# INFRASTRUCTURE

## *as* CODE

# Why infrastructure-as-code matters: a short story.

# You are starting a new project

**I know, I'll use Ruby on Rails!**

```
> gem install rails
```

```
> gem install rails
Fetching: i18n-0.7.0.gem (100%)
Fetching: json-1.8.3.gem (100%)
Building native extensions.  This could take a while...
ERROR: Error installing rails:
ERROR: Failed to build gem native extension.

        /usr/bin/ruby1.9.1 extconf.rb
creating Makefile

make
sh: 1: make: not found
```

**Ah, I just need to install make**

```
> sudo apt-get install make
...
Success!
```

```
> gem install rails
```

```
> gem install rails
Fetching: nokogiri-1.6.7.2.gem (100%)
Building native extensions.  This could take a while...
ERROR:  Error installing rails:
ERROR: Failed to build gem native extension.

        /usr/bin/ruby1.9.1extconf.rb
checking if the C compileraccepts...    yes
Building nokogiri usingpackaged libraries.
Using mini_portile version2.0.0.rc2
checking forgzdopen()  in -lz... no
zlib is missing; necessary for building libxml2
*** extconf.rb failed ***
```

**Hmm. Time to visit StackOverflow.**

```
> sudo apt-get install zlib1g-dev
```

...

Success!

```
> gem install rails
```

```
> gem install rails
Building native extensions.  This could take a while...
ERROR:  Error installing rails:
ERROR: Failed to build gem native extension.

        /usr/bin/ruby1.9.1 extconf.rb
checking if the C compiler accepts... yes
Building nokogiri using packaged libraries.
Using mini_portile version 2.0.0.rc2
checking for gzdopen() in -lz... yes
checking for iconv... yes

Extracting libxml2-2.9.2.tar.gz into tmp/x86_64-pc-linux-
gnu/ports/libxml2/2.9.2... OK
*** extconf.rb failed ***
```

**y u never install correctly?**

(Spend 2 hours trying random StackOverflow suggestions)

```
> gem install rails
```

```
> gem install rails
...
Success!
```

**Finally!**

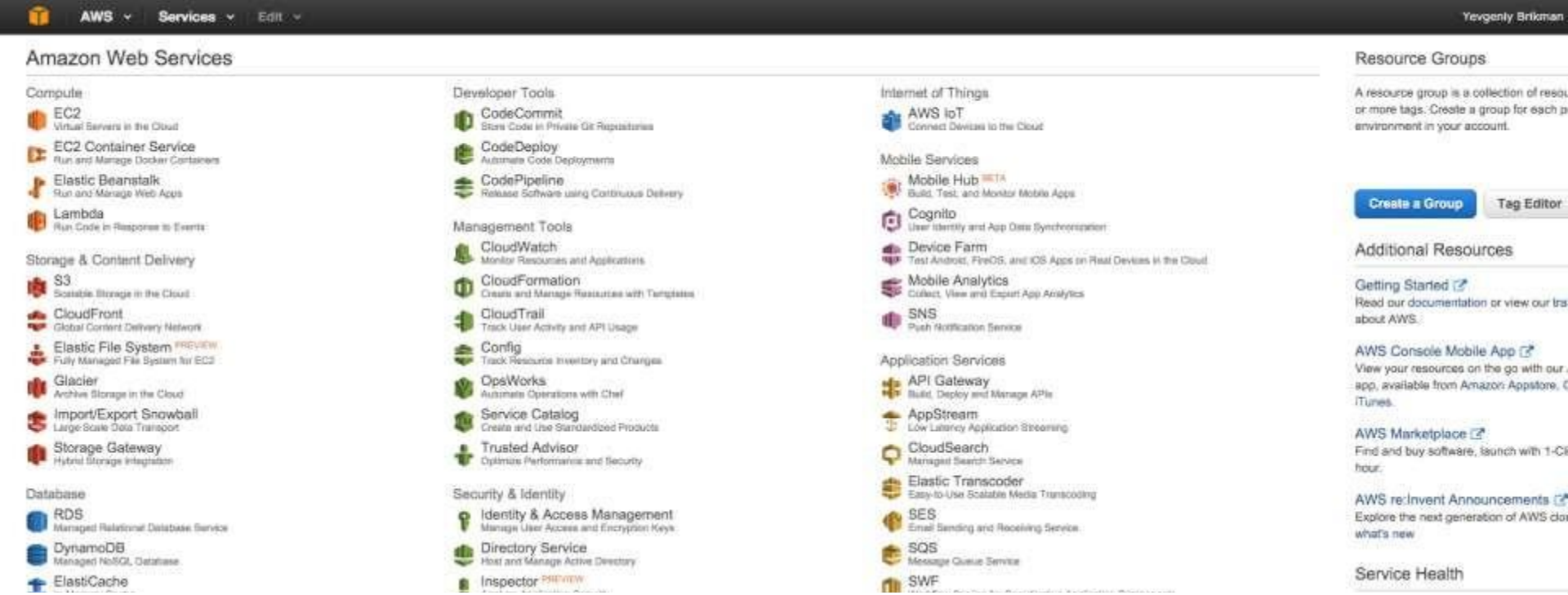
```
> rails new my-project
> cd my-project
> rails start
```

```
> rails new my-project
> cd my-project
> rails start

/source/my-project/bin/spring:11:in `<top (required)>':
undefined method `path_separator' for Gem:Module
(NoMethodError)
  from bin/rails:3:in `load'
  from bin/rails:3:in `<main>'
```


FFFFFFF
FFFFFFF
FFFFFFF
FFFFF
FFFFF
FFFFF
UUUU
UUUU
UUUU-

**Eventually**, you get it working

# Now you have to deploy your Rails app in production




# You use the AWS Console to deploy an EC2 instance

```
> ssh ec2-user@ec2-12-34-56-78.compute-1.amazonaws.com

       __|  __|_  )
       _|  (     /   Amazon Linux AMI
      ___|\___|___|

[ec2-user@ip-172-31-61-204 ~]$ gem install rails
```

```
> ssh ec2-user@ec2-12-34-56-78.compute-1.amazonaws.com

       __|  __|_  )
       _|  (     /   Amazon Linux AMI
      ___|\___|___|

[ec2-user@ip-172-31-61-204 ~]$ gem install rails
ERROR:  Error installing rails:
ERROR: Failed to build gem native extension.

        /usr/bin/ruby1.9.1 extconf.rb
```


FFFFFFF
FFFFFFF
FFFFFFF
FFFFF
FFFFF
FFFFF
UUUU
UUUU
UUUU-

**Eventually** you get it working

# Critical Ruby On Rails Issue Threatens 240,000 Websites

**Bug allows attackers to execute arbitrary code on any version of Ruby published in the last six years.**

All versions of the open source Ruby on Rails Web application framework released in the past six years have a critical vulnerability that an attacker could exploit to execute arbitrary code, steal information from databases and crash servers. As a result, all Ruby users should immediately upgrade to a newly released, patched version of the software.

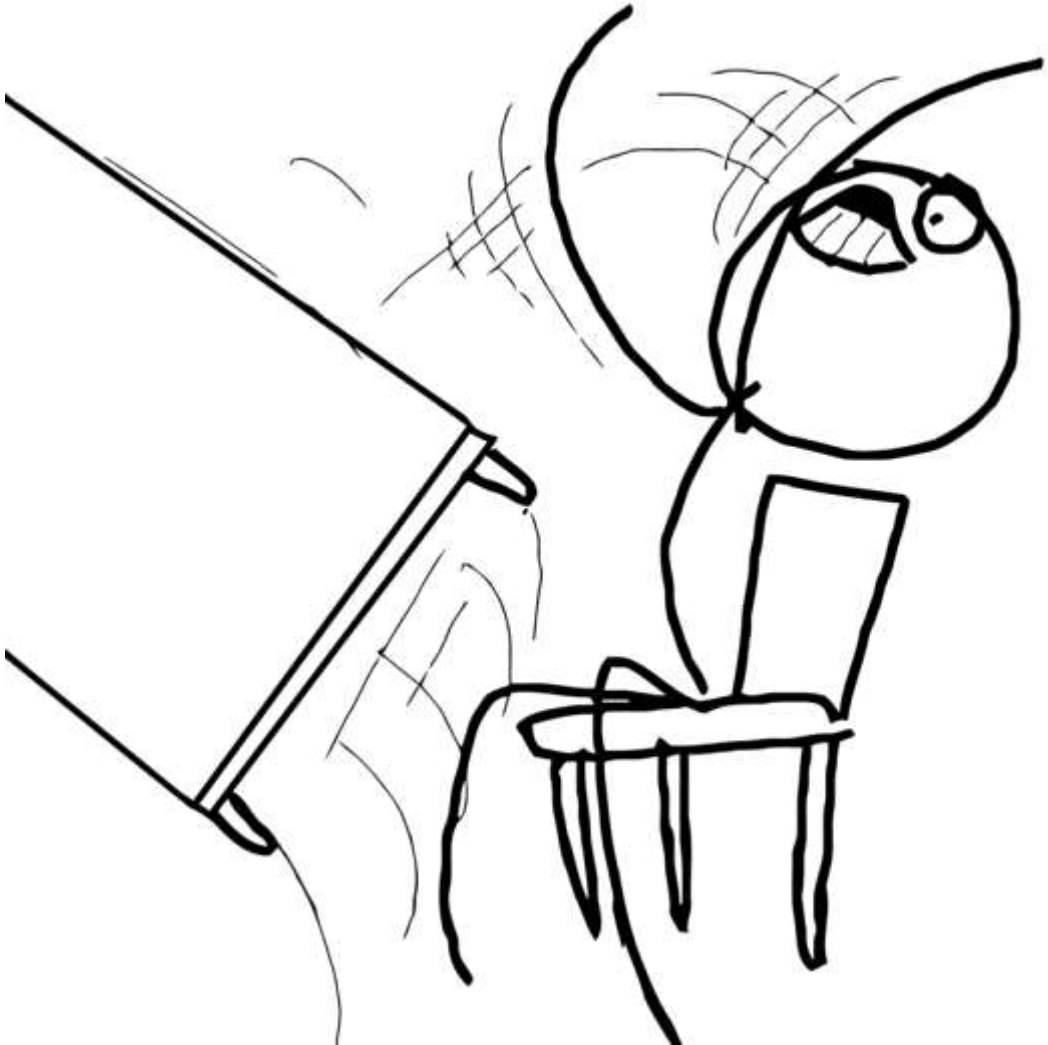
That warning was sounded Tuesday in a Google Groups post made by Aaron Patterson, a key Ruby programmer. "Due to the critical nature of this vulnerability, and the fact that portions of it have been disclosed publicly, all users running an affected release should either upgrade or use one of the work arounds immediately," he wrote. The patched versions of Ruby on Rails (RoR) are 3.2.11, 3.1.10, 3.0.19 and 2.3.15.

