Building resilient microservices with Kubernetes, Docker, and Envoy

Phil Lombardi, Rafael Schloming, Richard Li, and Ed Rousseau

DATAWIRE

Before we begin ...

- You're running latest version of Docker on your laptop
- You've created an account on hub.docker.com
- You've downloaded some key Docker images
  - docker pull datawire/ambassador-envoy
  - docker pull prom/prometheus
  - docker pull python:3-alpine
- You've set up a working environment (we've pre-installed everything for you)
  - Ubuntu Docker image
  - git clone https://github.com/datawire/shopbox
  - cd shopbox
  - ./shopbox.sh
- You have your favorite terminal & text editor ready to go!

Go to the presentation here: https://d6e.co/2xQyXzN

About us

- Been working with microservices for past 2.5 years, both with our own cloud services and with consulting
- We build open source tools for developers building microservices, based on our own experiences
- Host microservices.com, which has talks from dozens of practitioners on microservices

Our schedule for today

Introduction: Microservices, Kubernetes, Docker, Envoy	20 minutes	Presentation
Core Concepts of Docker & Kubernetes	70 minutes	Workshop
Break around 3pm	30 minutes	
Building & deploying a microservice in production	60 minutes	Workshop
Wrap-up	20 minutes	Presentation / Q&A
Feedback	5 minutes	

Our focus today

1. Trying to communicate the key concepts
2. Minimize time spent on the mechanics, since they're impossible to remember until you do them often enough
3. Try to give you a mental model to understand the different (overwhelming) choices in the Kubernetes ecosystem, and how to start.
4. If you don't understand the purpose of a specific exercise, please ask!

Also …

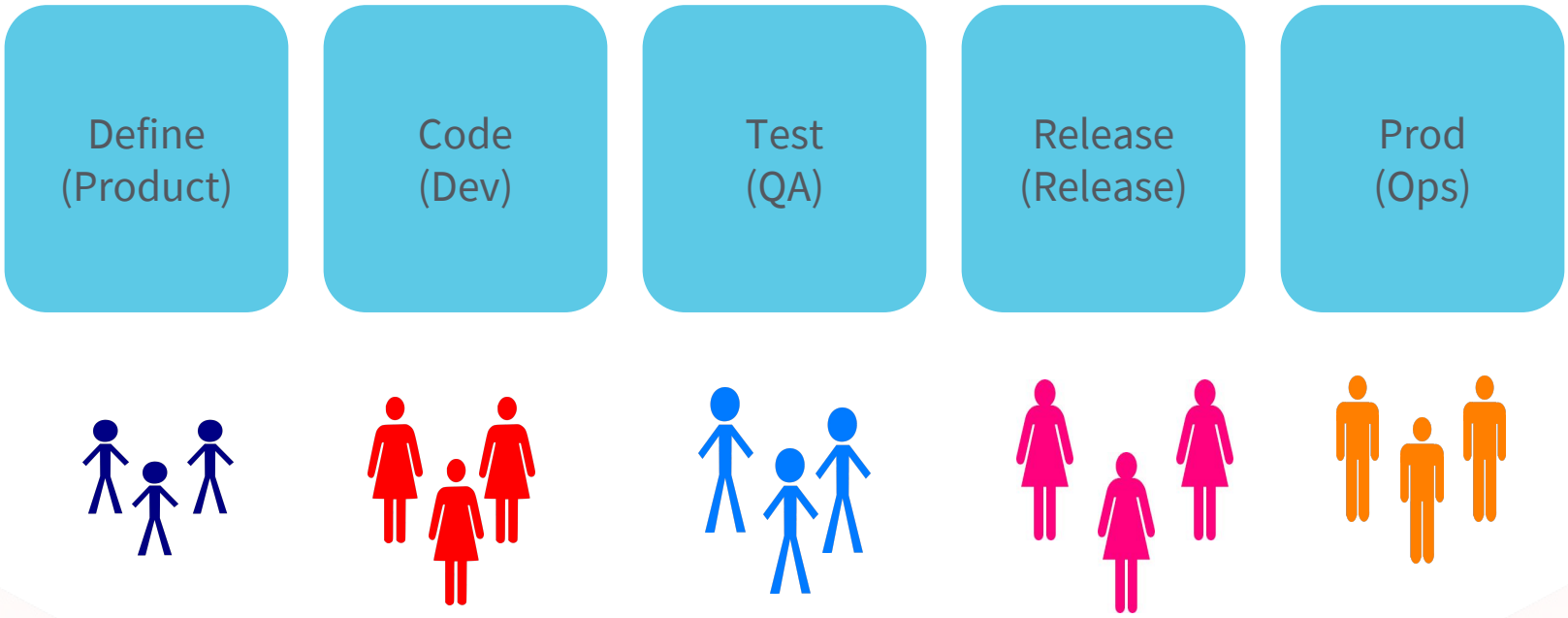
Minikube is a popular choice for these trainings, since you can run things locally. But minikube is a big performance hog and isn't quite as realistic with some of the things we're doing, so we're going to try to use the Internet. (We have done some work to minimize bandwidth consumption.)

Microservices


How do you ship better (cloud) software faster?

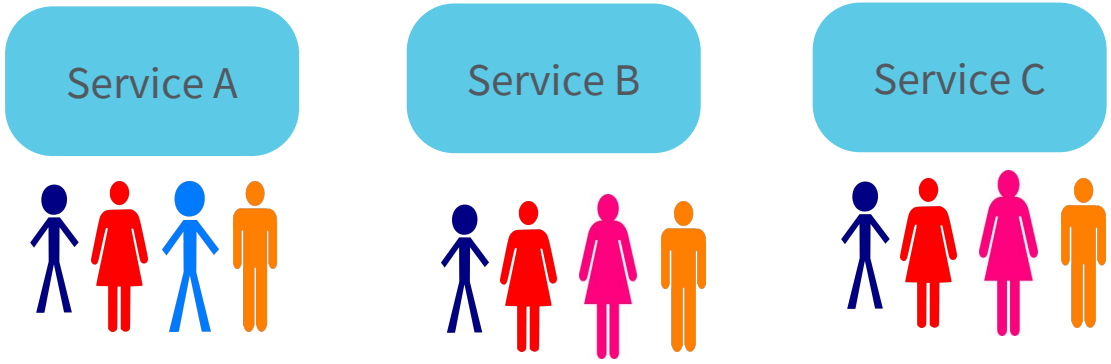
Microservices. (Multiple, asynchronous launches.)

