

Webinar

Kubernetes Anti-patterns

Kostis Kapelonis

**An expert is a person who has made all the
mistakes that can be made in a very narrow
field**

Niels Bohr

Background, Problems and Solutions



Kostis Kapelonis

Now Developer advocate at Codefresh
Interests: Kubernetes, CI, CD, GitOps

Ex Java dev (10+ years)
Ex Release manager (5+ years)
Manning author (Java testing with Spock)



Original blog

A collection of all “questionable” practices I have seen companies using without understanding the alternatives

- <https://codefresh.io/kubernetes-tutorial/kubernetes-antipatterns-1/>
- <https://codefresh.io/kubernetes-tutorial/kubernetes-antipatterns-2/>
- <https://codefresh.io/kubernetes-tutorial/kubernetes-antipatterns-3/>

Also published in Medium and dev.to

HOME > BLOG



CONTINUOUS DEPLOYMENT/DELIVERY

Kubernetes Deployment Antipatterns – part 1

12 min read



Kostis Kapelonis · Jan 20, 2021

In our previous guide, we documented [10 Docker anti-patterns](#). This guide has been very popular as it can help you in your first steps with container images. Creating container images for your application, however, is only half the story. You still need a way to deploy these containers in production, and the de facto solution for doing this is by using Kubernetes clusters.

The editorial

Articles, announcements, and more that give you a high-level overview of challenges and features.

[Datadog and the Container Report, with Michael Gerstenhaber](#)

Craig Box, Kubernetes Podcast from Google

[Kubernetes Deployment Antipatterns — part 1](#)

Kostis Kapelonis

[Kubernetes Pods Advanced Concepts Explained](#)

Regis Wilson, Release

[Discover and invoke services across clusters w](#)

Emeka Nwafor, Product Manager, and Jeremy Cloud



Kubernetes @kubernetesio · 21m
★ @codepipes shares part 3 in a series on #Kubernetes deployment antipatterns



Kubernetes Deployment Antipatterns—part 3
This is the third and last part in our Kubernetes Anti-patterns series. See also part 1 and part 2 for the previous anti-patterns.
[medium.com](#)

7 retweets, 13 likes

JANUARY 2021

Kubernetes Deployment Antipatterns—part 3

10 min read · In Container Hub · View story · Details

4.2K

Kubernetes Deployment Antipatterns—part 2

10 min read · In Container Hub · View story · Details

2.9K

Kubernetes Deployment Antipatterns—part 1

12 min read · In Container Hub · View story · Details

7.2K

DECEMBER 2020

Using Helm to Deploy a Kubernetes Applica...

9 min read · In Container Hub · View story · Details

1.1K

Disclaimer!

What this talk is about

The Kubernetes cluster is already there (and setup correctly)

All advice is for application deployment and not cluster deployment

We are interested in applications and not cluster infrastructure

There are different anti-patterns for how to deploy the cluster itself



Anti-pattern list

1. Using containers with the latest tag in Kubernetes deployments
2. Baking the configuration inside container images
3. Coupling applications with Kubernetes features/services for no reason
4. Mixing application deployment with infrastructure deployment (e.g. having Terraform deploying apps with the Helm provider)
5. Performing ad-hoc deployments with kubectl edit/patch by hand
6. Using Kubectl as a debugging tool
7. Misunderstanding Kubernetes network concepts
8. Using permanent staging environments instead of dynamic environments
9. Mixing production and non-production clusters
10. Deploying without memory and CPU limits
11. Misusing health probes
12. Not using Helm (and not understanding what Helm brings to the table)
13. Not having deployment metrics to understand what the application is doing
14. Not having a secret strategy/treating secrets in an ad-hoc manner
15. Attempting to go all in Kubernetes (even with databases and stateful loads)



Cloud providers
Artifact stores
Unit & integration testing
Endless integrations
Login hell
Vulnerability scans
Containers
Git providers
GitOps
Declarative infrastructure
iOS & Android builds
Parallel builds & deployments
PR Reviews
Containers
Self-hosted
Easy fixes = scalability
Kubernetes
Shared dependencies
Provisioning environments
Microservices = tons of pipelines
Scaling pipeline variations
Monorepos
Blue/green deployments

Anti-pattern 1: Using containers with the latest tag in Kubernetes deployments

More complex deployment patterns

Easy fixes = scalability

Microservices = tons of pipelines

Don't use latest tag

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: my-bad-deployment
5 spec:
6   template:
7     metadata:
8       labels:
9         app: my-badly-deployed-app
10    spec:
11      containers:
12        - name: dont-do-this
13          image: docker.io/myusername/my-app:latest
```



REJECTED



Latest tag does
NOT mean the
most recent or
the last one built

Latest is not a
special tag in
Docker (or
Kubernetes).

It is just the default
tag used if you don't
specify a tag
yourself



sonarsource/sonarcloud-quality-gate ☆

↓ Pulls 500K

By [sonarsource](#) • Updated 8 months ago
Bitbucket Pipelines Pipe: SonarCloud Quality Gate check

Container

Overview **Tags**

Filter Tags

Sort by Newest

TAG

0.1.5.61-QA

Last pushed 8 months ago by [sonardocker](#)

docker pull sonarsource/sonarcloud-qu...

DIGEST

f95a4e58cc2d

OS/ARCH

linux/amd64

COMPRESSED SIZE

45.68 MB

TAG

latest

Last pushed 8 months ago by [sonardocker](#)

docker pull sonarsource/sonarcloud-qu...

DIGEST

fefdf8131284

OS/ARCH

linux/amd64

COMPRESSED SIZE

45.68 MB

Different image

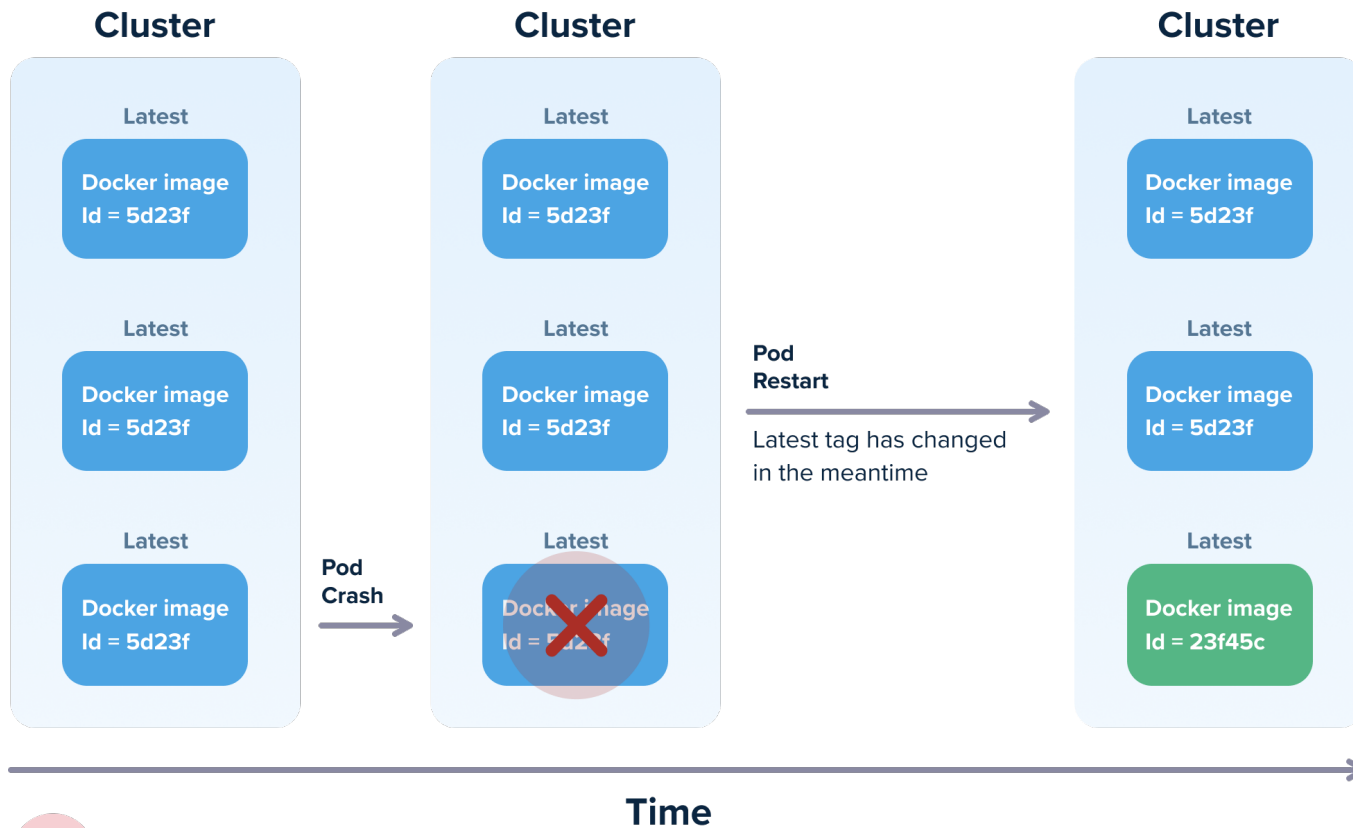
REJECTED

How to detect this anti-pattern

Latest is a transient tag

1. It can be any version of your app
2. You don't really know which application version was deployed
3. Worst case scenario: latest definition changes in the middle of a deployment





⊗ Don't



Solution

Use specific tags in Deployments

Strategy 1 = use the Git hash as a tag

- myapp: ccdd07d
- myapp: a70bfe1
- myapp: 95be785

Strategy 2 = use application version (semver)

- myapp: 0.1
- myapp: 0.2
- myapp: 0.3



Strategy3 : Use date/build number

- myapp: 8789
- myapp: 8790
- myapp: 8791



Gotcha!



<https://unsplash.com/photos/ABNhXfQFtdU>

Big gotcha!

All Docker tags are mutable (!!!)

Tags can be overwritten. So version 0.1 that John has might be different then version 0.1 that Mary has

docker: how to show the diffs between 2 images

Asked 7 years, 4 months ago · Active 6 months ago · Viewed 52k times

▲ 63
▼
16
🕒

I have a Dockerfile with a sequence of RUN instructions that execute "apt-get install"s; for example, a couple of lines:

```
RUN apt-get install -y tree  
RUN apt-get install -y git
```

After having executed "docker build", if I then execute "docker images -a", I see the listing of all the base-child-child-.... images that were created during the build.

I'd like to see a list of all of the packages that were installed when the "apt-get install -y git" line was executed (including the dependent packages that may have also been installed, besides the git packages).

Note: I believe that the "docker diff" command shows the diffs between a container and the image from which it was started. Instead I'd like the diffs between 2 images (of the same lineage): the "tree" and "git" image IDs. Is this possible?

Thanks.

docker

Share Follow

edited Jan 18 '14 at 14:20

The Over

✎ CSS
Stac

✎ Pros
Our

Featured

☐ The
Ope
Ads

☐ Plan
Frid:

☰ Take

Answer

Create

Paid or
Stark i

An all too common scenario

The problems of mutable tags

1. Mary(dev) deploys image with tag 3.7 on QA Kubernetes cluster
2. Alex (QA) tests image with tag 3.7 and finds a bug
3. John (dev) deploys another image with same tag 3.7 (**oops**)
4. Mary can no longer find the bug as image is different than what Alex tested

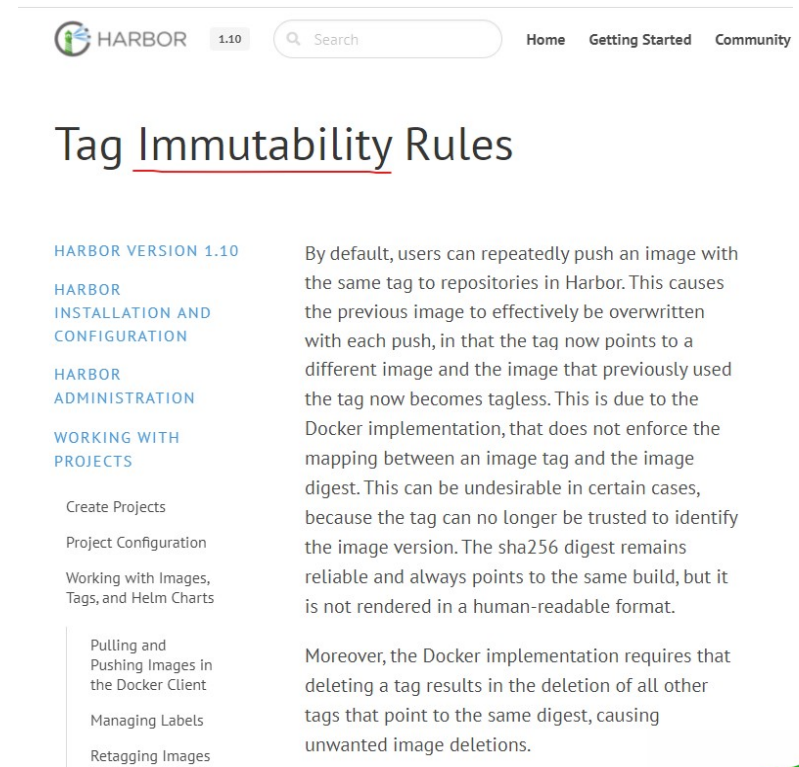


Solution to gotcha

Use immutable tags

Only push container tags ONCE. This way you know exactly what is in each container image

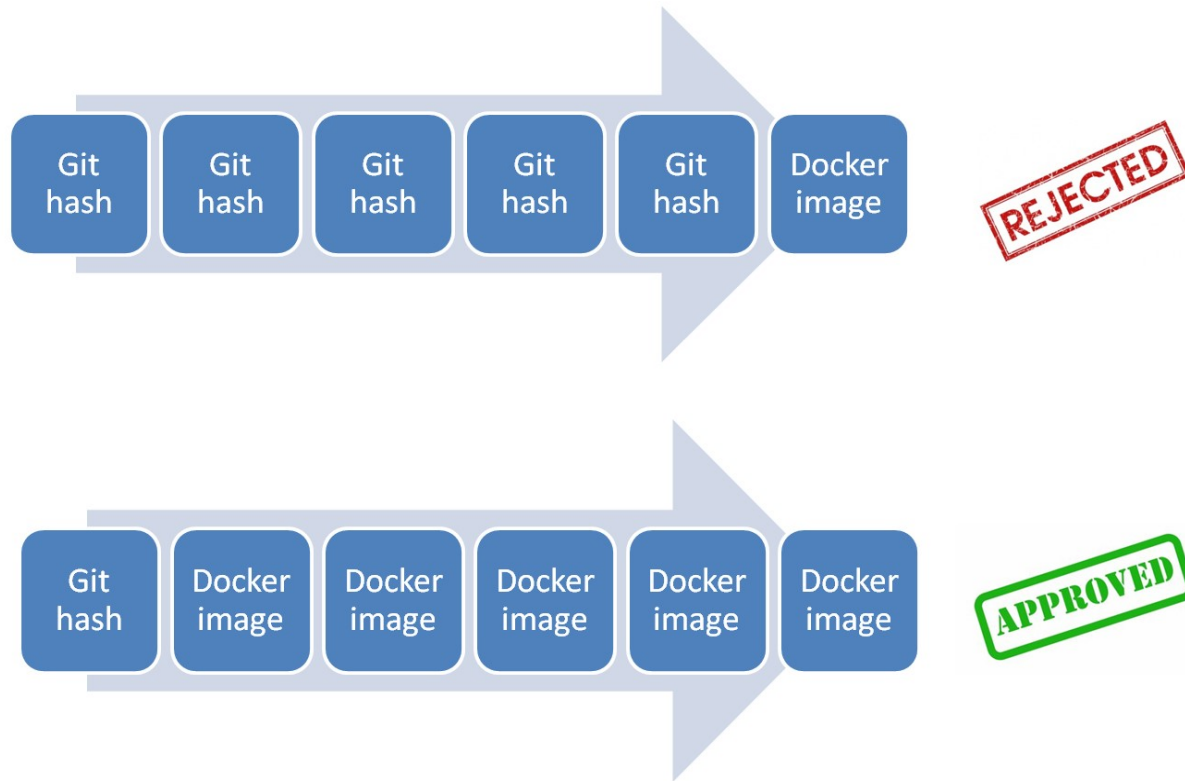
Check your Registry documentation



The screenshot shows the Harbor documentation page for "Tag Immutability Rules". The page header includes the Harbor logo, version "1.10", a search bar, and navigation links for "Home", "Getting Started", and "Community". The main heading is "Tag Immutability Rules". A sidebar on the left lists navigation options: "HARBOR VERSION 1.10", "HARBOR INSTALLATION AND CONFIGURATION", "HARBOR ADMINISTRATION", and "WORKING WITH PROJECTS". Under "WORKING WITH PROJECTS", there are links for "Create Projects", "Project Configuration", "Working with Images, Tags, and Helm Charts", "Pulling and Pushing Images in the Docker Client", "Managing Labels", and "Retagging Images". The main content area explains that by default, users can repeatedly push an image with the same tag to repositories in Harbor, which causes the previous image to be overwritten. It notes that the tag now points to a different image and the previous one becomes tagless. This is due to the Docker implementation, which does not enforce the mapping between an image tag and the image digest. The sha256 digest remains reliable and always points to the same build, but it is not rendered in a human-readable format. A final paragraph states that the Docker implementation requires deleting a tag results in the deletion of all other tags that point to the same digest, causing unwanted image deletions.



Build your image once in CI



Use specific container tags in deployments. We suggest the application version strategy (semver)

Treat Docker tags as immutable.

Force immutable tags on the Registry level.

Demand for flexible processes

Endless integrations

GitOps

Anti-pattern 2: Baking the configuration
inside container images

Containers

Easy fixes = scalability

More complex
deployment patterns

Microservices = tons of pipelines

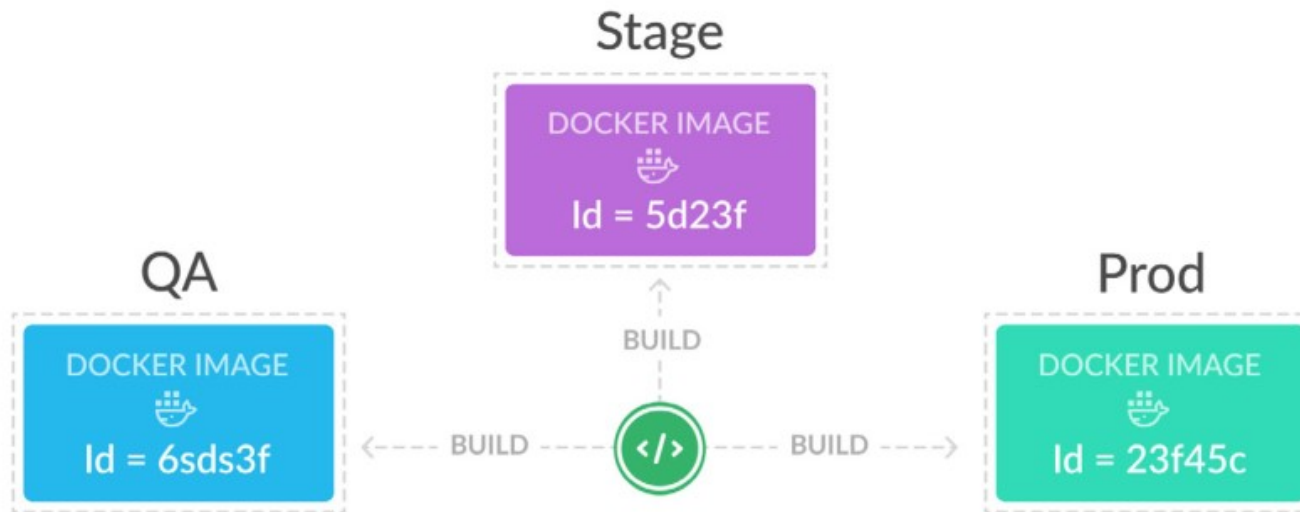
Blue/green deployments

Scaling pipeline variations

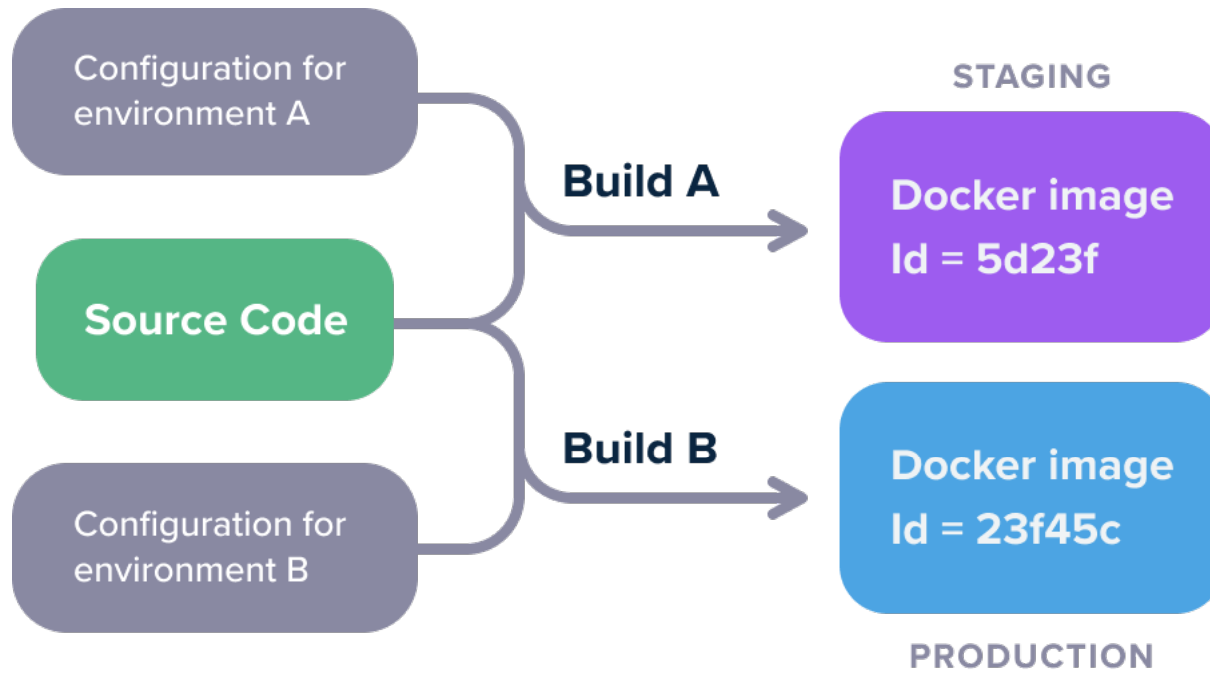
Monorepos

Cloud providers
Artifact stores
Unit & integration testing
Declarative infrastructure
PR Reviews
Login hell
Git providers
iOS & Android builds
Parallel builds & deployments
Security scans
Kubernetes
Shared dependencies
Provisioning environments
Canary releases

Different images per cluster



REJECTED



 **Don't**



How to detect this anti-pattern

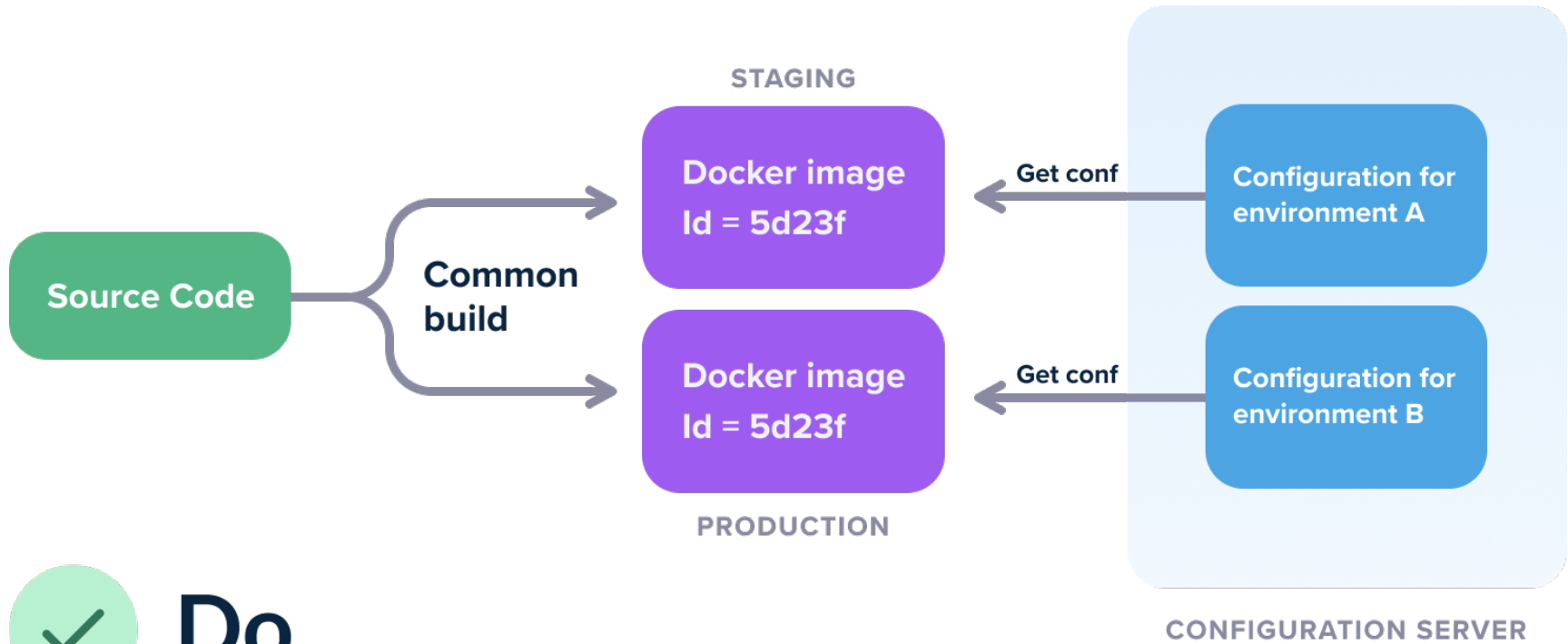
Hardcoded configuration smells

1. Tags myapp:staging, myapp:qa, myapp:prod
2. Git branches staging, production, qa
3. Config folder in Git with prod, qa, staging subfolders in application source code



A single Docker image should be deployed to all clusters (QA/Staging/Prod)

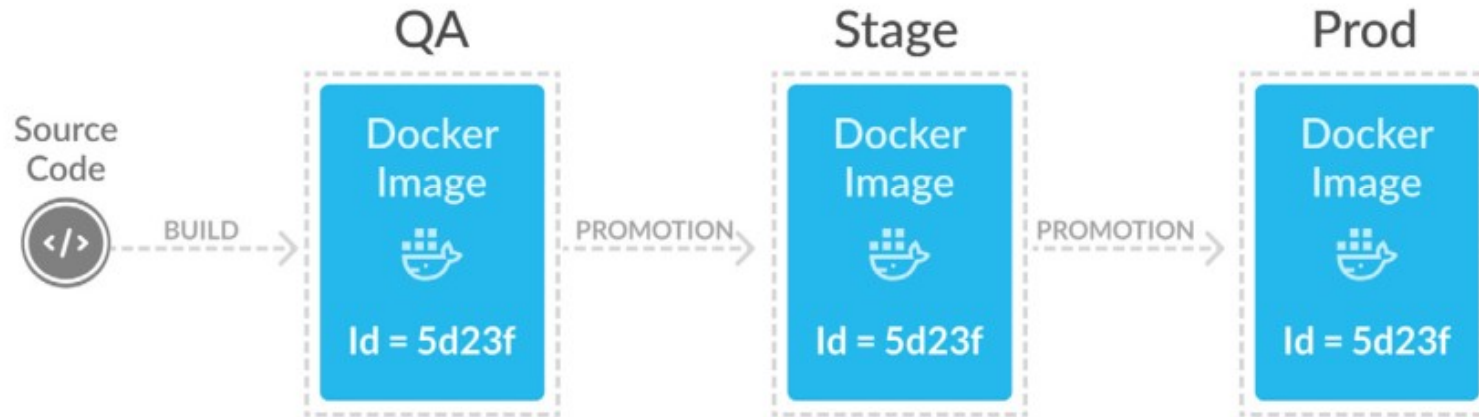
Configuration is loaded externally and never hardcoded in the container



 **Do**



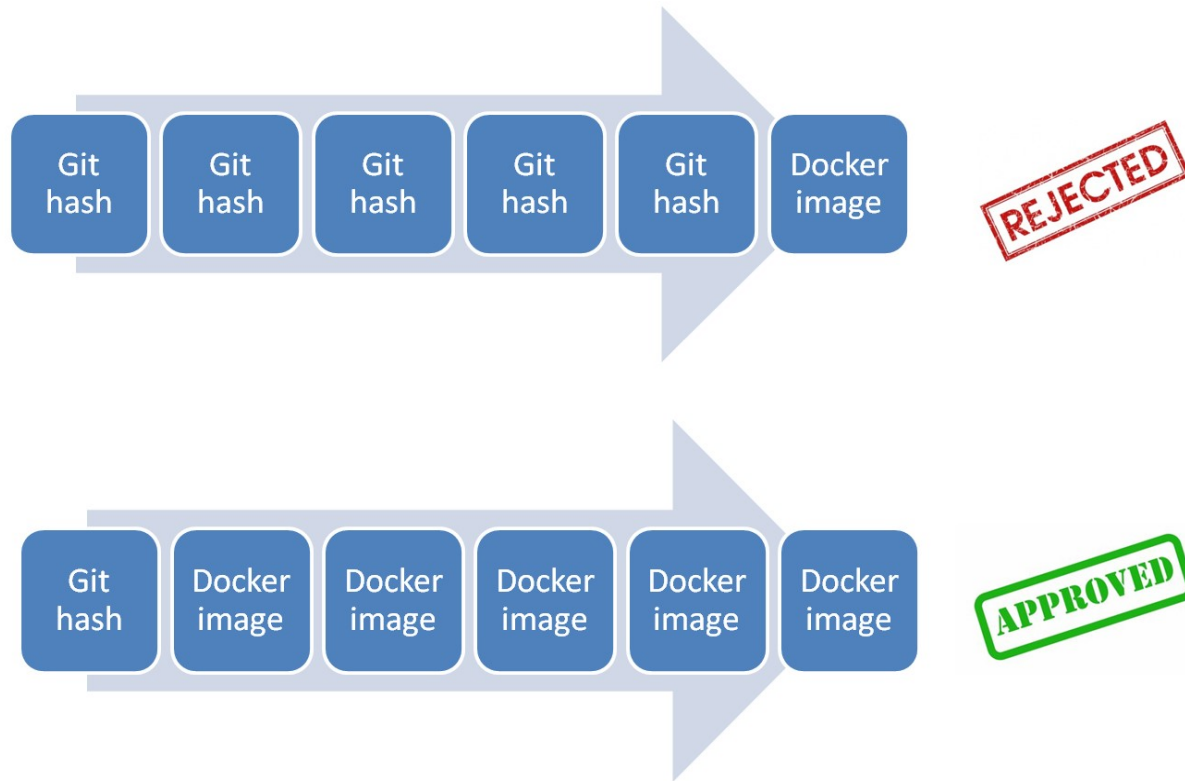
Promote the same image





<https://unsplash.com/photos/QMjCzOGegIA>

Build your image once in CI



Solution to hardcoded configuration

Decouple configuration

1. Kubernetes configmaps
2. Consul
3. etcd
4. Zookeeper
5. Bitnami Sealed secrets/ Mozilla Sops
6. Hashicorp vault



This was good advice even before k8s



III. Config

Store config in the environment

An app's *config* is everything that is likely to vary between deploys (staging, production, developer environments, etc). This includes:

- Resource handles to the database, Memcached, and other backing services
- Credentials to external services such as Amazon S3 or Twitter
- Per-deploy values such as the canonical hostname for the deploy

Apps sometimes store config as constants in the code. This is a violation of twelve-factor, which requires strict separation of config from code. Config varies substantially across deploys, code does not.

A litmus test for whether an app has all config correctly factored out of the code is whether the codebase could be made open source at any moment, without compromising any credentials.

Note that this definition of "config" does not include internal application config, such as `config/routes.rb` in Rails, or how code modules are connected in Spring. This type of config does not vary between deploys, and so is best done in the code.

<https://12factor.net/config>



All clusters get a single image.

Test the same image developers created

Each cluster has different runtime configuration

Cloud providers
Artifact stores
Unit & integration testing
Endless integrations
Login hell
Vulnerability scans
Containers
Git providers
GitOps
Declarative infrastructure
PR Reviews
iOS & Android builds
Parallel builds & deployments
Self-hosted
Canary releases
Kubernetes
Shared dependencies
Provisioning environments
More complex deployment patterns
Microservices = tons of pipelines
Rolling updates
Blue/green deployments
Scaling pipeline variations
Monorepos
Easy fixes = scalability

Anti-pattern 3: Coupling applications with Kubernetes features/services for no reason

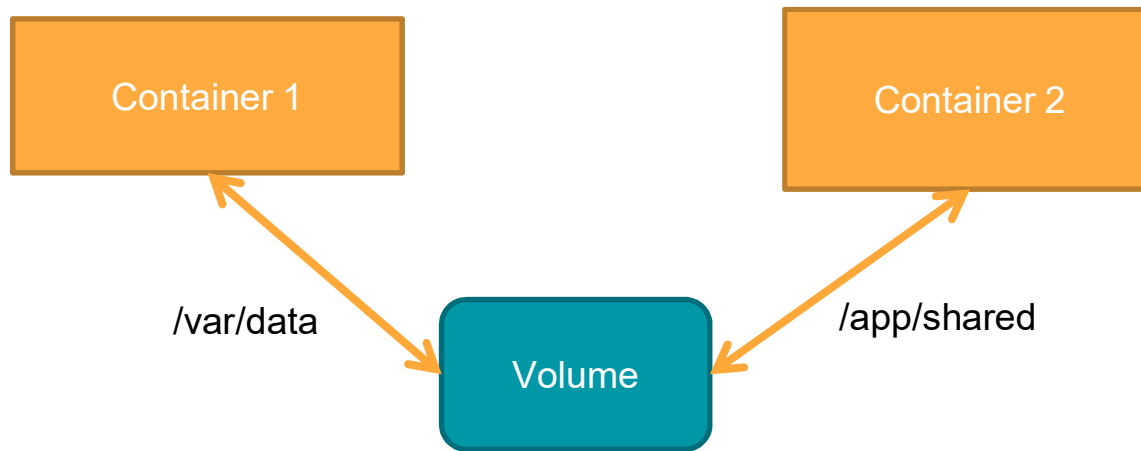
Assuming a prod namespace

```
func main() {  
  
    resp, err := http.Get("my-backend.prod.svc.cluster.local")  
    if err != nil {  
        panic(err)  
    }  
    defer resp.Body.Close()  
  
    // Print the HTTP response status.  
    fmt.Println("Response status:", resp.Status)  
}
```



REJECTED

Poor man's message queue



REJECTED

Common mistakes

Coupling to Kubernetes

1. Expect a certain volume configuration
2. Expect a certain naming of services/DNS
3. Read information directly from labels and annotations
4. Query the pod itself (e.g. for the IP address)
5. Need a sidecar or init (even in local development)
6. Call other services directly with their API (e.g. vault)



Getting secrets from vault

```
final Map<String, String> secrets = new HashMap<String, String>();
secrets.put("value", "world");
secrets.put("other_value", "You can store multiple name/value pairs under a sing

// Write operation
final LogicalResponse writeResponse = vault.logical()
    .write("secret/hello", secrets);

...

// Read operation
final String value = vault.logical()
    .read("secret/hello")
    .getData().get("value");
```

```
public class Example {
    // inject the actual template
    @Autowired
    private VaultOperations operations;

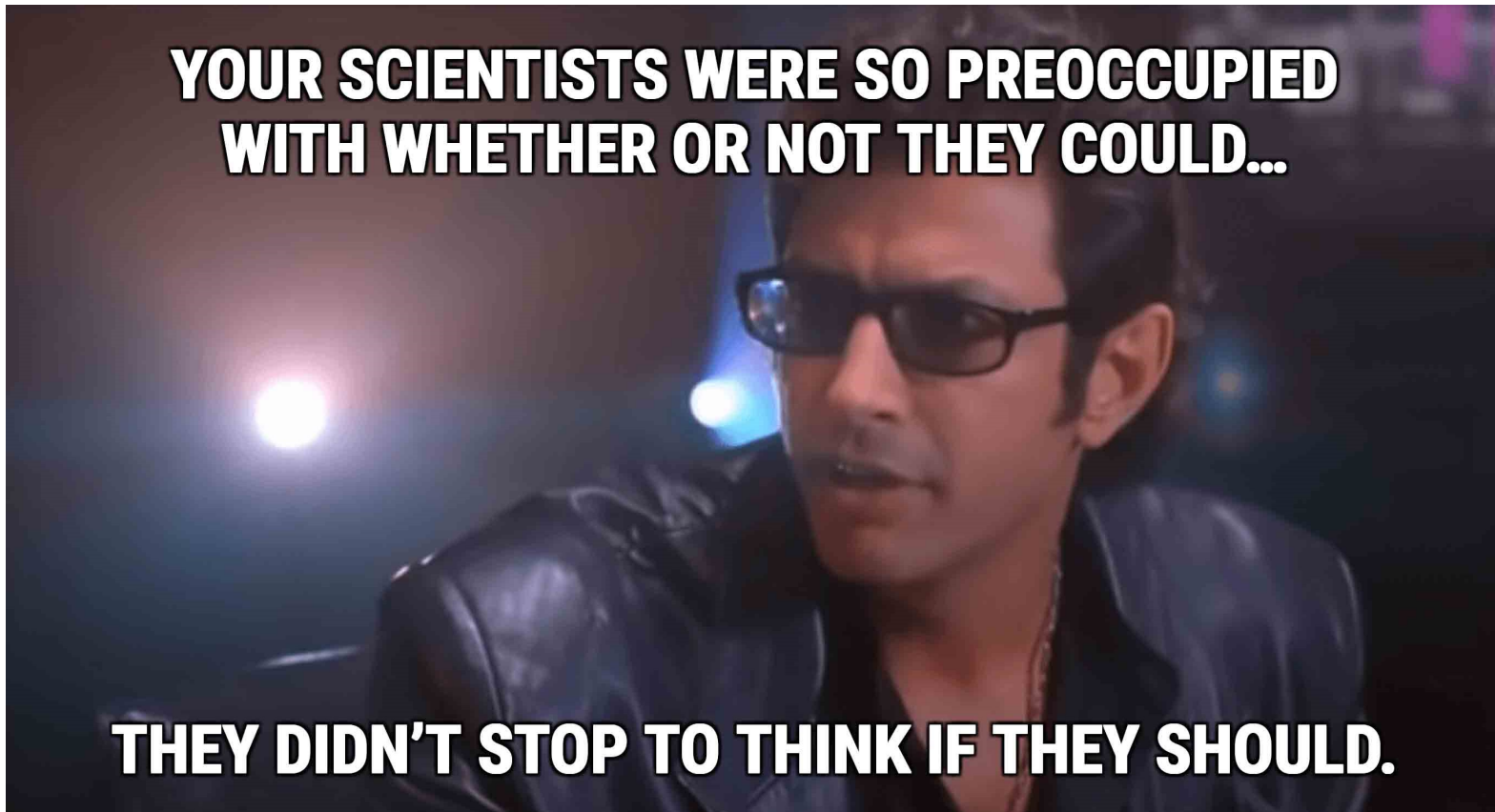
    public void writeSecrets(String userId, String password) {
        Map<String, String> data = new HashMap<String, String>();
        data.put("password", password);