As a result, more than 240,000 websites that use Ruby on Rails Web applications are at risk of being exploited by attackers. High-profile websites

Now you urgently have to update all your Rails installs

```
> bundle update rails
```

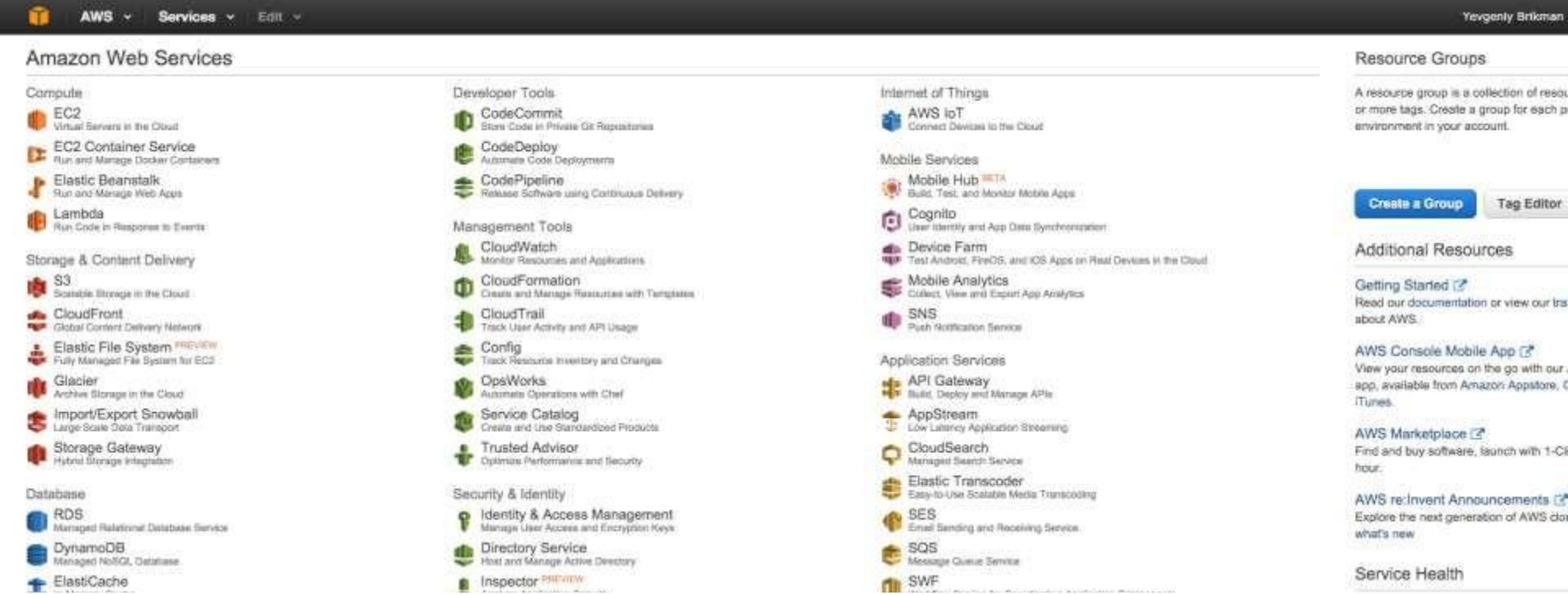
```
> bundle update rails
Building native extensions.  This could take a while...
ERROR:  Error installing rails:
ERROR: Failed to build gem native extension.

        /usr/bin/ruby1.9.1extconf.rb
checking if the C compileraccepts...    yes
Building nokogiri usingpackaged  libraries.
Using mini_portile version2.0.0.rc2
checking forgzdopen() in-lz...    yes
checking foriconv...  yes

Extracting libxml2-2.9.2.tar.gz into tmp/x86_64-pc-linux-
gnu/ports/libxml2/2.9.2... OK
*** extconf.rb failed ***
```



# The problem isn’t Rails

```
> ssh ec2-user@ec2-12-34-56-78.compute-1.amazonaws.com

       __|  __|_  )
       _|  (     /   Amazon Linux AMI
      ___|\___|___|

[ec2-user@ip-172-31-61-204 ~]$ gem install rails
```

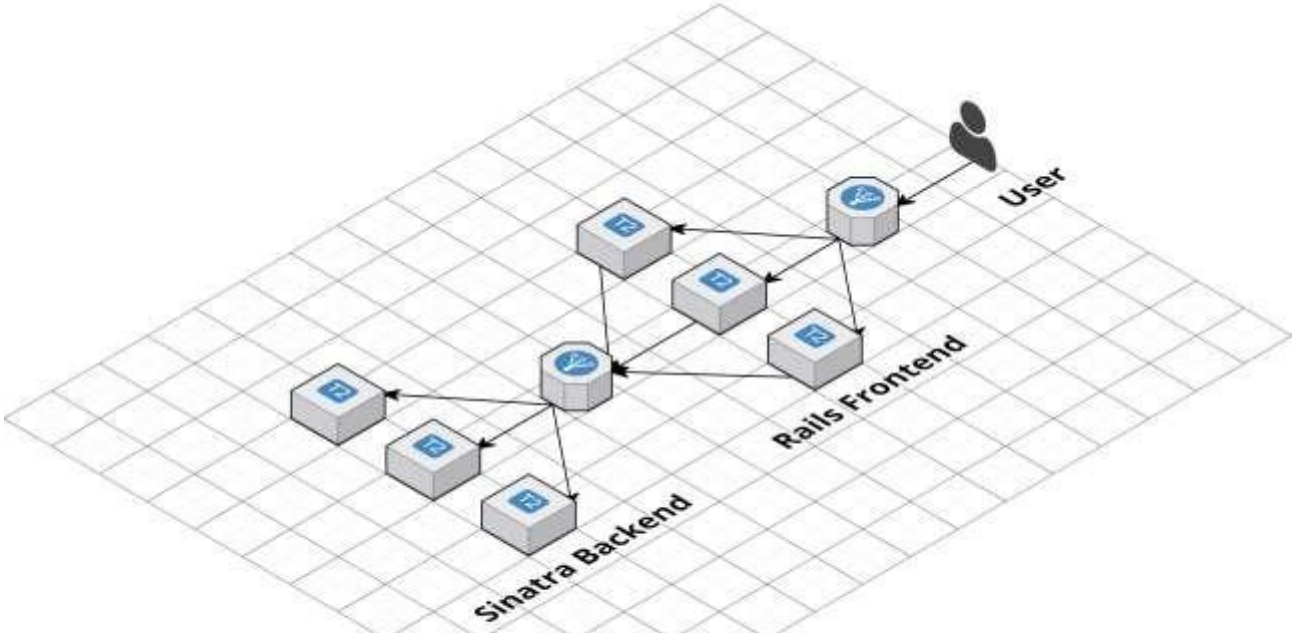
# The problem is that you're configuring servers manually