Take the standard development process:



… and make it distributed.

> No central release, test, dev cycle. Each person/team operates an **independent** development process.
> Team needs to have the skills / knowledge to operate all aspects of the development process.

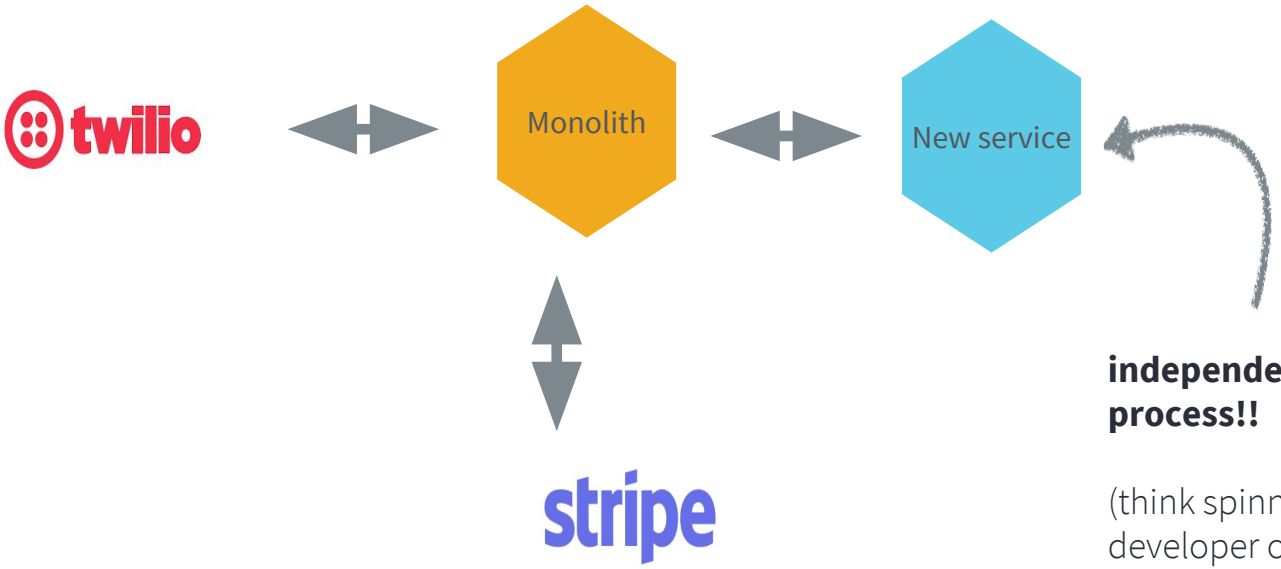


Microservices is a **distributed development process** for cloud-native software.

> **Not an architecture!** Your architecture supports the process, and not the other way around.

How do you adopt a distributed development process?

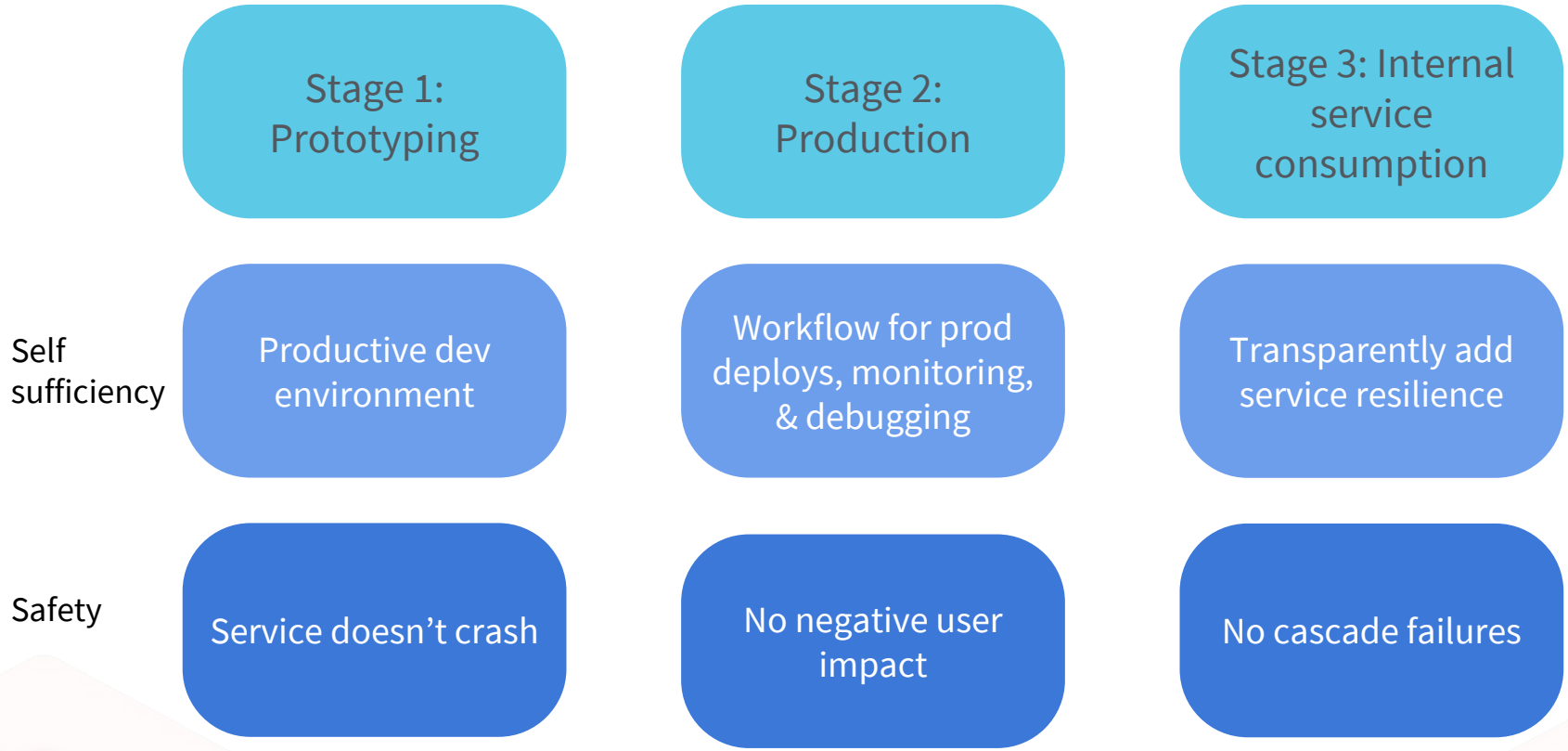
Start by creating an independent process! (with a from-scratch API service)



Give the developer / dev team the ability to SUCCESSFULLY operate independently.

