- DO -

- DON’T -

Kubernetes

Deployment antipatterns

codefresh

Table of Contents

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.

We soon realized that we must also create a similar guide for Kubernetes deployments. This will hopefully give you the whole picture of how to create a container image **and** how to properly deploy it (or at least warn you of some common pitfalls).

Notice that in this guide we talk specifically about **application deployments** on Kubernetes and not Kubernetes clusters themselves. This means that we assume that the Kubernetes cluster is already there (and it is properly set up) and you simply want to deploy an application on it. In the future, we will complete the trilogy by also documenting anti-patterns for the creation of the clusters (i.e. talk about the infrastructure level instead of the application level).

Unlike other guides that simply complain about how things can go wrong, we always associate each anti-pattern with the respective solution. This way you can actually check your own deployment process and fix any issues without hunting down extra information.

Here is the list of bad practices that we will examine today:

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)

By the way, if you still haven't looked at the container anti-patterns guide, you should do this now, as some of the bad practices mentioned above will reference it.

Anti-pattern 1

Using containers with the latest tag in Kubernetes deployments

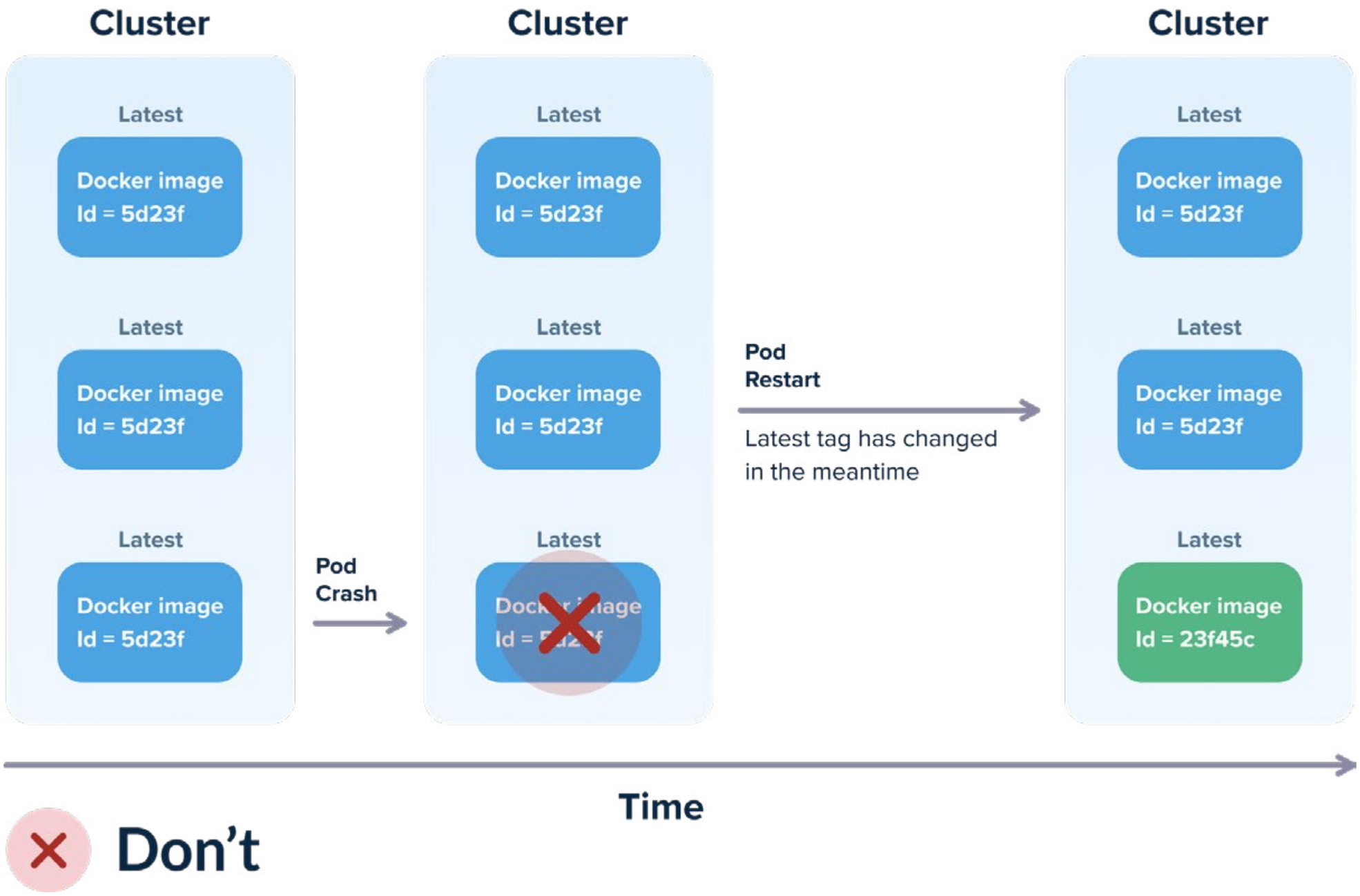
If you have spent any time building containers, this should come as no surprise. Using the “latest” tag for Docker images is a bad practice on its own as “latest” is just a name of the tag and it doesn’t actually mean “most recent” or “lastly built”. Latest is also the default tag if you don’t specify one when talking about a container image.