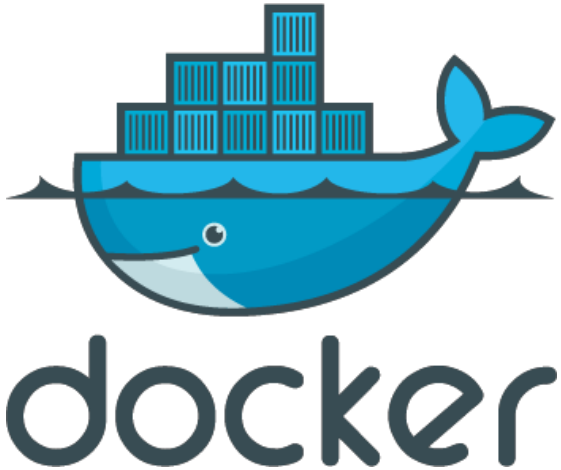

# And that you're deploying infrastructure manually

# A better alternative: infrastructure-as-code

**In this talk, we'll go through a real-world example:**




# We'll configure & deploy two microservices on Amazon ECS





# With two infrastructure-as-code tools: Docker and Terraform

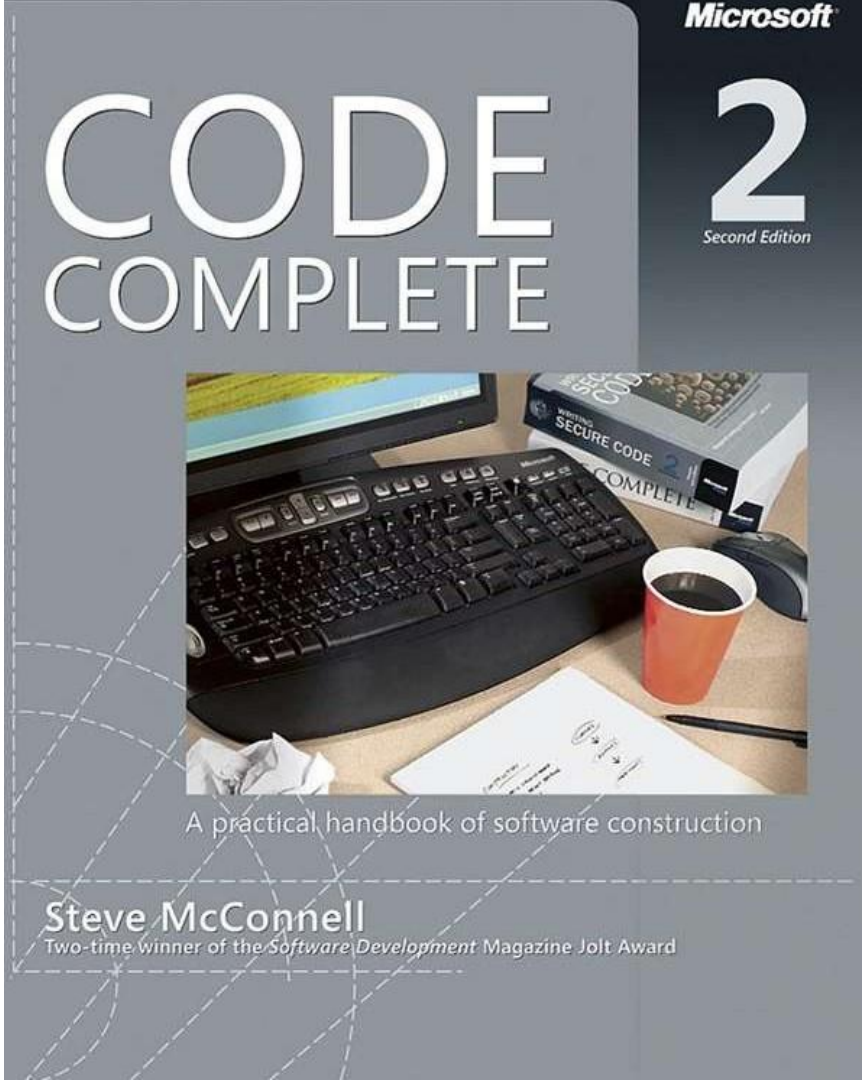
# Outline

1. Microservices
2. Docker
3. Terraform
4. Recap


Outline
1. Microservices
2. Docker
3. Terraform
4. ECS
5. Recap

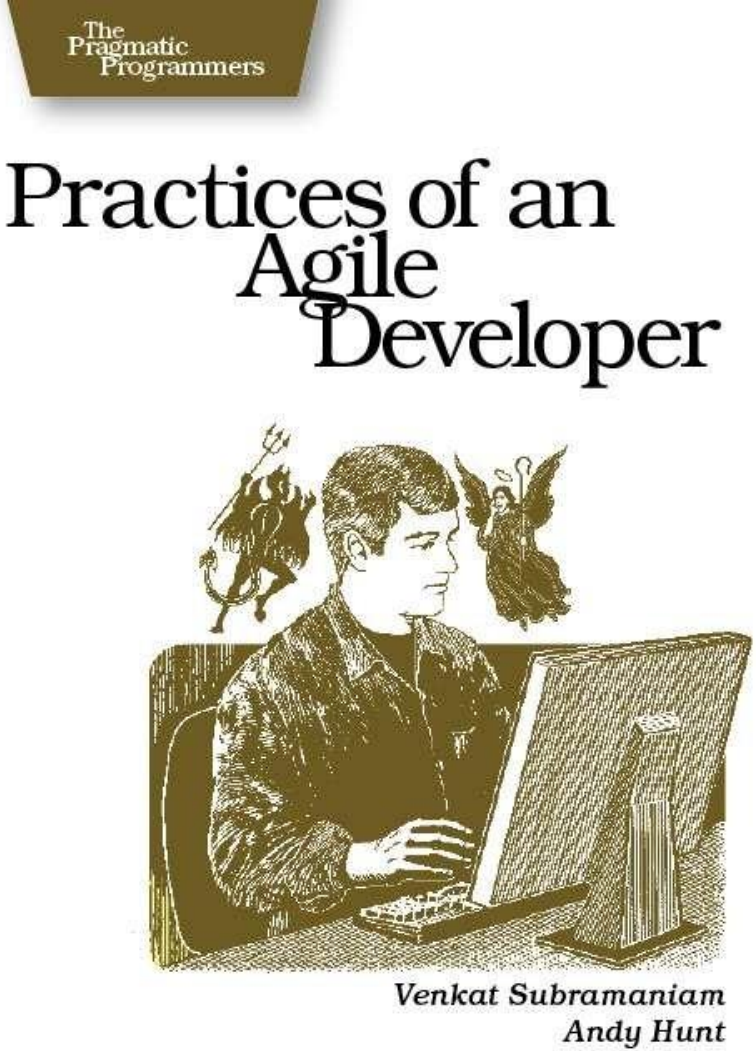
**Code is the enemy**: the more you have, the slower you go

| Project Size<br>Lines of code | Bug Density<br>Bugs per thousand lines of code |
|---|---|
| < 2K | 0 – 25 |
| 2K – 6K | 0 – 40 |
| 16K – 64K | 0.5 – 50 |
| 64K – 512K | 2 – 70 |
| > 512K | 4 – 100 |



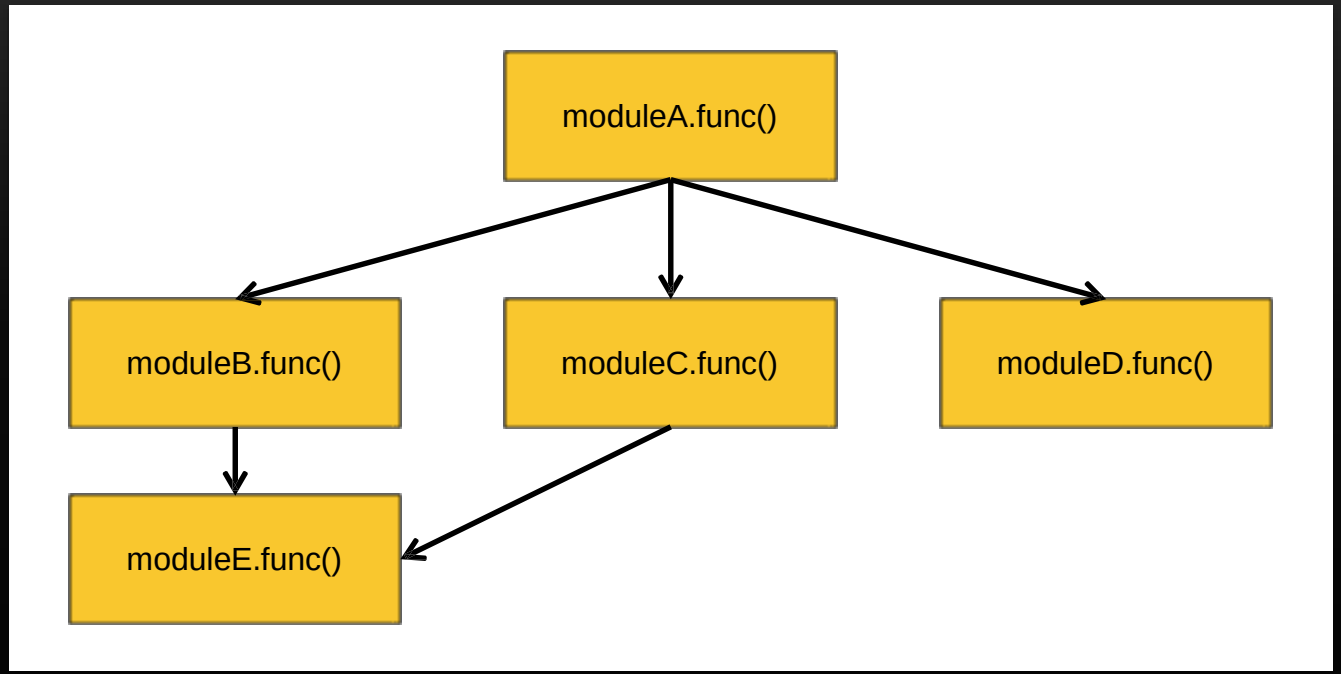
As the code grows, the number of bugs grows even faster

*“Software development doesn't happen in a chart, an IDE, or a design tool; it happens in your head.”*