1. **Self-sufficiency**. Each team needs to be self-sufficient in all aspects of the development cycle to avoid bottlenecks.
2. **Safety**. Since it's hard to be an expert in every area of the development, need to insure an oops isn't catastrophic.

The definition of self-sufficient safety varies based on the evolution of your microservice.



Together, Kubernetes, Docker, and Envoy give your developers the basic infrastructure for self-sufficiency & safety.

1. Microservices is a **distributed development workflow** that helps you go faster.
2. An efficient workflow for your team provides **self-sufficiency** and **safety.**
3. The **Kubernetes ecosystem, Docker, and Envoy** provide the foundational components you need to build that workflow.

Docker, Kubernetes, and Envoy

DATAWIRE

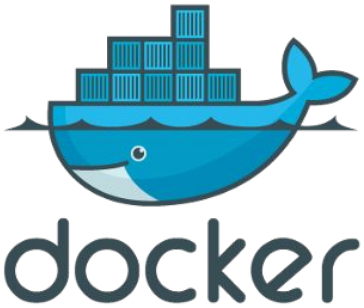
Core Concept 1: Containers

DATAWIRE

What is a container?

- Lightweight Linux environment. It is a form of virtualization… but very different from a full virtual machine.
- Immutable, deployable artifact.
- Runnable.
- Popularized by Docker but there are many runtime implementations (e.g. LXC, Rkt).

What is Docker?



- A tool, ecosystem and platform for building, pushing and running containers.
- The most popular container runtime currently.
- Default container runtime in Kubernetes.

Why Containers?

- Easy and fast to produce.
- Great way to isolate different components in a complex system.
- Ensures a reproducible runtime for your app along the dev -> build -> test -> prod pipeline.
- Easy to share in a team or with external partners.

Your Development Environment

Let's Get Started...

- We've built an Ubuntu container image for you
  - Includes all the client-side tools we'll use for the training (e.g., kubectl)
- We'll use Kubernaut for Kubernetes clusters
  - On-demand, ephemeral clusters (designed for CI … or training!)
- The container mounts a local directory into /workspace in the image your files are synchronized with your laptop.

Let's Get Started...

Run the below commands in your terminal if you haven't already:

```
$ git clone https://github.com/datawire/shopbox

$ cd shopbox

$ ./shopbox.sh
```

Pre-configured dev environment with kubectl (with tab completion), kubernaut, and various utilities we will use today.

Back to containers ...

Let’s build a service as a container

A simple web application: Quote of The Moment (“QOTM”).

- Requires Python
- Uses Flask

GET STARTED

```
$ git clone https://github.com/datawire/qotm

$ cd qotm
```

The Dockerfile

```
FROM python:3-alpine
MAINTAINER Datawire <dev@datawire.io>
LABEL PROJECT_REPO_URL         = "git@github.com:datawire/qotm.git" \
      PROJECT_REPO_BROWSER_URL = "https://github.com/datawire/qotm" \
      DESCRIPTION              = "(Obscure) Quote of the Moment!" \
      VENDOR                   = "Datawire, Inc." \
      VENDOR_URL               = "https://datawire.io/"

WORKDIR /service
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . ./
EXPOSE 5000
ENTRYPOINT ["python3", "qotm/qotm.py"]
```

Let's build it!

Run the below command to build the image (the trailing period on the below command is required!):

```
$ docker build -t <your-docker-user>/qotm:1 .
```

Each Docker image consists of **layers**. A layer is an ordered union of files and directory changes to an image.

Because layers are cached, putting the parts of the Dockerfile least likely to change first (e.g., the OS) can make a huge difference in build speed.

What Just Happened?

1. Docker executed the instructions in the Dockerfile. Each command created a new layer.

2. Docker composes an image from all the layers.

3. The docker engine pointed a named reference **<your-docker-user>/qotm:1** at the final image.

Tagging is Important and Useful

- Tags allow you to easily reference and reuse an image.

- You can create multiple tags to point at the same image which can be useful in sharing contexts.

- Tags can be pushed to a Docker registry so other people can reuse your image!

Run the image!

Images are an inert, immutable file. When you want to run an image, a container is produced.

RUN THE IMAGE

```
$ docker run --rm -dit -p 5000:5000 <your-docker-user>/qotm:1

# open another shell (that is *not* running shopbox)

$ curl localhost:5000
```

Share the image

We've got an image running on your laptop, but you really want to share it -- with Kubernetes, with your colleagues, with your family & friends ...

PUSH THE IMAGE

```
$ docker login
```

```
$ docker push <your-docker-user>/qotm:1
```

You can see the image on your public Docker Hub account at https://hub.docker.com.

One last thing ...

Docker does a great job of running the container. Why can't I just use Docker everywhere?

WHAT HAPPENS IF WE CRASH?

```
$ curl localhost:5000/crash

$ curl localhost:5000
```

Uh oh … we need something that is a little bit smarter!

Core Concept 2: Kubernetes

DATAWIRE

What is Kubernetes?

- Runs massive numbers of containers based on lessons learned by Google.

