = false
  end
  ...

  config.vm.provision "docker" do |d|
    d.pull_images "jfrog/bintray"
    d.run "jfrog/bintray"
  end
end
```


Vagrant
VirtualBox
Provider
Docker
provision
CentOS 6
Container
mongo
couch
elastic
...
Service
Manager
???

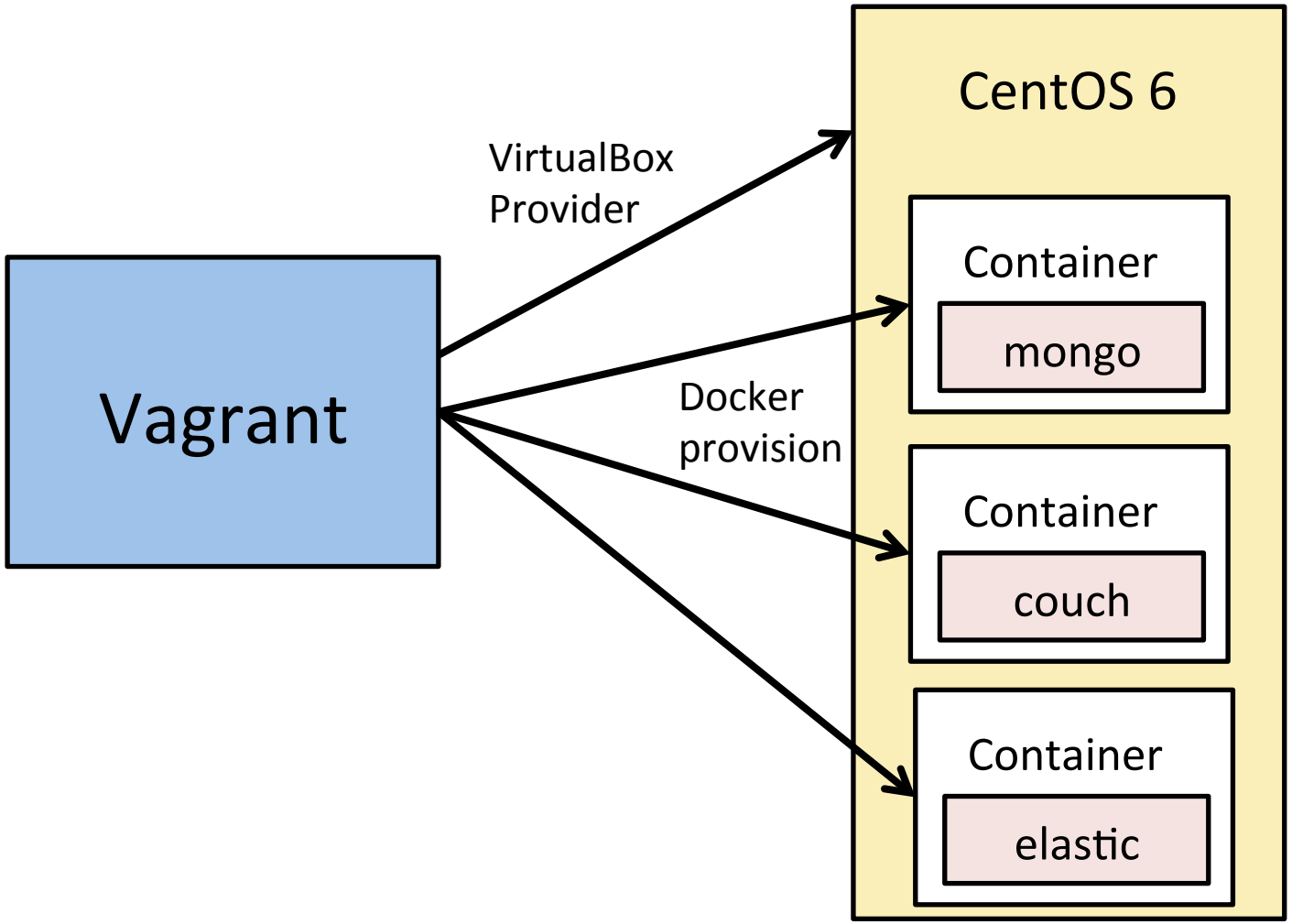


- One single container or multiple containers - with one you also need to add a process manager; for instance Monit or Supervisor.

OK, we have an idea

Solution III:

Vagrant + Docker
Separate containers with Vagrant provisioners


Vagrant
VirtualBox
Provider
Docker
provision
CentOS 6
Container
mongo
Container
couch
Container
elastic