Using the “latest” tag in a Kubernetes deployment is even worse as by doing this you don’t know what is deployed in your cluster anymore.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-bad-deployment
spec:
  template:
    metadata:
      labels:
        app: my-badly-deployed-app
    spec:
      containers:
      - name: dont-do-this
        image: docker.io/myusername/my-app:latest
```

If you apply this deployment you have now lost all information on which container tag is actually deployed. Container tags are mutable, so the “latest” tag does not really mean anything to anyone. Maybe this container image was created 3 minutes ago, maybe it was 3 months ago. You will need to hunt down all your logs for your CI system or even download the image locally to inspect it so that you know what version it contains.

The “latest” tag is even more dangerous if you couple it with an always pull policy. Let’s say that your pod is dead, and Kubernetes decides to restart it in order to make it healthy (remember that is why you are using Kubernetes in the first place).



Kubernetes will reschedule the pod and if your pull policy allows this, it will pull the "latest" image again from your Docker registry! This means that, if in the meantime the "latest" tag has changed, you now have a new version in this particular pod which is different from what the other pods have. In most cases, this is not what you want.

It also goes without saying that performing "deployments" by killing pods manually and waiting for them to pull again your "latest" image is a recipe for success (if you do happen to use this form of "deployment").

The correct deployment format in Kubernetes should follow a proper tagging strategy. The specific strategy is not that important as long as you have one.

Some suggestions are:

- Using tags with Git hashes (e.g. docker.io/myusername/my-app:acef3e). This is straightforward to implement but may be overkill since a Git hash is not easily readable by non-technical people.
- Using tags with the application version following semantic versions (e.g. docker.io/myusername/my-app:v1.0.1). This method has many advantages for both developers and non-developers and is our personal recommendation.
- Using tags that signify a consecutive number such as a build number or build date/time. This format is very hard to work with but can be easy to adopt with legacy applications.

The important thing is that you should agree that container tags should be treated as immutable. A Docker image that is marked as v2.0.5 should be created only once and should be promoted from one environment to another.

If you see a deployment that uses the image with tag v2.0.5, you should be able to...

- pull this image locally and be certain that it is the exact same one that is running on the cluster;
- easily track down the Git hash that it created it.

If your deployment workflows depend in any way on using “latest” tags, you are sitting on a time bomb.

Anti-pattern 2

Baking the configuration inside container images

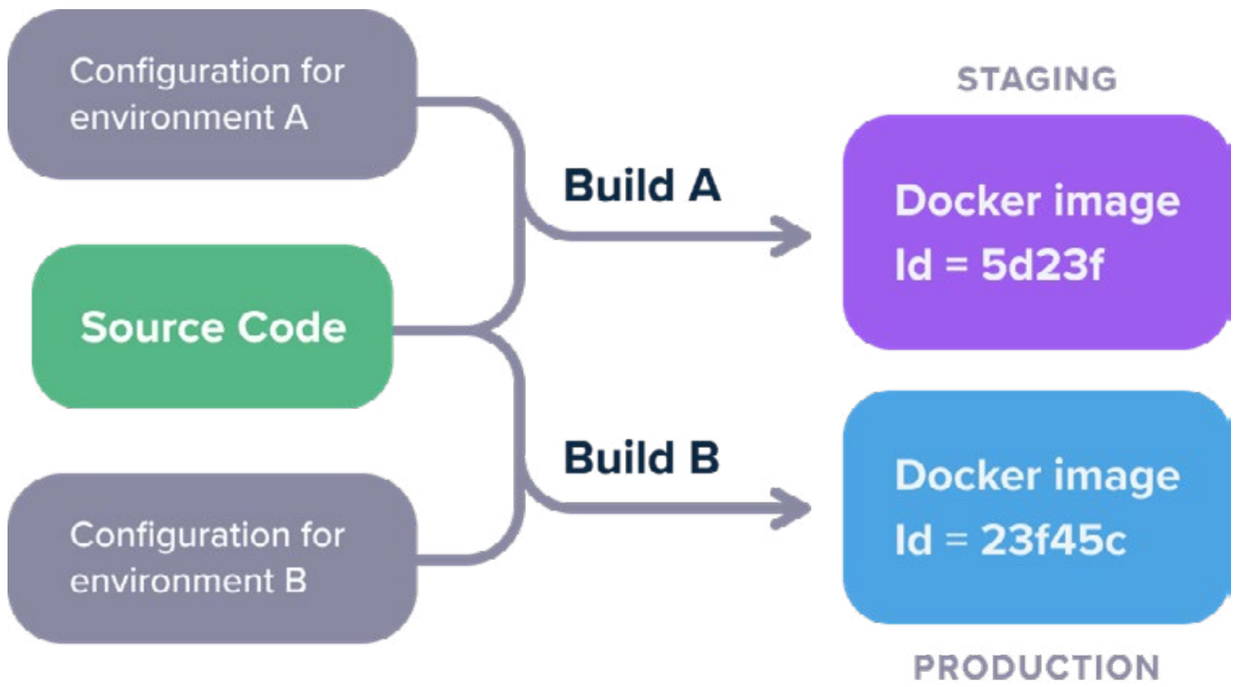
This is actually another anti-pattern that comes from building container images. Your images should be “generic” in the sense that they should be able to run in any environment.

This was a good practice even before containers appeared and is already documented as part of the 12-factor app. Your container images should be built once and then promoted from one environment to another. No configuration should be present in the container itself.

If your container image:

- has hardcoded IP address
- contains passwords and secrets
- Mentions specific URLs to other services
- Is tagged with string such as “dev”, “qa”, “production”