- Schedules and runs all kinds of containers
  - long-lived (e.g. services)
  - short-lived (e.g. pre-launch hooks, cronjobs etc)

- Kubernetes can be thought of as a Distributed OS or process manager

The Office Tower Analogy

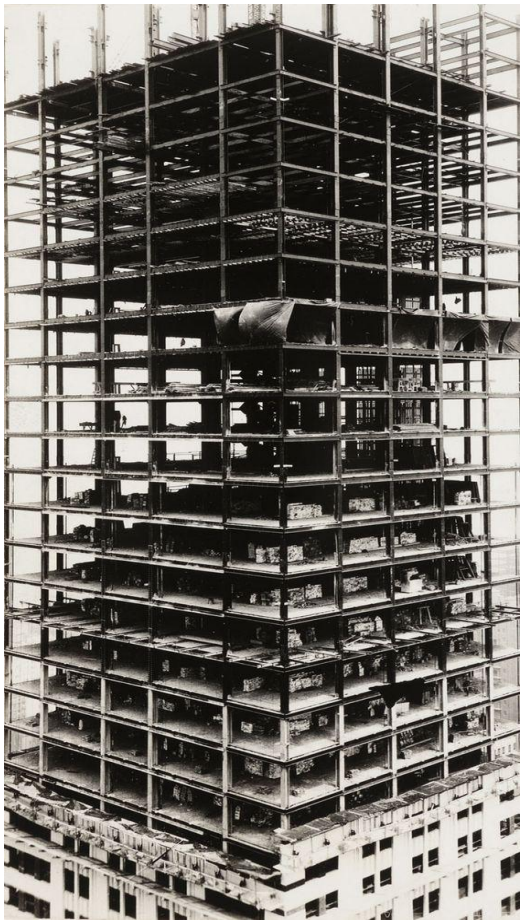
Your product is the building as a whole.

Your business logic is

the offices and workers

The Office Tower Analogy

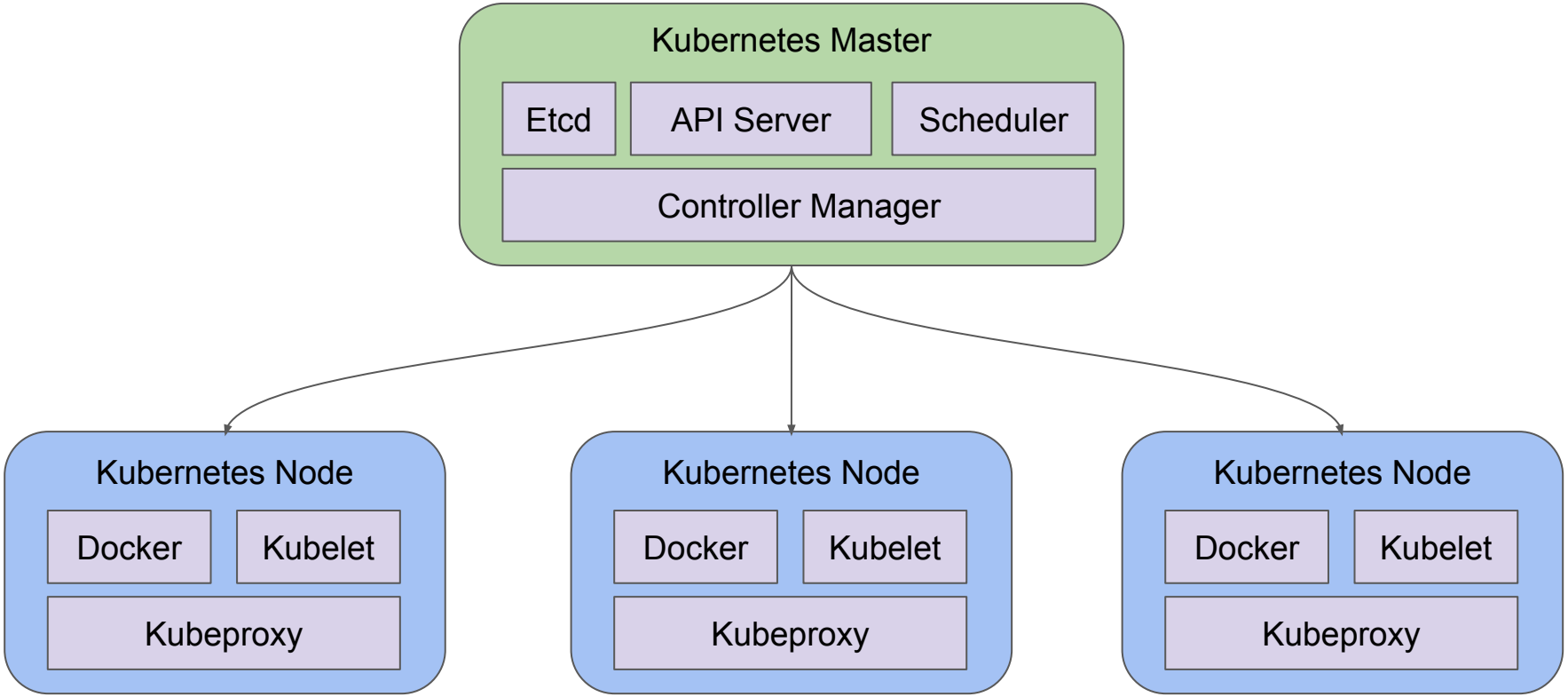
Kubernetes provides the infrastructure to build your app around.



It is the foundational app platform for your team to build your businesses apps around.

Kubernetes Architecture

Types of nodes: **Masters** and **Workers**



Container packages your code in a portable way

Pod gives your code a temporary home *inside* the cluster

Deployment keeps your code running, even when it is updated

Service provides a stable address that can reach many pods

4 basic concepts

Your Development Environment

Kubernaut

- You can get your own Kubernetes cluster easily with Google, Microsoft etc.
- Or you can install Kubernetes yourself in AWS
- To simplify things, we're going to let you borrow some of our Kubernetes clusters :)
- We're going to use Kubernaut which provides on-demand K8S clusters for our internal CI/CD systems.

Kubernaut

Visit https://kubernaut.io/token

```
$ kubernaut set-token <TOKEN>

$ kubernaut claim

$ export KUBECONFIG=${HOME}/.kube/kubernaut
```

Back to Kubernetes

Let’s see if there’s anything running!

```
$ kubectl get services

$ kubectl get pods
```

Let’s run our container

```
$ kubectl run qotm --image=<your-docker-user>/qotm:1

$ kubectl get pods
```

We see a pod! How do we talk to the pod?

Pod gives your code a temporary home *inside* the cluster

We need to talk to the pod!

We can tell Kubernetes to forward requests from outside the cluster to the pod, and vice versa.

```
$ kubectl port-forward <pod-name> 5000 &

$ curl localhost:5000
```

What happens when a pod crashes?