```
Vagrant.configure("2") do |config|
  config.vm.box = "centos6”

  config.vm.provider :virtualbox do |vb|
     vb.gui = false
  end
  ...

  config.vm.provision "docker" do |d|
    d.pull_images "jfrog/mongo"
    d.run "jfrog/mongo”

    d.pull_images "jfrog/couch"
    d.run "jfrog/couch”
  end
end
```

But…

Developers break things

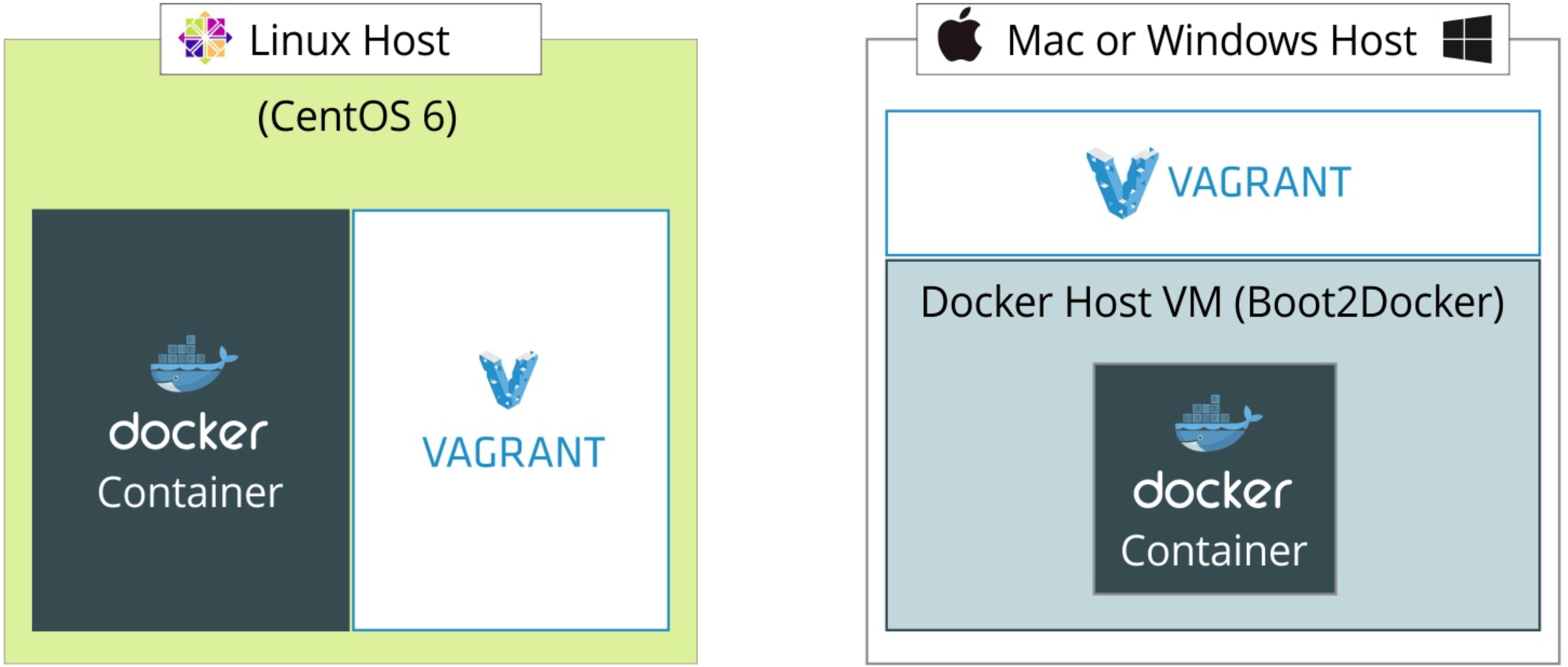
vagrant destroy

- Remember, vagrant destroy undoes the box completely
- I.E. empty box, no images, no configuration
- Next vagrant up will download all the images from scratch (specially painful over VPN)

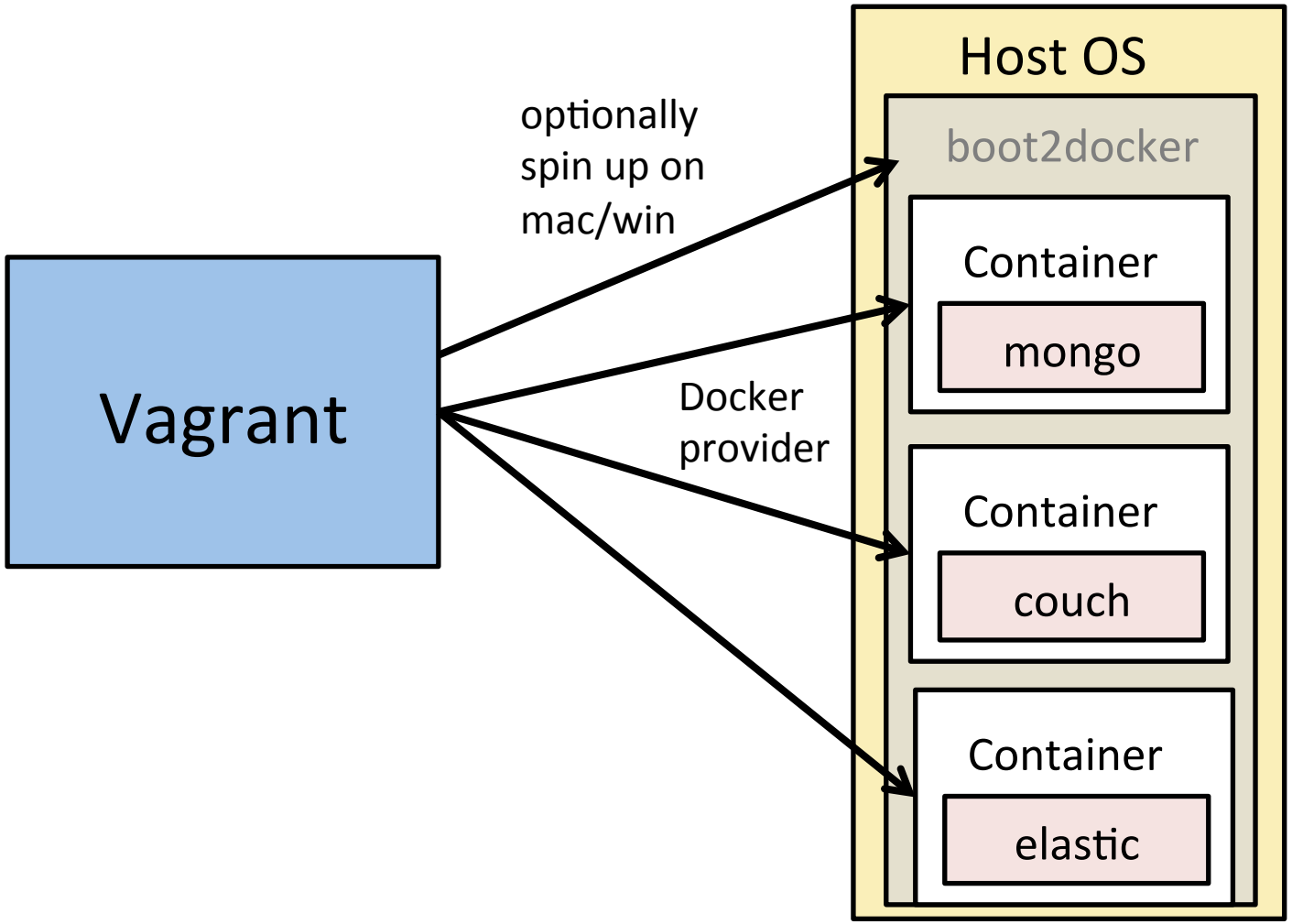
Solution IV:

Vagrant + Docker
Separate containers with Vagrant providers