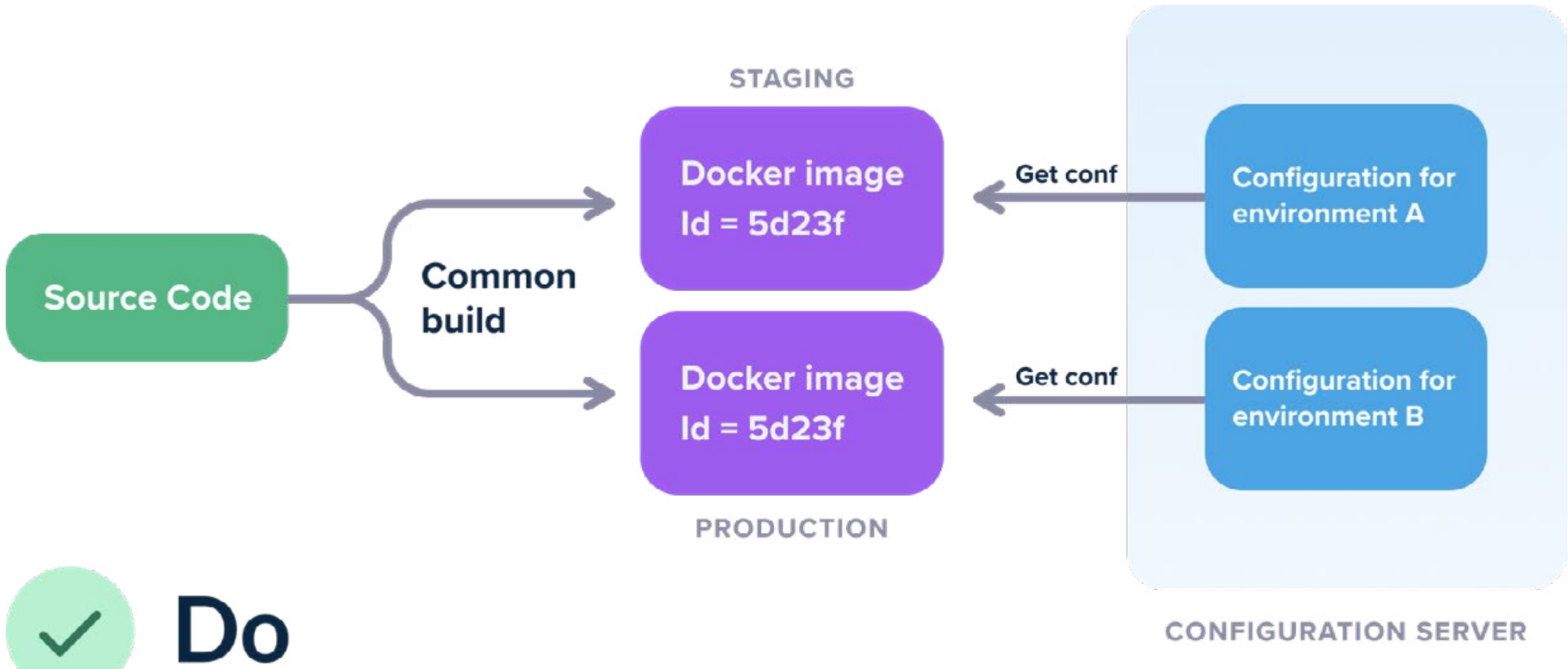
..then you have fallen into the trap of building environment-dependent container images.



✕ Don’t

This means that for each different environment, you have to rebuild your image so you deploy to production something different than what was tested before.

The solution to this problem is very simple. Create “generic” container images that know nothing about the environment they are running on. For configuration, you can use any external method such as Kubernetes configmaps, Hashicorp Consul, Apache Zookeeper, etc.



Do

Now you have a single image that gets deployed in all your clusters. It is much easier to understand what it contains and how it was created.

A secondary advantage is that if you do need to change the configuration on your cluster, you can simply change the external configuration system instead of rebuilding the full container image from scratch. Depending on the programming language and framework that you use, you can even update the live configuration without any restarts or redeployments.

Anti-pattern 3

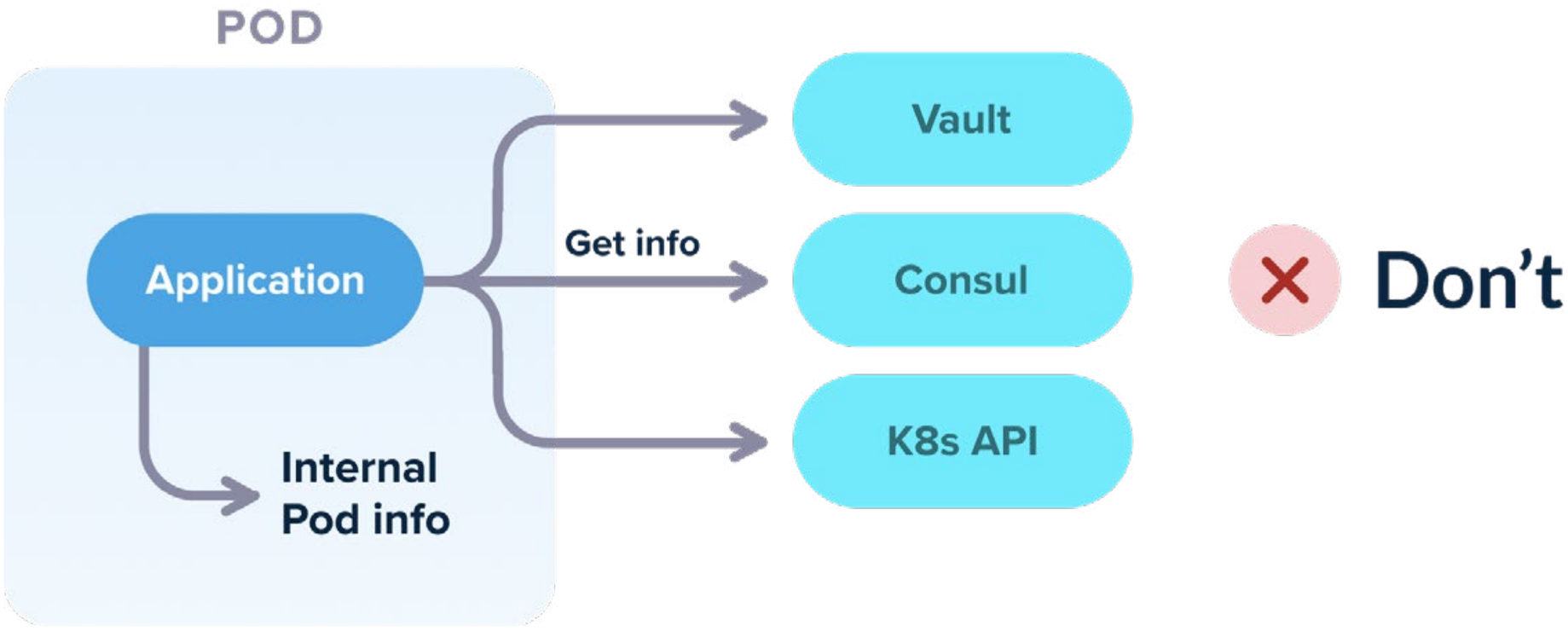
Coupling applications with Kubernetes features/services for no reason

In the previous section, we explained why you should not store configuration inside a container and how a container should not know anything about the cluster it is running on.

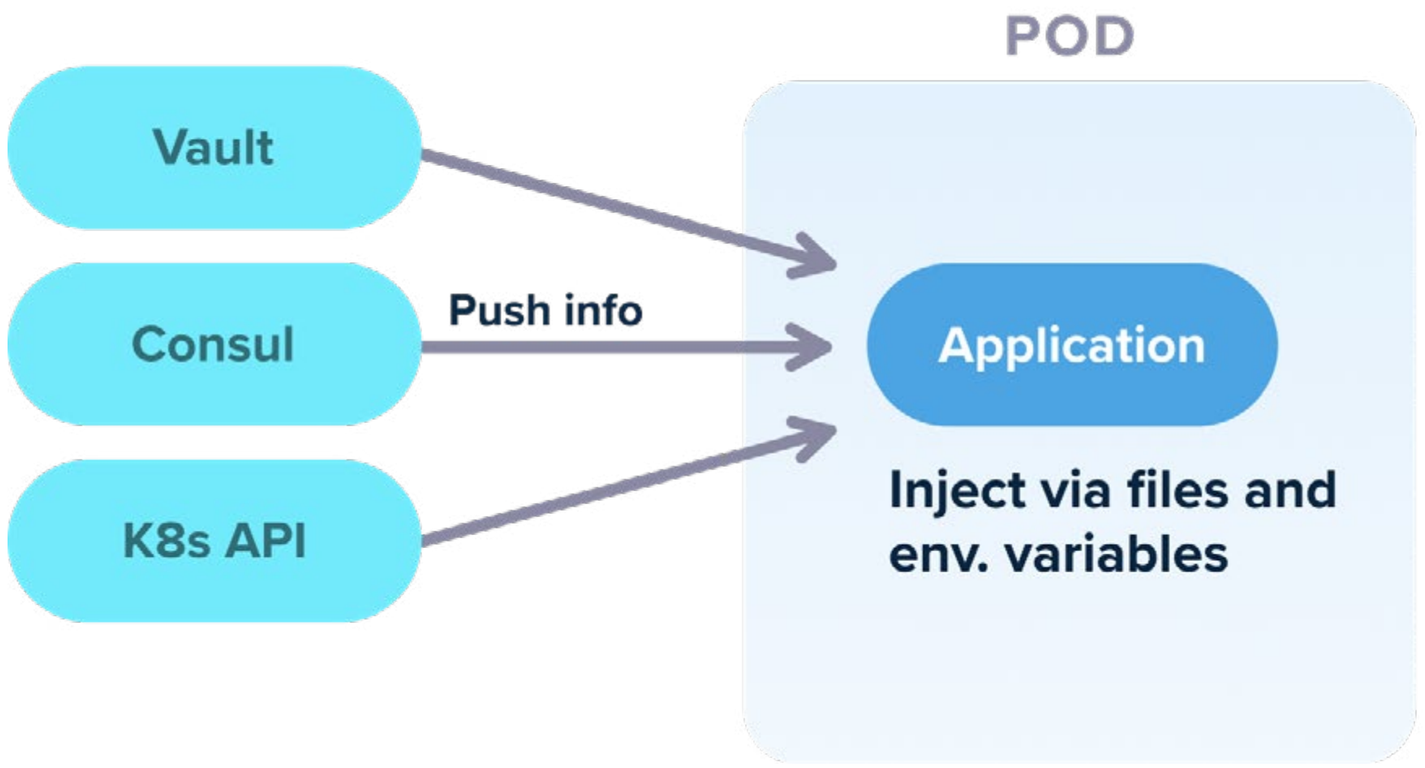
We can take this to an extreme by **requiring each container to not even know that it is running inside Kubernetes** at all. Unless you are developing an application that is destined to handle a cluster, your application should not tamper with the Kubernetes API or other external services that are assumed to be inside the cluster.

This scenario is very common with overenthusiastic teams that adopt Kubernetes and fail to isolate their application from the cluster. Some classic examples are application that:

- expect a certain volume configuration for data sharing with other pods
- expect a certain naming of services/DNS that is set up by Kubernetes networking or assume the presence of specific open ports
- get information from Kubernetes labels and annotations
- query their own pod for information (e.g to see what IP address they have)
- need an init or sidecar container in order to function properly even in local workstations
- call other Kubernetes services directly (e.g. using the vault API to get secrets from a Vault installation that is assumed to also be present on the cluster)
- read data from a local kube config
- use directly the Kubernetes API from within the application



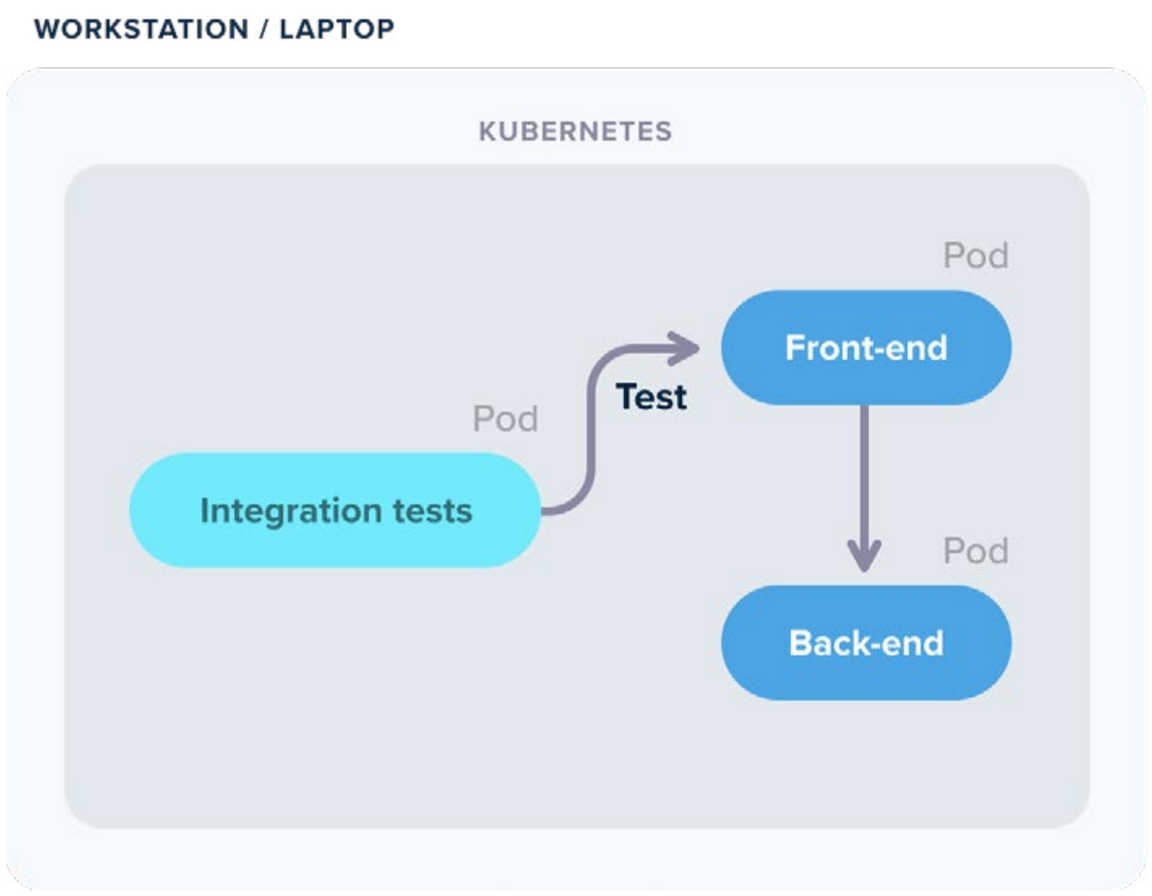
Now of course, if your application is Kubernetes specific (let's say that you are creating an autoscaler or operator) then it indeed needs to access Kubernetes services directly. But for the other 99% of standard web applications out there, your application should be completely oblivious to the fact that it is running inside Kubernetes.



Do

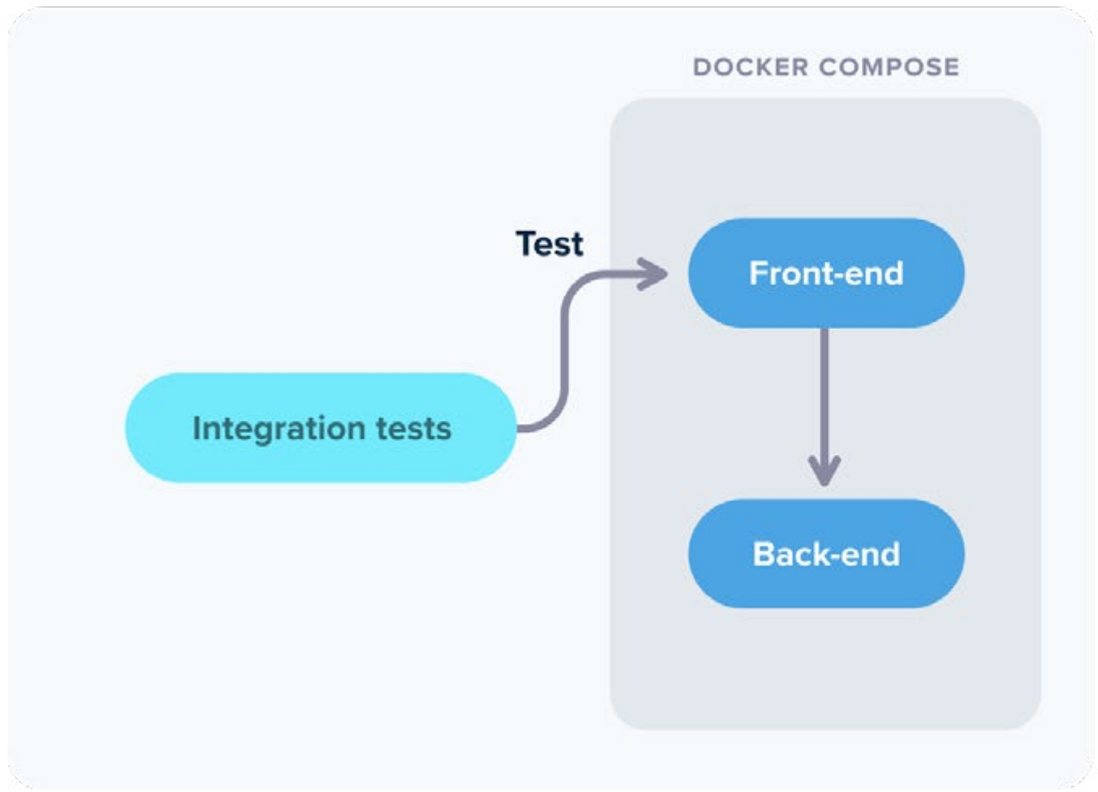
The litmus test that shows if your application is tied to Kubernetes or not is the ability to run your application with Docker compose. If creating a Docker compose file for your app is dead simple, then it means that you are following the 12-factor app principles and your application can be installed on any cluster without the need for special settings.

It is important to also understand the premise of local Kubernetes testing. There are several solutions today for local Kubernetes deployments (minikube, microk8s, kind etc). You might look at these solutions and think that if you are a developer working on an application that is deployed to Kubernetes you also need to run Kubernetes yourself.