Let’s crash the pod again.

```
$ curl localhost:5000/crash
```

```
$ curl localhost:5000
```

Note: don’t run this loop too often. If you crash your server too frequently, Kubernetes will assume it is having deeper problems, and will introduce a delay before attempting to restart.

What just happened?

The Kubernetes pod automatically restarted the container!

- By default, Kubernetes will detect failures and auto-restart (with exponential backoff, capped at 5 minutes)

- Kubernetes also lets you extend this with custom liveness and readiness probes

Managing pods

- What if we want to update the software on our pod?
- What if we want more than one pod running, for availability or scalability reasons?

Deployment keeps your code running, even when it is updated

Deployments

Kubernetes provides extensive control over how pods are run. These are specified by a deployment object.

22 lines (21 sloc) | 365 Bytes — Raw | Blame | History

```yaml
---
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: qotm
spec:
  replicas: 2
  strategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: qotm
    spec:
      containers:
      - name: qotm
        image: datawire/qotm:1.2
        imagePullPolicy: Always
        ports:
        - name: http-api
          containerPort: 5000
```

Let's try using a deployment

To save bandwidth, this deployment.yaml points to a prebuilt QOTM image we've already uploaded. Feel free to edit it to point to your Docker repo.

```
$ kubectl get pods

$ kubectl delete deployment qotm

$ kubectl apply -f kubernetes/qotm-deployment.yaml

$ kubectl get pods
```

We see we're now running three pods!

How do we talk to these pods?

It would be silly to set up port-forwards to each pod … and load balancing would be nice.

Service provides a stable address that can reach many pods

Service

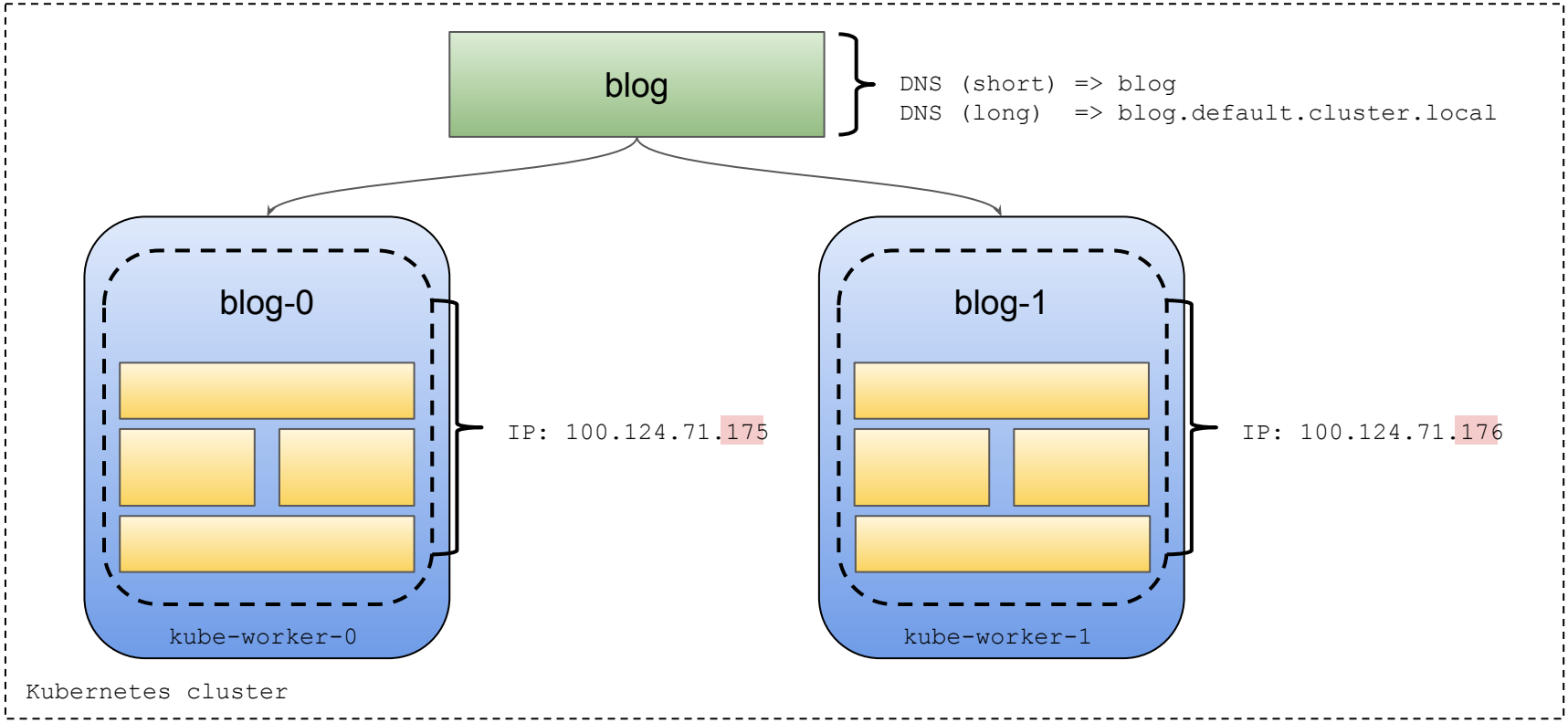
Kubernetes provides extensive control over how pods are run. These are specified in a service file.

13 lines (12 sloc) | 152 Bytes

```yaml
---
apiVersion: v1
kind: Service
metadata:
  name: qotm
spec:
  type: NodePort
  selector:
    app: qotm
  ports:
  - port: 80
    targetPort: http-api
```

Services Illustrated

A Service becomes a DNS A record pointing the pod IP addresses



Service Flavors

- Many different flavors of “Service” in Kubernetes
  - ClusterIP
  - NodePort
  - LoadBalancer
  - ExternalName - often forgotten, but very useful!