& use vagrant provider-management

Vagrant is smart!

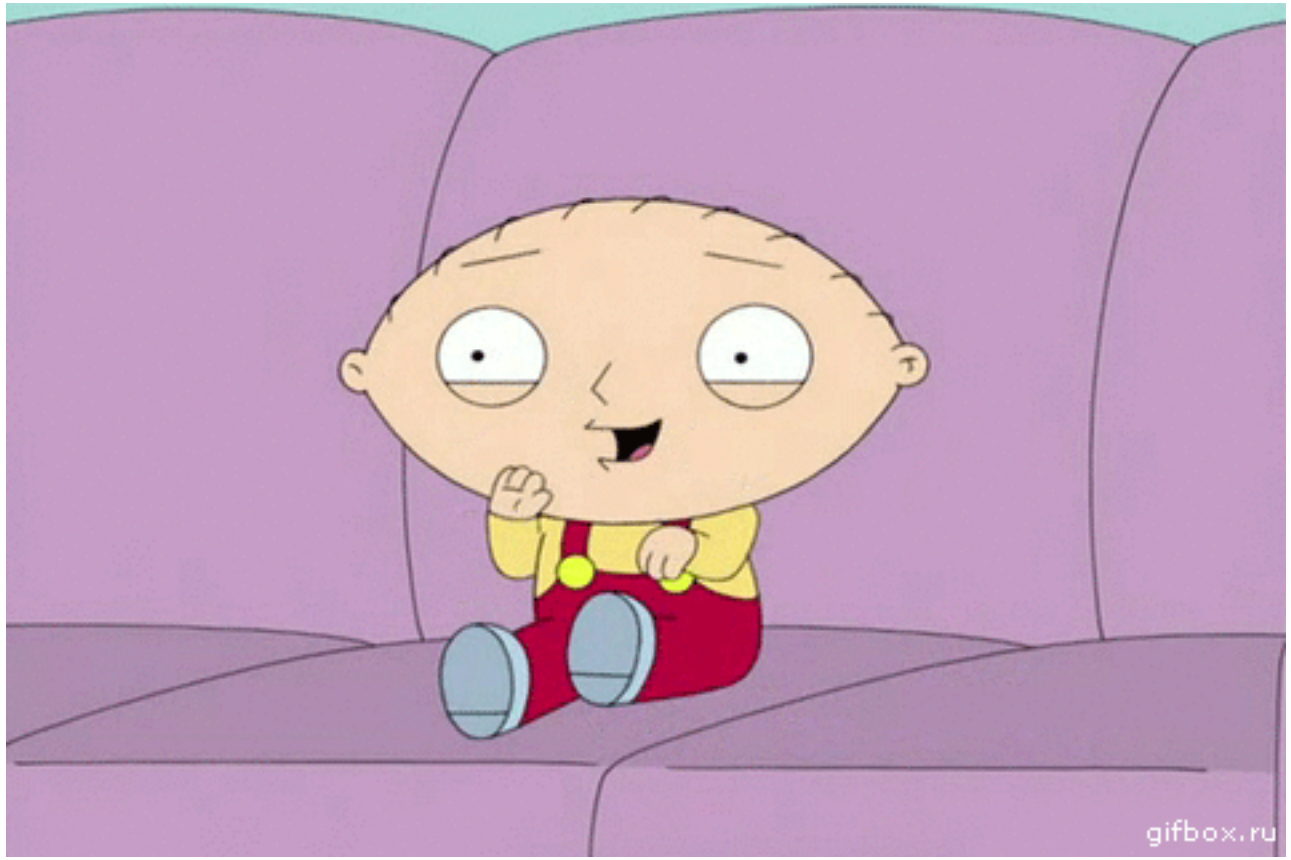


Tiny Core Linux
(a.k.a. boot2docker)


Host OS
boot2docker
optionally
spin up on
mac/win
Vagrant
Docker
provider
Container
mongo
Container
couch
Container
elastic

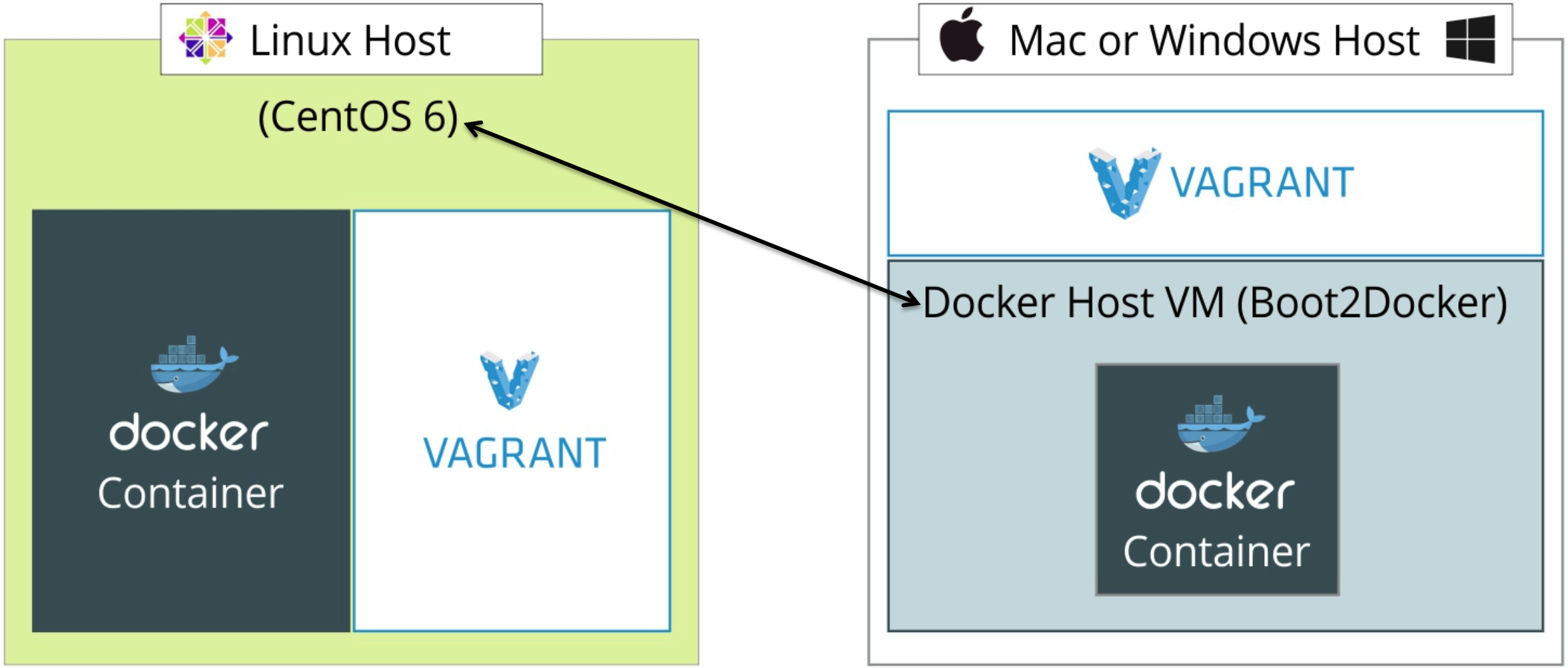
```
Vagrant.configure("2") do |config|
  //Look ma, no box!