Don’t

This could not be further from the truth. If your application is correctly designed you shouldn’t need Kubernetes for running integration tests locally. Just launch the application on its own (with Docker or Docker-compose) and hit it directly with the tests.



Do

It is ok if some of your dependencies are running on an external Kubernetes cluster. But the application itself should not need to run inside Kubernetes while you are testing its functionality.

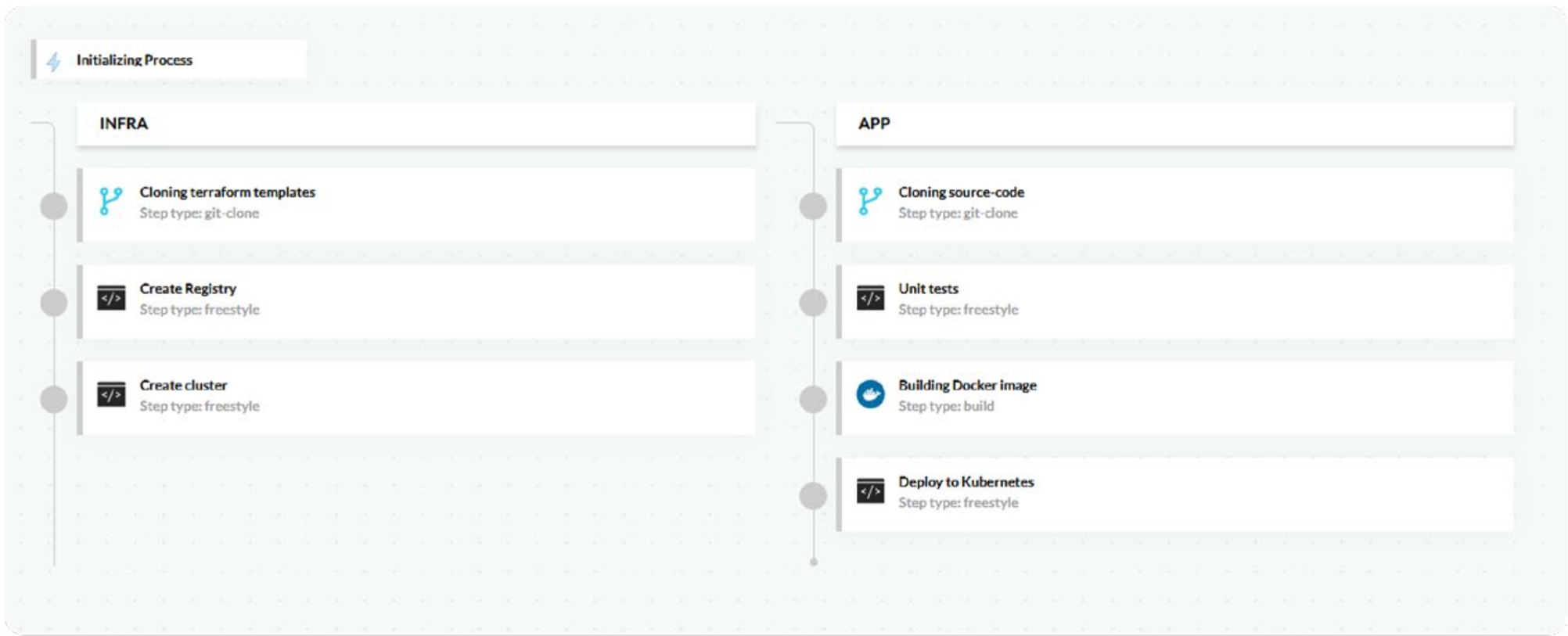
Alternatively, you can also use any of the dedicated solutions for local Kubernetes development such as Okteto, garden.io, and tilt.dev.

Anti-pattern 4

Mixing application deployment with infrastructure deployment

In recent years, the rise of Terraform (and similar tools like Pulumi) has given rise to the "infrastructure as code" movement that allows teams to deploy infrastructure in the same way as code.

But just because you can deploy infrastructure in a pipeline, doesn't mean that infrastructure and application deployment should happen all at once.



Don't

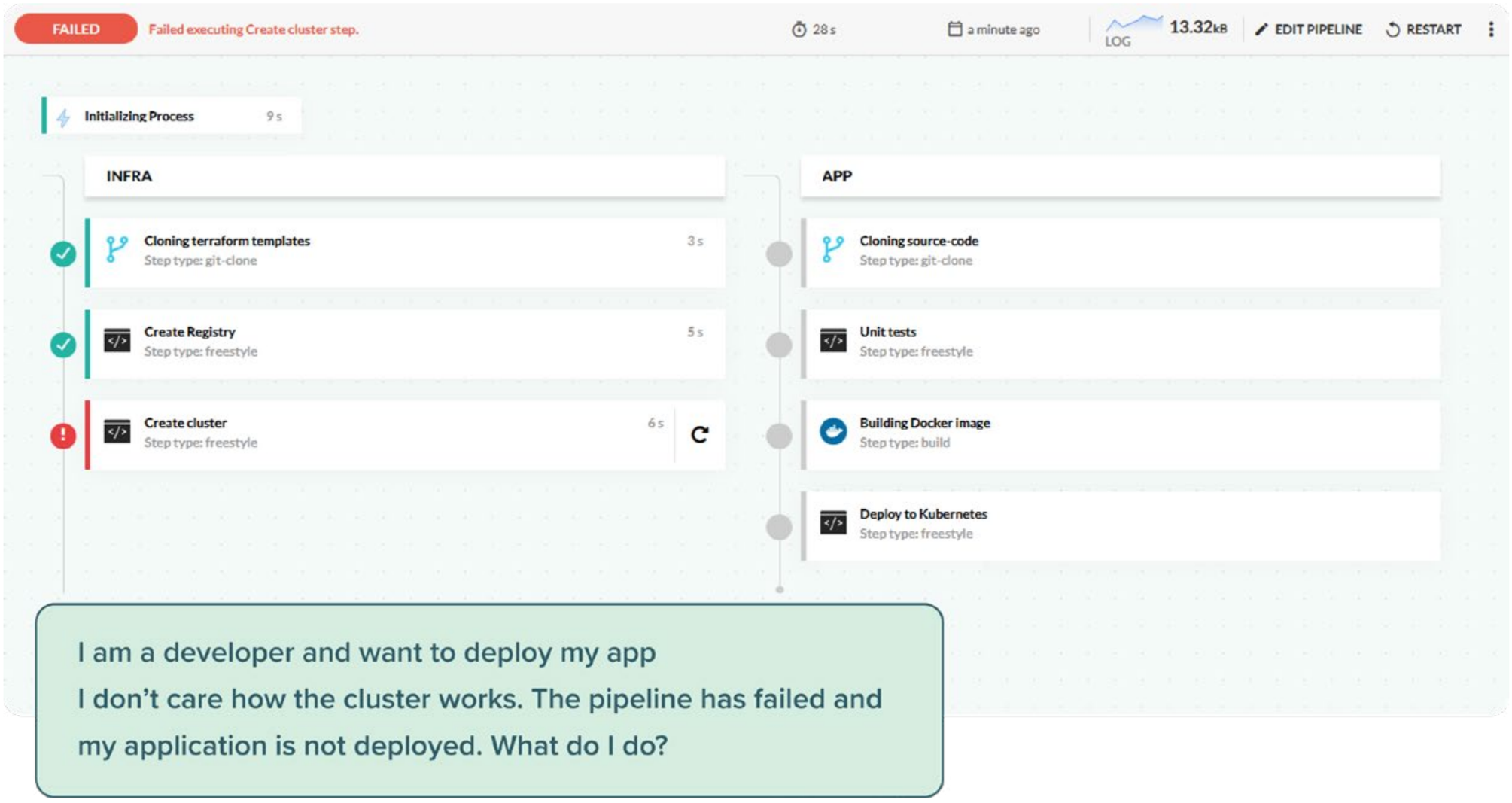
We see a lot of teams that create a single pipeline that both creates infrastructure (i.e. creating a Kubernetes cluster, container registry, etc.) and then deploys an application on top of it.

While this works great in theory, (as it means you are starting from scratch with each deployment) it is pretty wasteful in terms of resources and time.

In most cases, the application code will change much faster than the infrastructure. It is hard to generalize for all companies, but in most cases the rate the application changes might be 2x-10x more often than the infrastructure.

If you have a single pipeline that does both, then you are destroying/creating infrastructure that never changed simply because you want to deploy a new application version.

A pipeline that deploys everything (infra/app) might take 30 minutes, while a pipeline that deploys only the application might take only 5 minutes. You are spending 25 extra minutes on each deployment for no reason at all when the infrastructure has not changed.

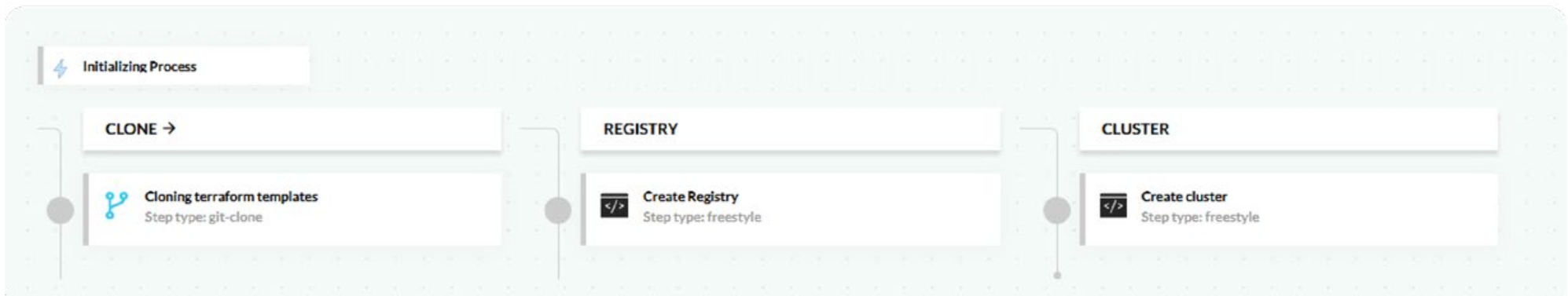


The second disadvantage is that if the single pipeline breaks, it is not clear who must look at it. If I am a developer and want to deploy my application on Kubernetes, I am not interested in Terraform errors, virtual networks, or storage volumes.

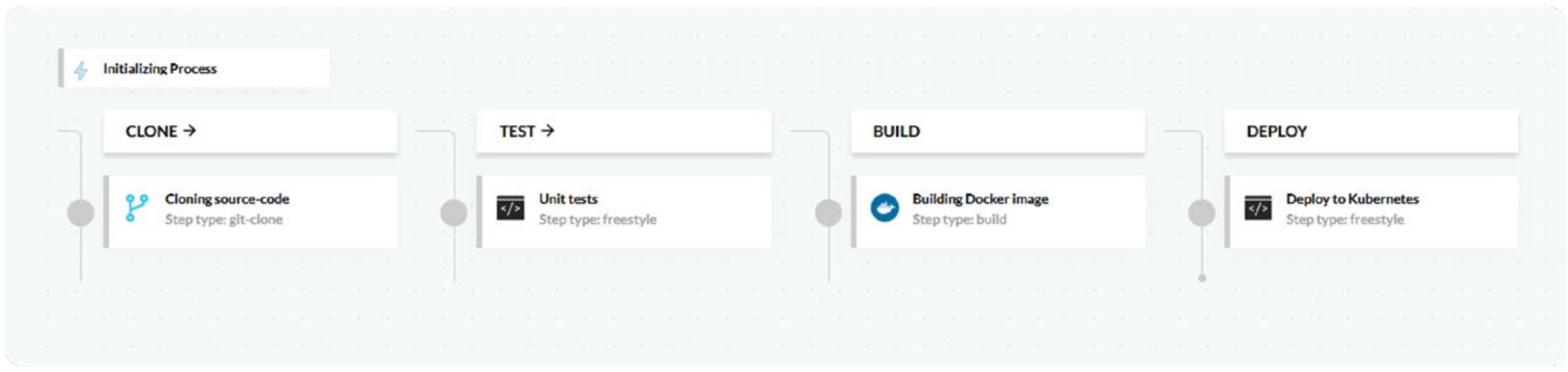
The whole point of DevOps is to empower developers with self-service tools. Forcing them to deal with infrastructure when they don't need to, is a step backward.

The correct solution is of course to split deployment or infrastructure on their own pipelines. The infrastructure pipeline will be triggered less often than the application one, making application deployments faster (and cutting down on lead time).

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



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



Developers will also know that when the application pipeline breaks, they don't need to deal with infrastructure errors or care about how the Kubernetes cluster was created. Operators can fine-tune the infrastructure pipeline without affecting developers at all. Everybody can work independently.

We sometimes see this anti-pattern (mixing infrastructure with application) where companies believe that this is the only way forward as the application needs something provided by the infrastructure pipelines.