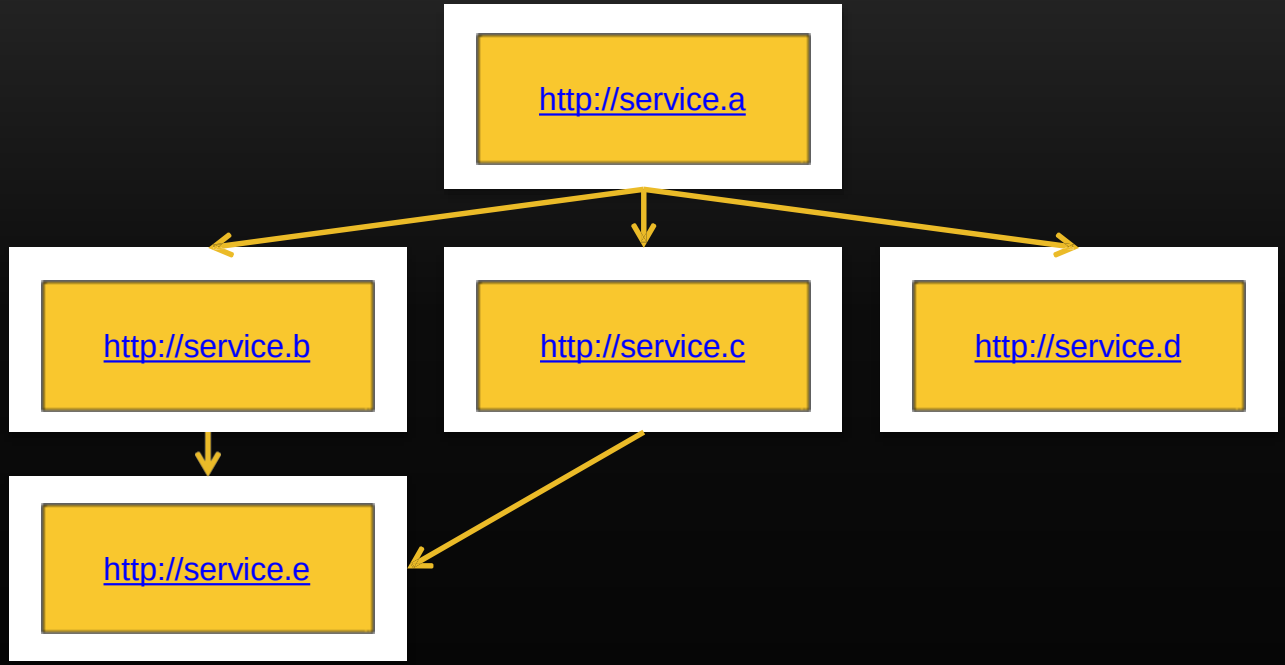


**The mind can only handle so much complexity at once**

One solution is to break the code into microservices




# In a monolith, you use function calls within one process




# With services, you pass messages between processes

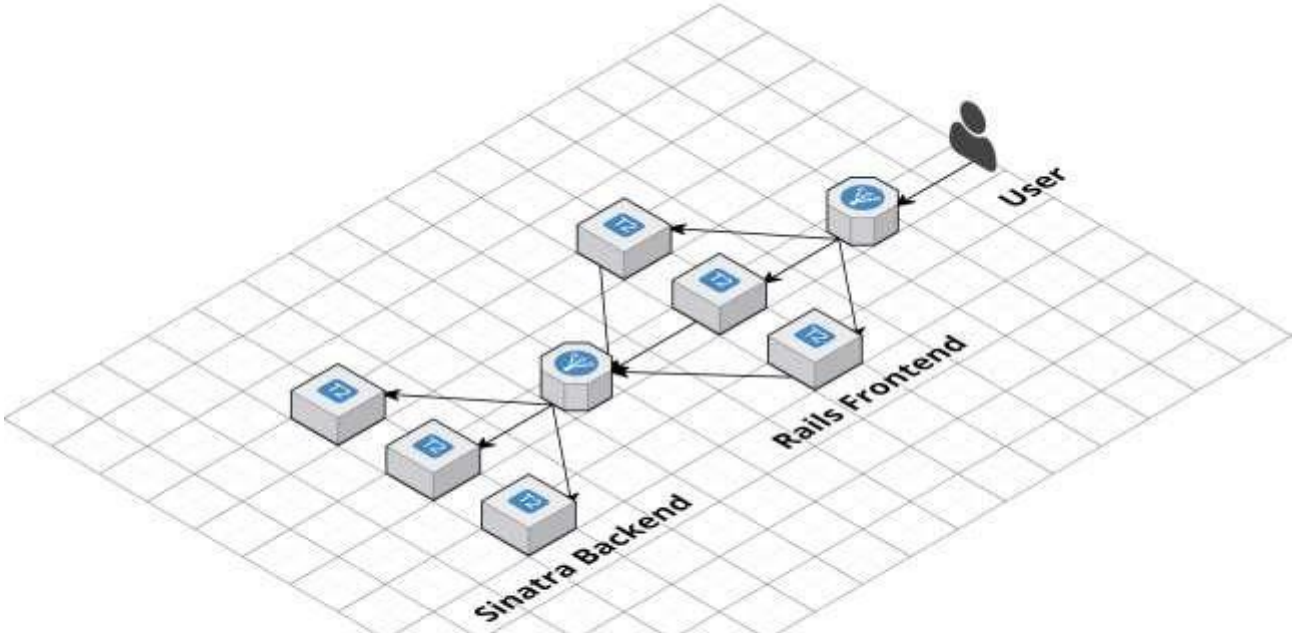
# Advantages of services:

1. Isolation
2. Technology agnostic
3. Scalability

# Disadvantages of services:

1. Operational overhead
2. Performance overhead
3. I/O, error handling
4. Backwards compatibility
5. Global changes, transactions, referential integrity all very hard

For more info, see: Splitting Up a Codebase into Microservices and Artifacts




# For this talk, we'll use two example microservices

```
require 'sinatra'

get "/" do
  "Hello, World!"
end
```

# A sinatra backend that returns “Hello, World”

```
class ApplicationController <ActionController::Base
  def index
    url = URI.parse(backend_addr)
    req = Net::HTTP::Get.new(url.to_s)
    res = Net::HTTP.start(url.host, url.port) {|http|
      http.request(req)
    }
    @text = res.body
  end
end
```

# A rails frontend that calls the sinatra backend

```
<h1>Rails Frontend</h1>
<p>
  Response from the backend: <strong><%= @text %></strong>
</p>
```

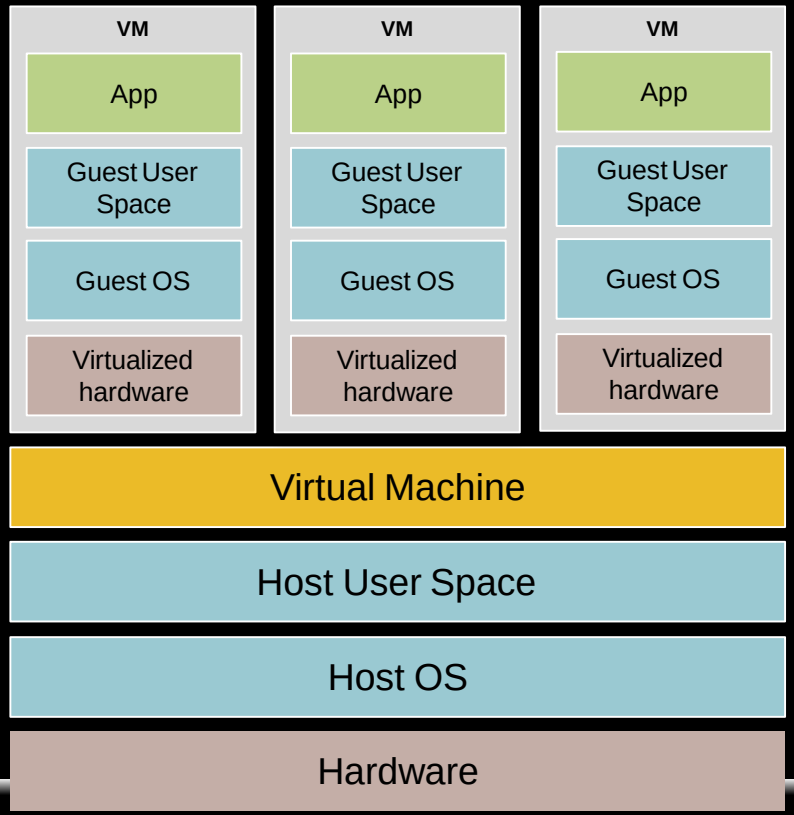
# And renders the response as HTML

# Outline

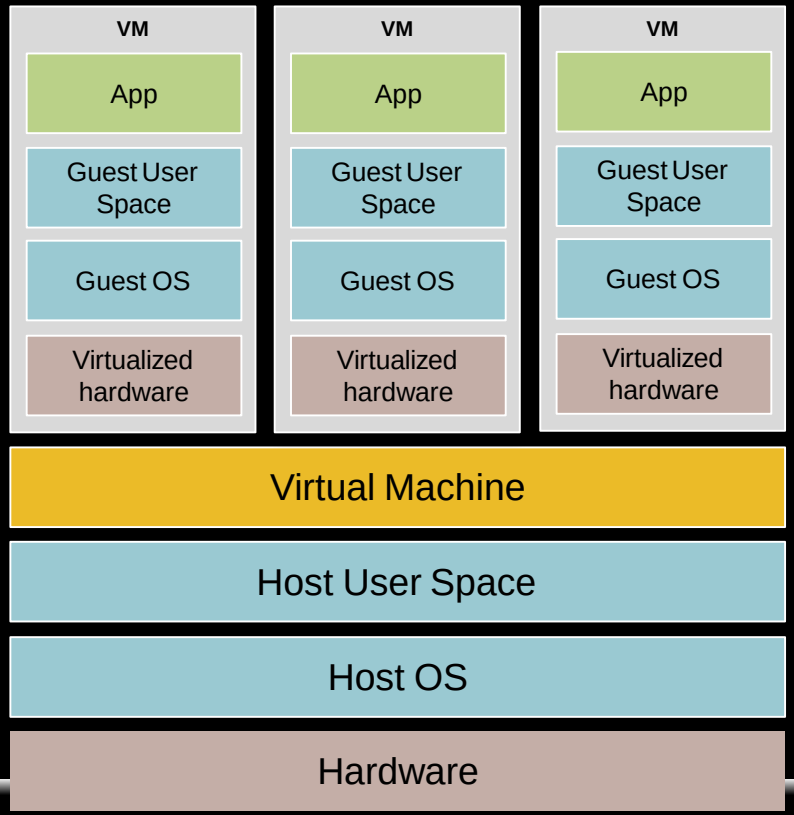
1. Microservices
2. **Docker**
3. Terraform
4. ECS
5. Recap