  config.vm.define "mongodb" do |container|
    container.vm.provider "docker" do |d|
      ...
    end
  end

  config.vm.define "couch" do |container|
    container.vm.provider "docker" do |d|
      ...
    end
  end
end
```

How did we like it?



But…

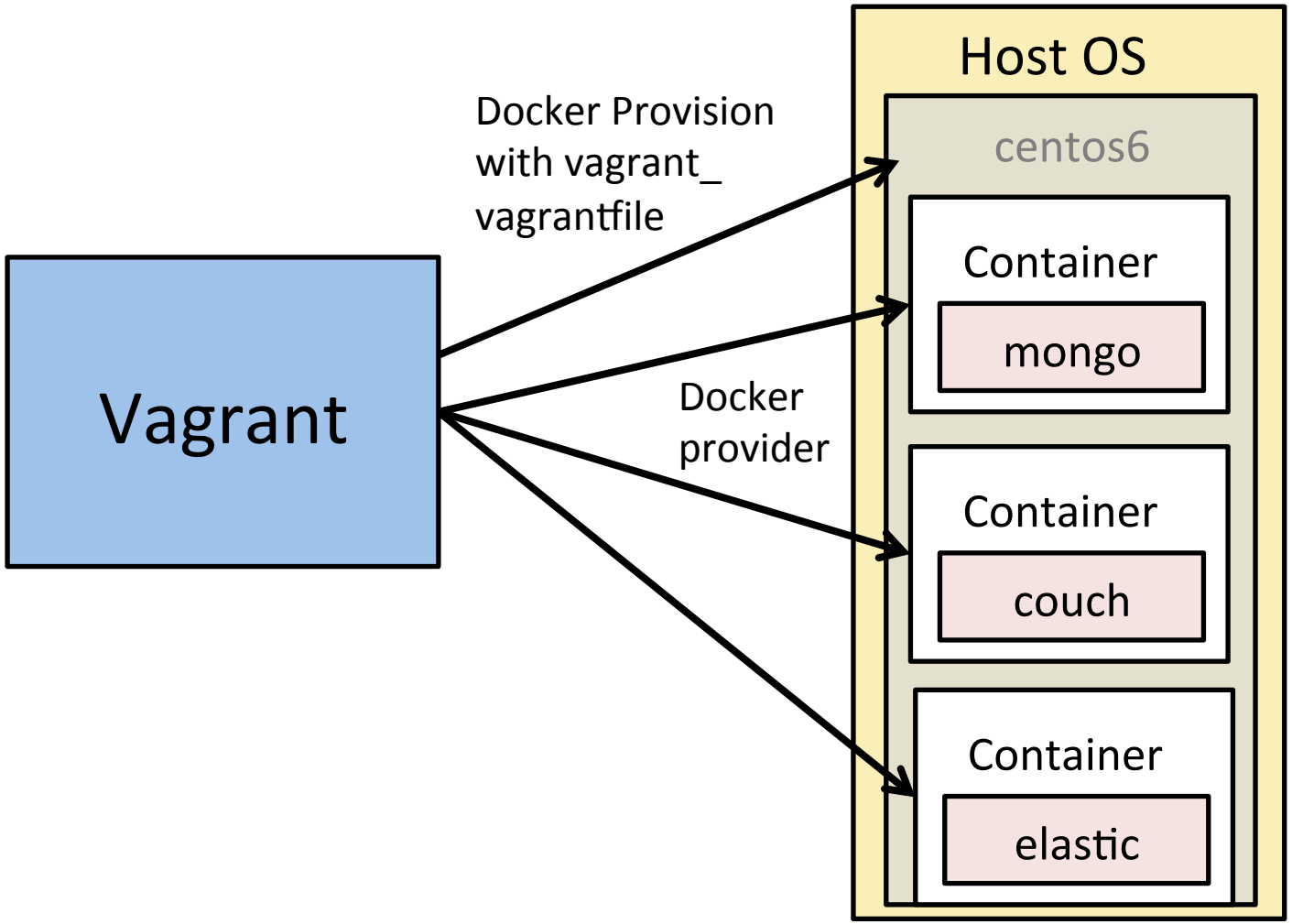
The problem is right here:



OK, we have an idea

Solution V:

Vagrant + Docker
Give up on boot2docker, use a box provider


Host OS
centos6
Container
mongo
Container
couch
Container
elastic
Vagrant
Docker Provision
with vagrant_
vagrantfile
Docker
provider