The classic example is creating something with Terraform and then passing the output of the deployment (e.g. an IP address) to the rest of the pipeline as input to the application code. If you have this limitation it means that you are suffering from the previous anti-pattern (coupling application to the details of the infrastructure) and you need to remove this coupling (i.e. your application code should not need a specific IP address to be deployed).

Notice that the same approach can be expanded to database upgrades. If you use pipelines for database changesets, then they should be independent of the application source code. You should be able to update only the DB schema or only the application code on their own, without having to do both for every deployment.

Anti-pattern 5

Performing ad-hoc deployments with kubectl edit/ patch by hand

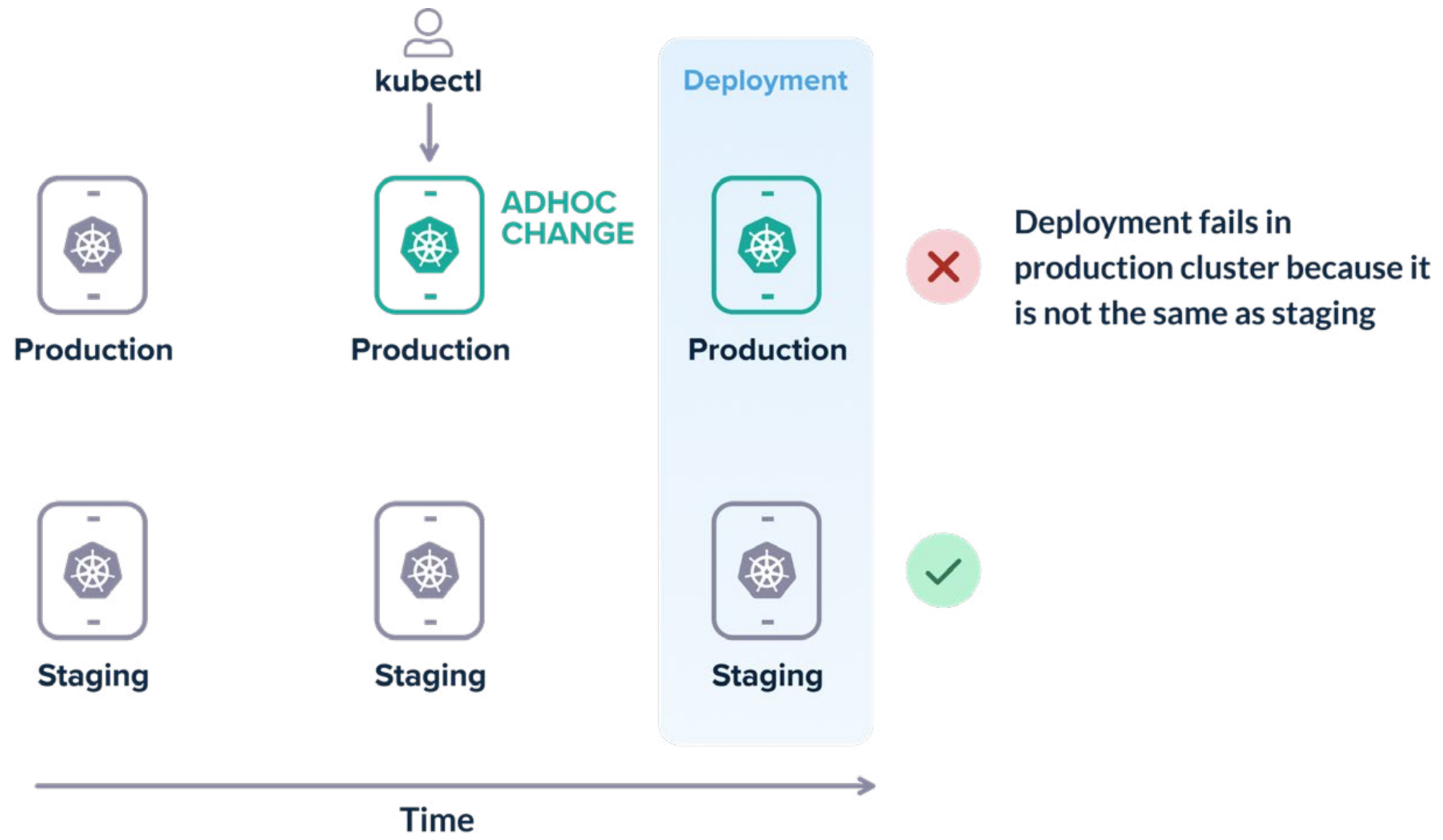
Configuration drift is a well known problem that existed even before Kubernetes appeared. It happens when two or more environments are supposed to be the same, but after certain ah-hoc deployments or changes they stop having the same configuration.

As time goes on, the problem becomes even more critical and can result in extreme scenarios where the configuration of a machine is not known any more and has to be reverse-engineered from the live instance.

Kubernetes can also suffer from this problem. The `kubectl` command is very powerful and comes with built-in apply/edit/patch commands that can change resources in place on a live cluster.

Unfortunately this method is easily abused by both cowboy developers and ninja operators. When ad-hoc changes happen in the cluster, they are never recorded anywhere else.

One of the most frequent reasons for failed deployments is environment configuration. A production deployment fails (even though it worked in the staging environment) because the configuration of the two environments is not the same anymore.



Falling into this trap is very easy. Hotfixes, “quick workarounds” and other questionable hacks are always the main reasons behind ad-hoc changes.

Kubectl should never be used for deployments by hand. All deployments should be taken care of by the deployment platform and ideally should also be recorded in Git following the GitOps paradigm.

If all your deployments happen via a Git commit:

- You have a complete history of what happened in your cluster in the form of Git commit history
- You know exactly what is contained on each cluster at any point in time and how environments differ among themselves
- You can easily recreate or clone an environment from scratch by reading the Git configuration.
- Rolling back configuration is trivial as you can simply point your cluster to a previous commit.

Most importantly, if a deployment fails, you can pinpoint really fast what was the last change that affected it and how it changed its configuration.
The patch/edit capabilities of `kubectl` should only be used for experimentation only. Changing live resources on a production cluster by hand is a recipe for disaster. Apart from having a proper deployment workflow, you should also agree with your time that abusing kubectl in this way should be avoided at all times.

Anti-pattern 6

Using Kubectl as a debugging tool

While we are still on the topic of kubectl, is it important to mention its second-biggest shortcoming. Kubectl is not a debugging tool and should not be used as such.

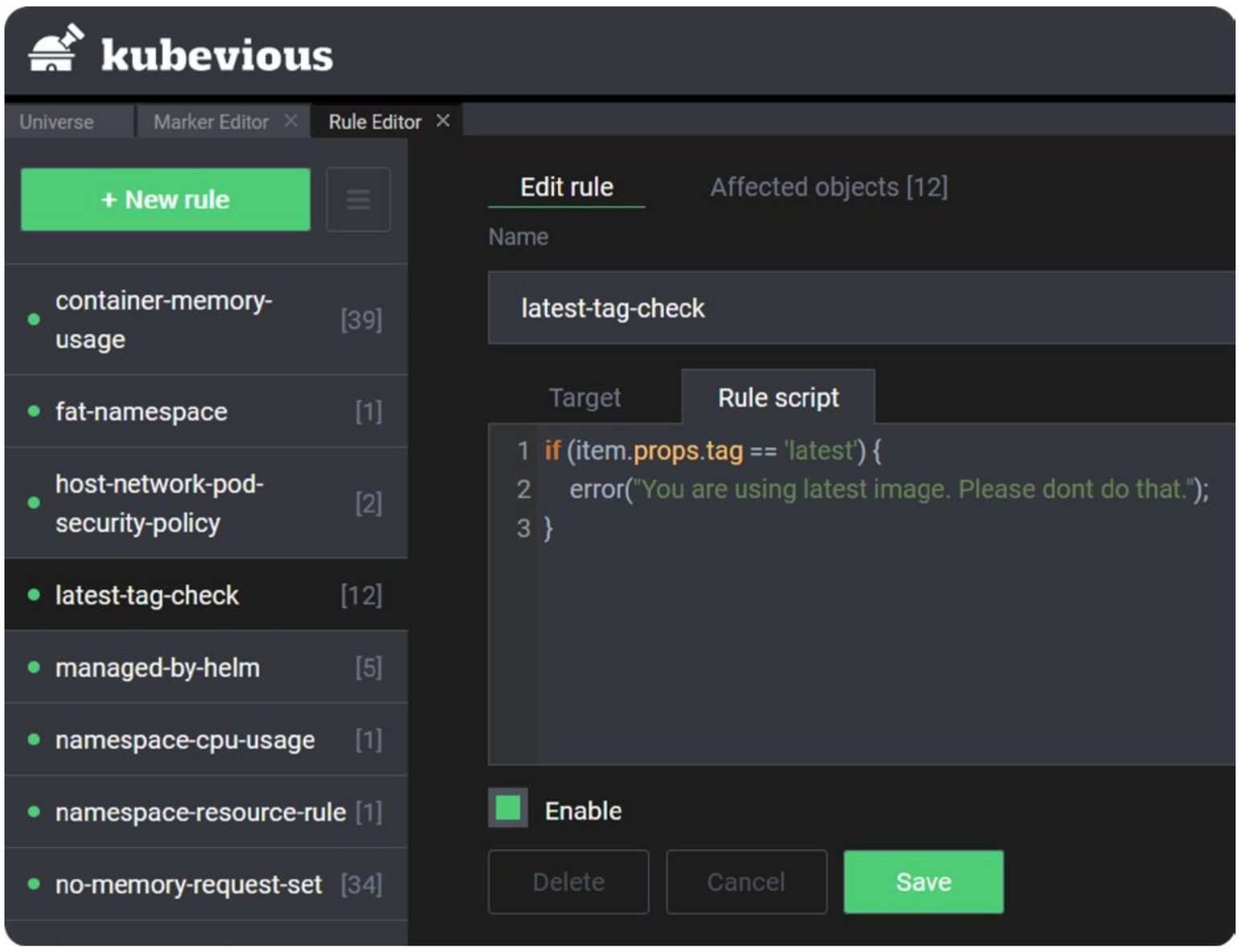
Every company that started adopting Kubernetes has eventually run into a problem that prompted the “10-questions-game” with kubectl. If you have a critical problem in your production cluster your first impulse should not be opening a terminal with kubectl. If you are doing this you have already lost the battle, especially if it is 3am, production is down and you are on call.