        operations.write(userId, data);
    }

    public Person readSecrets(String userId) {
        VaultResponseSupport<Person> response = operations.read(userId, Person.class);
        return response.getBody();
    }
}
```

COPY

REJECTED



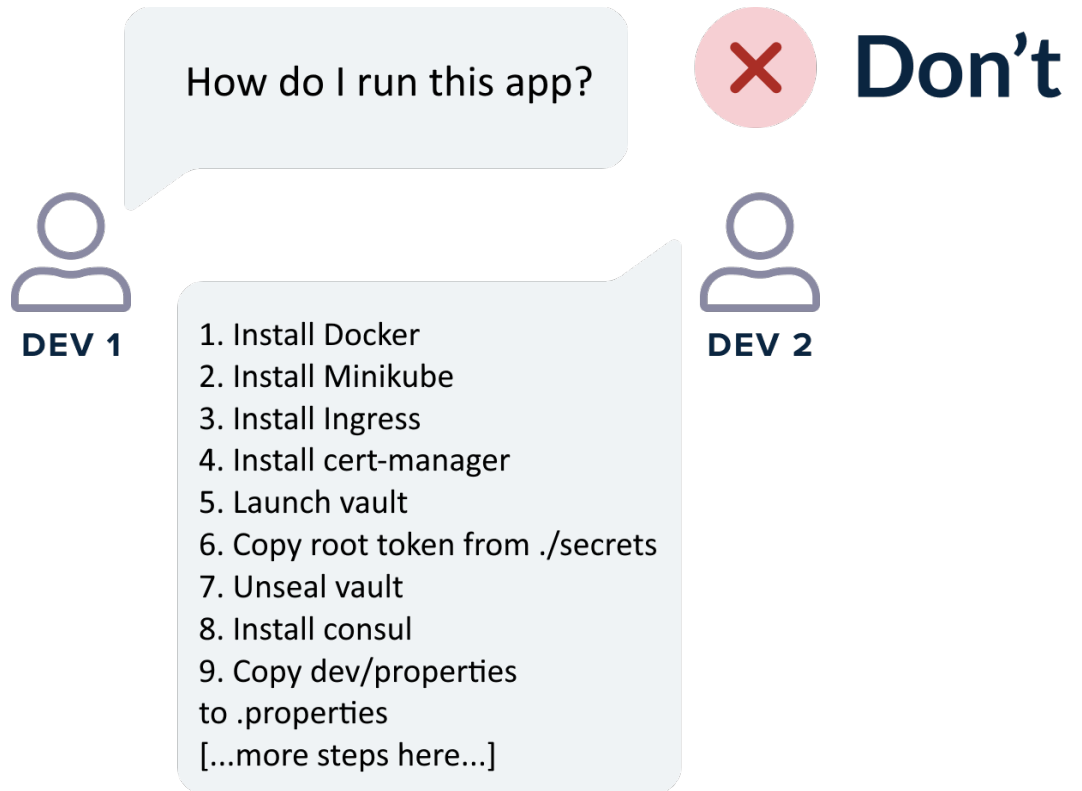
**YOUR SCIENTISTS WERE SO PREOCCUPIED
WITH WHETHER OR NOT THEY COULD...**

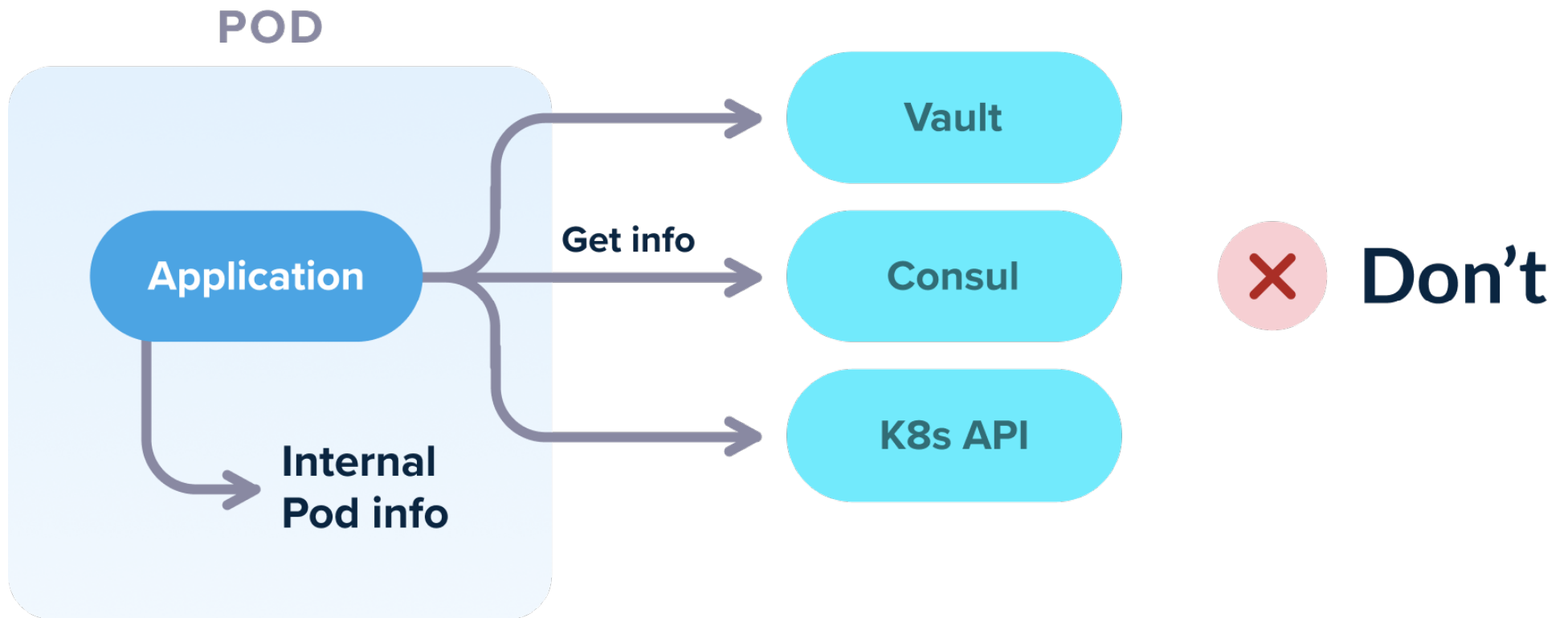
THEY DIDN'T STOP TO THINK IF THEY SHOULD.

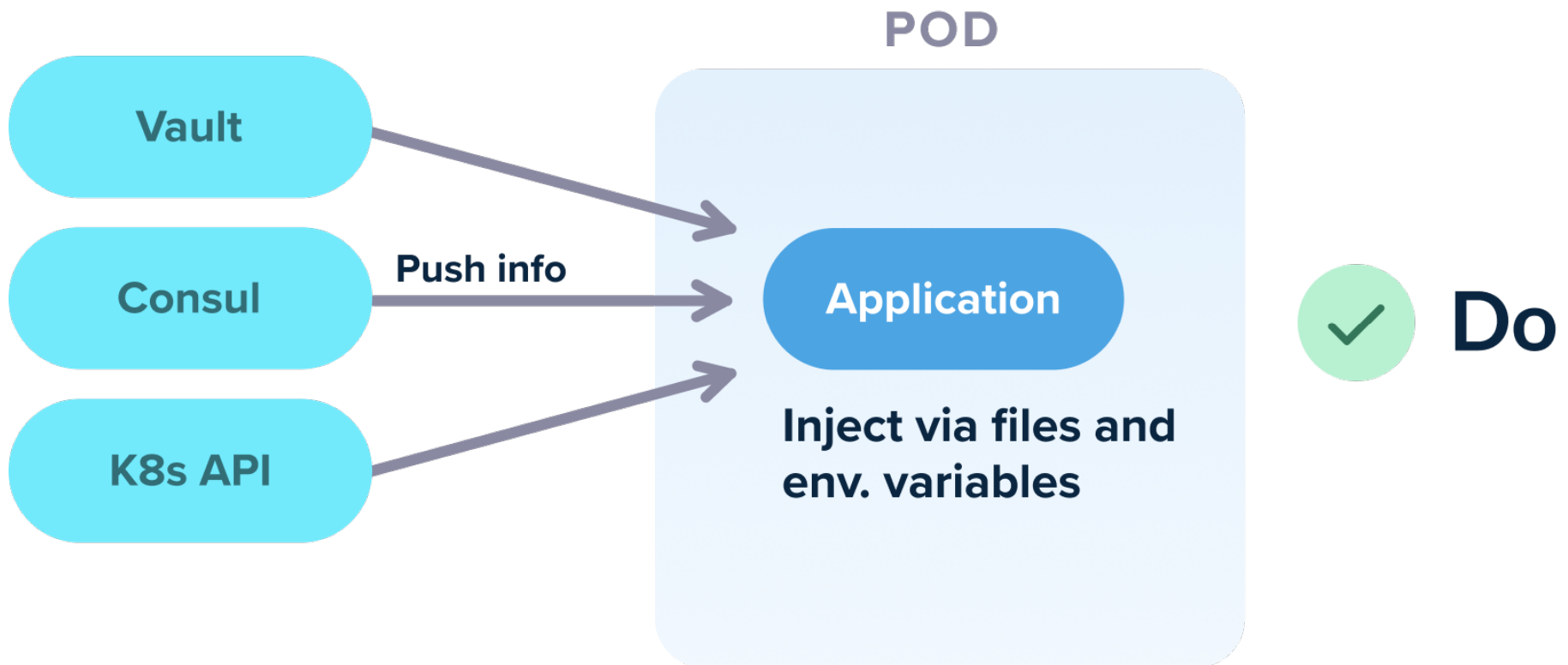


Making your life hard

- Developers have a hard time running the app.
- CI pipelines are super complex
- Integration testing is a mess
- There are too many moving parts









DEV 1

How do I run this app?



Do



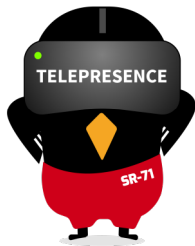
DEV 2

Run "docker-compose up"



Use dedicated solutions

Kubernetes local development tools



- <https://codefresh.io/kubernetes-tutorial/telepresence-2-local-development/>
- <https://codefresh.io/kubernetes-tutorial/okteto/>
- <https://codefresh.io/kubernetes-tutorial/local-kubernetes-development-tilt-dev/>
- <https://codefresh.io/howtos/local-k8s-draft-skaffold-garden/>



Don't use special
Kubernetes
services/APIs

Look at special
tools for local
dev

Your application
shouldn't even
know that it is
running inside
Kubernetes

Demand for flexible processes

Endless integrations

GitOps

Anti-pattern 4: Mixing application deployment with infrastructure deployment

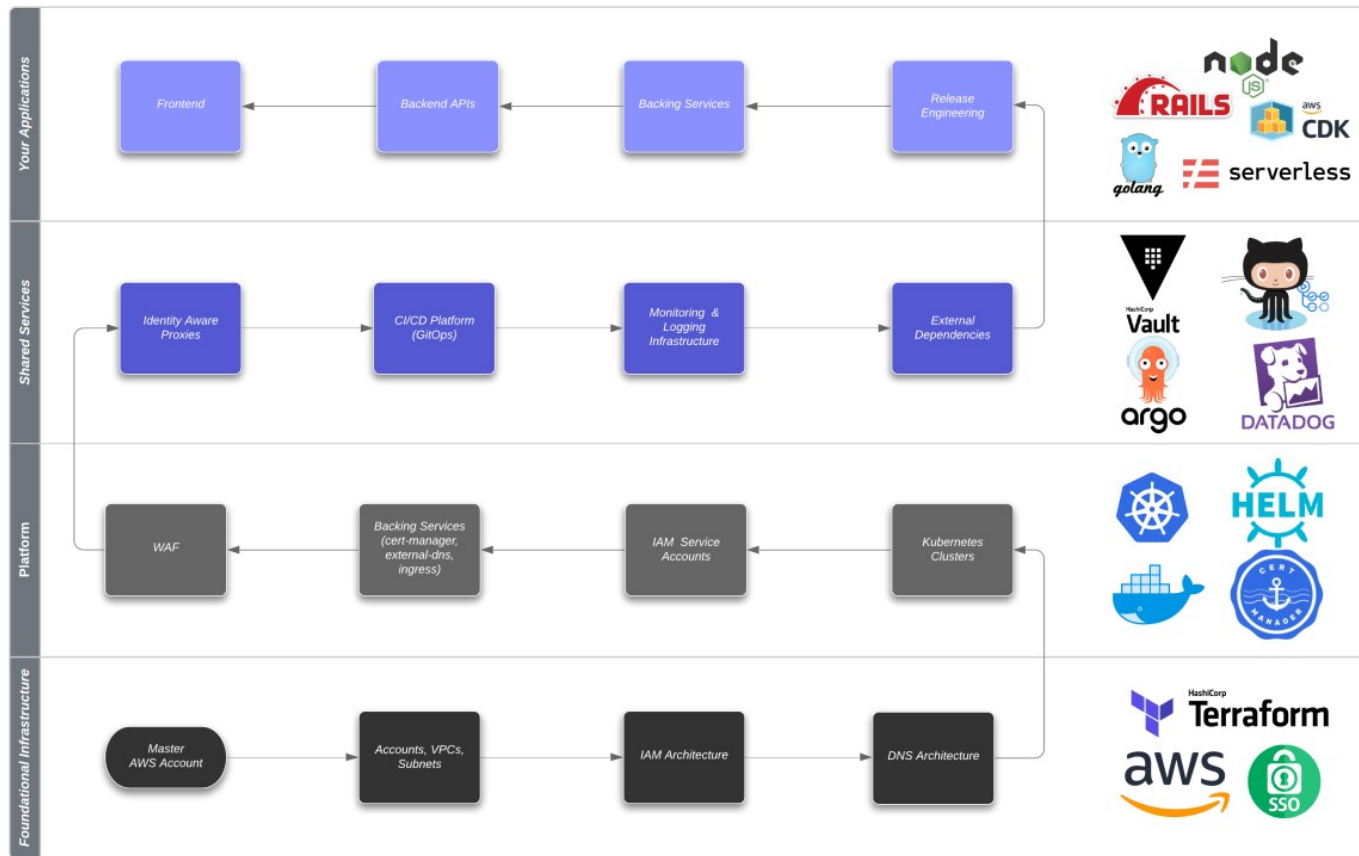
More complex deployment patterns

Microservices = tons of pipelines

Easy fixes = scalability

Cloud providers
Artifact stores
Unit & integration testing
Declarative infrastructure
PR Reviews
Login hell
Git providers
iOS & Android builds
Containers
Vulnerability scans
Parallel builds & deployments
Self-hosted
Canary releases
Kubernetes
Shared dependencies
Provisioning environments
Blue/green deployments
Scaling pipeline variations
Monorepos

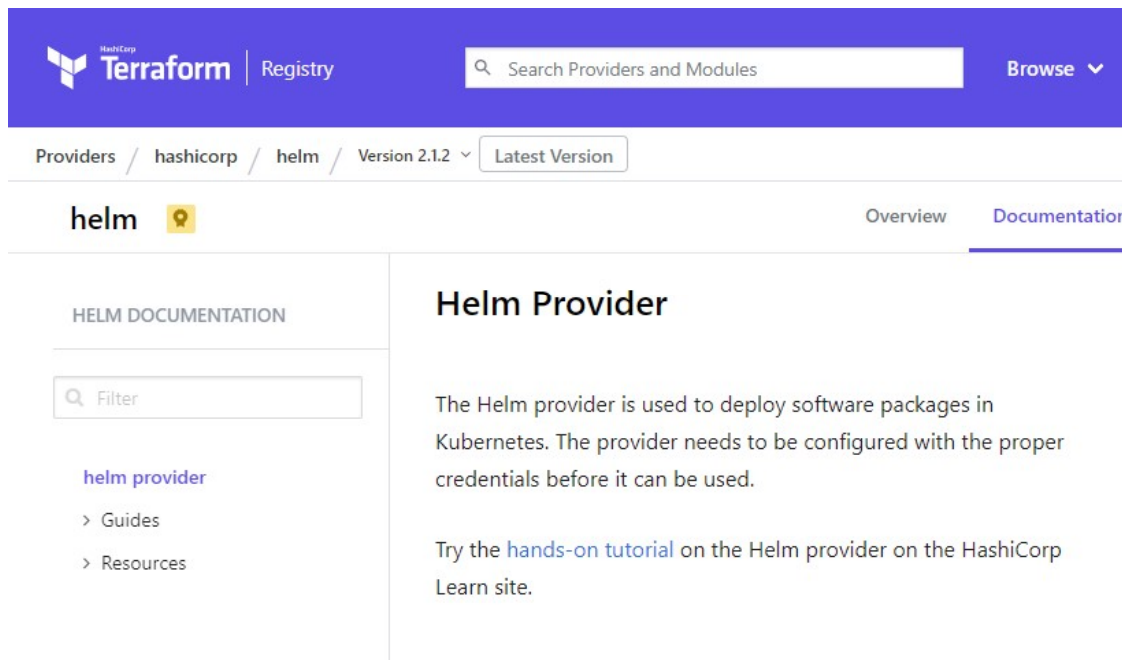
4 LAYERS OF INFRASTRUCTURE



Cloud Posse

<https://cloudposse.com/big-picture/>

Terraform Kubernetes provider

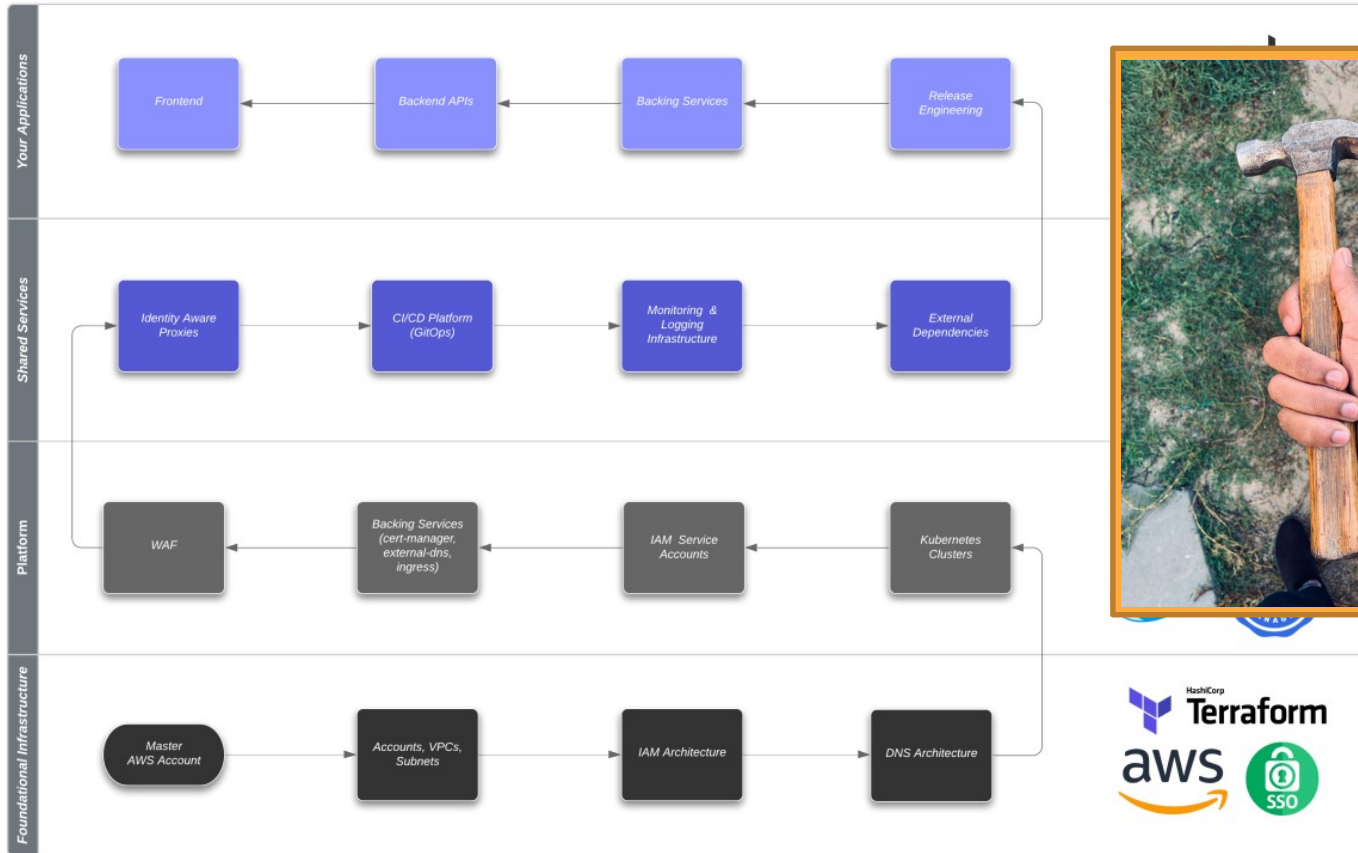


The screenshot shows the Terraform Registry page for the Helm provider. The page has a purple header with the Terraform logo and 'Registry' text. A search bar contains 'Search Providers and Modules' and a 'Browse' dropdown. Below the header, the breadcrumb path is 'Providers / hashicorp / helm / Version 2.1.2' with a 'Latest Version' button. The main content area is titled 'helm' and has tabs for 'Overview' and 'Documentation'. The 'Documentation' tab is active, showing 'HELM DOCUMENTATION' with a search filter. A list of links includes 'helm provider', '> Guides', and '> Resources'. The main text describes the Helm provider as used for deploying software packages in Kubernetes, requiring configuration with credentials. It also links to a 'hands-on tutorial' on the HashiCorp Learn site.

REJECTED

4 LAYERS OF INFRASTRUCTURE

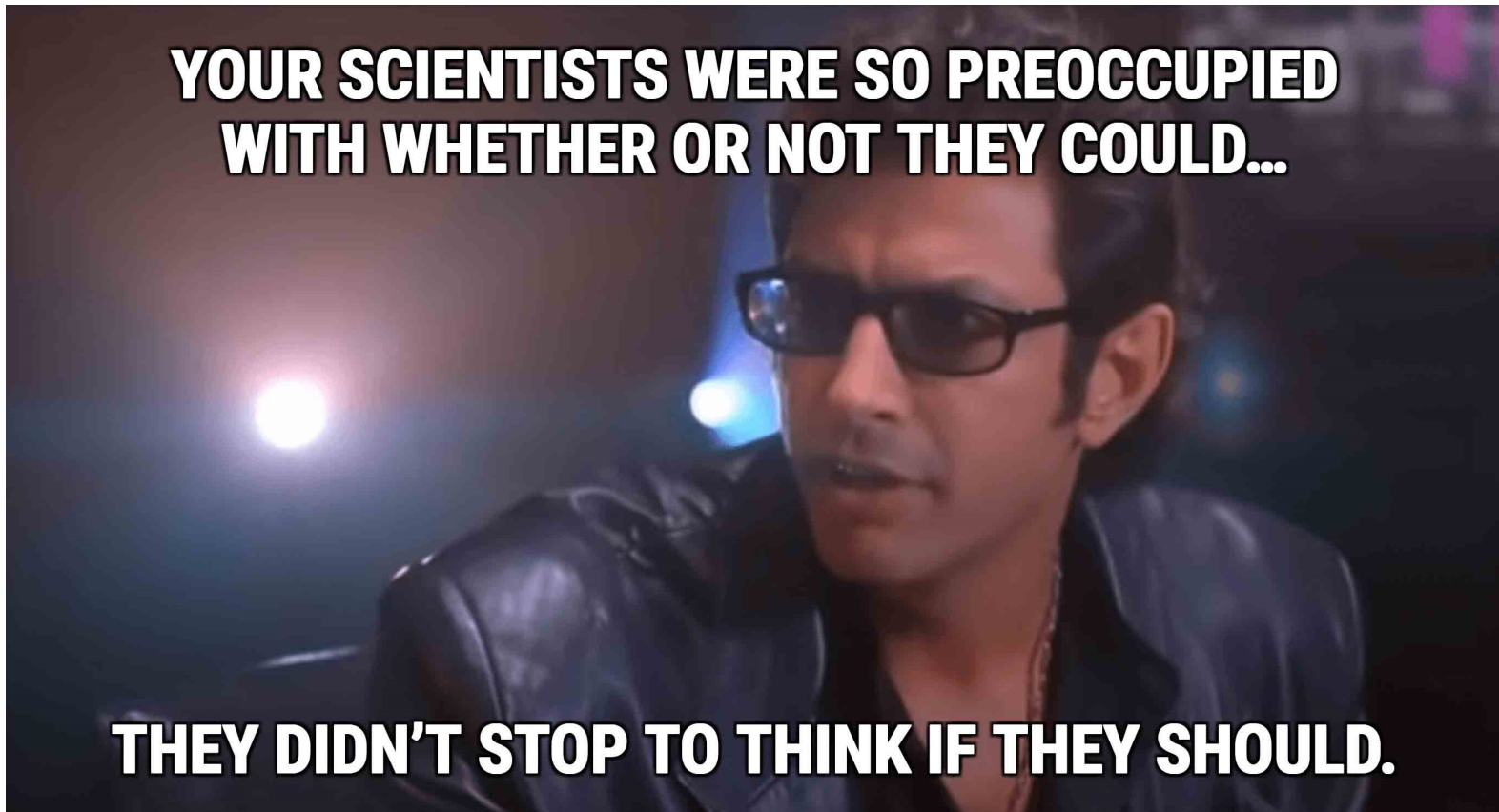
codefresh



Cloud Posse

<https://unsplash.com/photos/B4YHKz6lLrQ>



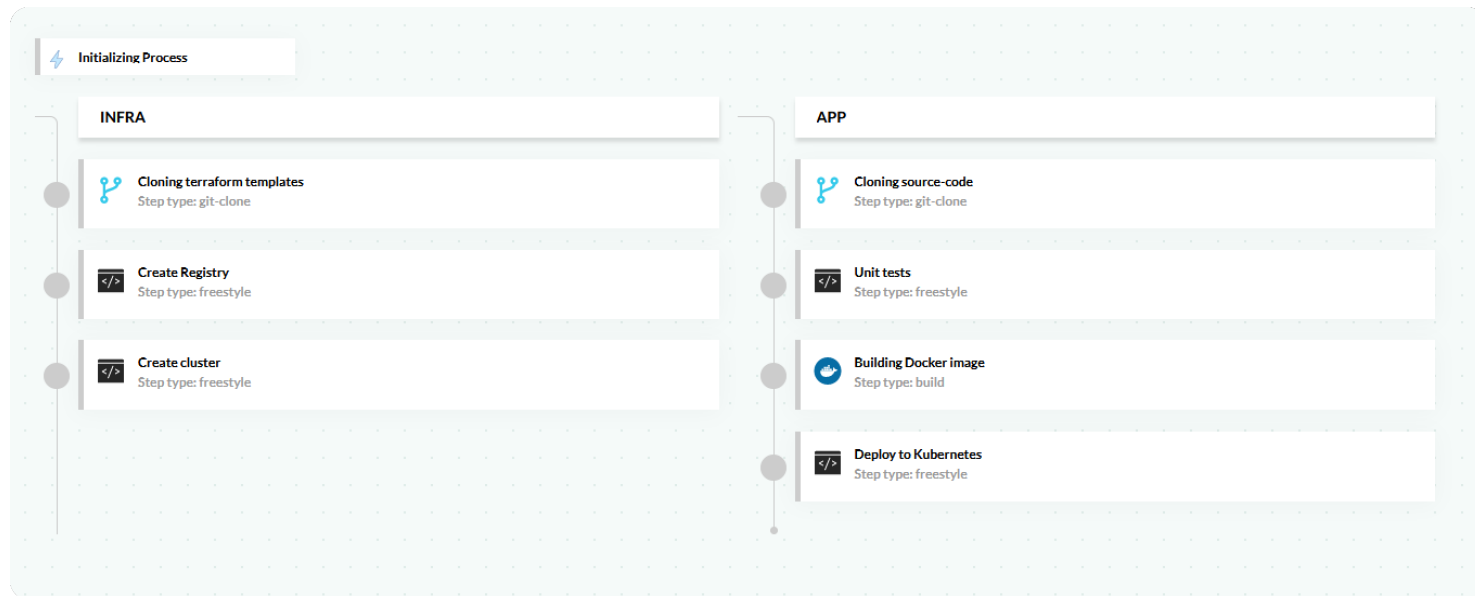