Let’s try using a service

```
$ kubectl apply -f kubernetes/qotm-service.yaml

$ kubectl get services # get qotm port highlighted below

NAME         CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
kubernetes   10.96.0.1        <none>        443/TCP        2d
qotm         10.107.109.252   <nodes>       80:<port>/TCP   6s

$ kubectl cluster-info # get cluster hostname

$ curl http://<cluster-hostname>:<port>

# or use this script for convenience:

$ curl $(url.sh qotm)
```

Recap: The source -> Kubernetes workflow

A. Build a container image that contains your code, dependencies, and configuration, based on the Dockerfile.

B. Tag the image.

C. Push image to a container registry.

D. Update Kubernetes manifest with tag.

E. Apply Kubernetes manifest to cluster.

F. Repeat for all dependencies.

Container packages your code in a portable way

Pod gives your code a temporary home *inside* the cluster

Deployment keeps your code running, even when it is updated

Service provides a stable address that can reach many pods

Let’s take a break!
(hopefully it’s around 3pm)

DATAWIRE

Core Concept 3: Envoy / L7

DATAWIRE

The definition of self-sufficient safety varies based on the evolution of your microservice.

Stage 1: Prototyping	Stage 2: Production	Stage 3: Internal service consumption
Productive dev environment	Workflow for prod deploys, monitoring, & debugging	Transparently add service resilience
Service doesn't crash	No negative user impact	No cascade failures

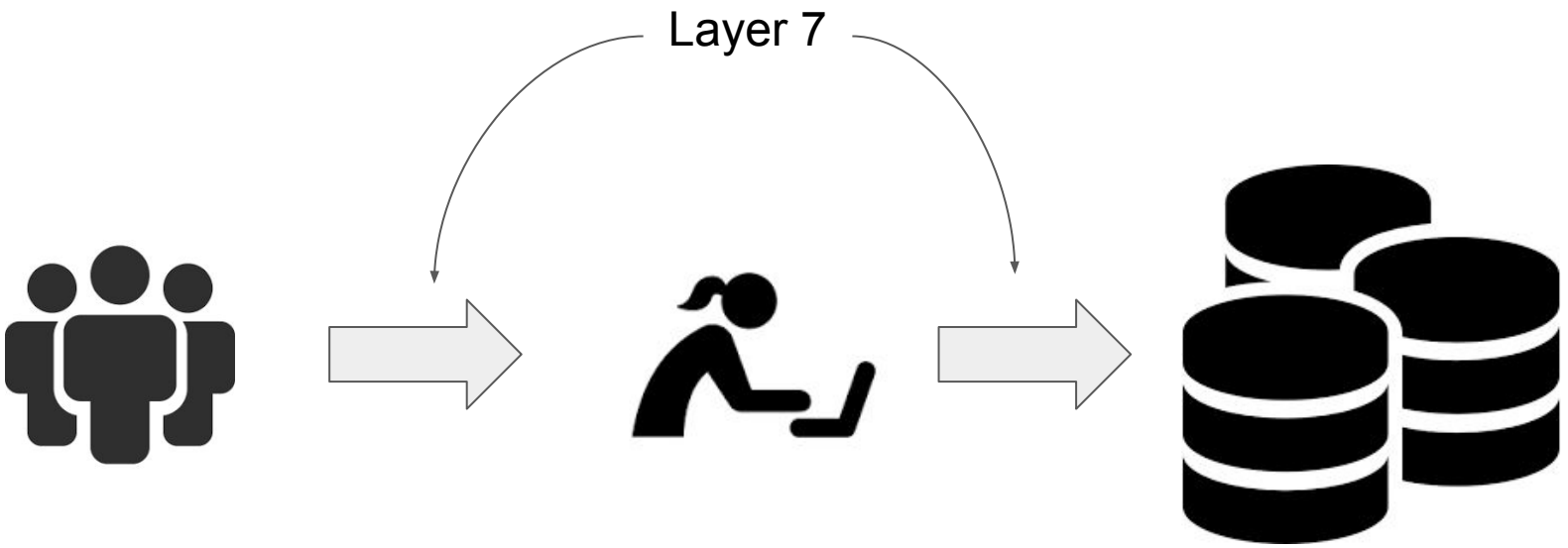
Measuring user impact is a L7 problem!

- What is L7?
  - We really mean application-level protocols
  - HTTP, gRPC, Thrift, redis://, pgsql://, ...
- In a microservices architecture, your services are an L7 network
- For you to write a service that talks to your users and/or other services, you need to understand & manage L7

L7 is now a **development** concern

- Everything has always been wired together with L7
- But in development, you could leave it (mostly) to operations
- Now, with microservices, L7 is a **development** concern as well:
  - More services
  - More remote dependencies
  - Greater release frequency
- So what does this mean for you?