```
kubectl get ns
kubectl get pods -n sales
kubectl describe pod prod-app-1233445 -n sales
kubectl get svc - n sales
kubectl describe...
```

All your Kubernetes clusters should have proper monitoring/tracing/logging systems in place that can be used for pinpointing issues in a timely manner. If you need to run kubectl to inspect something it means that you have a gap in your observability tools and the thing that you need to inspect should be added to your monitoring tools.

Even if you simply want to inspect a cluster that you are not familiar with you should use a dedicated tool for this purpose. There are many tools for inspecting Kubernetes clusters today.

Kubevious for example is a comprehensive Kubernetes dashboard with a built-in rule engine that allows you to search and mark Kubernetes resources according to custom rules.



Metrics and tracing are so important that will be discussed in another anti-pattern later in our list.

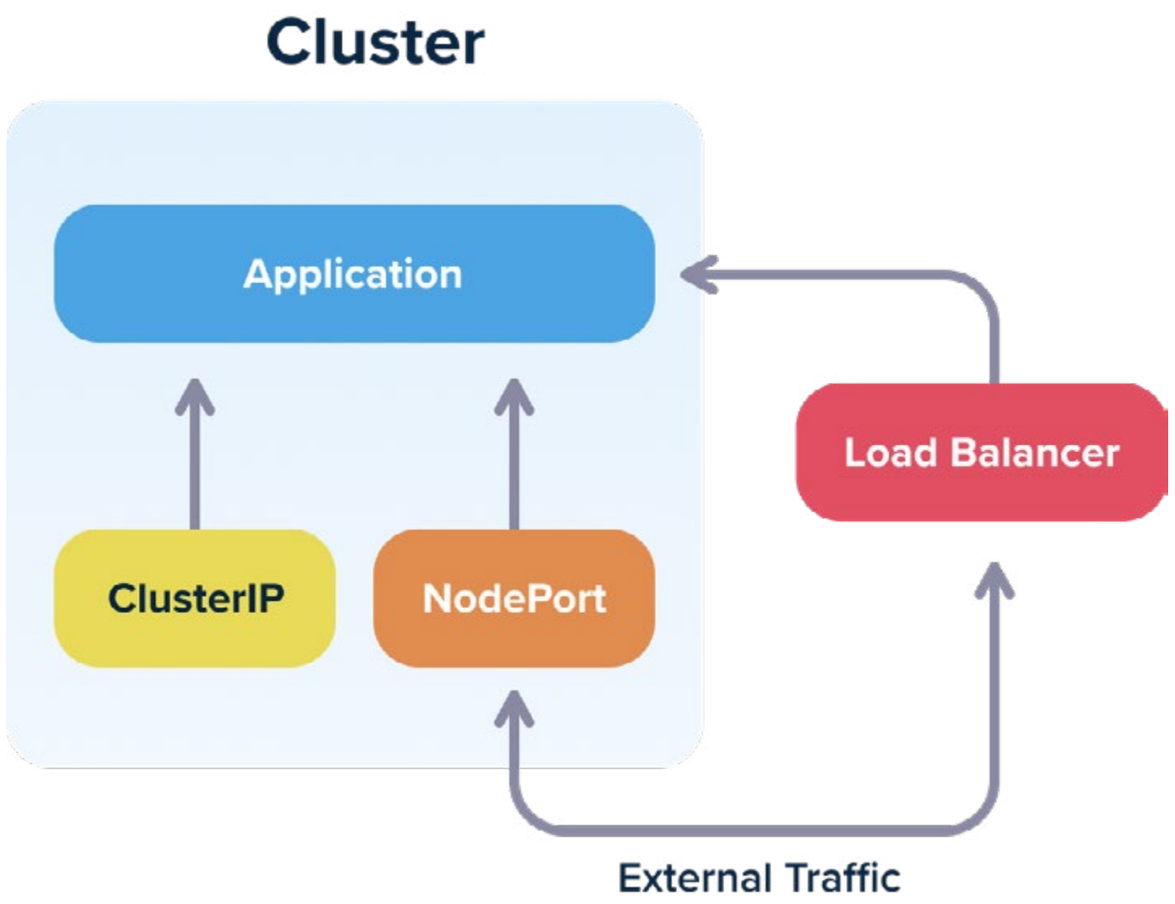
Anti-pattern 7

Misunderstanding Kubernetes network concepts

Gone are the days, where a single load balancer was everything you needed for your application. Kubernetes introduces its own networking model and it is your duty to learn and understand the major concepts. At the very least you should be familiar with load balancers, clusterIPs, nodeports and ingress (and how they differ).

We have seen both ends of the spectrum, where organizations create an overkill setup with a heavy-weight ingress controller (when a simple load balancer would suffice) or creating multiple load balancers (wasting money on the cloud provider) instead of a single ingress setup.

Understanding the different service options is one of the most confusing aspects for people starting with Kubernetes networking. ClusterIP services are internal to the cluster, NodePorts are both internal and external and Load balancers are external to the cluster, so make sure that you understand the implications of each service type.



And this is only for getting traffic inside your cluster. You should also pay attention to how traffic works within the cluster itself. DNS, security certificates, virtual services are all aspects that should be handled in detail for a production Kubernetes cluster.

You should also spend some time to understand what a service mesh is and what problems it solves. We do not advocate that every cluster should have a service mesh. But you should understand how it works and why you would need it.

You might argue that a developer should not have to learn about these networking concepts just to deploy an application, and you would be correct. We need an abstraction on top of Kubernetes for developers, but we don't have it yet.

Even as a developer you should know how traffic reaches your application. If a request needs to perform 5 hops between pods, nodes and services and each hop has a possible latency of 100ms, then your users face a possible delay of 500ms when visiting a web page. You should be aware of this, so that spending effort to optimize response time is focused on the true bottlenecks.

Also as a developer you should know what kubectl proxy does behind the scenes and when to use it.

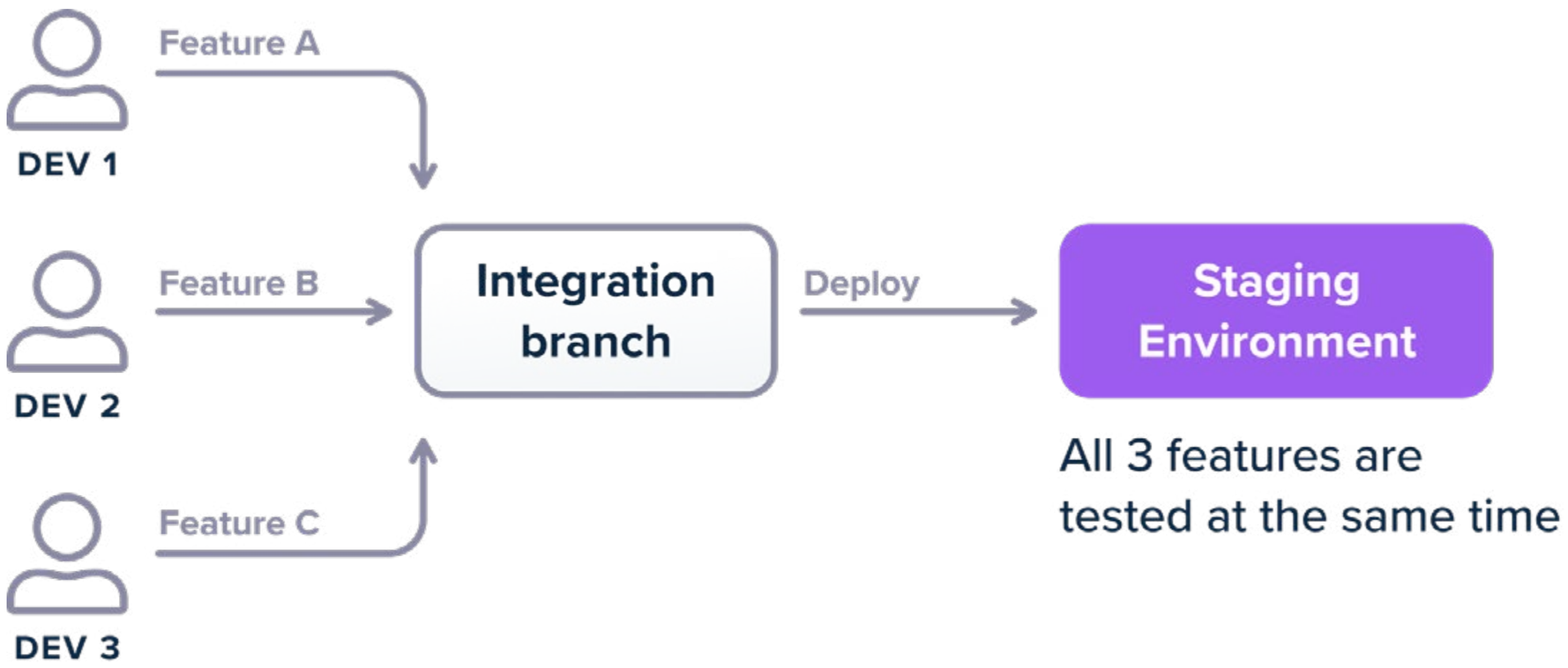
Anti-pattern 8

Using permanent staging environments instead of dynamic environments

With virtual machines (and even more so with bare metal servers) it is customary for a software team to have multiple predefined test environments that are used to verify an application before it reaches production.

One of the most common patterns is having at least 3 environments (QA/staging/production) and depending on the size of the company you might have more. The most important of these environments is the “integration” one (or whatever the company calls it) that gathers all features of developers after they are merged to the mainline branch.

Leaving aside the aspects of cost (if you have predefined test environments you always pay for them in terms of computing capacity even when they are not used) the most pressing issue is feature isolation.



Don’t

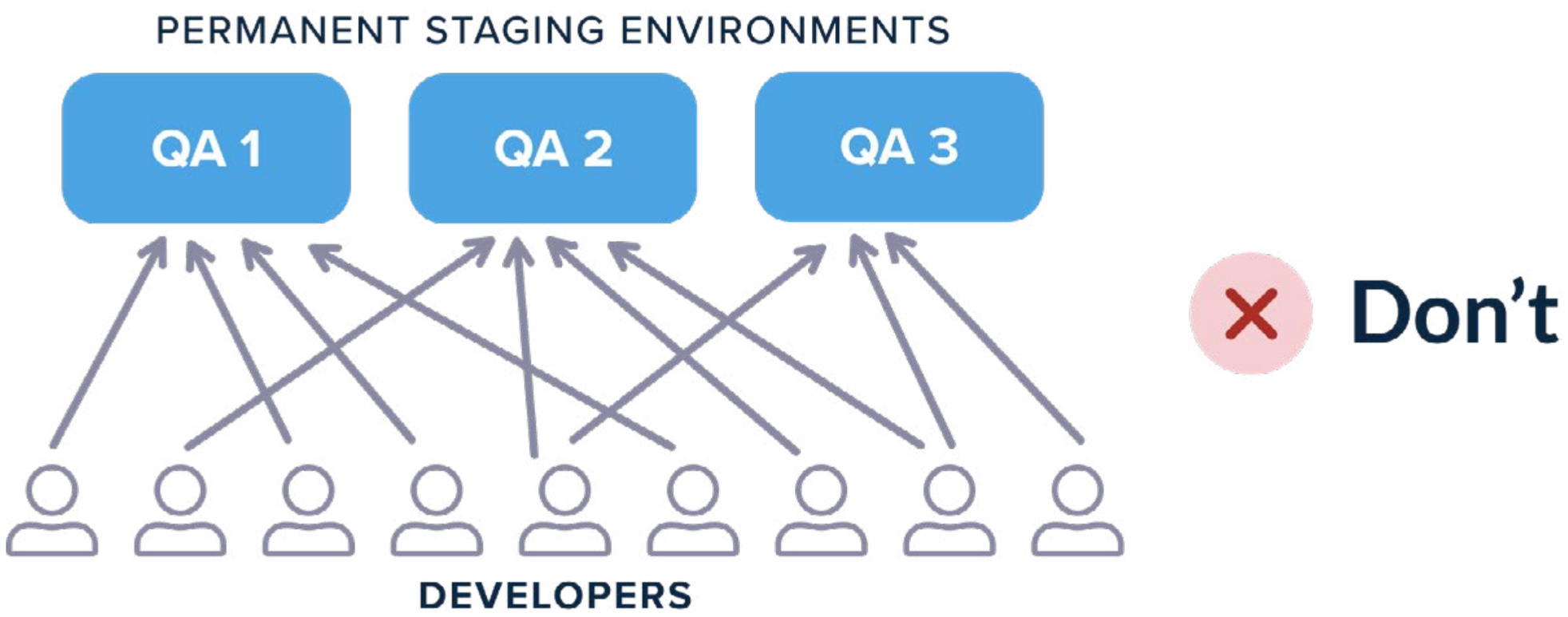
If you use a single environment for integration then when multiple developers merge features and something breaks, it is not immediately which of the feature(s) caused the problem. If 3 developers merge their features on a staging environment and the deployment fails (either the build fails, or the integration tests fail or the metrics explode) then there are several scenarios:

1. Feature A is problematic, B and C are fine
2. Feature B is problematic, A and C are fine
3. Feature C is problematic, B and C are fine
4. All features are fine on their own, but the combination of A and B is problematic
5. All features are fine on their own, but the combination of A and C is problematic
6. All features are fine on their own, but the combination of B and C is problematic
7. All features are fine on their own, but the combination of all 3 is problematic

Depending on the size of your programming team and the complexity of your software, distinguishing between these scenarios is a lengthy process. If a GUI button changes position it is probably easy to find out which developer is responsible for the commit. But if your recommendation engine suddenly became 5x slower, how quick can you identify the responsible feature if all 3 developers worked on the recommendation engine for this sprint?

To overcome this problem, companies almost always adopt the “booking paradigm”:

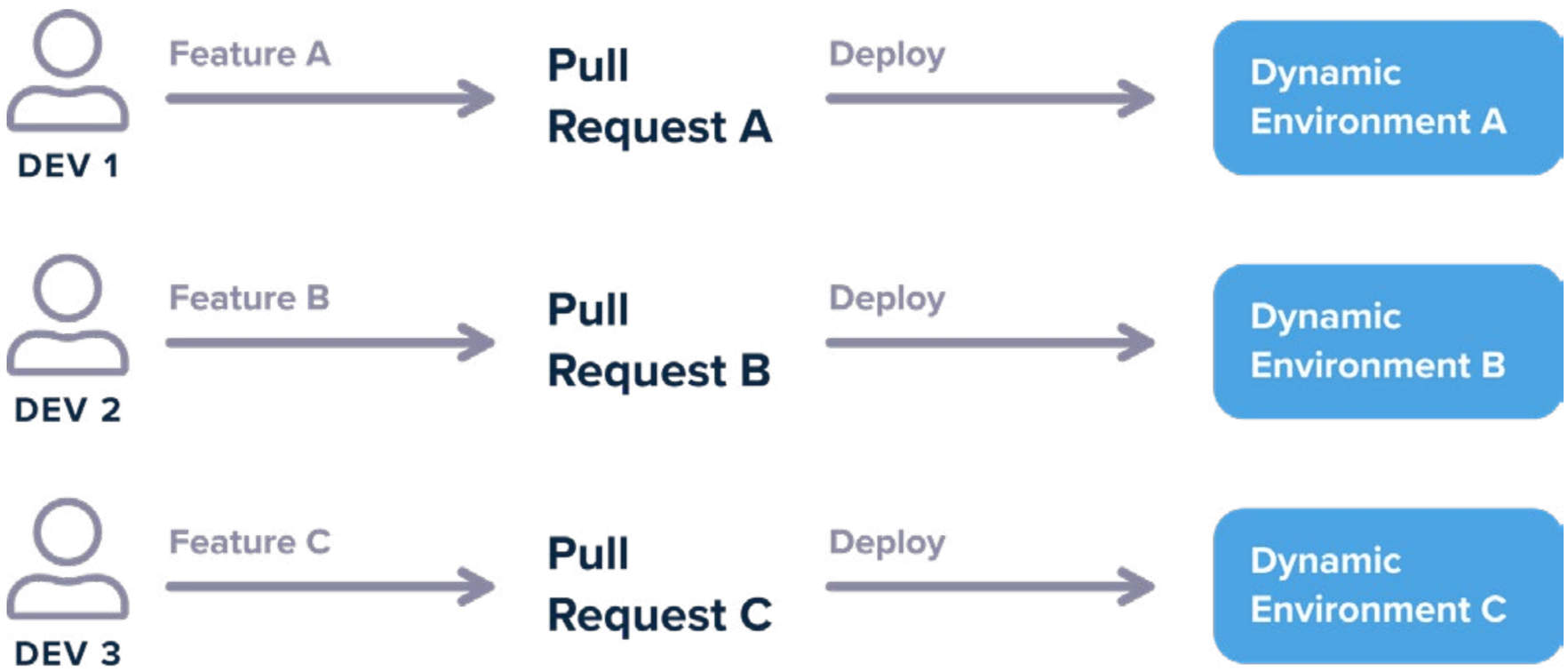
1. The staging environment is “booked” by each individual developer so that they can test their feature in isolation.
2. The company creates multiple “staging” environments that developers use for testing their features (as a single environment can quickly become a bottleneck).



This situation is still problematic because developers not only have to coordinate between themselves for choosing environments, but also because you have to keep track of the cleaning actions of each environment. For example, if multiple developers need a database changeset along with their feature, they need to make sure that the database of each staging environment contains only their own changes and not the changes of the previous developer that used that environment.

Additionally, multiple permanent staging environments can quickly suffer from the well known problem of configuration drift where environments are supposed to be the same, but after several ad-hoc changes, they are not.

The solution of course, is to abandon the manual maintenance of static environments and move to dynamic environments that are created and destroyed on demand. With Kubernetes this is very easy to accomplish:



Do Each feature is tested on its own

There are many ways to achieve this scenario, but at the very least every Open Pull Request should create a dynamic environment that contains only that Pull Request and nothing else. The environment is automatically destroyed when the pull request is merged/closed or after a specified amount of time.

The big strength of automatic environments however is the complete freedom of developers. If I am a developer and just finished with feature A and my colleague finished feature B, I should be able to do:

```
git checkout master
git checkout -b feature-a-b-together
git merge feature-a
git merge feature-b
git push origin feature-a-b-together
```

And then magically a dynamic environment should be present at: feature-a-b-together.staging.company.com or staging.company.com/feature-a-b-together.

Behind the scenes you can use Kubernetes namespaces and ingress rules for this isolation.

Note that it is ok if your company has permanent staging environments for specialized needs such as load testing, security penetration analysis, A/B deployments etc. But for the basic scenario of “I am a developer and want to see my feature running and run integration tests against it”, dynamic environments is the way to go.

If you are still using permanent environments for feature testing you are making life hard for your developers and are also wasting system resources when your environments are not actively used.

Anti-pattern 9

Mixing production and non-production clusters

Even though Kubernetes is specifically designed for cluster orchestration you shouldn't fall into the trap of creating a single big cluster for all your needs. At the very least you should have two clusters, a production one and a non-production one.

First of all, mixing production and non-production is an obvious bad idea for resource starvation. You don't want a rogue development version to overstep on the resource of the production version of something.

But as far as developers are concerned the biggest problem has to do with security. Unless you take active steps against it, all communication inside a Kubernetes cluster is allowed by default. And contrary to popular belief a pod from one namespace can freely communicate with a pod on another namespace. There are also some Kubernetes resources that are not namespaced at all.

Kubernetes namespaces are not a security measure. If your team has advanced Kubernetes knowledge then it is indeed possible to support multi-tenancy inside a cluster and secure all workloads against each other. But this requires significant effort and in most cases, it is much simpler to create a second cluster exclusively for production.

If you combine this anti-pattern with the second one (baking configuration inside containers) it should be obvious that a lot of bad scenarios can happen.

The most classic one:

1. A developer creates a feature namespace on a cluster that also houses production
2. They deploy their feature code in the namespace and start running integration tests
3. The integration tests write dummy data, or clean the DB, or tamper with the backend in a destructive manner
4. Unfortunately the containers had production URLs and configuration inside them, and thus all integration tests actually affected the production workloads!

To avoid falling into this trap, it is much easier to simply designate production and non-production clusters. Unfortunately several tutorials imply that namespaces can be used for environment division and even the official Kubernetes documentation has examples with prod/dev namespaces.

Note that depending on the size of your company you might have more clusters than two such as:

1. Production
2. Shadow/clone of production but with less resources
3. Developer clusters for feature testing (see the previous section)
4. Specialized cluster for load testing/security (see previous section)
5. Cluster for internal tools

No matter the size of your company you always should have at least 2 (production and everything else that is not production)

Anti-pattern 10

Deploying without memory and cpu limits

By default an application that is deployed to Kubernetes has no specified resource limits. This means that your application can potentially take over the whole cluster. This would be problematic in a staging cluster and catastrophic in a production cluster as the slightest memory leak or CPU hiccup will wreak havoc in the rest of the applications.

This means that all your applications (regardless of the target cluster) should have associated resource limits.

Maybe as a developer you don't need to know all the details on how the limits are set, but it is your responsibility to give some hints and advice to the person that is managing your cluster on what are the expectations of the application.

Unfortunately, simply looking at the average memory and CPU usage of an application is not enough. Average resources are just that - average. You need to examine your application and see with bursts of traffic and load and understand what is the behaviour of the resources under extreme conditions. After all, this is exactly the kind of condition that you don't want your application to be restarted without reason.

One of the benefits of Kubernetes is the elasticity of resources. If the cluster is killing/restarting your application just when it is starting to handle significant load (because let's say your eshop is getting hammered with traffic), you have lost the advantage of using a cluster in the first place.

On the other hand, setting too high limits is a waste of resources and makes your cluster less efficient.

You also need to examine your programming languages for specific usage patterns and how resources are used by the underlying platform. Legacy Java applications for example have notorious problems with memory limits.

Note that once you gain confidence with your application and how it uses Kubernetes resources you can also automate the whole resource game with a vertical cluster auto-scaler.

Anti-pattern 11

Misusing Health probes

Apart from resource limits, if you are moving your application to Kubernetes you should take into account the health probe settings.

By default, a Kubernetes application has no health probes unless you explicitly set them up. Like resource limits, you should consider health probes an essential part of your Kubernetes deployments. This means that all your applications should have resource limits AND health probes when deployed in any cluster of any type.

Health probes decide when and if your application is ready to accept traffic. As a developer you need to understand how Kubernetes is going to use your readiness and liveness endpoints and the implications of each one (especially what a timeout for each one means).

In summary:

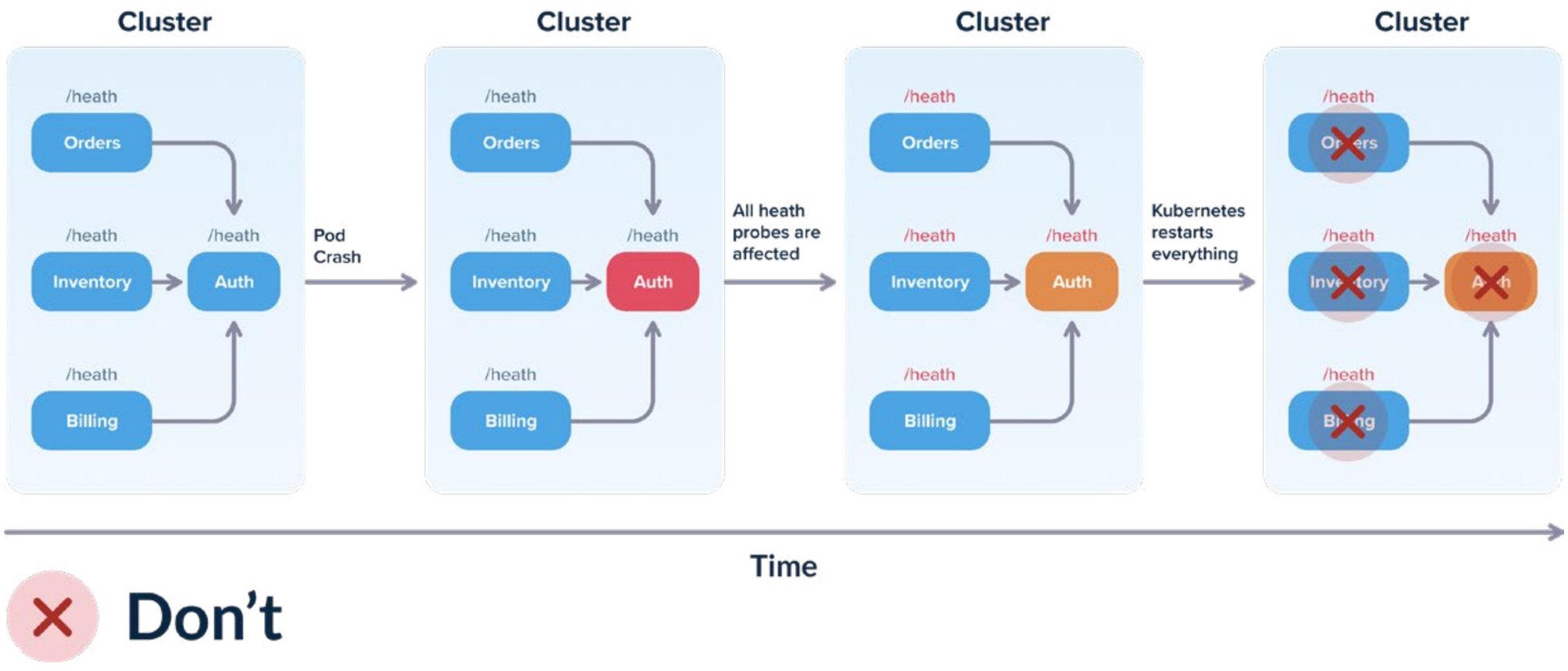
- Startup probe => Checks the initial boot of your applications. It runs only once
- Readiness probe => Checks if your application can respond to traffic. Runs all the time. If it fails Kubernetes will stop routing traffic to your app (and will try later)
- Liveness probe => Checks if your application is in a proper working state. Runs all the time. If it fails Kubernetes will assume that your app is stuck and will restart it.

Spend some time to understand the effects of each health probe and evaluate all the best practices for your programming framework.

Some of the common pitfalls are:

- Not accounting for external services (e.g. DBS) in the readiness probe
- Using the same endpoint for both the readiness and the liveness probe
- Using an existing “health” endpoint that was created when the application was running on a Virtual machine (instead of a container)
- Not using the health facilities of your programming framework (if applicable).
- Creating too complex health checks with unpredictable timings (that cause internal denial of service attacks inside the cluster)
- Creating cascading failures when checking external services in the liveness health probe

Creating cascading failures by mistake is a very common problem that is destructive even with Virtual machines and Load balancers (i.e., it is not specific to Kubernetes).



Don't

You have 3 services that all use the Auth service as a dependency. Ideally the liveness probe for each service should only check if the service itself can respond to queries. However, if you setup the liveness probe to check for the dependencies then the following scenario can happen:

1. Initially, all 4 services work correctly.
2. The auth service has a hiccup (maybe its DB is overloaded) and is not responding to requests fast enough.
3. All 5 services correctly detect that the auth service has issues.
4. Unfortunately, they set their OWN health status to down (or unavailable) even though they all work fine.
5. Kubernetes runs the liveness probes and decides that all 4 services are down and restarts all of them (while in reality only one of them had issues).

For more advanced cases, you should also become familiar with circuit breakers as they allow you to decouple the "liveness" of an application with its ability to respond.

Anti-pattern 12

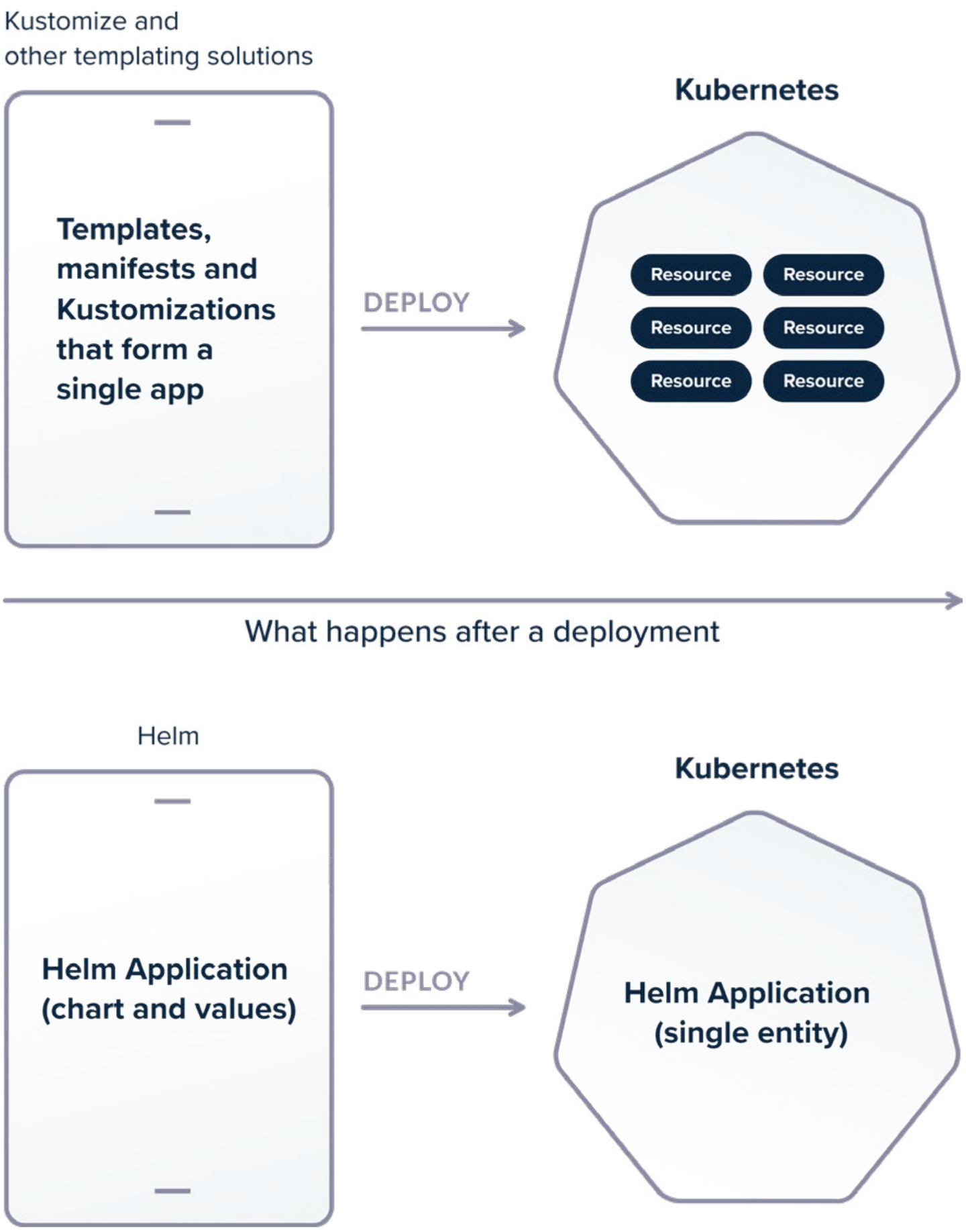
Not using the Helm package manager

There is only one package manager right now for Kubernetes and that is Helm. You can think of Helm as the apt/rpm utility for your Kubernetes clusters.

Unfortunately, people often misunderstand the capabilities of Helm and choose another "alternative". Helm is a package manager that also happens to include templating capabilities. It is not a templating solution and comparing it to one of the various templating solutions misses the point.

All templating solutions for Kubernetes suffer from the same issue. They can do their magic during application deployment, but after the deployment has finished Kubernetes only knows a set of manifests and nothing more. Any concept of "application" is lost and can only be recreated by having the original source files at hand.