**YOUR SCIENTISTS WERE SO PREOCCUPIED
WITH WHETHER OR NOT THEY COULD...**

THEY DIDN'T STOP TO THINK IF THEY SHOULD.



Single pipeline for Infra and app

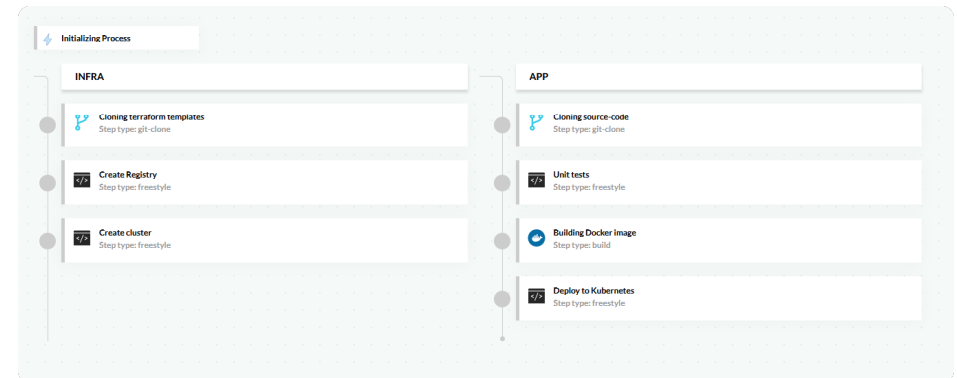


✗ Don't



Mixing infrastructure and application deployment

1. You are wasting time for everybody (dev/ops)
2. You are making life difficult for developers
3. Your deployments are very complex
4. Who should look at a broken pipeline? Dev or ops?

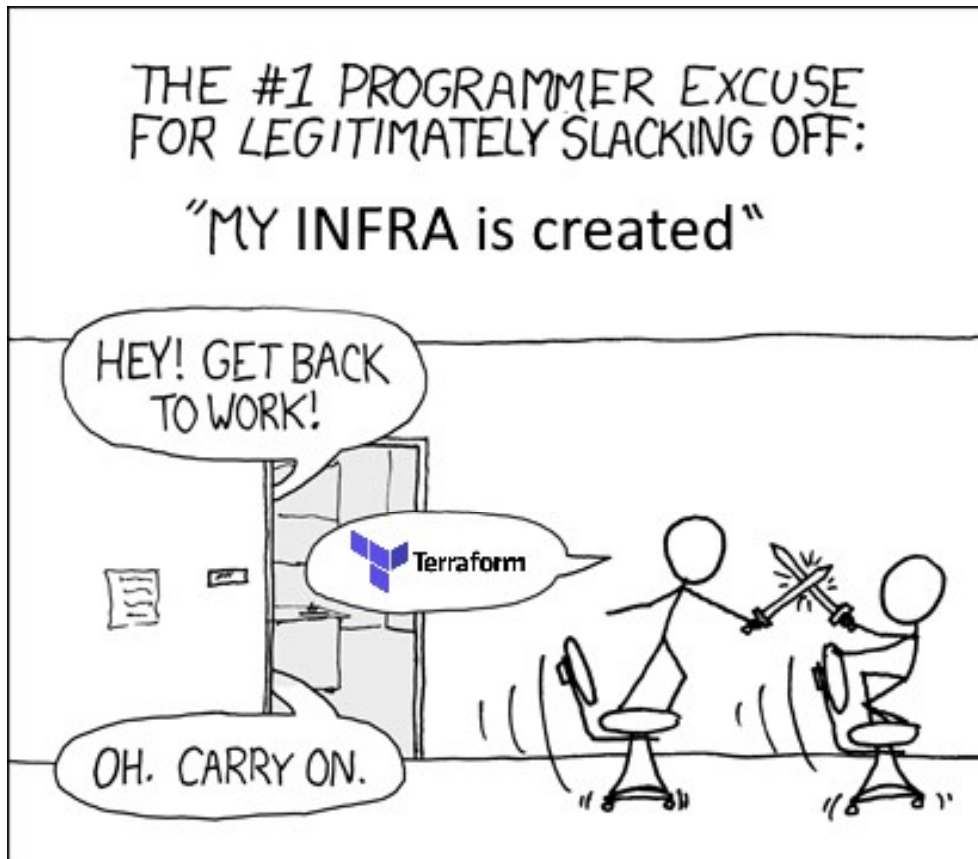


✗ Don't

Infrastructure and applications have a different change frequency

1. In most cases applications change 2x-10x more often than infrastructure
2. Deployment of infrastructure/app might take 30 minutes
3. Deployment of application might take 5 minutes
4. For each app deployment you WASTE 25 minutes

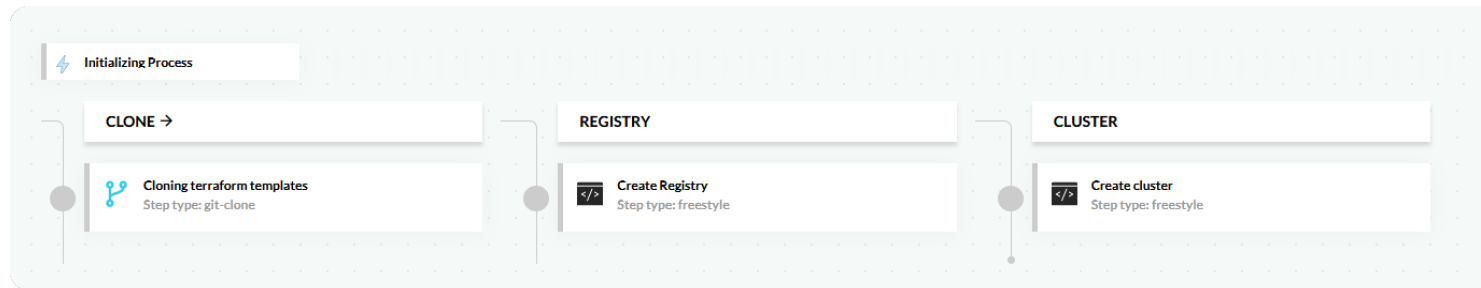




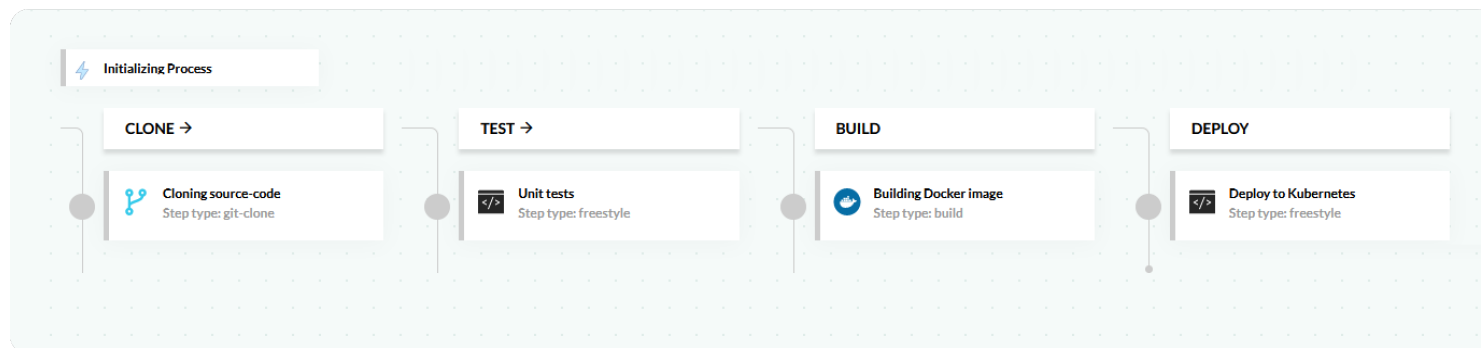
<https://xkcd.com/303/>

REJECTED

✓ **Do** Infrastructure pipeline, takes 25 minutes runs 3 times a day



✓ **Do** Application pipeline, takes 5 minutes and runs 20 times a day



Developers don't
care about
infrastructure
(and they
shouldn't have to
care)

Provide
Developers with
actionable errors
in pipelines

FAILED Failed executing Create cluster step. 28 s a minute ago LOG 13.32kB EDIT PIPELINE RESTART codefresh

Initializing Process 9 s

Group	Step Name	Step Type	Duration	Status
INFRA	Cloning terraform templates	git-clone	3 s	Success
	Create Registry	freestyle	5 s	Success
	Create cluster	freestyle	6 s	Failed
APP	Cloning source-code	git-clone		Not Started
	Unit tests	freestyle		Not Started
	Building Docker image	build		Not Started
	Deploy to Kubernetes	freestyle		Not Started

I am a developer and want to deploy my app
I don't care how the cluster works. The pipeline has failed and
my application is not deployed. What do I do?

REJECTED

Applications
should be deployed
on their own

Infrastructure
deployment should
be separate

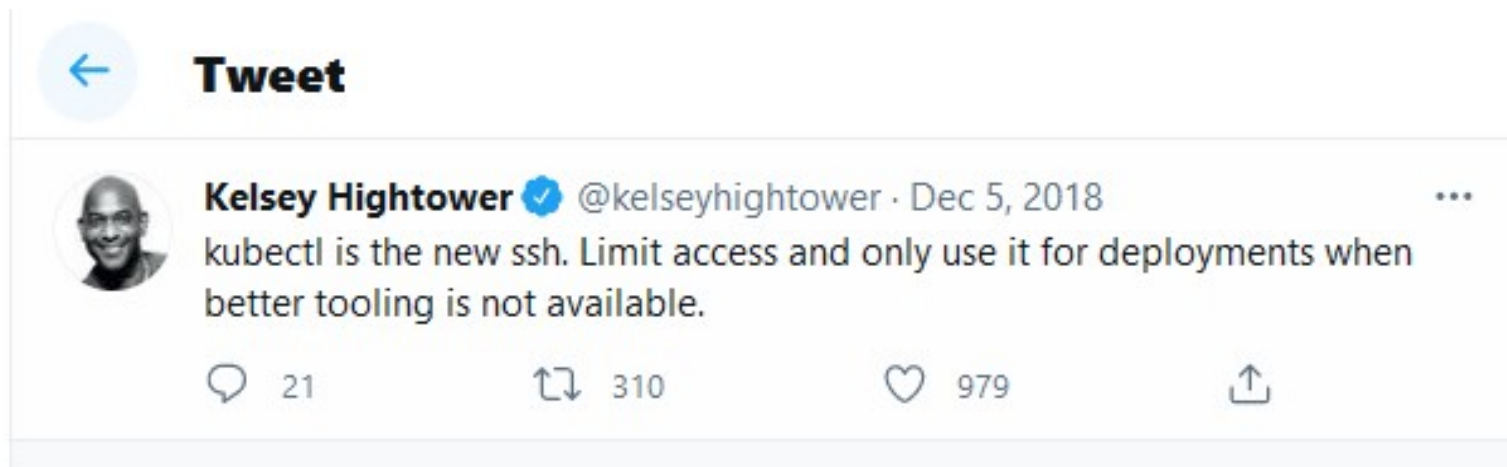
Don't abuse
Terraform for
application
deployments

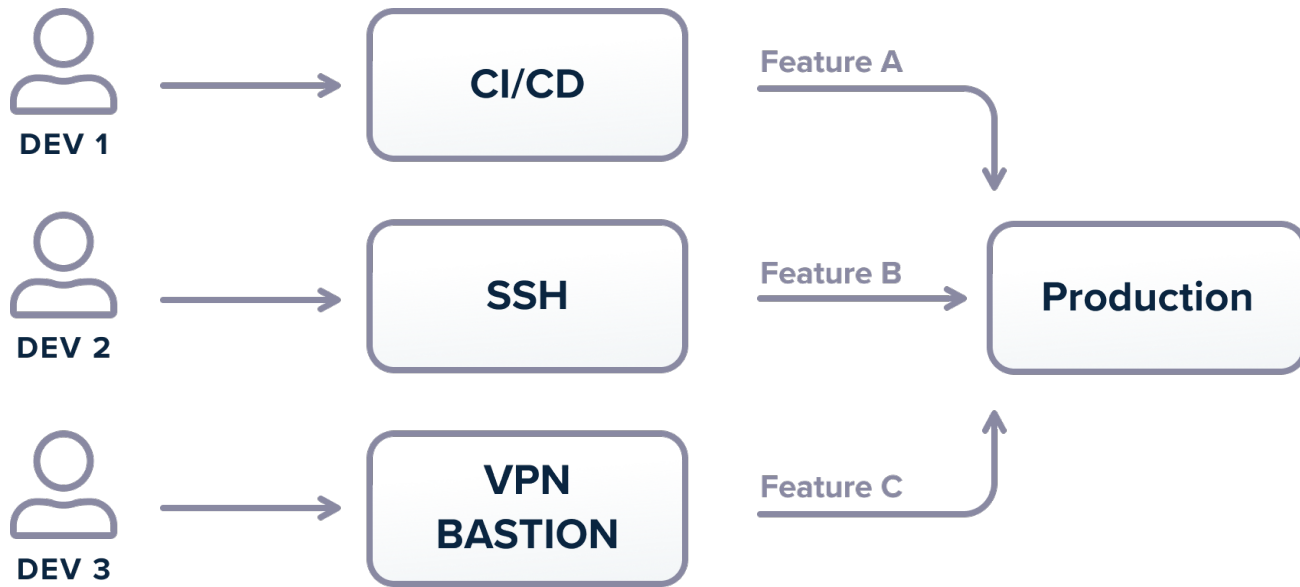
Cloud providers
Artifact stores
Unit & integration testing
Endless integrations
Login hell
Vulnerability scans
Containers
Git providers
GitOps
Declarative infrastructure
iOS & Android builds
Parallel builds & deployments
PR Reviews
Containers
Self-hosted
Canary releases
Kubernetes
Shared dependencies
Provisioning environments
More complex deployment patterns
Microservices = tons of pipelines
Scaling pipeline variations
Monorepos
Blue/green deployments

Anti-pattern 5: Performing ad-hoc deployments with kubectl edit/patch by hand

Easy fixes = scalability

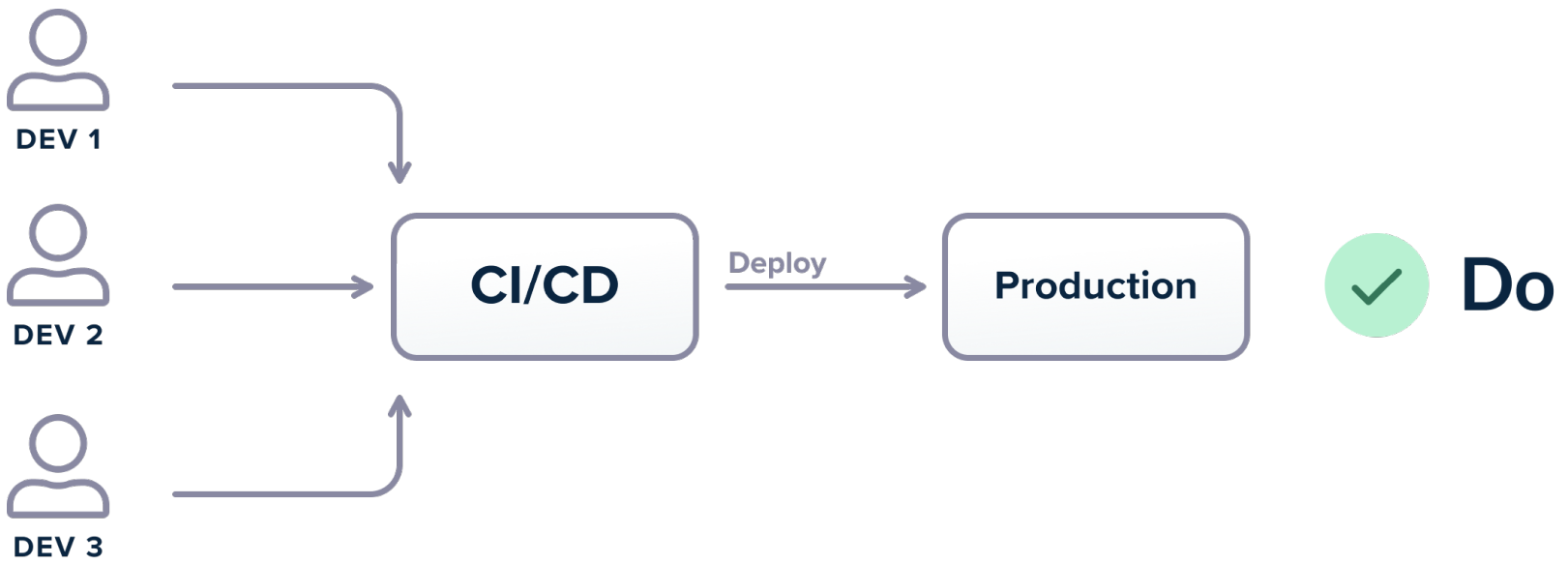
Kubectl is the new SSH





Don't





Deploying via SSH
was never a good
practice

This was true even
with VMs

Only CI/CD
should deploy to
production

3. Access The Argo CD API Server

By default, the Argo CD API server is not exposed with an external IP. To access the API server, choose one of the following techniques to expose the Argo CD API server:

Service Type Load Balancer

Change the argocd-server service type to `LoadBalancer`:

```
kubectl patch svc argocd-server -n argocd -p '{"spec": {"type": "LoadBalancer"}}'
```

Create Grafana Enterprise configuration

Create a Grafana configuration file with the name `grafana.ini`. Then paste the content below.

Note: You will have to update the `root_url` field to the url associated with the license you were given.

```
[enterprise]
license_path = /etc/grafana/license/license.jwt
[server]
root_url = /your/license/root/url
```

Create Configmap for Grafana Enterprise Config

Create a Kubernetes Configmap from your `grafana.ini` file with the following command:

```
kubectl create configmap ge-config --from-file=/path/to/your/config.ini
```

Edit the file with the command:

```
kubectl edit cm prometheus-server
```

And add this new job:

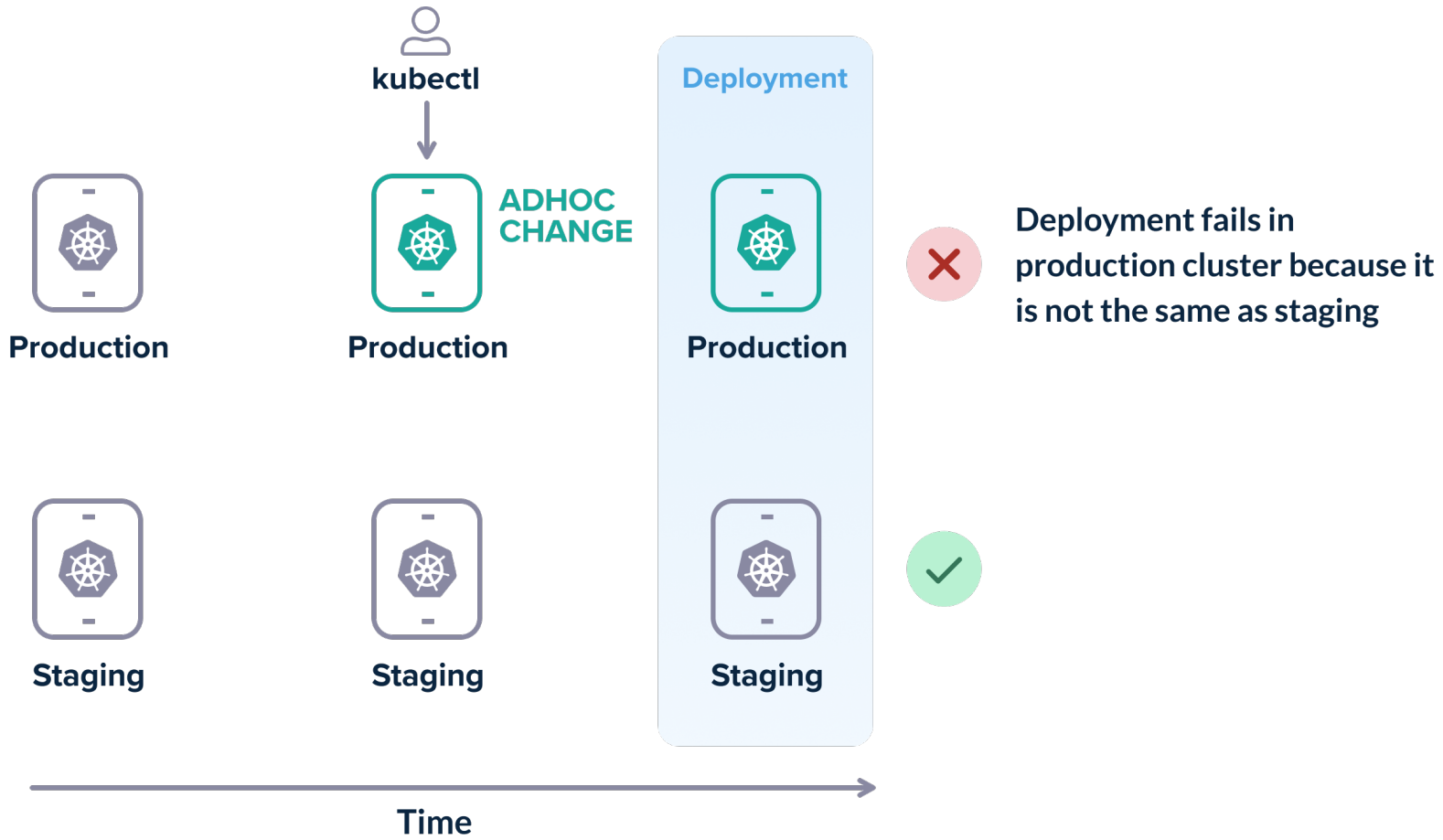
```
- job_name: 'traefik'
  static_configs:
  - targets: ['traefik-prometheus:9100']
```



Don't deploy to production with manual kubectl commands

1. Kubectl apply/edit/patch are only for demos and POCs
2. Never change live resources on a cluster
3. You never know what is installed in your cluster
4. Perfect recipe for disaster (configuration drift)





Git is the single
source of truth.
All changes should
pass from Git.
Change resources
by git commit/push

Use GitOps

Deploy with a Git commit

1. You know exactly what is in the cluster
2. You have a complete history of what/when/by whom
3. You can create/clone your cluster in minutes
4. Roll back by simply going to a previous commit

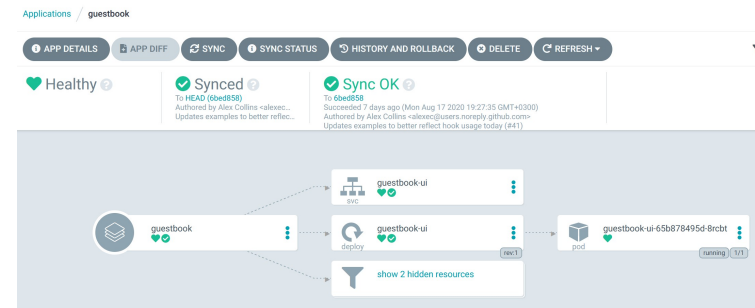
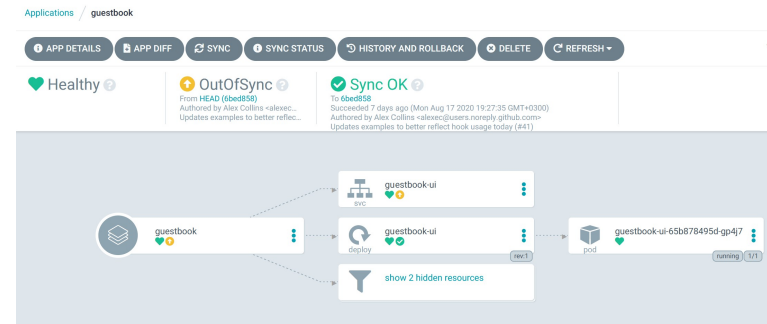


Avoid configuration drift with GitOps

SUMMARY PARAMETERS MANIFEST **DIFF** EVENTS

Compact diff Inline Diff