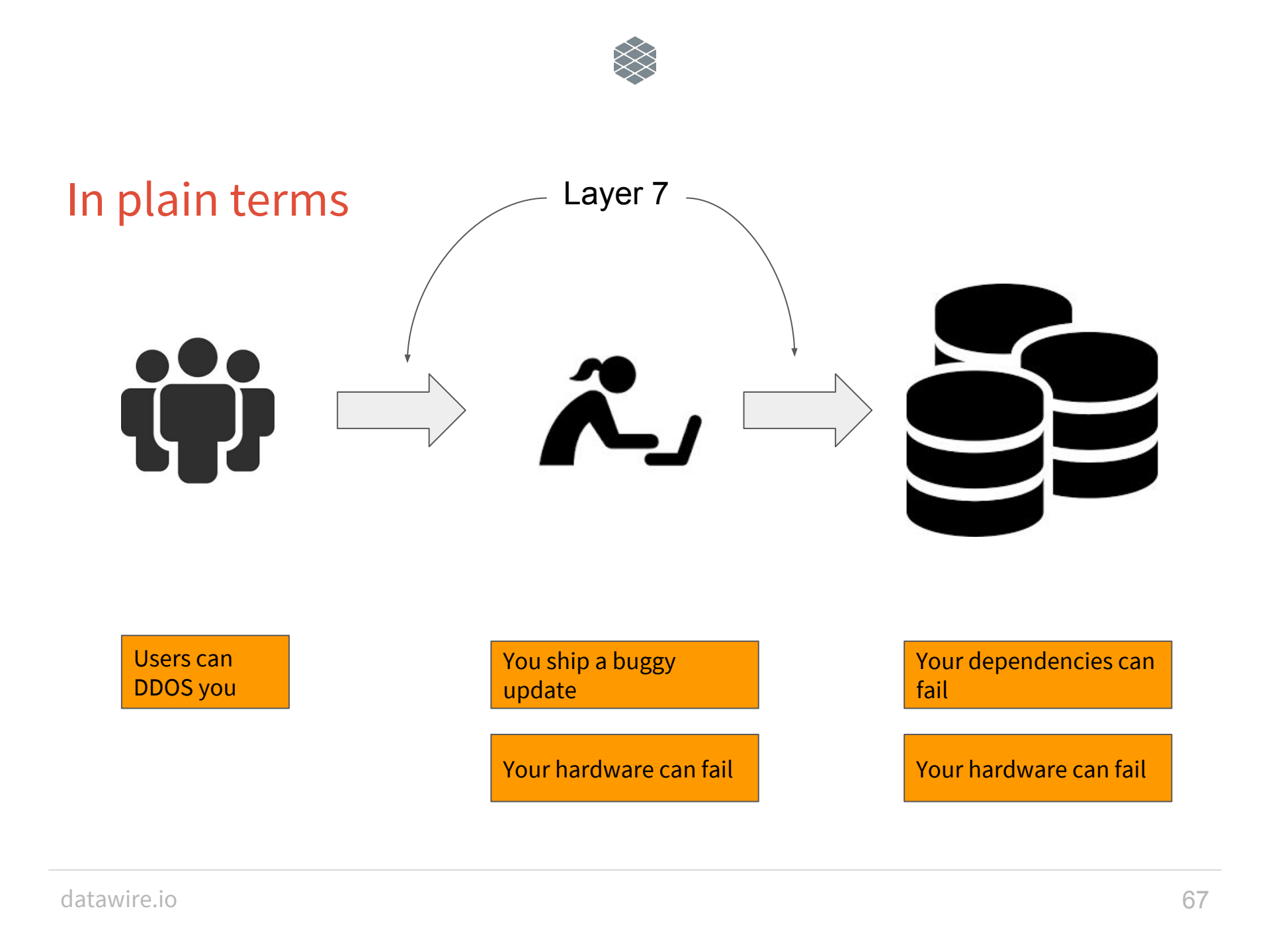
You: stuck in the middle



- Users consume your service over L7
- You consume your dependencies over L7
- Literally everything in this picture is a source of failure

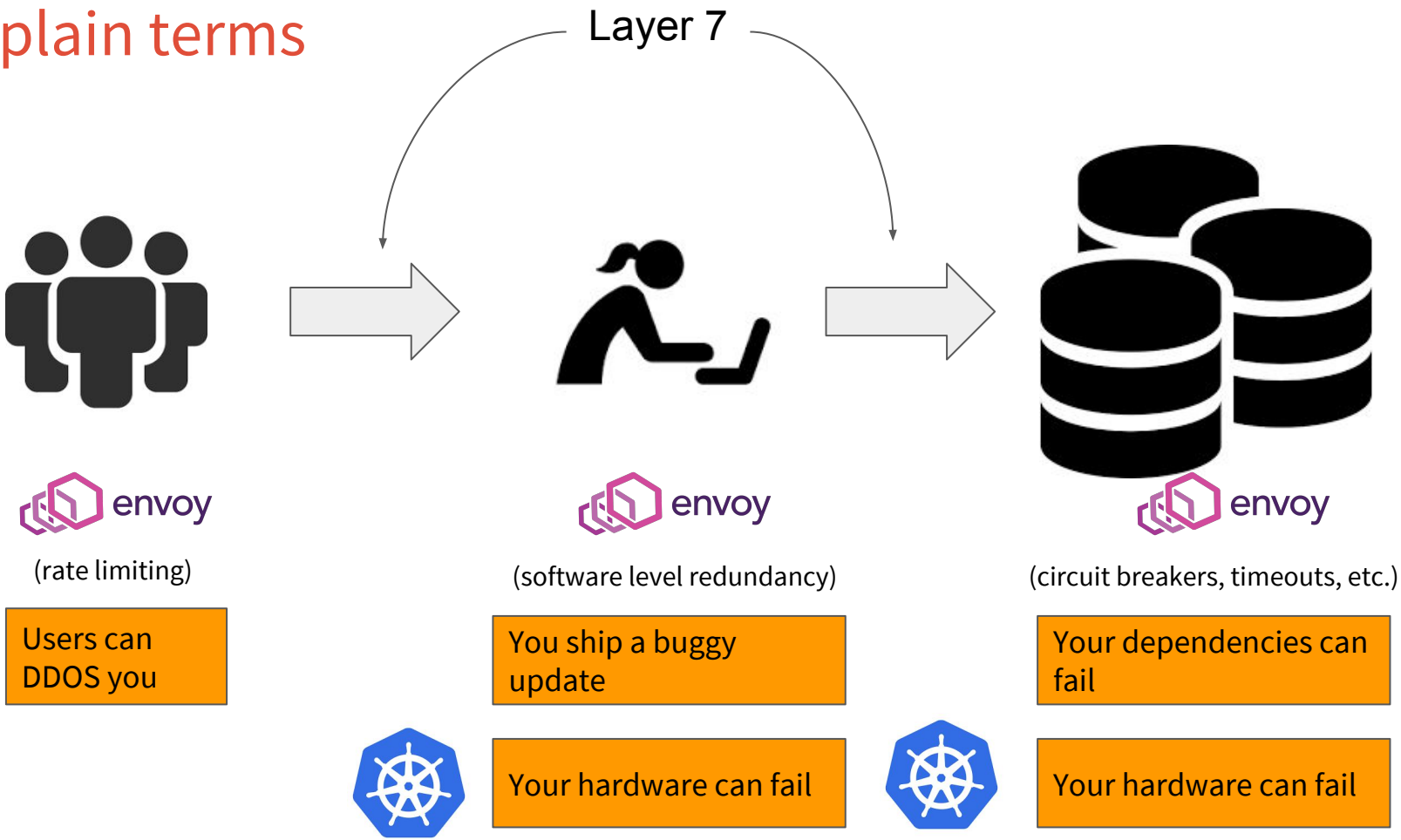
All your distributed systems problems are amplified

- Single points of failure
- Catastrophic failure modes
- Cascade failures



In plain terms

In plain terms



How do we protect us from ourselves?