**Docker allows you to build and run code in containers**

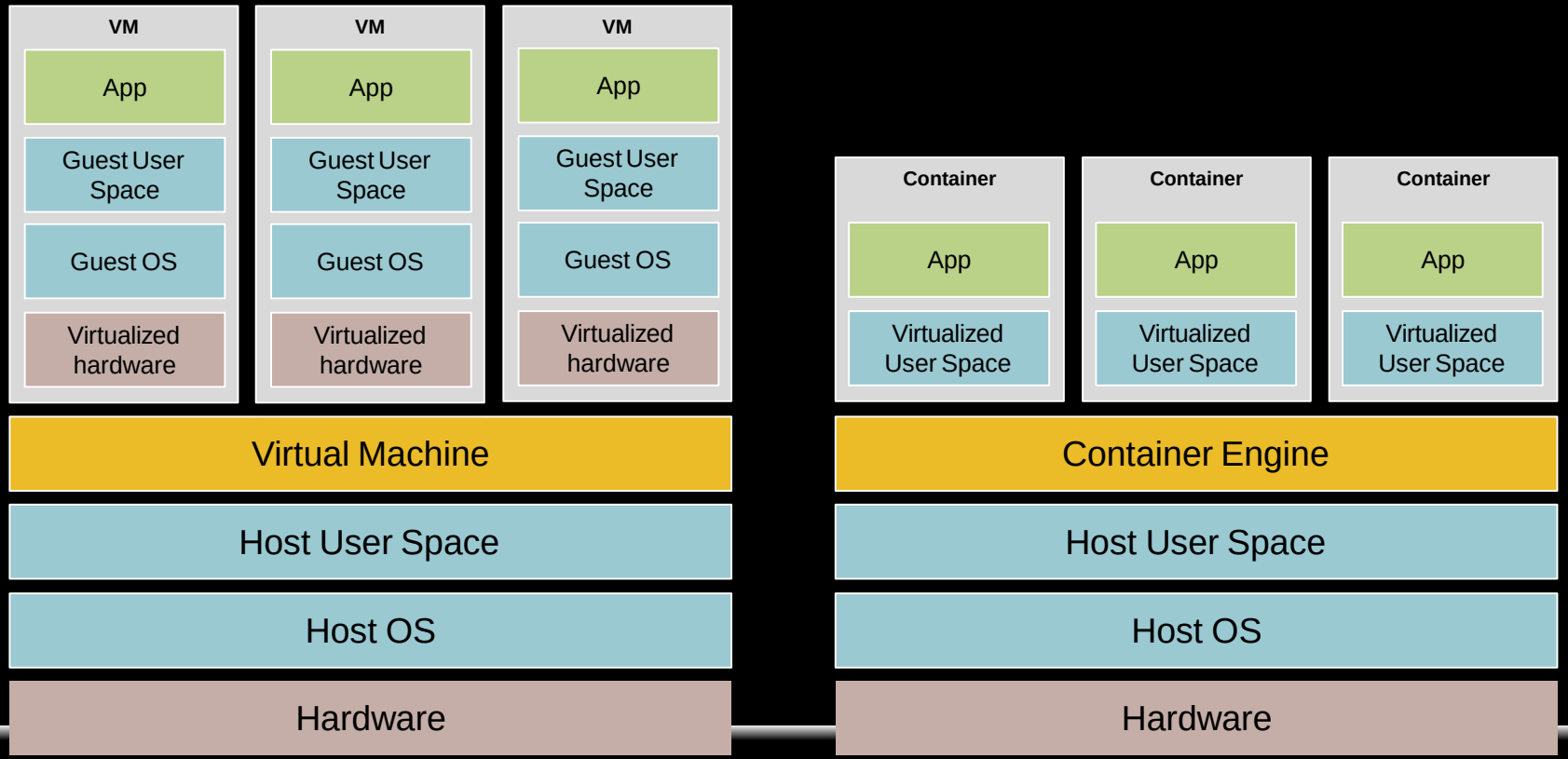
# Containers are like lightweight Virtual Machines (VMs)



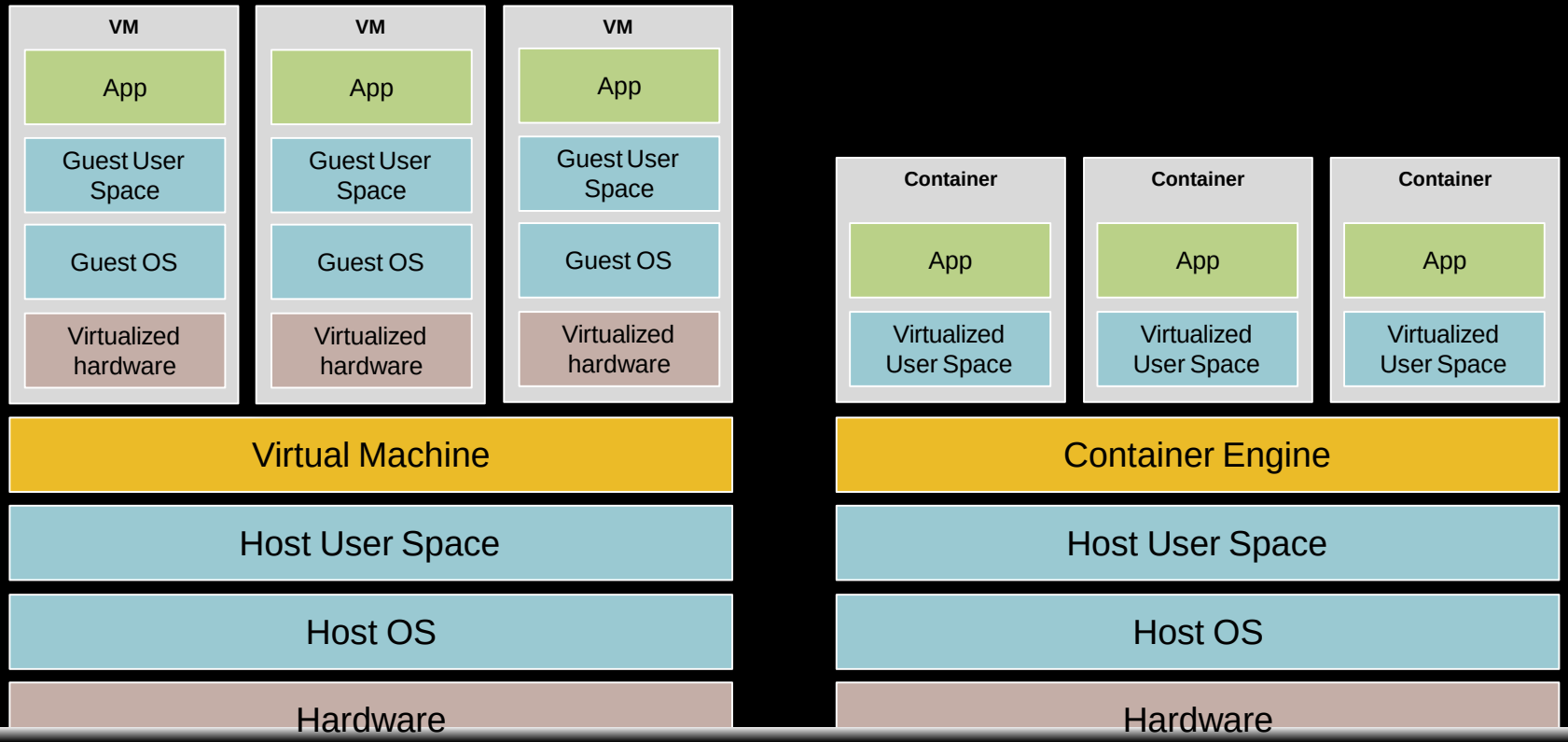

# VMs virtualize the hardware and run an entire guest OS on top of the host OS




**This provides good isolation, but lots of CPU, memory, disk, & startup overhead**




# Containers virtualize User Space (shared memory, processes, mount, network)




# Isolation isn’t as good but much less CPU, memory, disk, startup overhead

```
> docker run –it ubuntu bash

root@12345:/# echo "I'm in $(cat /etc/issue)”

I'm in Ubuntu 14.04.4 LTS
```

# Running Ubuntu in a Docker container

```
> time docker run ubuntu echo "Hello, World"
Hello, World

real  0m0.183s
user  0m0.009s
sys 0m0.014s
```

**Containers boot very quickly.**
**Easily run a dozen at once.**

You can define a Docker image as code in a Dockerfile

```
FROM gliderlabs/alpine:3.3

RUN apk --no-cache add ruby ruby-dev
RUN gem install sinatra --no-ri --no-rdoc

RUN mkdir -p /usr/src/app
COPY . /usr/src/app
WORKDIR /usr/src/app

EXPOSE 4567
CMD ["ruby", "app.rb"]
```