```
 /Service/default/guestbook-ui
1  apiVersion: v1                1  apiVersion: v1
2  kind: Service                 2  kind: Service
3  metadata:                    3  metadata:
4  labels:                      4  labels:
5  app.kubernetes.io/instance: guestbook  5  app.kubernetes.io/instance: guestbook
6  name: guestbook-ui           6  name: guestbook-ui
7  spec:                        7  spec:
8  ports:                       8  ports:
9  - port: 8000                 9  - port: 8000
10 targetPort: 80               10 targetPort: 80
11 selector:                    11 selector:
12 app: guestbook-ui           12 app: guestbook-ui
```



Avoid manual
deployments with
SSH

Avoid manual
deployments with
kubectl

Always use Git to
know what is in
your cluster

Recap

Top 5 – anti-patterns

1. Don't use latest tag. Treat tags as immutable
2. Don't create different images per environment
3. Don't couple the application to K8s (or Vault)
4. Don't mix infrastructure with application deployment
5. Use kubectl apply/patch/edit only for demos/POVs



The modern approach to
DevOps automation

Open a FREE account today at codefresh.io

Demand for flexible processes

Endless integrations

GitOps

Anti-pattern 6: Using Kubectl as a debugging tool

Containers

Microservices = tons of pipelines

Cloud providers
Artifact stores
Unit & integration testing
Declarative infrastructure
PR Reviews
Plugin hell
Git providers
iOS & Android builds
Parallel builds & deployments
Security scans
Containers
Self-hosted
Canary releases
Kubernetes
Easy fixes = scalability
More complex deployment patterns
Shared dependencies
Provisioning environments
Rolling updates
Blue/green deployments
Scaling pipeline variations
Monorepos

Kubectl is the new SSH



You shouldn't use
SSH for debugging
VM applications

You shouldn't
use kubectl for
debugging
Kubernetes
applications

It is 3am. You are getting paged for your “sales” app

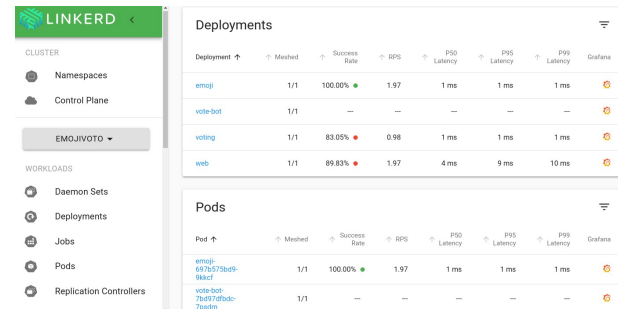
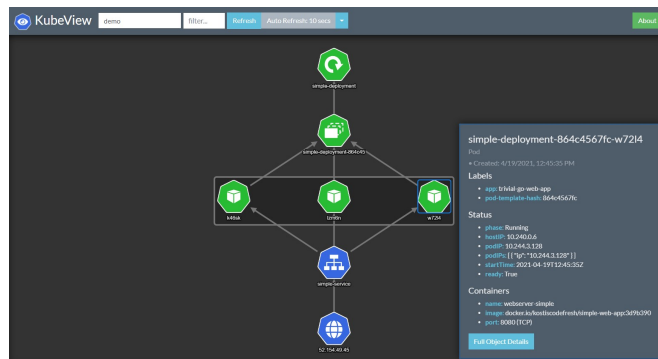
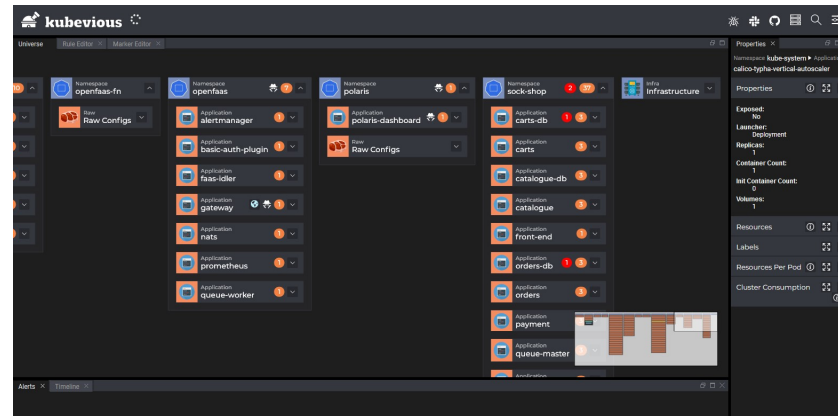
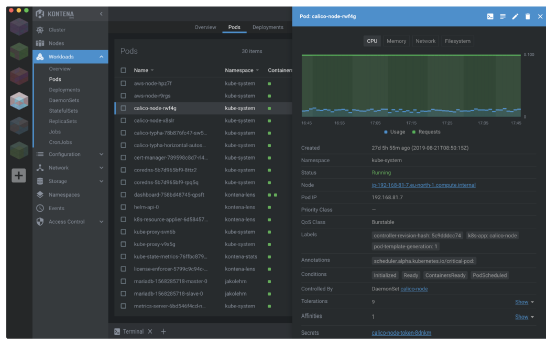
1. Open terminal
2. `kubectl get ns`
3. `kubectl get pods -n sales`
4. `kubectl describe pod prod-app-123 -n sales`
5. `kubectl svc -n sales`
6. `kubectl describe ...`
7. (more kubectl commands...)



If you need kubectl
to inspect
something you
have a gap in your
observability tools

There are
dedicated tools
for Kubernetes
debugging today

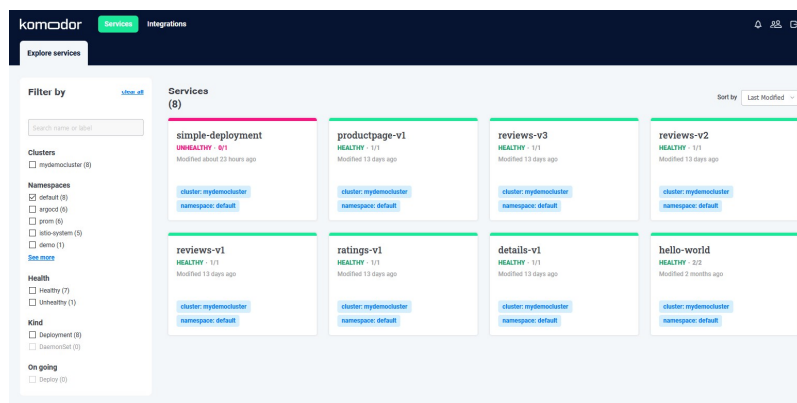
General purpose dashboards



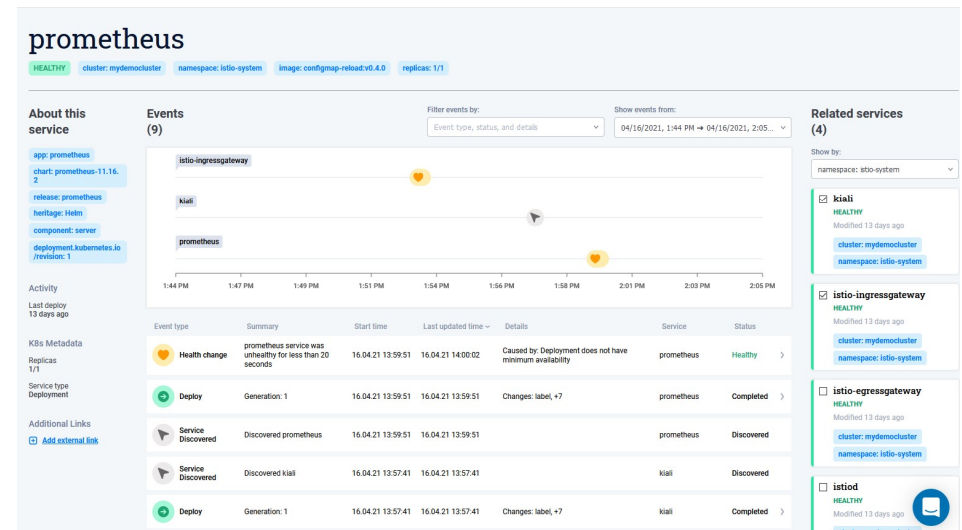
<https://codefresh.io/kubernetes-tutorial/kubevious-kubernetes-dashboard/>



Komodor - Kubernetes troubleshooting



The Komodor dashboard displays a grid of service cards. The 'simple-deployment' card is highlighted in pink and shows an 'UNHEALTHY' status with '0/1' replicas. Other cards include 'productpage-v1', 'reviews-v3', 'reviews-v2', 'reviews-v1', 'ratings-v1', 'details-v1', and 'hello-world', all showing 'HEALTHY' status.



The Prometheus service details page shows a 'HEALTHY' status with the following metadata: cluster: mydemocluster, namespace: istio-system, image: configmap-reload:v0.4.0, replicas: 1/1. The 'Events' section shows a timeline with a 'Health change' event at 1:54 PM and a 'Deploy' event at 1:58 PM. The 'Event details' table is as follows:

Event type	Summary	Start time	Last updated time	Details	Service	Status
Health change	prometheus service was unhealthy for less than 20 seconds	16.04.21 13:59:51	16.04.21 14:00:02	Caused by: Deployment does not have minimum availability	prometheus	Healthy
Deploy	Generation: 1	16.04.21 13:59:51	16.04.21 13:59:51	Changes: label +7	prometheus	Completed
Service Discovered	Discovered prometheus	16.04.21 13:59:51	16.04.21 13:59:51		prometheus	Discovered
Service Discovered	Discovered kiali	16.04.21 13:57:41	16.04.21 13:57:41		kiali	Discovered
Deploy	Generation: 1	16.04.21 13:57:41	16.04.21 13:57:41	Changes: label +7	kiali	Completed

<https://codefresh.io/devops/troubleshooting-kubernetes-with-komodor/>



Setup metrics and dashboards. Create runbooks

Predict incidents instead of putting out fires

Use kubectl as a last resort. After the incident add new metric to your dashboard

Cloud providers
Artifact stores
Unit & integration testing
Endless integrations
Login hell
Priority scans
Containers
Git providers
GitOps
Declarative infrastructure
PR Reviews
iOS & Android builds
Parallel builds & deployments
Self-hosted
Easy fixes = scalability
Provisioning environments
Microservices = tons of pipelines
Scaling pipeline variations
Monorepos
Blue/green deployments
More complex deployment patterns
Canary releases
Kubernetes
Shared dependencies

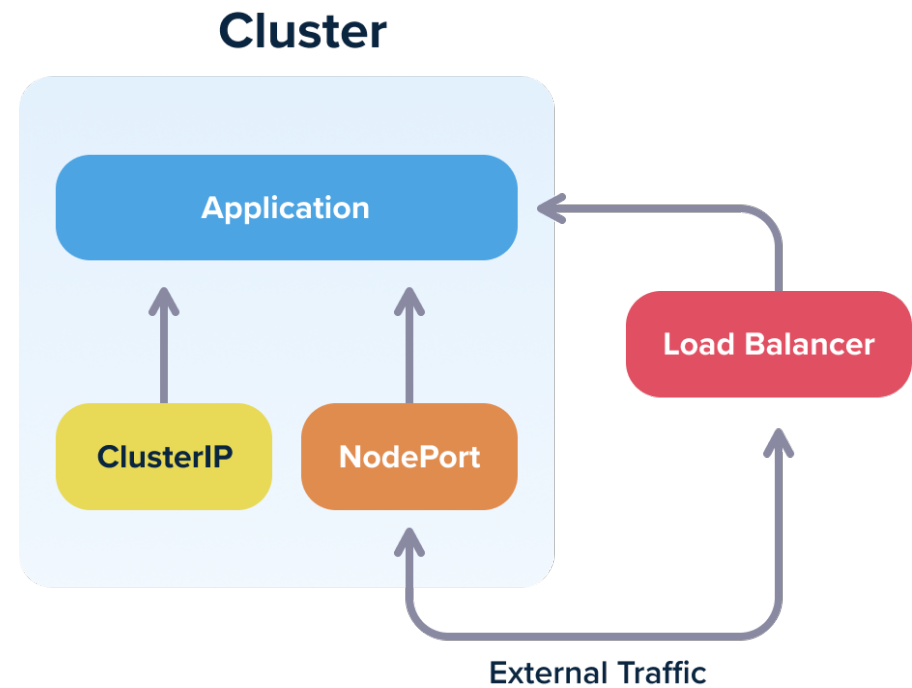
Anti-pattern 7: Misunderstanding Kubernetes network concepts

VMs:
LoadBalancer
Reverse Proxy

Kubernetes:
Service
Load balancer
Ingress
ClusterIP
NodePort
Service Mesh
Endpoint

Learn the basics

1. ClusterIP is internal traffic
2. Nodeport is internal/external
3. Loadbalancer is external and also affects billing in cloud installations



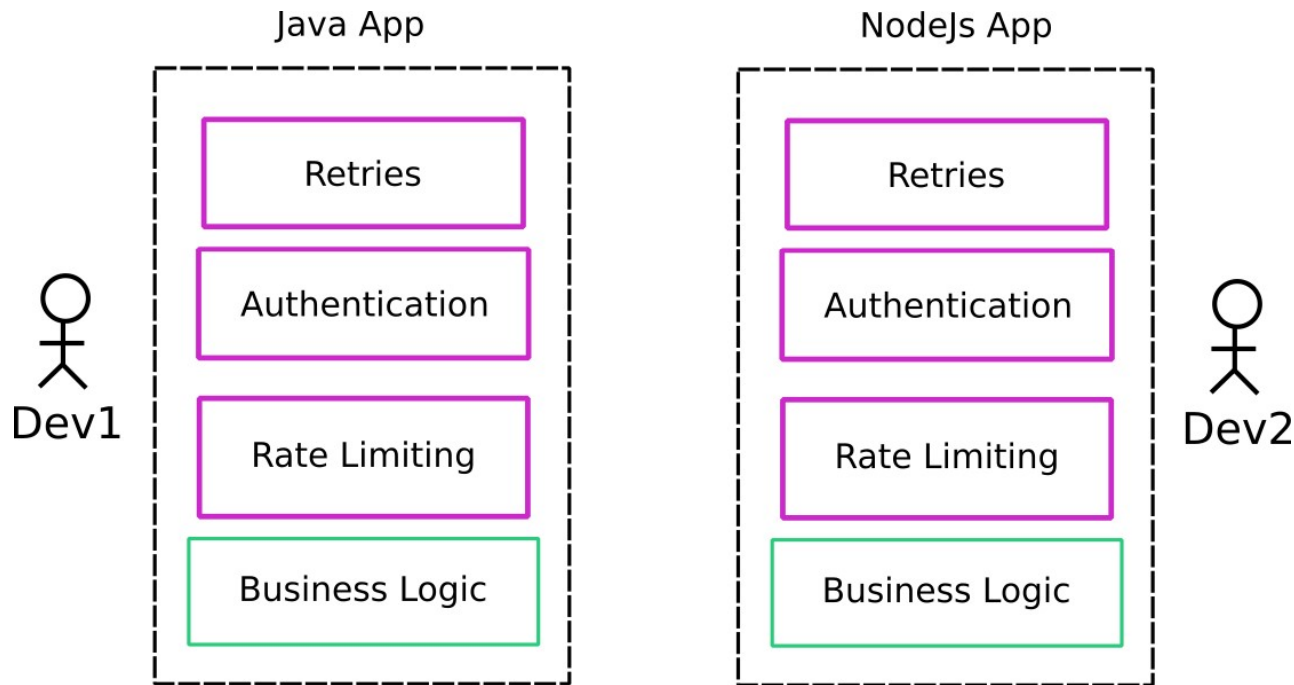
Learn the network topology

1. Loadbalancer per service (easy but expensive)
2. Single Ingress (cheap but inflexible)
3. Multiple Ingresses (powerful but complex)
4. With or without service mesh
5. With or without API gateway

If you are a
developer and each
microservice has
100ms latency

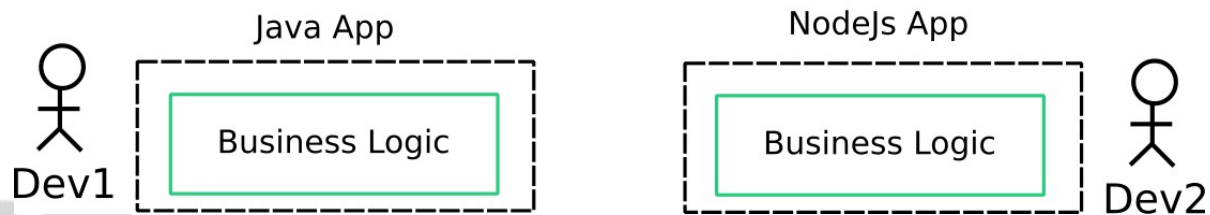
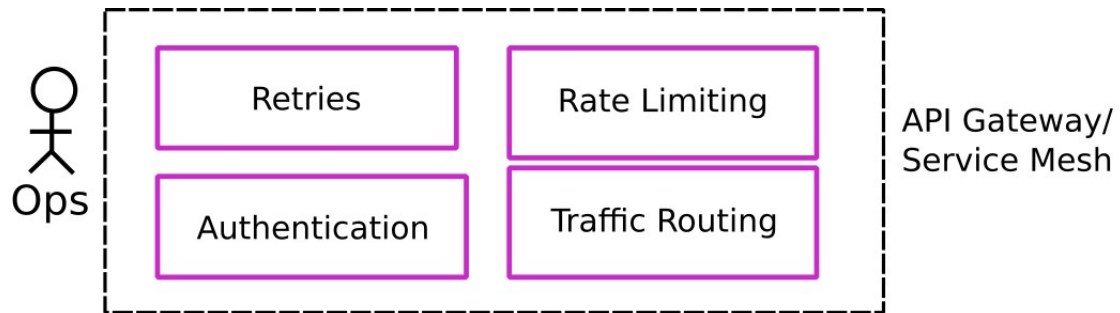
5 hops inside the
cluster is 0.5
seconds. Are
your customers
ready for that?

Before Service Mesh/Gateway



REJECTED

After Service Mesh/Gateway



Obsolete Programming libraries

1. Service discovery
2. Custom Load balancing
3. Authentication (e.g. OAuth)
4. Rate limiting
5. Retries/timeouts
6. Circuit breakers
7. Utilization metrics
8. Encryption, certificates

Hystrix: Latency and Fault Tolerance for Distributed Systems

oss lifecycle **maintenance** build error maven central **1.5.18** License Apache 2

Hystrix Status

Hystrix is no longer in active development, and is currently in maintenance mode.

Hystrix (at version 1.5.18) is stable enough to meet the needs of Netflix for our existing applications. Meanwhile, our focus has shifted towards more adaptive implementations that react to an application's real time performance rather than pre-configured settings (for example, through [adaptive concurrency limits](#)). For the cases where something like Hystrix makes sense, we intend to continue using Hystrix for existing applications, and to leverage open and active projects like [resilience4j](#) for new internal projects. We are beginning to recommend others do the same.



Understand how
traffic reaches your
application

Evaluate a
gateway or
service mesh.
Know the trade-
offs

Demand for flexible processes

Endless integrations

GitOps

Anti-pattern 8: Using permanent staging environments instead of dynamic ones

More complex deployment patterns

Microservices = tons of pipelines

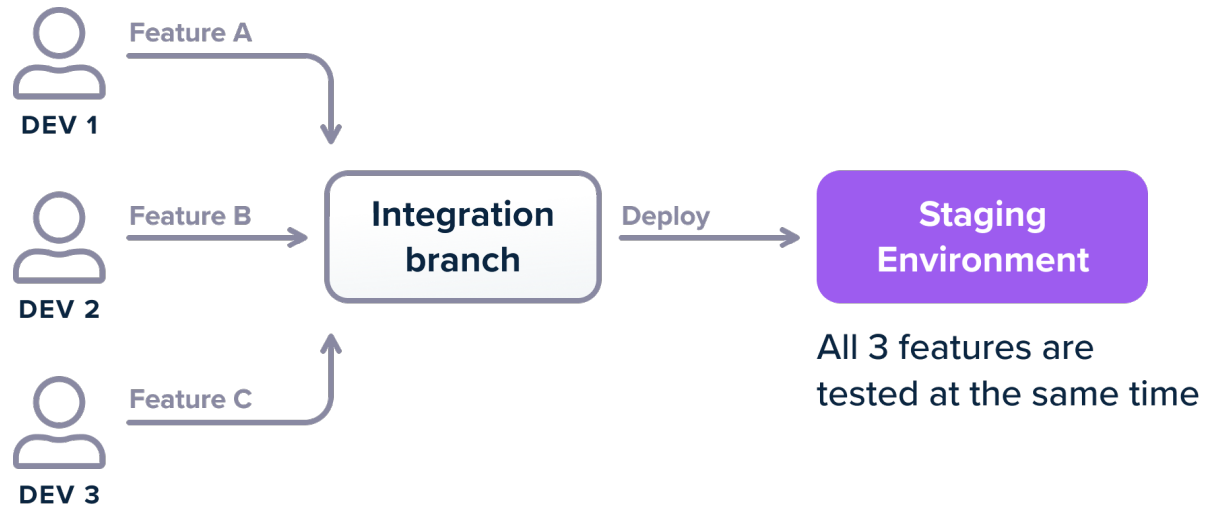
Easy fixes = scalability

Cloud providers
Artifact stores
Unit & integration testing
Declarative infrastructure
PR Reviews
Login hell
Git providers
iOS & Android builds
Containers
Vulnerability scans
Parallel builds & deployments
Self-hosted
Canary releases
Kubernetes
Shared dependencies
Provisioning environments
Blue/green deployments
Scaling pipeline variations
Monorepos

Most companies
are still stuck with
static
environments

Adopting
Kubernetes
impacts testing
environments like
never before

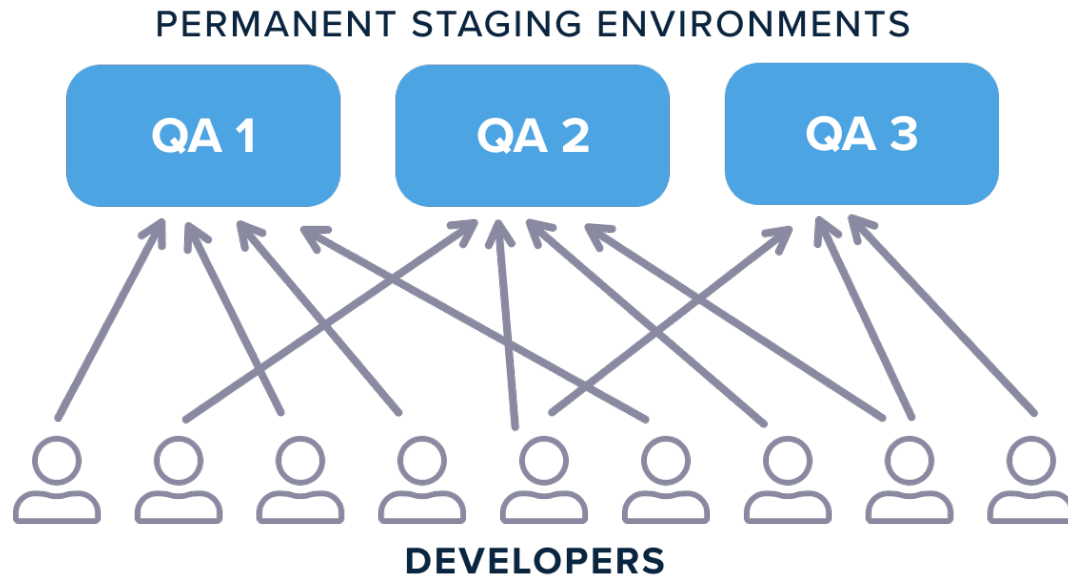
Single staging environment



 Don't



Multiple staging environments

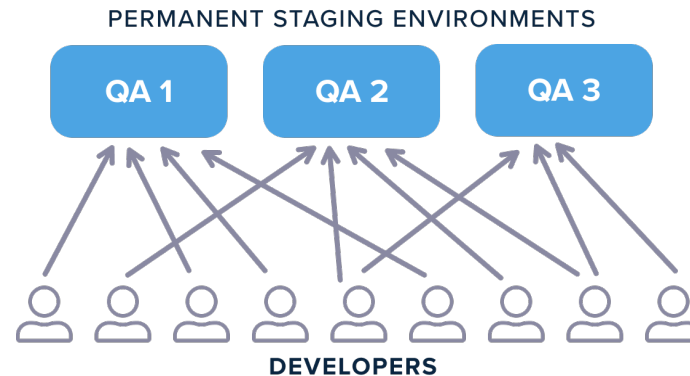


✘ Don't



Multiple staging environments

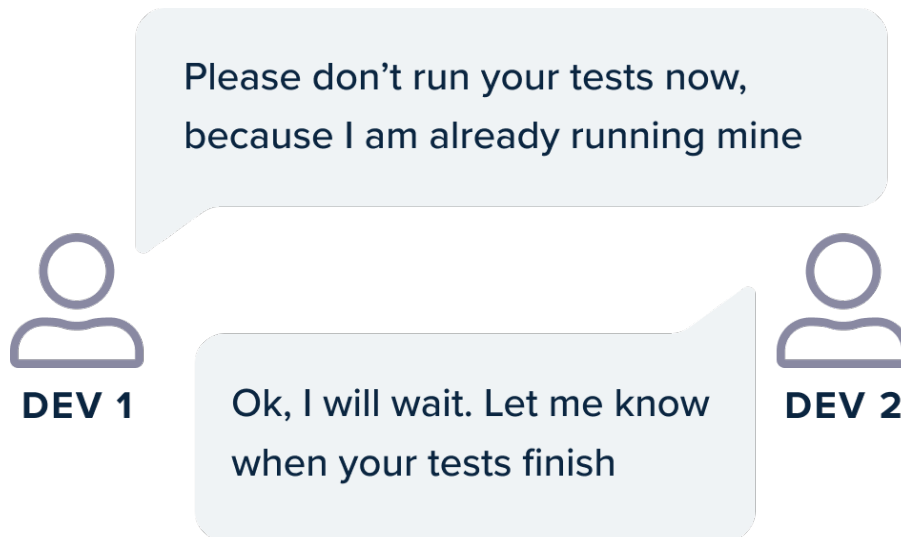
1. Feature conflicts
2. Decreased team velocity
3. Complex clean/setup
4. Wasted resources



✘ Don't



Decreased Velocity



 **Don't**

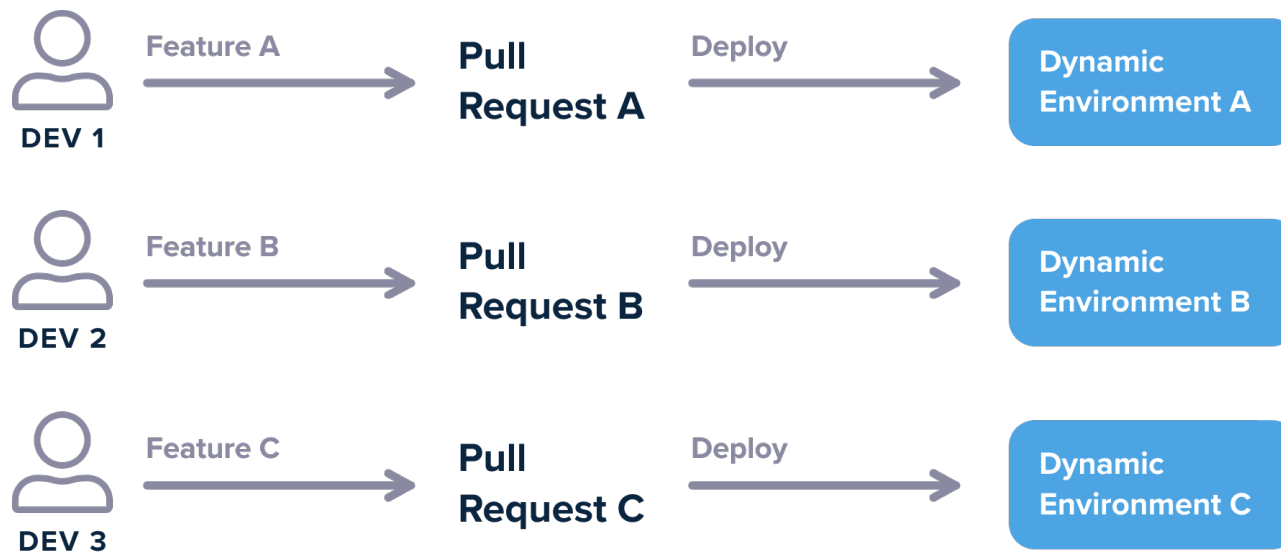


You pay for
resources even
when
environments are
not used

Complex
cleanup/reset
process

Bugs manifest if
wrong
configuration is
present

Dynamic environments

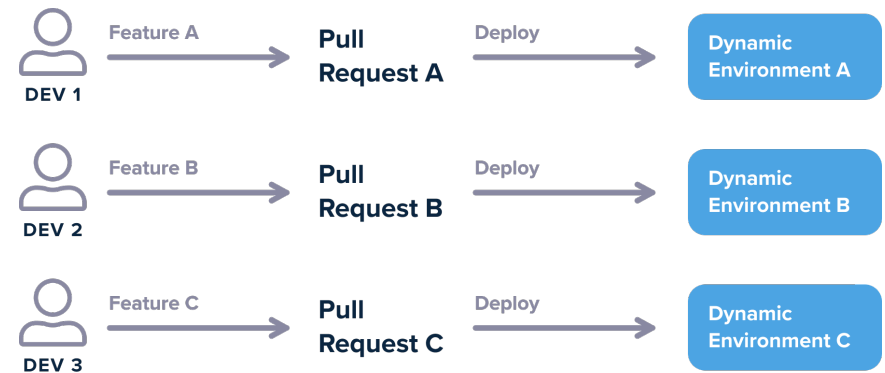


Do Each feature is tested on its own



Dynamic environments

1. Feature isolation
2. Better resource utilization
3. Easy cleanup
4. Adapt to any Git Flow





 **Do** Each feature is tested on its own









Naming patterns (host/path)

- Pr23 -> pr23.staging.com
 - Pr45 -> pr45.staging.com
 - Pr39 -> pr39.staging.com
-
- Pr23 -> staging.com/pr23
 - Pr45 -> staging.com/pr45
 - Pr39 -> staging.com/pr39


Changed title #2


 Open kostis-codefresh wants to merge 1 commit into `main` from `pr-2345` 

 Conversation 1  Commits 1  Checks 0  Files changed 1

 kostis-codefresh commented 9 minutes ago Member  ...

No description provided.

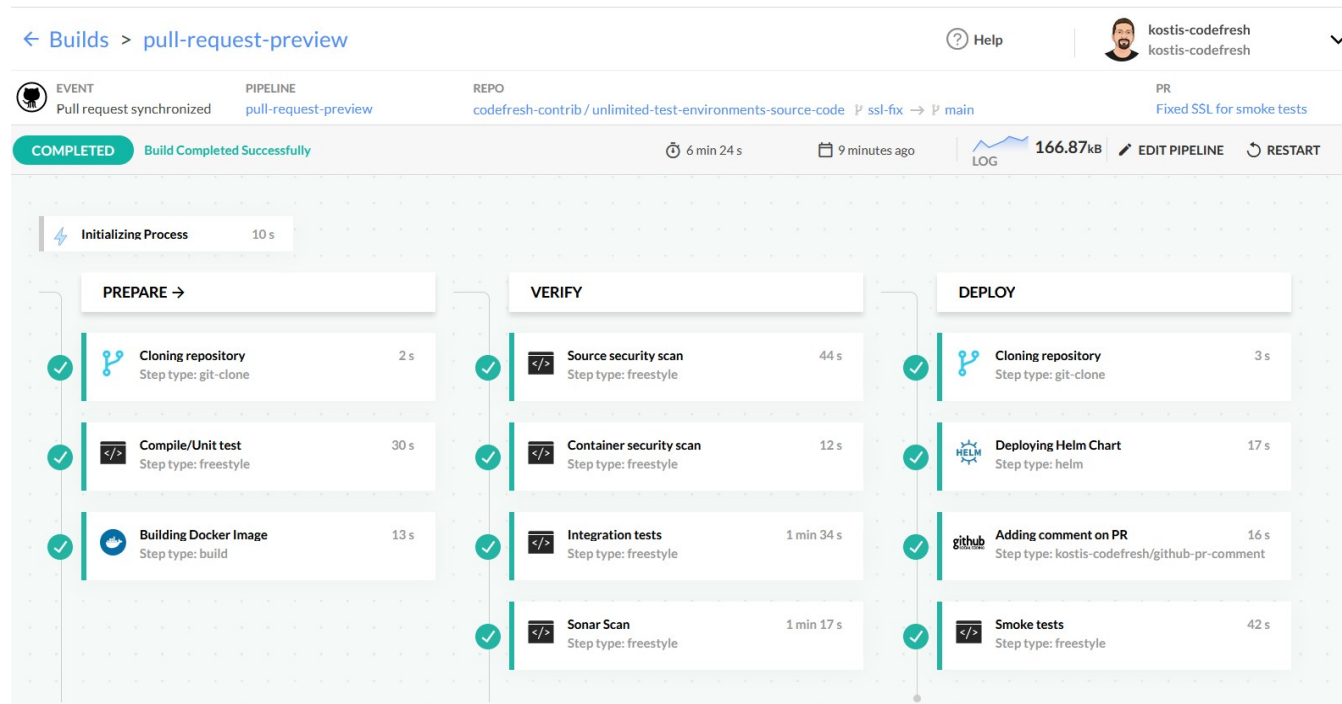
  Changed title Verified  84d4dc9

 kostis-codefresh commented 38 seconds ago Member Author  ...

[CI] Staging environment is at <https://kostis.sales-dev.codefresh.io/pr-2345/>



Quality gates and smoke tests



The screenshot displays a Codefresh pipeline run for a pull request. The pipeline is titled "pull-request-preview" and is in a "COMPLETED" state. The build finished successfully 6 minutes and 24 seconds ago, with a size of 166.87kB. The pipeline consists of three main stages: PREPARE, VERIFY, and DEPLOY. Each stage contains several steps, all of which are marked as successful with green checkmarks.

Stage	Step Name	Step Type	Duration
PREPARE →	Cloning repository	git-clone	2 s
	Compile/Unit test	freestyle	30 s
	Building Docker Image	build	13 s
VERIFY	Source security scan	freestyle	44 s
	Container security scan	freestyle	12 s
	Integration tests	freestyle	1 min 34 s
	Sonar Scan	freestyle	1 min 17 s
DEPLOY	Cloning repository	git-clone	3 s
	Deploying Helm Chart	helm	17 s
	Adding comment on PR	kostis-codefresh/github-pr-comment	16 s
	Smoke tests	freestyle	42 s

<https://codefresh.io/docs/docs/ci-cd-guides/preview-environments/>



Fully automated for devs

1. git checkout master
2. git checkout -b feature-a-b-together
3. git merge feature-a
4. git merge feature-b
5. git push origin feature-a-b-together
6. (open PR in Github)

After some minutes <http://staging.example.com/feature-a-b-together> is up



Use dynamic environments instead of static ones

Everything should be created and destroyed on demand

Demand for flexible processes

Endless integrations

GitOps

Anti-pattern 9: Mixing production and non-production clusters

More complex deployment patterns

Microservices = tons of pipelines

Easy fixes = scalability

Cloud providers
Artifact stores
Unit & integration testing
Declarative infrastructure
PR Reviews
Login hell
Git providers
iOS & Android builds
Containers
Vulnerability scans
Parallel builds & deployments
Self-hosted
Canary releases
Kubernetes
Shared dependencies
Provisioning environments
Blue/green deployments
Scaling pipeline variations
Monorepos

Production should be separate