- If all our redundant hardware runs the same code, our own bugs quickly become the biggest source of catastrophic failure

Create **software** level redundancy

- Redundant hardware protects us from mechanical failures
- We need redundant software implementations to protect us from our own bugs
- Canary testing is the most basic version of this
  - run multiple versions of your code to improve resiliency (like genetic diversity)

Envoy helps us do this as well, but we need to wire it into our developer workflow

- This is what we will focus on

Envoy

- Modern, L7 proxy, designed for distributed cloud architectures
  - L7 observability
  - Resilience
    - Global rate limiting
    - Advanced load balancing
    - Circuit breakers
  - HTTP/2 & gRPC support
  - APIs for managing fleets of Envoys
- Adopted by the CNCF (which also hosts Kubernetes, Prometheus, Docker, among other projects)
- Originally written by the engineering team at Lyft, and now with committers from Google, IBM, Datawire, and others
- Alternatives: NGINX Plus, HAProxy

First, some setup

- We're going to deploy multiple times from source to prod
- This means
  - Your deployment file needs to be templated, so you can point to multiple different versions in production
  - You don't want to manually run the docker build, etc commands by hand
- So we're going to use a simple open source tool, Forge, to automate this process

- We've already set up the QOTM service to support Forge, which means:
  - A service.yaml file that specifies the service name
  - A Jinja2-templated Kubernetes manifest
- So run **`forge setup`** from your ***workspace*** directory to configure Forge for your environment

Now, we can automatically deploy

```
# let’s delete the original services

$ kubectl delete svc qotm
$ kubectl delete deploy qotm

# deploy from source to cluster, using templated k8s manifest

$ forge deploy
```

Canary testing

- Route X% of your traffic to new version
- Monitor your metrics to make sure no difference between old version and new version
- Gradually ramp up traffic to new version

Benefits

- Immediate rollback to old version
- Minimize impact of any error

Costs

- Need extra capacity for canary testing
- Need a L7 router (you can only do coarse canaries with K8S)

Let’s deploy Envoy!

At the top-level workspace directory (adjacent to your existing QOTM service)

```
$ git clone https://github.com/datawire/envoy-canary.git
$ forge deploy
$ API=$(url.sh api)
$ curl $API/qotm/ # don’t forget trailing slash
```

What did we just do?

- We deployed:
  - Envoy, pre-configured to collect stats, and work with Kubernetes
  - A (stub) auth service, which Envoy calls on a per request basis
- We are now routing to your QOTM through Envoy
  - QOTM we made a ClusterIP service instead of a NodePort

- If we are having bandwidth problems, we can manually deploy the following images:
  - datawire/auth-training:1
  - datawire/envoy-training:1

Let’s set up monitoring with Prometheus

```
$ git clone https://github.com/datawire/prometheus-canary
$ forge deploy
$ url.sh prometheus
```

In your browser, visit http://<prometheus-url>.

Now, let's deploy a canary

In `qotm/qotm.py`, add a `time.sleep(0.5)` to the `statement()` function.

```
$ CANARY=true forge deploy
$ while true; do curl $API/qotm/; done
```

Monitor the canary

In Prometheus, execute this query:

```
{__name__=~"envoy_cluster_qotm_upstream_rq_time_timer|envoy_cluster_qotm_canary_upstream_rq_time_timer"}
```

Hit execute periodically to see changes (you might also want to reduce the granularity of the time window to 5 minutes).

Recap

The story so far ...

1. Adopting a fast, distributed workflow is critical to accelerating productivity.
2. Start building your workflow by thinking about the single developer/team, for a single service.
3. We showed how Kubernetes, Docker, Envoy, and monitoring (e.g., Prometheus) can be used to build your workflow.
4. Your workflow depends on the stage of your service.
5. Managing L7 is really important, and gives you new, critical capabilities such as canary testing and transparent monitoring.

Recapping the workflow

Some additional topics

- Service meshes
- Development environments
- Stateful services
- Organizational adoption

Service mesh

Stage 1: Prototyping	Stage 2: Production	Stage 3: Internal service consumption
Productive dev environment	Workflow for prod deploys, monitoring, & debugging	Transparently add service resilience
Service doesn't crash	No negative user impact	No cascade failures

Service meshes

- When you have stage 3 services, you want to think about a service mesh
  - But you should start with 1 service, so don't worry about the service mesh right away!
- Provide two critical functions
  - Observability (e.g., tracing) across all of your services
  - Resilience across all of your services
- Function by deploying a sidecar proxy (e.g., Envoy) next to each of your services
- Use iptables or equivalent to insure all service **egress** traffic is routed through sidecar
- Sidecar adds in trace IDs, circuit breaking, etc.

Development Environments for Microservices

	Run entire system locally	Run business logic locally, cloud resources remote	Single service local, all other services remote	All remote development
Realism: How closely does this mirror production?	●	●●	●●●	●●●●
Fast feedback cycle for developers	●●●●	●●●	●●●	●
Low setup and maintenance cost for developers	●	●●	●●●	●●●●
Scalability as your application gets more complex	●	●●	●●●●	●●●●

DATAWIRE

Stateful services

- Databases and such can be deployed with a Kubernetes manifest -- same technique as Envoy or the existing services, but a different configuration
- Standard canary testing doesn't work as well for stateful services
  - Envoy supports shadowing of requests
  - (We're working on this so it's more useable)
- If you have non-K8S resources (e.g., AWS RDS, etc.) consider adding the Terraform/Ansible/etc. Scripts for creating these resources in another folder as part of your standard service

Organizational adoption

- Build an API service, just like Stripe or Twilio
- Staff with a single, spinoff team
- Define the purpose of the service from the perspective of the user
- Don’t allow the service team to make any changes to the existing code base, or vice versa

Thank you!

- (Anonymous) feedback survey -- would be VERY grateful if you could spend 5 minutes filling it out so we can get better
  - https://d6e.co/2yxVnmn
- Feel free to email us:
  - richard@datawire.io
  - rhs@datawire.io
  - plombardi@datawire.io
- If you're interested in any of our open source tools, check them out:
  - https://forge.sh for deployment
  - https://www.telepresence.io for fast development cycles
  - https://www.getambassador.io easiest way to deploy/configure Envoy on Kubernetes

appendix

DATAWIRE

Testing microservices

- Traditional approach to testing a microservices architecture is with a staging environment
  - First push services to staging
  - Then run integration tests
  - If tests pass, then push to production
- But this introduces a big tradeoff
  - In order to do a “true” integration testing, you need to synchronize your different service versions in staging … and push those versions into production
  - But this introduces a bottleneck!
- So you want to think about more distributed strategies for integration testing