# Here is the Dockerfile for the Sinatra backend

```
FROM gliderlabs/alpine:3.3

RUN apk --no-cache add ruby ruby-dev
RUN gem install sinatra --no-ri --no-rdoc

RUN mkdir -p /usr/src/app
COPY . /usr/src/app
WORKDIR /usr/src/app

EXPOSE 4567
CMD ["ruby", "app.rb"]
```

**It specifies dependencies, code, config, and how to run the app**

```
> docker build -t brikis98/sinatra-backend .
Step 0 : FROM gliderlabs/alpine:3.3
 ---> 0a7e169bce21

(...)

Step 8 : CMD ruby app.rb
---> 2e243eba30ed

Successfully built 2e243eba30ed
```

# **Build** the Docker image

```
> docker run -it -p 4567:4567 brikis98/sinatra-backend
INFO  WEBrick 1.3.1
INFO  ruby 2.2.4 (2015-12-16) [x86_64-linux-musl]
== Sinatra (v1.4.7) has taken the stage on 4567 for
development with backup from WEBrick
INFO  WEBrick::HTTPServer#start: pid=1 port=4567
```

# Run the Docker image

```
> docker push brikis98/sinatra-backend
The push refersto  a repository [docker.io/brikis98/sinatra-
backend] (len: 1)
2e243eba30ed: Image successfully pushed
7e2e0c53e246: Image successfully pushed
919d9a73b500: Image successfully pushed

(...)

v1: digest: sha256:09f48ed773966ec7fe4558 size: 14319
```

# You can share your images by pushing them to Docker Hub

Now you can reuse the same image in dev, stg, prod, etc

```
> docker pull rails:4.2.6
```

**And you can reuse images created by others.**

```
FROM rails:4.2.6

RUN mkdir -p /usr/src/app
COPY . /usr/src/app
WORKDIR /usr/src/app

RUN bundle install

EXPOSE 3000
CMD ["rails", "start"]
```

**The rails-frontend is built on top of the official rails Docker image**

**No more insane install procedures!**

```
rails_frontend:
  image: brikis98/rails-frontend
  ports:
    - "3000:3000"
  links:
    - sinatra_backend:sinatra_backend

sinatra_backend:
  image: brikis98/sinatra-backend
  ports:
    - "4567:4567"
```

# Define your entire dev stack as code with docker-compose

```
rails_frontend:
  image: brikis98/rails-frontend
  ports:
    - "3000:3000"
  links:
    - sinatra_backend

sinatra_backend:
  image: brikis98/sinatra-backend
  ports:
    - "4567:4567"
```

**Docker links provide a simple service discovery mechanism**

```
> docker-compose up
Starting infrastructureascodetalk_sinatra_backend_1
Recreating infrastructureascodetalk_rails_frontend_1

sinatra_backend_1 | INFO  WEBrick 1.3.1
sinatra_backend_1 | INFO  ruby 2.2.4 (2015-12-16)
sinatra_backend_1 | Sinatra has taken the stage on 4567

rails_frontend_1  | INFO  WEBrick 1.3.1
rails_frontend_1  | INFO  ruby 2.3.0 (2015-12-25)
rails_frontend_1  | INFO  WEBrick::HTTPServer#start: port=3000
```

# Run your entire dev stack with one command

# Advantages of Docker:

1. Easy to create & share images
2. Images run the same way in all environments (dev, test, prod)
3. Easily run the entire stack in dev
4. Minimal overhead
5. Better resource utilization

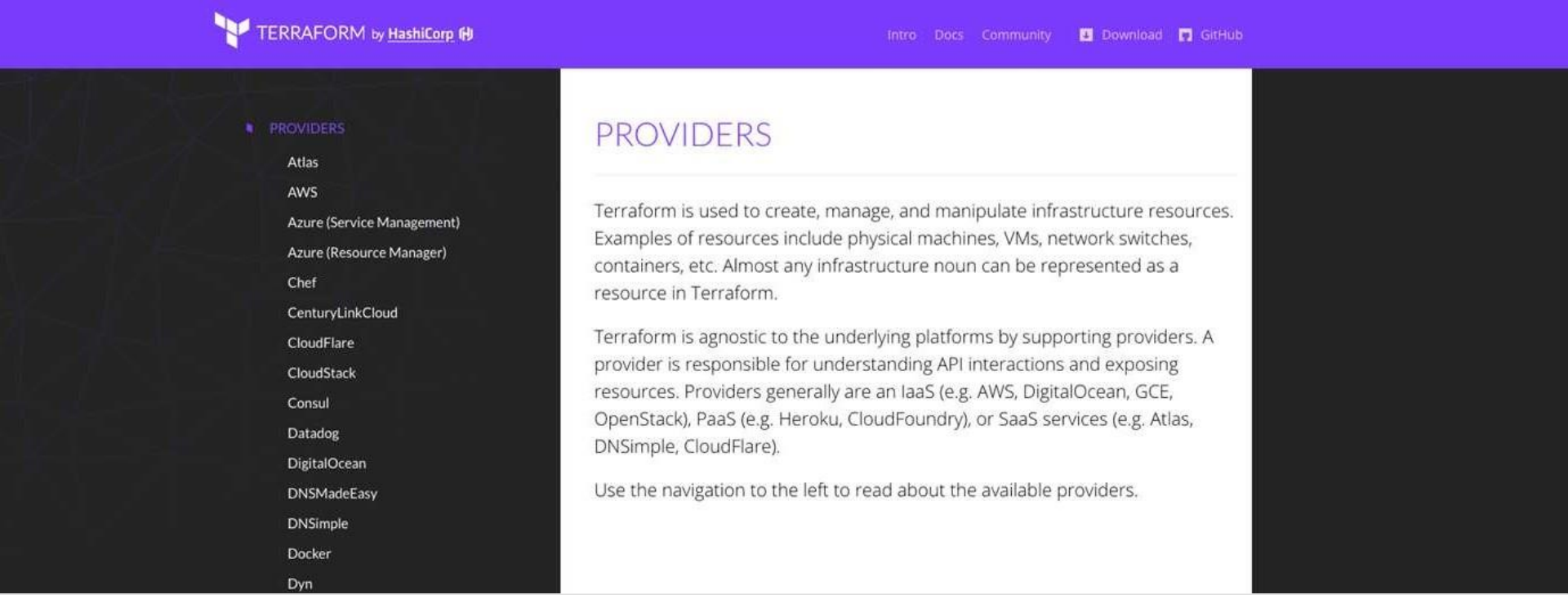
# Disadvantages of Docker:

1. Maturity. Ecosystem developing very fast, but still a ways to go
2. Tricky to manage persistent data in a container
3. Tricky to pass secrets to containers