Helm, on the other hand, knows about the whole application and stores application-specific information in the cluster itself. Helm tracks the application resources AFTER the deployment. The difference is subtle but important.



Let's say that you get kubectl access on a cluster that has 4 applications (where each one is composed of multiple resources) in the same namespace. Your task is to delete one of these applications.

If you have used a simple templating solution (such as Kustomize), you cannot easily understand the application structure by looking at the running resources. If you don't have the original templates (i.e. kustomizations and original patches) you need to manually inspect resources with kubectl and make the correlations between the application components.

Helm on the other hand tracks the application inside the cluster. You can simply do:

- helm ls (to look at the list of applications in the namespace)
- helm uninstall my-release (to remove the application)

That's it! No access to the original templates is needed or to a CI pipeline.

Comparing Helm to Kustomize/k8compt/kdeploy and other templating tools is unfair to Helm as Helm is much more than a templating solution.

One of the killer features of Helm is also the rollback function. You are paged at 3 am in the morning and want to perform the fastest rollback possible in a Kubernetes cluster that you are not familiar with. There is simply no time to track the original source files and identify what is the "previous" versions of the templates in Git.

With Helm it is trivial to do:

1. helm ls (to see releases)
2. helm history my-release (to see previous versions of deployments)
3. helm rollback my-release my-previous-version

All this, right from the cluster because Helm knows what an application is unlike templating solutions that stop their work after a deployment has finished.

Another misconception is that Helm packages are managed with Git Repos and changing a chart involves cloning/copying the chart locally. Helm charts should actually be managed with Helm repositories. It is ok if you also have a copy in Git (following the GitOps paradigm) but deploying a Helm chart to a cluster should happen from a Helm repository and not a Git repo.

It is also worth noting that since Helm 3, there is no server-side component anymore (the infamous Tiller), so if the last time you evaluated Helm you had concerns regarding the security of the cluster, you need to take a fresh look at Helm a second time now that Tiller is gone.

Unless you have a really strange workflow, not using Helm is like not using apt/rpm for package management. If you decide that Helm is not for you, it should be a conscious choice (after careful consideration) and not based on misinformation about selecting “a better templating solution than Helm”.

Anti-pattern 13

Not having deployment metrics

We have mentioned metrics several times in the previous anti-patterns. By “metrics” we actually mean the whole trilogy of:

- logging - to examine the events and details of requests (usually post-incident)
- tracing - to dive deep in the journey of a single request (usually post-incident)
- metrics - to detect an incident (or even better to predict it)

Metrics are more important in Kubernetes than traditional Virtual machines, because of the distributed nature of all services in a cluster (especially if you use micro-services). Kubernetes applications are fully dynamic (they come and go unlike virtual machines) and it is vital that you know how they adapt to your traffic.

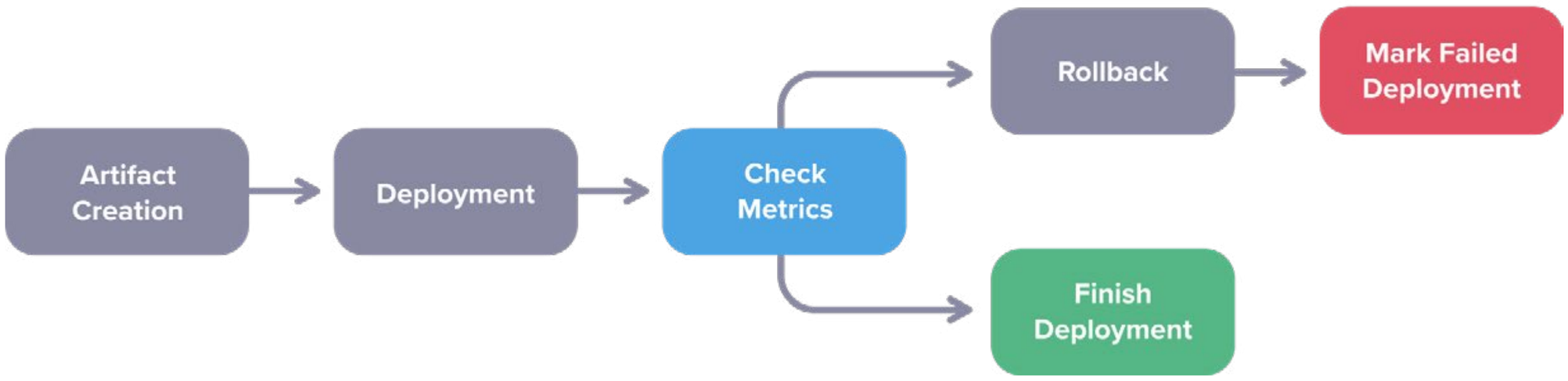
The exact solution that you choose for your metrics is not that important, as long as you have sufficient metrics for all your use cases such as:

- Getting critical information in a timely manner instead of using kubectl (see antipattern 6)
- Understanding how traffic enters your cluster and what is the current bottleneck (see antipattern 7)
- Verifying/adapting your resource limits (see antipattern 10)

The most important use case however is that you need to understand if your deployment has succeeded. Just because a container is up, doesn’t mean that your application is in a running state or can accept requests (see also anti-pattern 11).

Ideally, metrics should not be something that you look at from time to time. Metrics should be an integral part of your deployment process. Several organizations follow a workflow where the metrics are inspected manually after a deployment takes place, but this process is sub-optimal. Metrics should be handled in an automated way:

FULLY AUTOMATED ROLLBACKS



1. A deployment takes place
2. Metrics are examined (and compared to base case)
3. Either the deployment is marked as finished or it is rolled back

Note that there is no manual step involved in these actions.

Making your metrics affecting your deployments is not an easy task. It shows however what is the end goal and how important are metrics for Kubernetes deployments.

Anti-pattern 14

Not having a strategy for secrets

In the second anti-pattern we explained why baking configuration in containers is a bad practice. This is even more true for secrets. If you use a dynamic service for configuration changes, it makes sense to use the same (or similar) service for handling secrets.

Secrets should be passed during runtime to containers. There are many approaches to secret handling from simple storage to git (in an encrypted form) to a full secret solution like Hashicorp vault.

Some common pitfalls here are:

- Using multiple ways for secret handling
- Confusing runtime secrets with build secrets
- Using complex secret injection mechanisms that make local development and testing difficult or impossible.

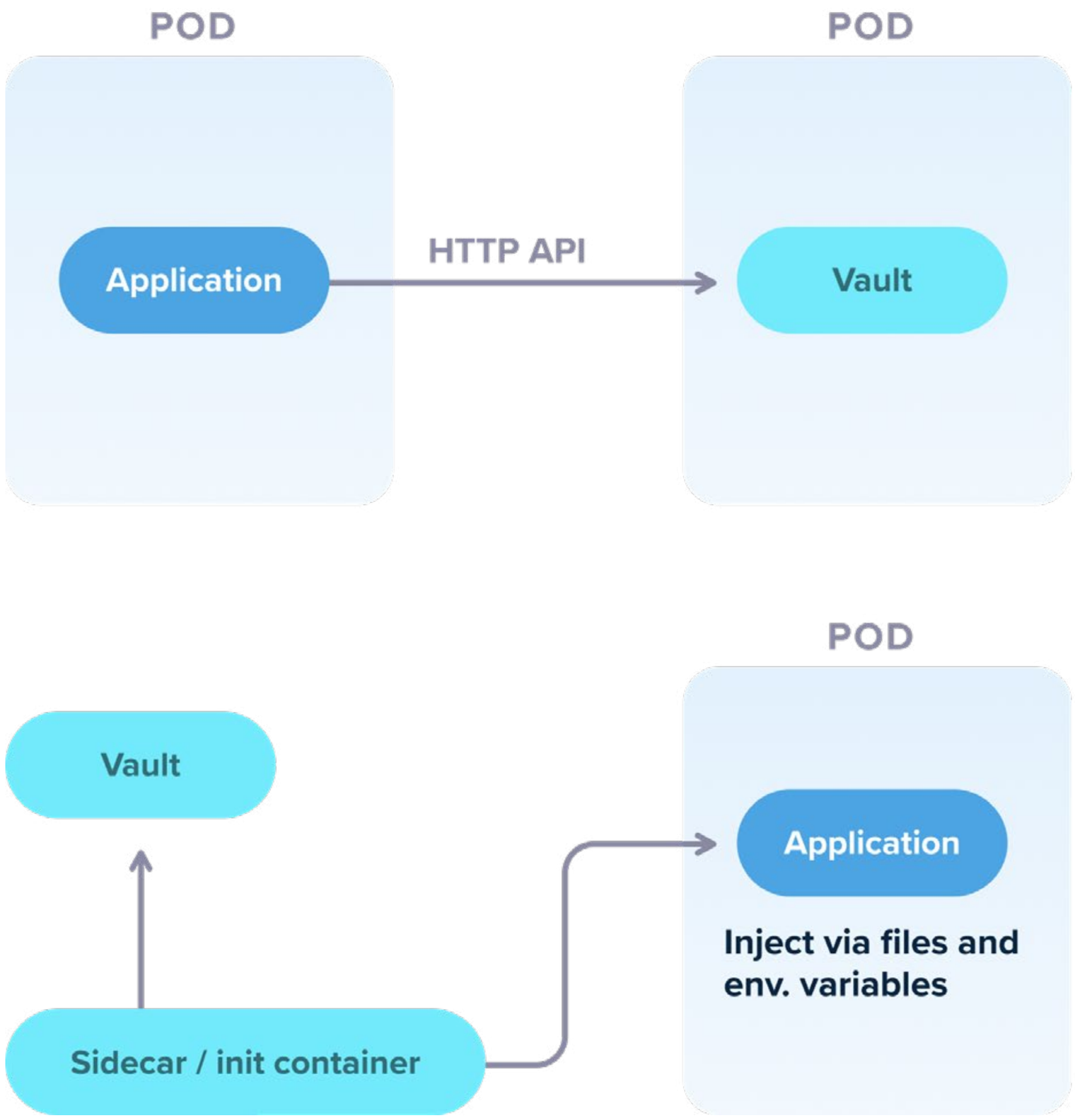
The important point here is to choose a strategy and stick to it. All teams should use the same method for secret handling. All secrets from all environments should be handled in the same way. This makes secret tracking easy (knowing when and where a secret was used).

You should also pass to each application only the secrets it actually needs.

Runtime secrets are the secrets that an application needs AFTER it is deployed. Examples would be database passwords, SSL certificates, and private keys.

Build secrets are secrets that an application needs ONLY while it is packaged. An example would be the credentials to your artifact repository (e.g. Artifactory or Nexus) or for file storage to an S3 bucket. These secrets are not needed in production and should never be sent to a Kubernetes cluster. Each deployed application should only get exactly the secrets it needs (and this is true even for non-production clusters).

As we also mentioned in anti-pattern 3, secret management should be handled in a flexible way that allows for easy testing and local deployment of your app. This means that the application should not really care about the source of the secrets and should only focus on their usage.



As an example, even though Hashicorp Vault has a flexible API for getting secrets and tokens, your Kubernetes application should NOT use that API directly in order to get the required information. If you do this, then testing the application locally becomes a nightmare, as a developer would need to set up a vault instance locally or mock the vault responses just to run the application.

Anti-pattern 15

Attempting to solve all problems with Kubernetes

As with all technologies before it, Kubernetes is a specific solution that solves a specific set of problems. If you already have those problems then you will find that adopting Kubernetes greatly simplifies your workflow and gives you access to a clustering solution that is well designed and maintained.

It is important however to understand the benefits and drawbacks of adopting Kubernetes. At least in the beginning it is much easier to use Kubernetes for stateless services that will exploit the elasticity and scalability of the cluster.

Even though technically Kubernetes supports stateful services as well, it is best to run such services outside the cluster as you begin your migration journey. It is ok if you keep databases, caching solutions, artifact repositories, and even Docker registries outside the cluster (either in Virtual machines or cloud-based services).

Moving ahead

Kubernetes is a complex solution that requires a new way of thinking across all fronts (networking, storage, deployments, etc). Deploying to a Kubernetes cluster is a completely different process than deploying to Virtual machines.

Instead of reusing your existing deployment mechanisms, you should spend some time to examine all the implications of Kubernetes applications and not fall into the traps we have seen in this guide:

1. Deploying images with the “latest” tag
2. Hardcoding configuration inside container images
3. Coupling the application with cluster constructs
4. Mixing infrastructure deployments with application releases
5. Doing manual deployments using kubectl
6. Using kubectl for debugging clusters
7. Not understanding the Kubernetes network model
8. Wasting resources on static environments instead of dynamic ones
9. Mixing production and non-production workloads in the same cluster
10. Not understanding memory and CPU limits
11. Misusing health probes
12. Not understanding the benefits of Helm
13. Not have effective application metrics
14. Handling secrets in an ad-hoc manner
15. Adopting Kubernetes even when it is not the proper solution.

Happy deployments!