```
→ Kostis kubectl get ns
NAME                STATUS   AGE
argo-rollouts       Active   91d
argocd               Active   211d
canary              Active   34d
dashboard           Active   39d
default             Active   426d
demo                Active   50d
example             Active   12d
istio-system        Active   53d
komodor             Active   75d
kube-node-lease     Active   426d
kube-public         Active   426d
kube-system         Active   426d
kubonav            Active   39d
kubernetes-dashboard Active   39d
kubeview            Active   166d
production          Active   10s
prom                Active   76d
qa                  Active   2s
staging             Active   5s
→ Kostis |
```

1. Many tutorials use production/staging as different namespaces
2. Use only for demos/POVs
3. Don't do this in real projects



Every pod in every namespace can access every other pod in every other namespace

You can lock down namespaces but it is complex and unneeded

Don't namespace production

1. Resource starvation
2. Cannot easily upgrade cluster
3. Mistakes will happen

```
→ Kostis kubectl get ns
NAME                STATUS  AGE
argo-rollouts       Active  91d
argocd               Active  211d
canary               Active  34d
dashboard            Active  39d
default              Active  426d
demo                 Active  50d
example              Active  12d
istio-system         Active  53d
komodor              Active  75d
kube-node-lease      Active  426d
kube-public          Active  426d
kube-system          Active  426d
kubonav              Active  39d
kubernetes-dashboard Active  39d
kubeview             Active  166d
production           Active  10s
prom                 Active  76d
qa                   Active  2s
staging              Active  5s
→ Kostis |
```



Don't namespace production

1. Developer creates a namespace
2. They deploy feature code and run tests
3. Integrations write dummy data or clean DB
4. A production URL was forgotten inside the code
5. Production DB is destroyed (!!!)

<https://unsplash.com/photos/7x18e4cF-nk>



REJECTED

Suggested clusters

1. Production
2. Shadow/Clone of production but with less resources
3. Developer cluster for feature testing
4. Specialized cluster for load/security testing
5. Cluster for internal tools (e.g. monitoring)
6. Test Cluster for SREs/sys admins



Treat namespaces as soft partitions in the cluster.

Production should run on its own cluster

Treating namespaces as a security measure is a recipe for disaster

Use at least 2 clusters (one is prod)

Demand for flexible processes

Endless integrations

GitOps

Anti-pattern 10: Deploying without memory and CPU limits

Containers

Microservices = tons of pipelines

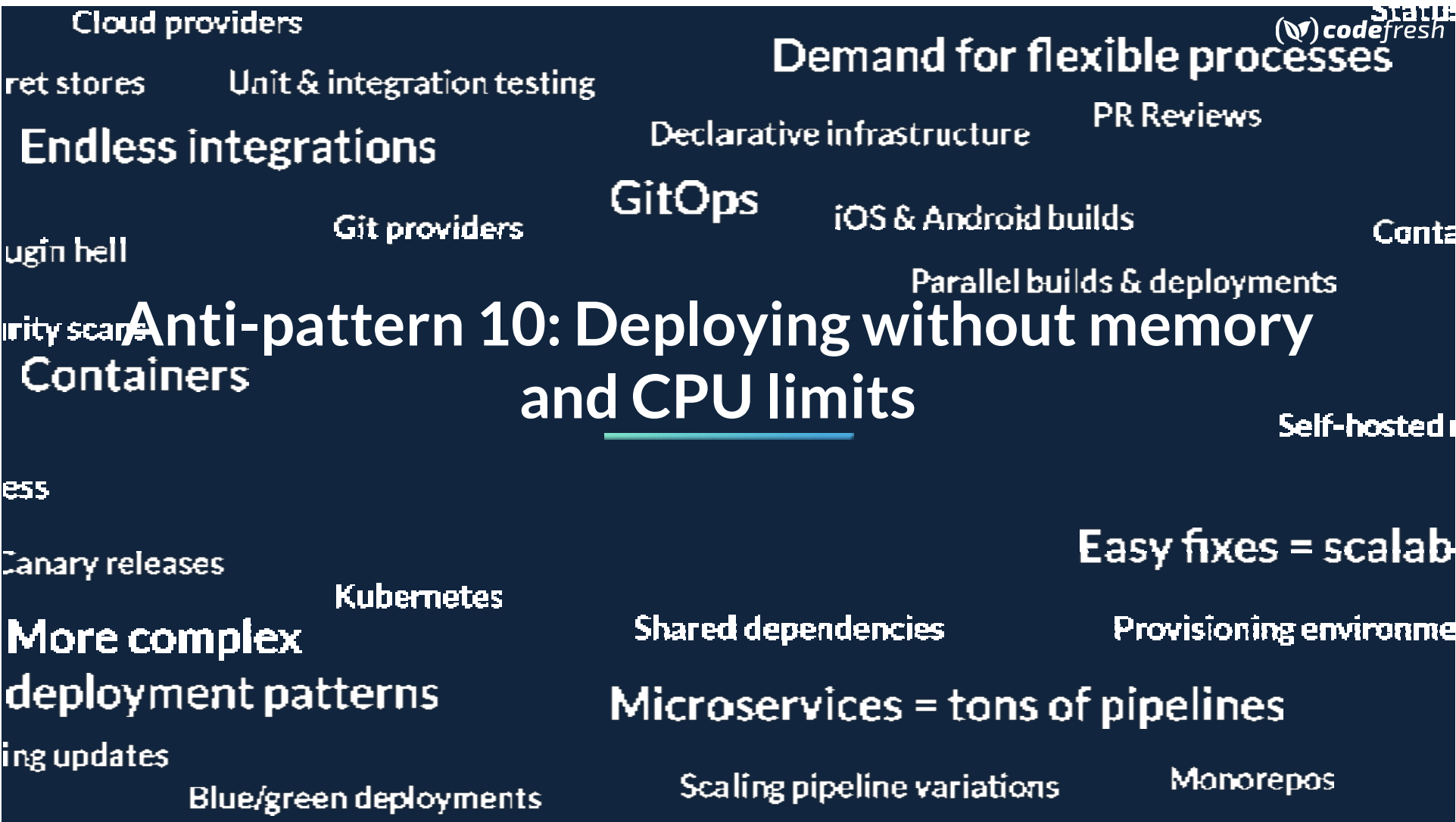
More complex deployment patterns

Easy fixes = scalability

Blue/green deployments

Scaling pipeline variations

Monorepos



By default an application deployed on Kubernetes has no resource limits

This means that a single rogue application can overwhelm the whole cluster

As a developer you need to give some hints to the Kubernetes admin for resource consumption

As an operator you need to make sure that all applications have limits (and monitor them)

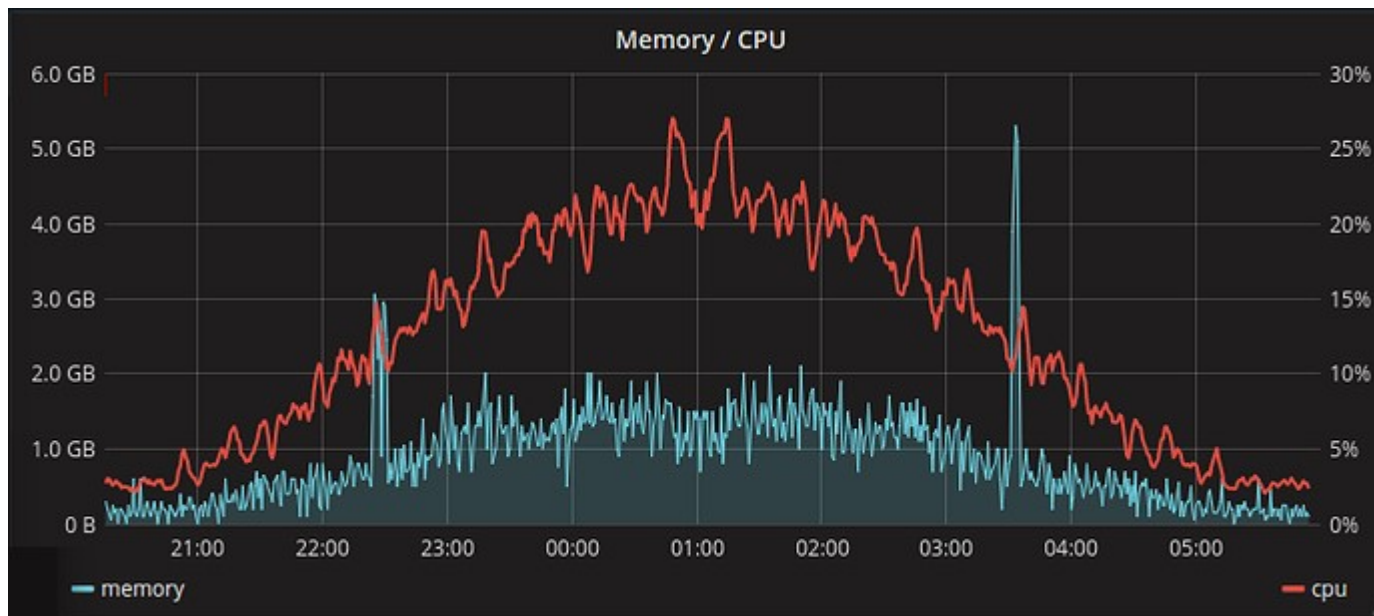
Setting resource limits

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: app
    image: images.my-company.example/app:v4
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
  - name: log-aggregator
    image: images.my-company.example/log-aggregator:v6
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

<https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/>



Don't use the average



REJECTED

How to define correct limits

1. Average consumption is “just average”
2. Take into account traffic bursts
3. Perform load testing
4. Learn about minimum/average/maximum
5. Fix your memory leaks 😊
6. Start with a guess and iterate on it (using metrics)
7. Check your programming language documentation



If you put large values you are wasting resources and increase your bills

If you put small values, your application performance will suffer (and the cluster will possibly kill your app)

Use your metrics



Take it to the next level



<https://unsplash.com/photos/pqHRNS8Mojc>

Cloud advantages

Embrace autoscaling

Cluster autoscaling
(increase your nodes?)

Horizontal autoscaling
(increase your pods)

Vertical autoscaling
(increase your resource
limits)



Just watch your apps auto-scale



<https://unsplash.com/photos/vvLBPW3uS4Q>

All applications should have resource limits (even non-prod clusters)

Make use of autoscaling facilities

Let your cluster work for you

Demand for flexible processes

Endless integrations

GitOps

Anti-pattern 11: Misusing health probes

More complex deployment patterns

Microservices = tons of pipelines

Easy fixes = scalability

Cloud providers
Artifact stores
Unit & integration testing
PR Reviews
Declarative infrastructure
Git providers
iOS & Android builds
Parallel builds & deployments
Container
Vulnerability scans
Self-hosted
Canary releases
Kubernetes
Shared dependencies
Provisioning environments
Blue/green deployments
Scaling pipeline variations
Monorepos

All applications
should have
resource limits
(even non-prod
clusters)

All applications
should have
health probes

(some coding
required)

Health endpoints

Kubernetes queries your app



Startup probe.

Readiness probe.

Liveness probe



Setting probe endpoints

- Startup probes
- Readiness probe
- Liveness probe

- Custom command
- Http endpoints
- Tcp port check

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
    name: liveness-http
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/liveness
    args:
    - /server
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
        httpHeaders:
        - name: Custom-Header
          value: Awesome
      initialDelaySeconds: 3
      periodSeconds: 3
```