```
Vagrant.configure("2") do |config|
  //Look ma, box is back!
  HOST_VAGRANTFILE = "#{checkout}/docker/Vagrantfile"

  config.vm.define "mongodb" do |container|
    container.vm.provider "docker" do |d|
      d.image = "#{repo_url}/bintray/mongo:2.6”
      d.vagrant_vagrantfile = "#{HOST_VAGRANTFILE}"
      ...
    end
  end

  config.vm.define "couch" do |container|
    container.vm.provider "docker" do |d|
      d.image = "#{repo_url}/bintray/couchdb:1.6.0”
      d.vagrant_vagrantfile = "#{HOST_VAGRANTFILE}"
      ...
    end
  end
end
```

```
Vagrant.configure("2") do |config|
  config.vm.provision "docker"

  config.vm.define "dockerhost"
  config.vm.box = "dockerhost"
  config.vm.box_url = "http://repo-local:8081/artifactory/bintray-
resolver/dockerhost-centos6.box"
  config.vm.box_download_checksum ="4ee5de076da2a4e1c7ecc7f201f408bb"
  config.vm.box_download_checksum_type  = "md5"
  config.vm.synced_folder "../", "/vagrant", create: true
  config.vm.network "forwarded_port", guest: 5984, host: 5984
  ...
  config.vm.provider :virtualbox do |vb|
    vb.name = "dockerhost"
    vb.customize ["modifyvm", :id,"--cpus", 2,"--memory", 1512]
  end
end
```

How did we like it?

But…

Race condition on vagrant up

- When running multiple providers in the same vagrantfile, must use: `--no-parallel`
- SSHing to and destroying host vm is not trivial
- Devs started to get to know (and like) Docker

Official B2D is not up to date

- We want to use fork: https://github.com/YungSang/boot2docker-vagrant-box

OK, we have an idea

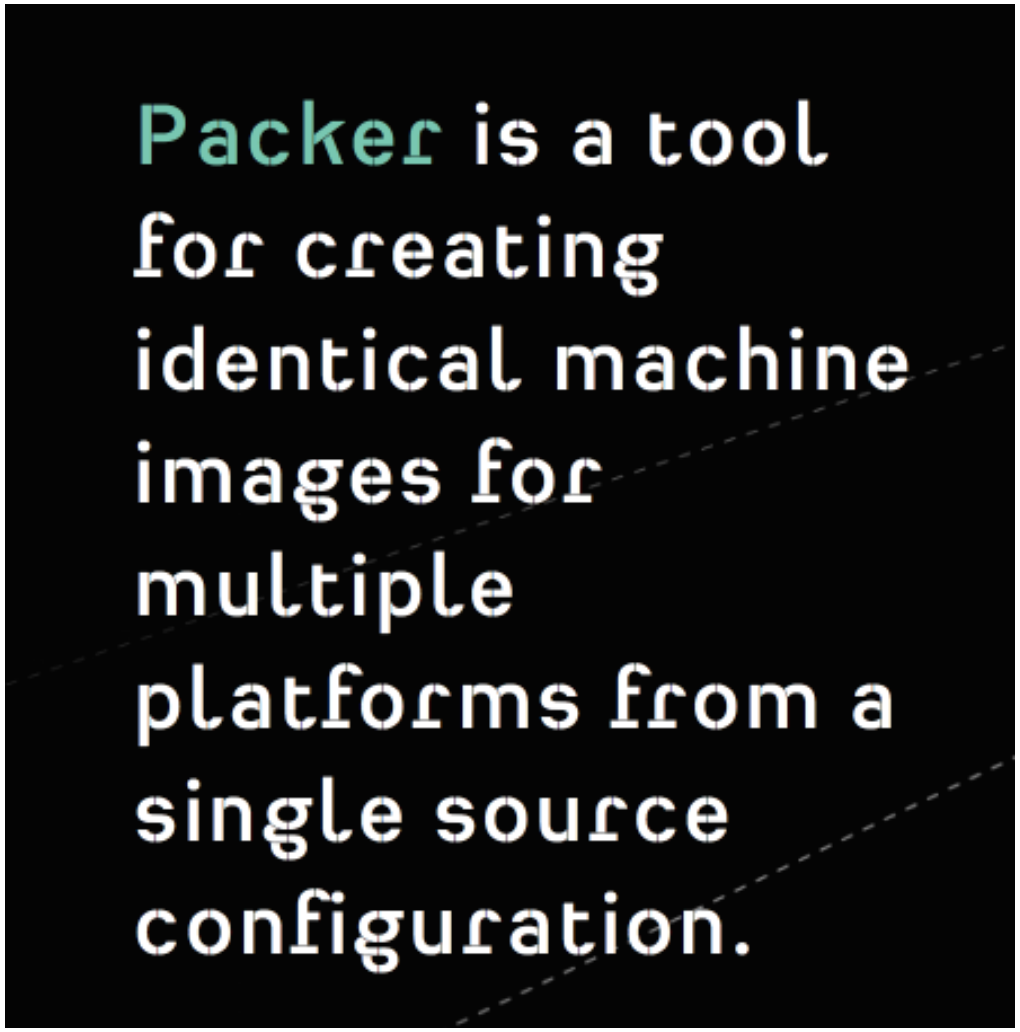
Solution VI:

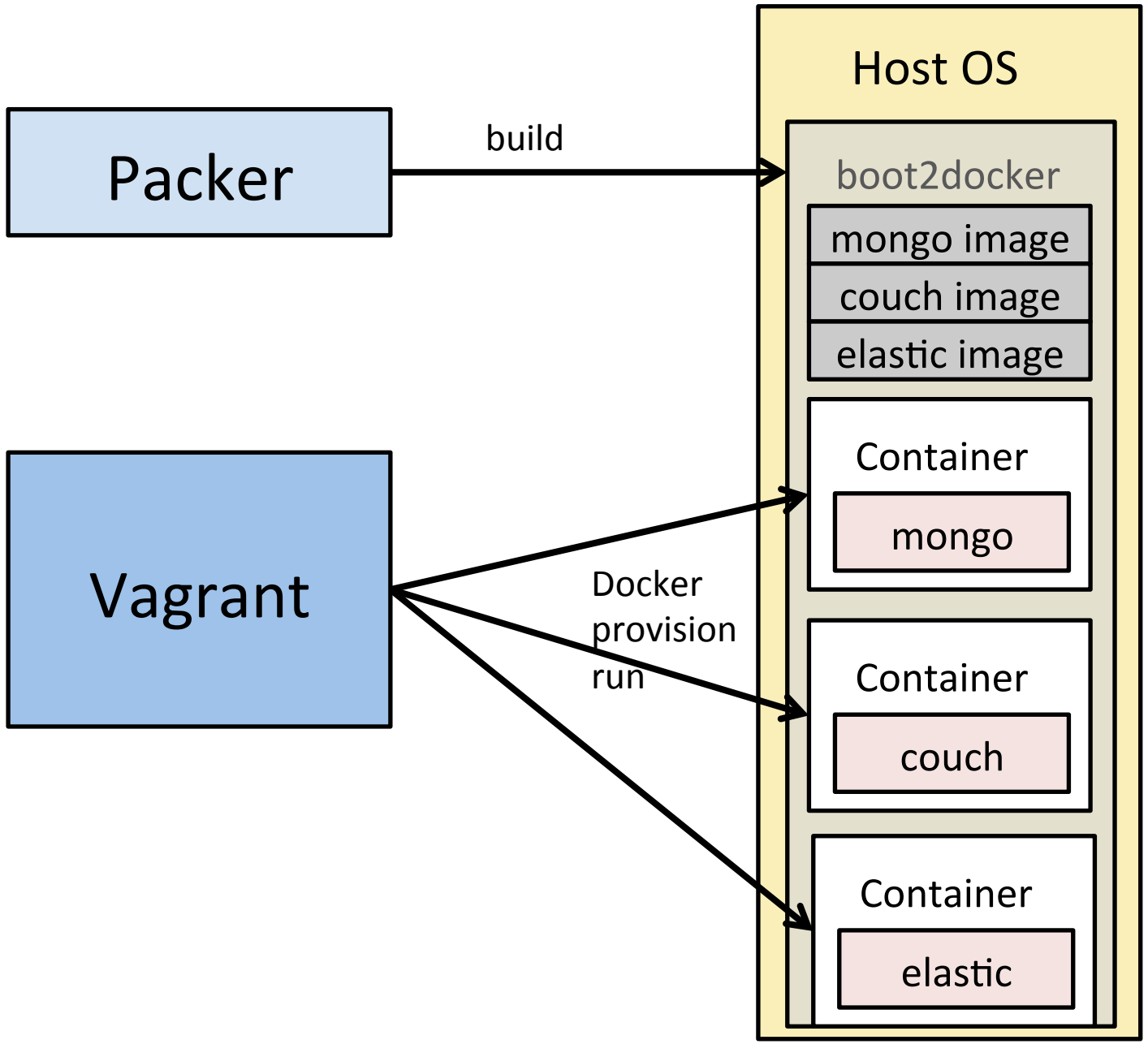
Vagrant + Docker
Full control: Make b2d and images part of the box

YungSang's boot2docker

- Well maintained – matches latest Docker versions
- Has 'docker enter' command – easy login to Docker

Build the VM with Packer




Packer
build
Host OS
boot2docker
mongo image
couch image
elastic image
Vagrant
Docker
provision
run
Container
mongo
Container
couch
Container
elastic