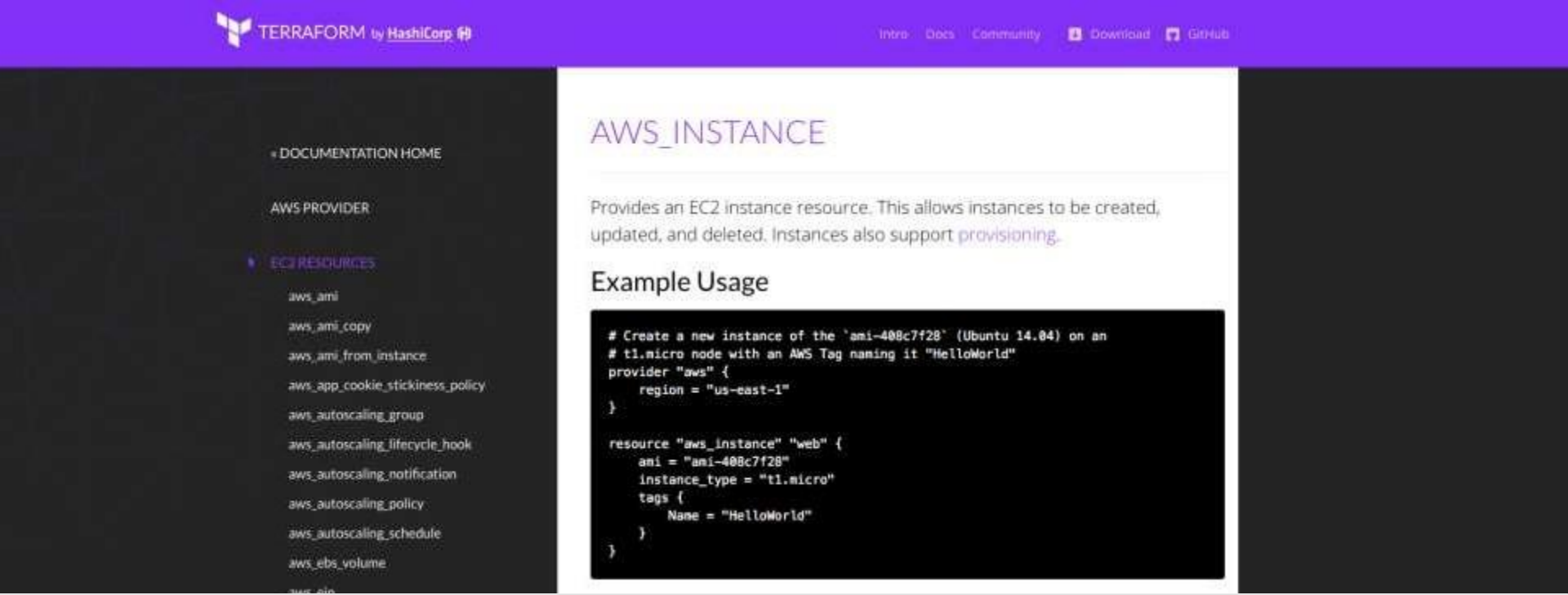
# Outline

1. Microservices
2. Docker
3. **Terraform**
4. ECS
5. Recap

Terraform is a tool for provisioning infrastructure




# Terraform supports many providers (cloud agnostic)




# And many resources for each provider

# You define infrastructure as code in Terraform templates

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "example" {
  ami = "ami-408c7f28"
  instance_type = "t2.micro"
}
```

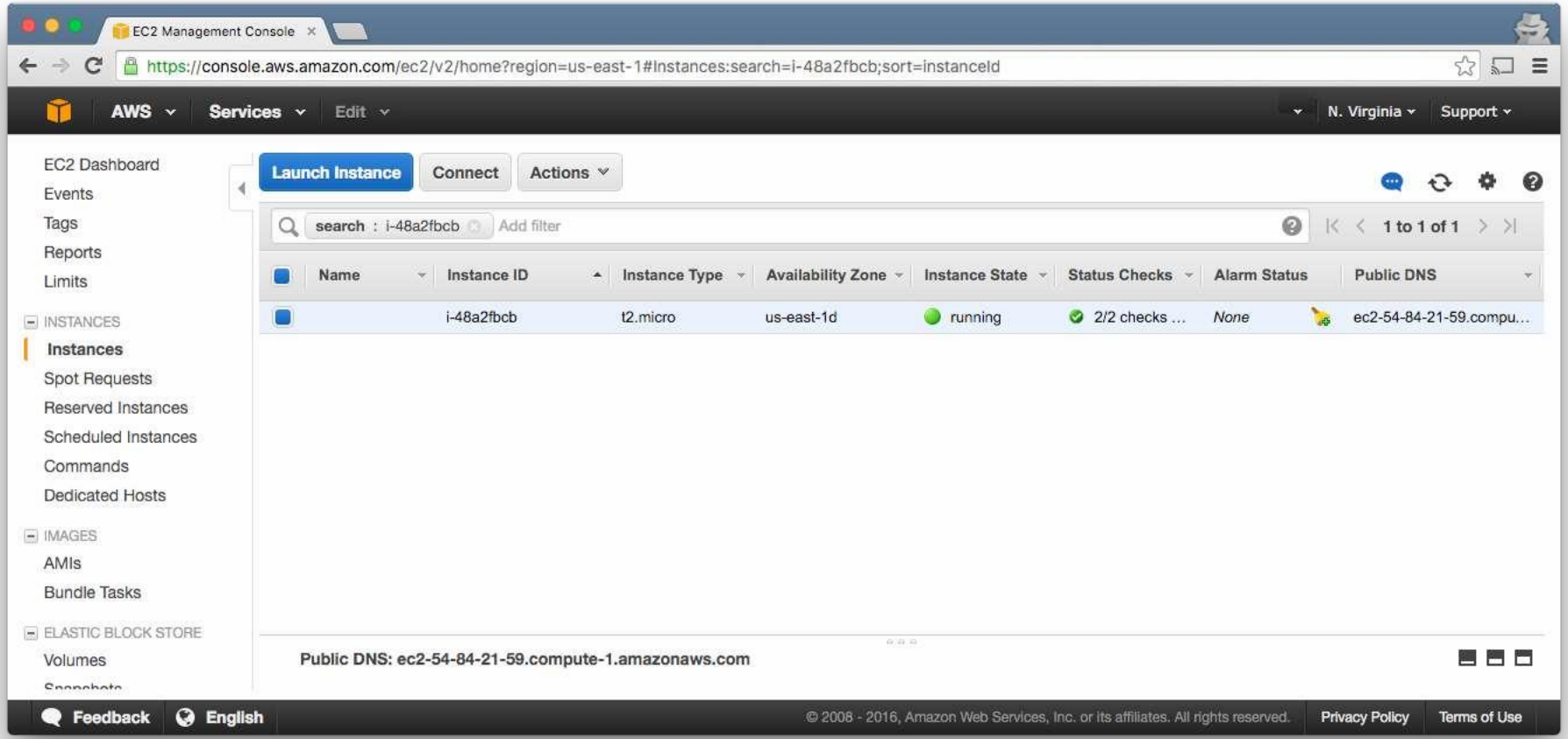
**This template creates a single EC2 instance in AWS**

```
> terraform plan
+ aws_instance.example
    ami:                          "" => "ami-408c7f28"
    instance_type:                "" => "t2.micro"
    key_name:                     "" => "<computed>"
    private_ip:                   "" => "<computed>"
    public_ip:                    "" => "<computed>"

Plan: 1 to add, 0 to change, 0 to destroy.
```

# Use the plan command to see what you're about to deploy

```
> terraform apply
aws_instance.example:  Creating...
     ami:                          "" => "ami-408c7f28"
     instance_type:                "" => "t2.micro"
     key_name:                     "" => "<computed>"
     private_ip:                   "" => "<computed>"
     public_ip:                    "" => "<computed>”
aws_instance.example: Creation complete

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

# Use the apply command to apply the changes




# Now our EC2 instance is running!

```
resource "aws_instance" "example" {
    ami ="ami-408c7f28"
    instance_type = "t2.micro"
    tags{
        Name = "terraform-example"
    }
}
```

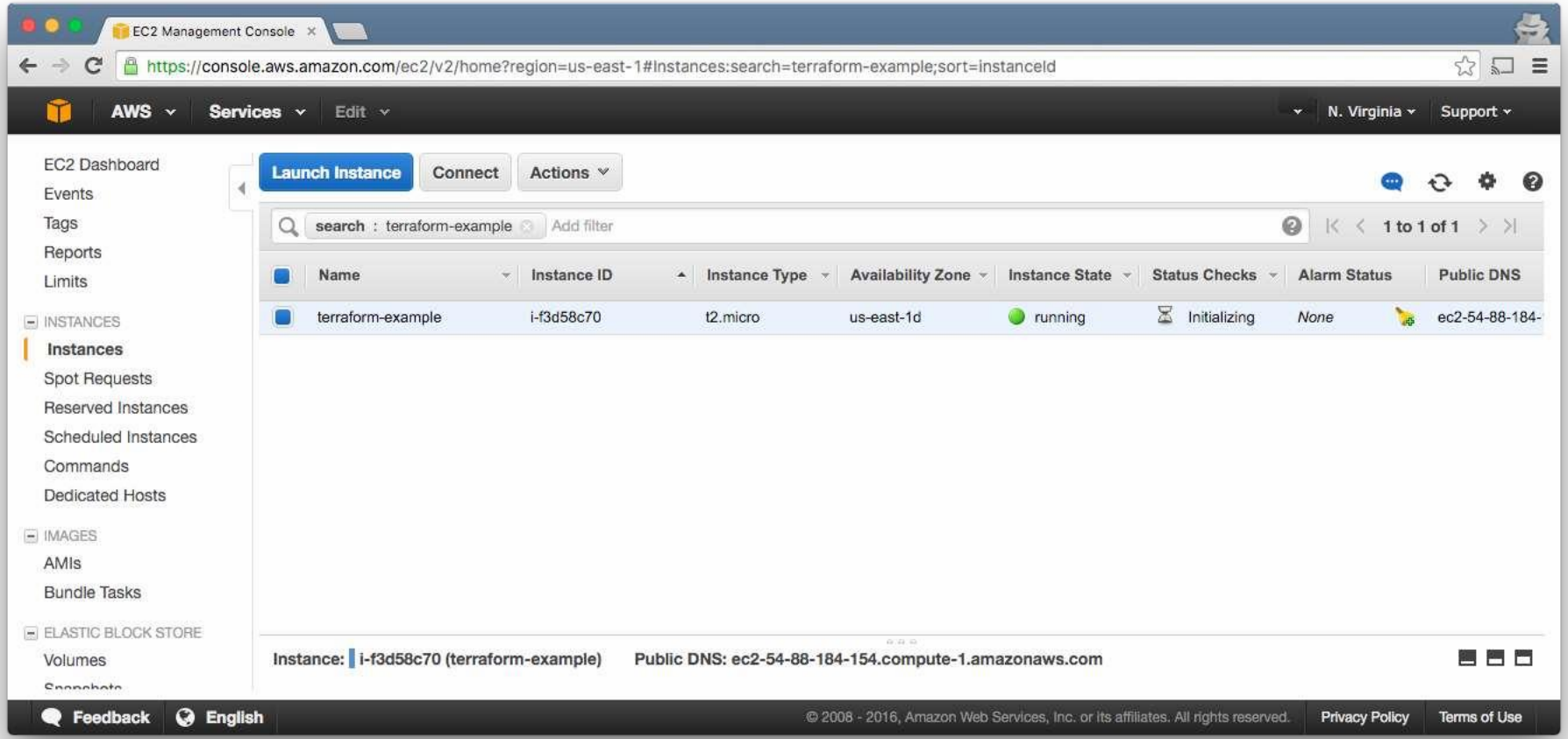
# Let’s give the EC2 instance a tag with a readable name

```
> terraform plan
~ aws_instance.example
    tags.#:     "0" => "1"
    tags.Name: "" => "terraform-example"


Plan: 0 to add, 1 to change, 0 to destroy.
```

# Use the plan command again to verify your changes

```
> terraform apply
aws_instance.example: Refreshing state...
aws_instance.example: Modifying...
  tags.#:    "0" => "1"
  tags.Name: "" => "terraform-example"
aws_instance.example: Modifications complete

Apply complete! Resources: 0 added, 1 changed, 0 destroyed.
```