<https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/>



Learn what the probes do



<https://unsplash.com/photos/kBVreEYUzp8>

Startup probe

1. Runs only once
2. Checks the initial boot of your application
3. Kubernetes will not send traffic to your app
4. Used in combination with liveness probe
5. Mostly for legacy applications



Readiness probe

1. Runs all the time
2. Checks if your application can respond to traffic
3. If it fails Kubernetes will **stop sending traffic** (and try again later)
4. Used when your application needs time to process requests
5. Could also check for external dependencies
6. Should be separate than liveness probe



Liveness probe

1. Runs all the time
2. Checks if your application is working (and not deadlocked)
3. If it fails Kubernetes **will restart the app**
4. Watchdog for stuck/deadlocked applications
5. Should NOT check external dependencies
6. Should be separate than readiness probe



Implement the HTTP endpoints



<https://unsplash.com/photos/tG36rvCeqng>

Common mistakes

- Not accounting for external services (e.g. DB) in the readiness probe
- Using the same endpoint for readiness and liveness
- Using the existing health endpoint that was created for a Virtual machine
- Not using the Health facilities of your framework
- Creating too complex healthchecks that cause denials of service
- Creating cascading failures (external services in liveness probe)



Check your programming framework

< Back to index

- 1. Enabling Production-ready Features
- 2. Endpoints
 - 2.1. Enabling Endpoints
 - 2.2. Exposing Endpoints
 - 2.3. Securing HTTP Endpoints
 - 2.4. Configuring Endpoints
 - 2.5. Hypermedia for Actuator Web Endpoints
 - 2.6. CORS Support
 - 2.7. Implementing Custom Endpoints
 - 2.8. Health Information
 - 2.9. Kubernetes Probes**
 - 2.9.1. Checking External State with Kubernetes Probes
 - 2.9.2. Application Lifecycle and Probe States
- 2.10. Application Information
- 3. Monitoring and Management over HTTP
- 4. Monitoring and Management over JMX
- 5. Loggers
- 6. Metrics
- 7. Auditing
- 8. HTTP Tracing
- 9. Process Monitoring
- 10. Cloud Foundry Support
- 11. What to Read Next

2.9. Kubernetes Probes

Applications deployed on Kubernetes can provide information about their internal state with **Container Probes**. Depending on your Kubernetes configuration, the kubelet will call those probes and react to the result.

Spring Boot manages your **Application Availability State** out-of-the-box. If deployed in a Kubernetes environment, actuator will gather the "Liveness" and "Readiness" information from the `ApplicationAvailability` interface and use that information in dedicated **Health Indicators**: `LivenessStateHealthIndicator` and `ReadinessStateHealthIndicator`. These indicators will be shown on the global health endpoint (`/actuator/health`). They will also be exposed as separate HTTP Probes using **Health Groups**: `/actuator/health/liveness` and `/actuator/health/readiness`.

You can then configure your Kubernetes infrastructure with the following endpoint information:

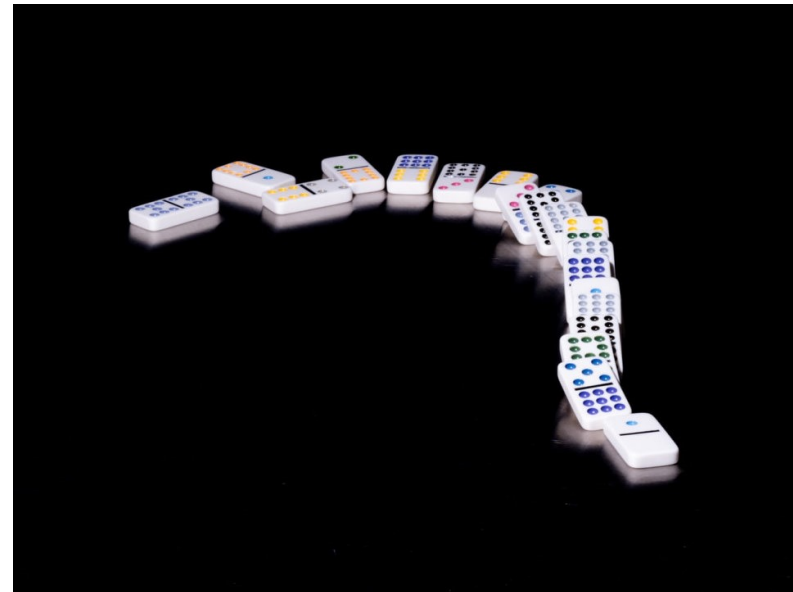
```
livenessProbe:
  httpGet:
    path: /actuator/health/liveness
    port: <actuator-port>
    failureThreshold: ...
    periodSeconds: ...

readinessProbe:
  httpGet:
    path: /actuator/health/readiness
    port: <actuator-port>
    failureThreshold: ...
    periodSeconds: ...
```

<https://docs.spring.io/spring-boot/docs/current/reference/html/actuator.html>

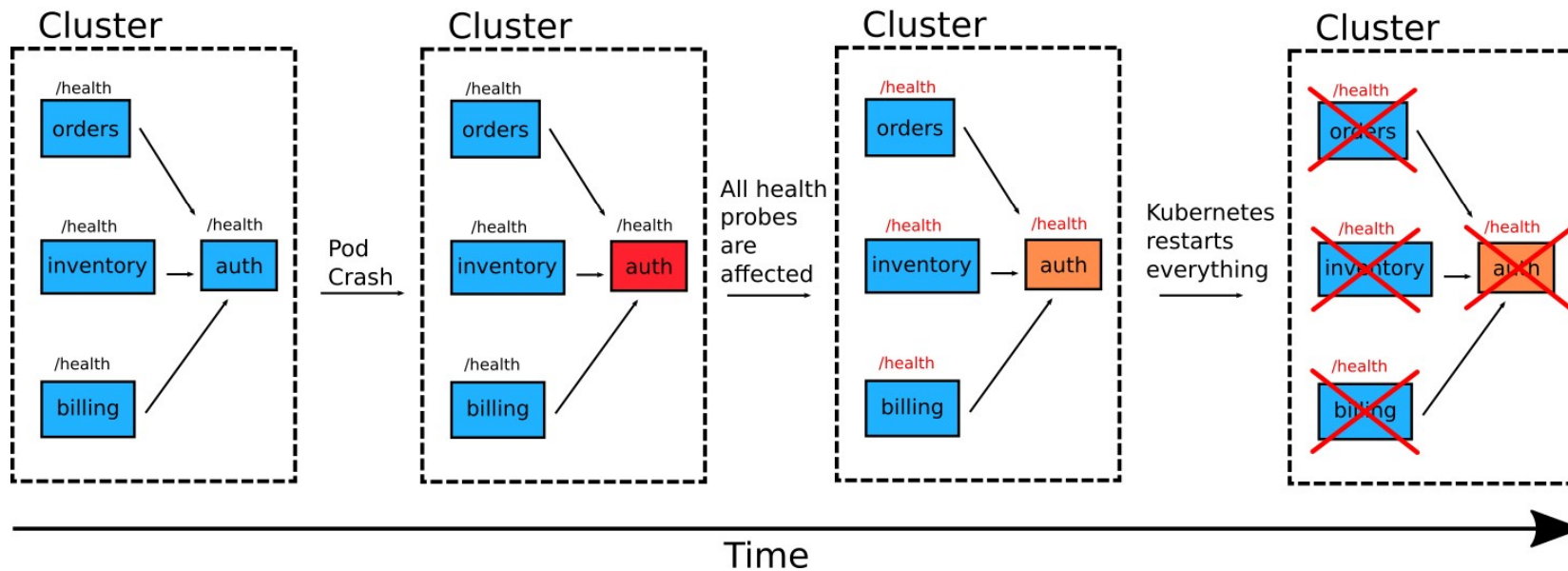


Cascading failures



<https://unsplash.com/photos/Em2hPK55o8g>

Using dependencies in liveness probes creates cascading failures **Don't**



Demand for flexible processes

Endless integrations

GitOps

Containers Anti-pattern 12: Not using Helm

More complex deployment patterns

Microservices = tons of pipelines

Easy fixes = scalability

Cloud providers
Artifact stores
Unit & integration testing
Declarative infrastructure
PR Reviews
Git providers
iOS & Android builds
Parallel builds & deployments
Container
Security scans
Self-hosted
Canary releases
Kubernetes
Shared dependencies
Provisioning environments
Blue/green deployments
Scaling pipeline variations
Monorepos

Demand for flexible processes

Endless integrations

GitOps

Anti-pattern 13: Not having deployment metrics

Containers

Easy fixes = scalability

More complex deployment patterns

Microservices = tons of pipelines

Cloud providers
Artifact stores
Unit & integration testing
Declarative infrastructure
PR Reviews
Login hell
Git providers
iOS & Android builds
Parallel builds & deployments
Priority scans
Kubernetes
Shared dependencies
Provisioning environments
Canary releases
Blue/green deployments
Scaling pipeline variations
Monorepos
Self-hosted

Demand for flexible processes

Cloud providers
Secret stores
Unit & integration testing

Endless integrations

Declarative infrastructure
PR Reviews

GitOps

iOS & Android builds

Login hell

Git providers

Containers

Security scans

Anti-pattern 14: Not having a strategy for secrets

Parallel builds & deployments

Containers

Self-hosted

CI/CD

Easy fixes = scalability

Canary releases

Kubernetes

Shared dependencies

Provisioning environments

More complex
deployment patterns

Microservices = tons of pipelines

Rolling updates

Blue/green deployments

Scaling pipeline variations

Monorepos

Demand for flexible processes

Endless integrations

GitOps

Anti-pattern 15: Attempting to go all in
Kubernetes (even with stateful loads)

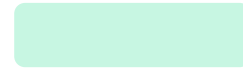
More complex
deployment patterns

Microservices = tons of pipelines

Easy fixes = scalability

Cloud providers
Artifact stores
Unit & integration testing
Declarative infrastructure
PR Reviews
Git providers
iOS & Android builds
Containers
Parallel builds & deployments
Self-hosted
Kubernetes
Shared dependencies
Provisioning environments
Blue/green deployments
Scaling pipeline variations
Monorepos

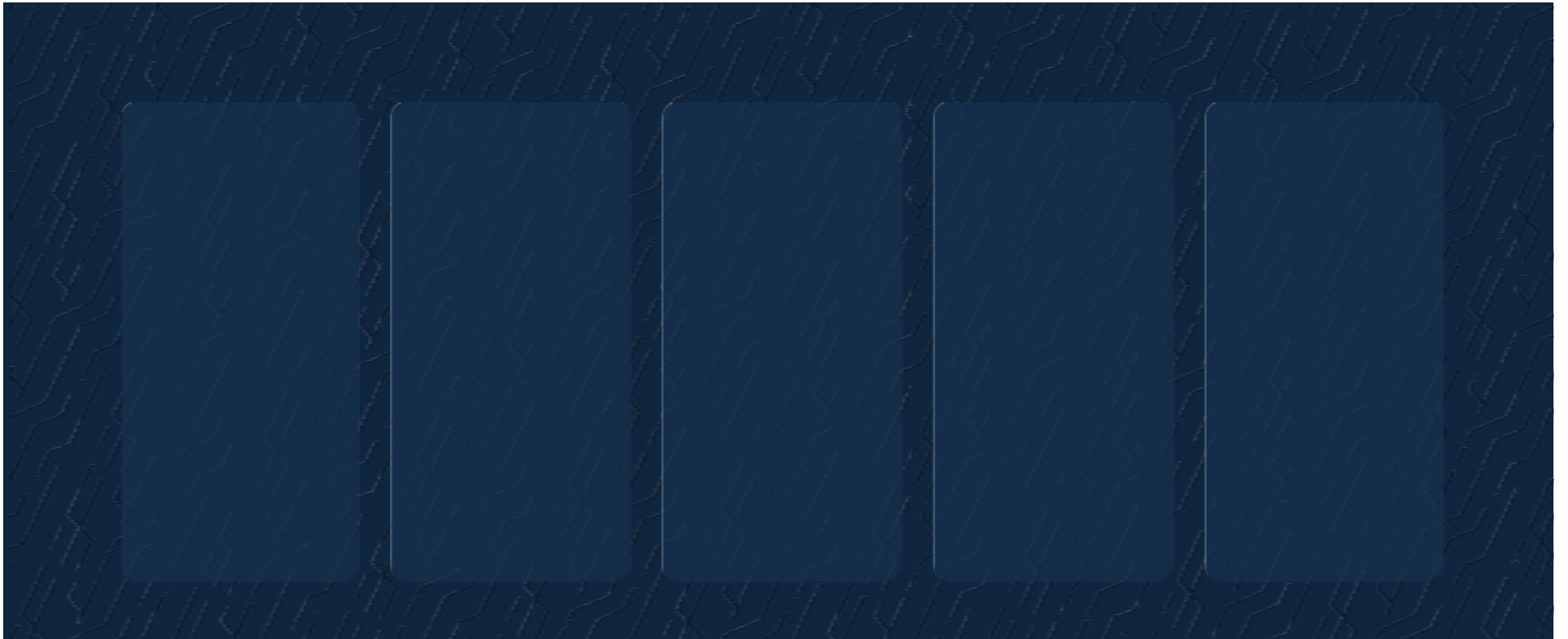
d
f
d
f
f



Sample Agenda

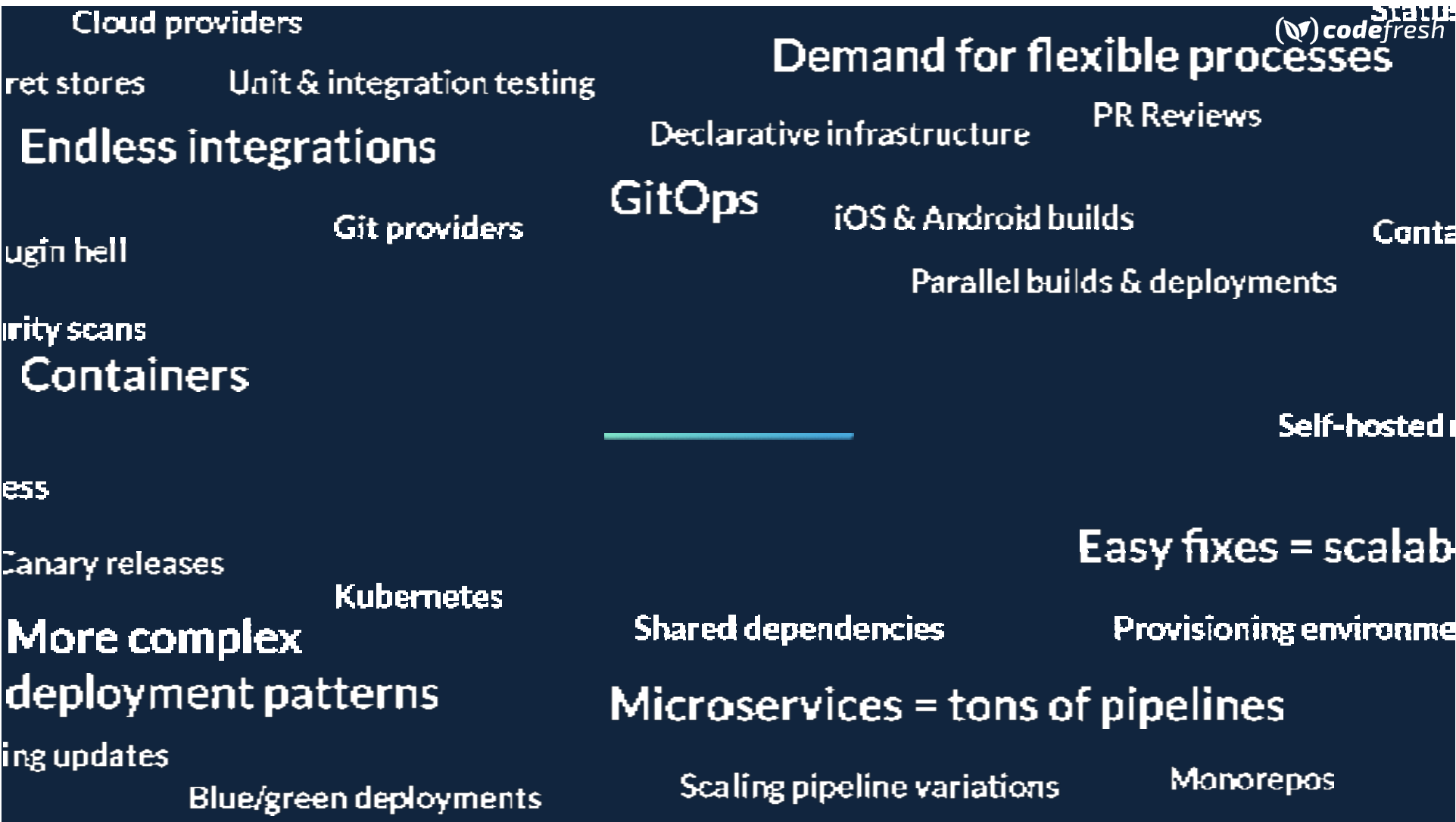
- 1** **10 MINUTES**
Introduction
- 2** **20 MINUTES**
Section 2
- 3** **15 MINUTES**
Live Demo
- 4** **10 MINUTES**
Q/A







Demand for flexible processes



Meetup title

Kubernetes Anti-patterns

Kostis Kapelonis













The #1 GitOps automation platform for Kubernetes apps

 **codefresh**



The modern approach to
DevOps automation

Open a FREE account today at codefresh.io

et stores Unit & integration testing PR Reviews
Endless integrations Declarative infrastructure
 Git providers **GitOps** iOS & Android builds
 gin hell Parallel builds & deployment
 ty scans
Containers *ICONS TO USE*
 ss
 nary releases Easy fixes =
More complex Kubernetes Shared dependencies Provisioning
deployment patterns **Microservices = tons of pipelines**
 g updates Scaling pipeline variations Monorepo
 Blue/green deployments