```
Vagrant.configure("2") do |config|
  config.vm.box = "bt-b2d-#{BOX_VERSION}"
  config.vm.box_url = "#{REPO}/bintray-boot2docker-vagrant/bt-
b2d-#{BOX_VERSION}.box"

  config.vm.provider :virtualbox do |vb, override|
    vb.gui = false
    ...
  end

  config.vm.provision :docker  do |d|
    d.run "mongodb",
        image: "#{DOCKER_REPO}bintray/mongo:dev",
        args: "-p 27017:27017 -v #{HOST_DATA_FOLDER}/mongodb/
log/:/data/log/ --privileged=true",
        ...

    d.run "couch",
        image: "#{DOCKER_REPO}bintray/couchdb:dev",
        args: "-p 5984:5984 --sig-proxy=false -v
#{HOST_DATA_FOLDER}/couchdb/log/:/usr/local/var/log/couchdb/”
```

```
"provisioners": [
    ...
    {
      "type": "shell",
      "inline": [
        "docker pull repo:80/bintray/mongo:dev",
        "docker pull repo:80/bintray/redis:dev",
        "docker pull repo:80/bintray/couchdb:dev",
        "docker pull repo:80/bintray/elasticsearch:dev"
      ],
      "pause_before": "30s"
    }
  ],
```

How did we like it?

But…

Still, some challenges

- Updating the base box
  - Solved by timestamping
  - Deciding when to repackage the box
- Developers Docker learning curve

How it works

How it works

1. Packer builds custom b2d image with base images preinstalled

How it works

1. Packer builds custom b2d image with base images preinstalled
2. Vagrant downloads the box from artifactory

How it works

1. Packer builds custom b2d image with base images preinstalled
2. Vagrant downloads the box from artifactory
3. Vagrant runs the b2d virtual box

How it works

1. Packer builds custom b2d image with base images preinstalled
2. Vagrant downloads the box from artifactory
3. Vagrant runs the b2d virtual box
4. Vagrant pulls & starts the containers

How it works

1. Packer builds custom b2d image with base images preinstalled
2. Vagrant downloads the box from artifactory
3. Vagrant runs the b2d virtual box
4. Vagrant pulls & starts the containers
5. Vagrant destroy keeps Base containers

No, Thank you!