# Use the apply command again to deploy those changes




# Now our EC2 instance has a tag!

```
resource "aws_elb" "example" {
    name = "example"
    availability_zones = ["us-east-1a", "us-east-1b"]
    instances=["${aws_instance.example.id}"]
    listener {
        lb_port = 80
        lb_protocol = "http"
        instance_port = "${var.instance_port}"
        instance_protocol = "http"
    }
```

# Let's add an Elastic Load Balancer (ELB).

```
resource "aws_elb" "example" {
    name = "example"
    availability_zones = ["us-east-1a", "us-east-1b"]
    instances=["${aws_instance.example.id}"]
    listener {
        lb_port = 80
        lb_protocol = "http"
        instance_port = "${var.instance_port}"
        instance_protocol = "http"
    }
```

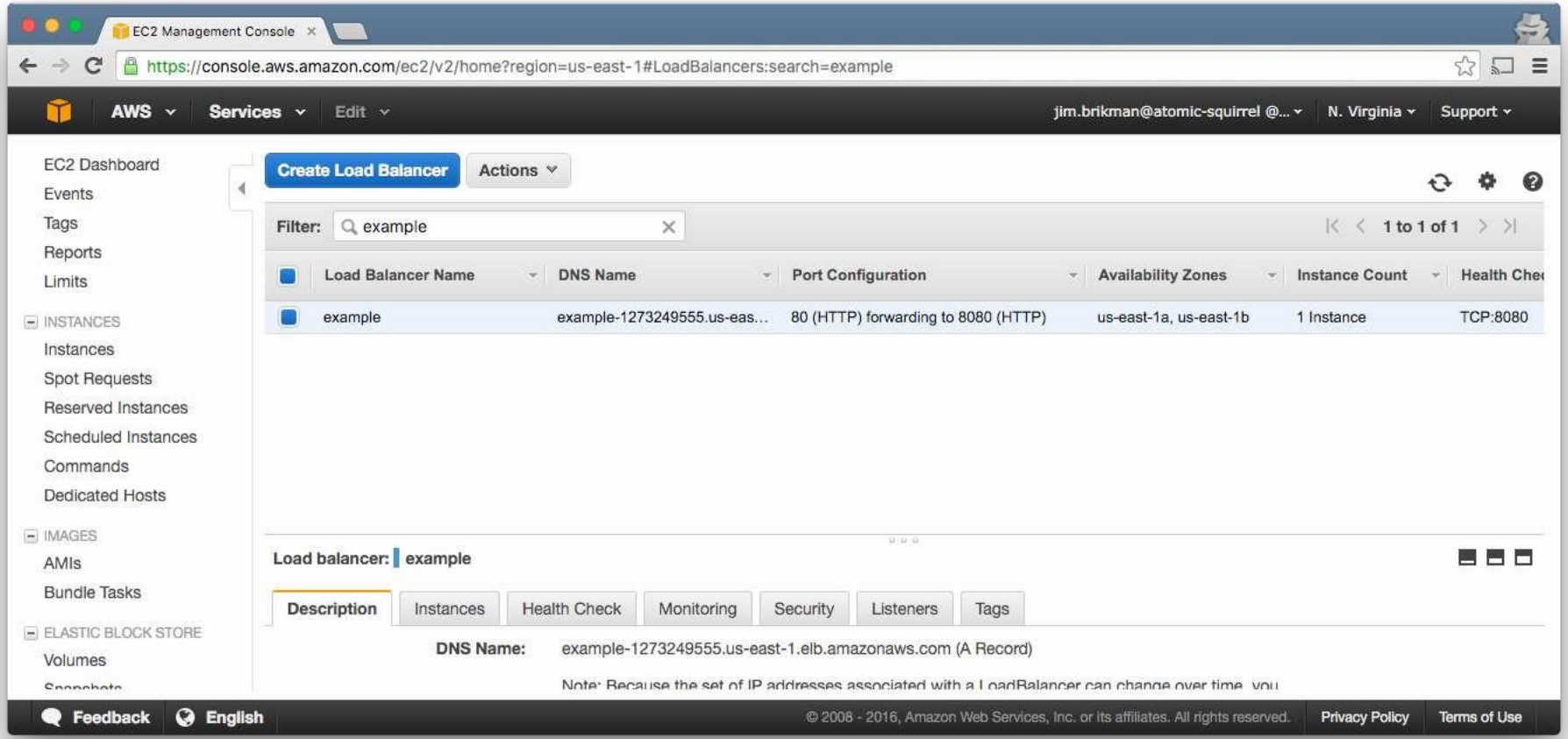
**Terraform supports variables, such as var.instance_port**

```
resource "aws_elb" "example" {
    name = "example"
    availability_zones = ["us-east-1a", "us-east-1b"]
    instances=["${aws_instance.example.id}"]
    listener {
        lb_port = 80
        lb_protocol = "http"
        instance_port = "${var.instance_port}"
        instance_protocol = "http"
    }
```

**As well as dependencies like aws_instance.example.id**

```
resource "aws_elb" "example" {
    name = "example"
    availability_zones = ["us-east-1a", "us-east-1b"]
    instances=["${aws_instance.example.id}"]
    listener {
        lb_port = 80
        lb_protocol = "http"
        instance_port = "${var.instance_port}"
        instance_protocol = "http"
    }
```

**It builds a dependency graph and applies it in parallel.**




# After running apply, we have an ELB!

```
> terraform destroy
aws_instance.example: Refreshing state... (ID: i-f3d58c70)
aws_elb.example: Refreshing state... (ID: example)
aws_elb.example: Destroying...
aws_elb.example: Destruction complete
aws_instance.example: Destroying...
aws_instance.example: Destruction complete

Apply complete! Resources: 0 added, 0 changed, 2 destroyed.
```

# Use the destroy command to delete all your resources

# Advantages of Terraform:

1. Concise, readable syntax
2. Reusable code: inputs, outputs, modules
3. Plan command!
4. Cloud agnostic
5. Very active development

# Disadvantages of Terraform:

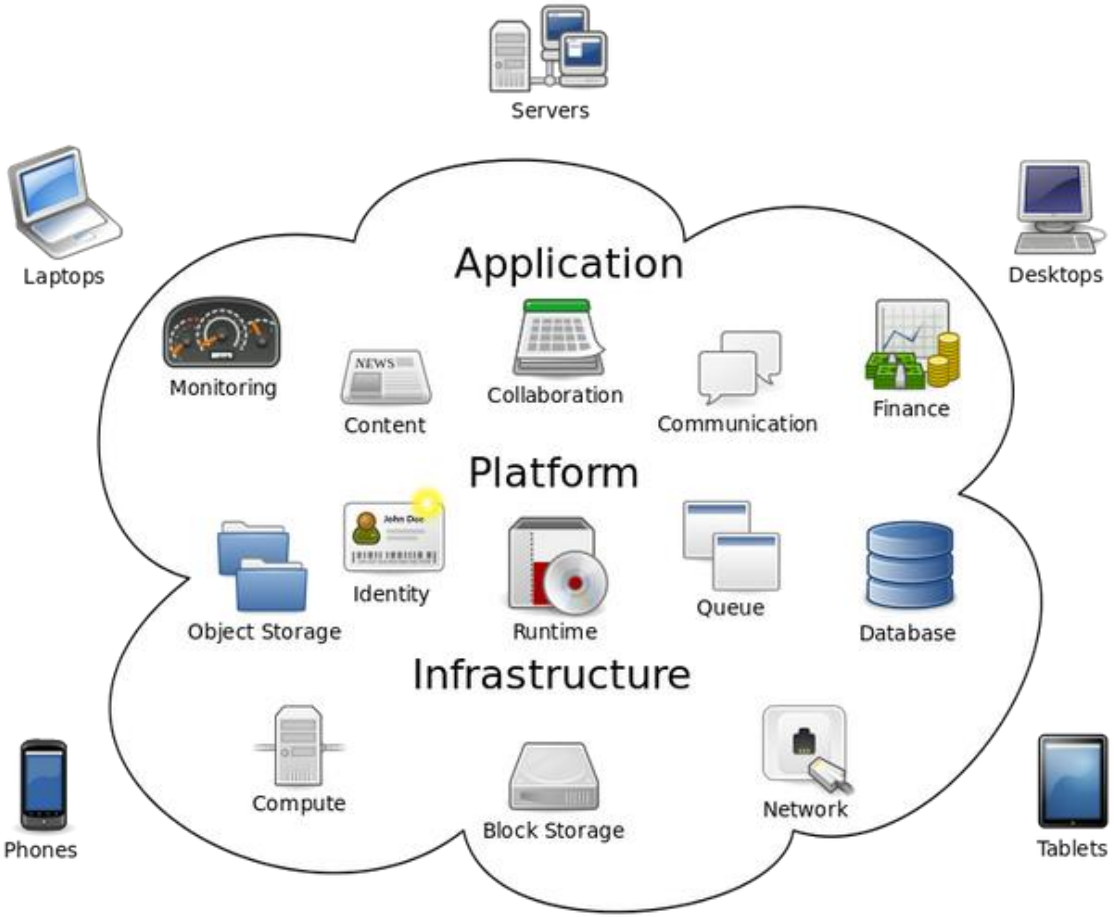
1. Maturity
2. Collaboration on Terraform state is hard (but terragrunt makes it easier)
3. No rollback
4. Poor secrets management

# Outline

1. Microservices
2. Docker
3. Terraform
4. Recap

## Benefits of infrastructure-as-code:

1. Reuse
2. Automation
3. Version control
4. Code review
5. Testing
6. Documentation




# Cloud computing


Questions?