KodeKloud

Visit www.kodekloud.com to learn more.

Hello and welcome to this lecture. In this lecture we look at minimizing the base image footprint.

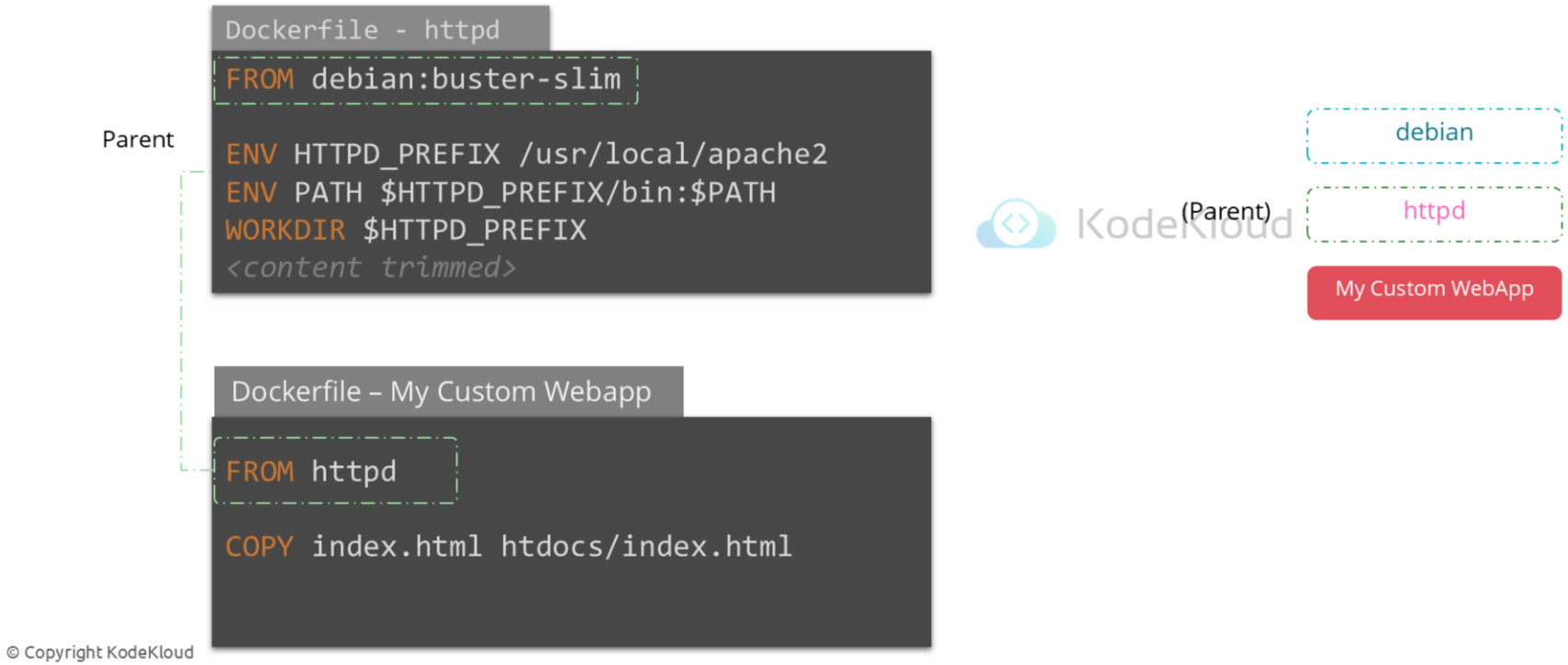
Base vs Parent Image

Let's start with base images first. These are some of the things that we discussed in the Docker Certification course. We discuss a lot more about images and builds in the DCA course, so check it out if you are interested.

Here I have my Dockerfile which is used to build an image of my custom web application. <c> The first line shows the image from which this image is built - httpd. <c> It is the parent image on which my custom application is built. So whatever image you use to build your custom image from is called the parent image. In this case httpd is a parent image.

But how is the httpd image itself built? Let’s look at the Dockerfile of the httpd image.

Base vs Parent Image

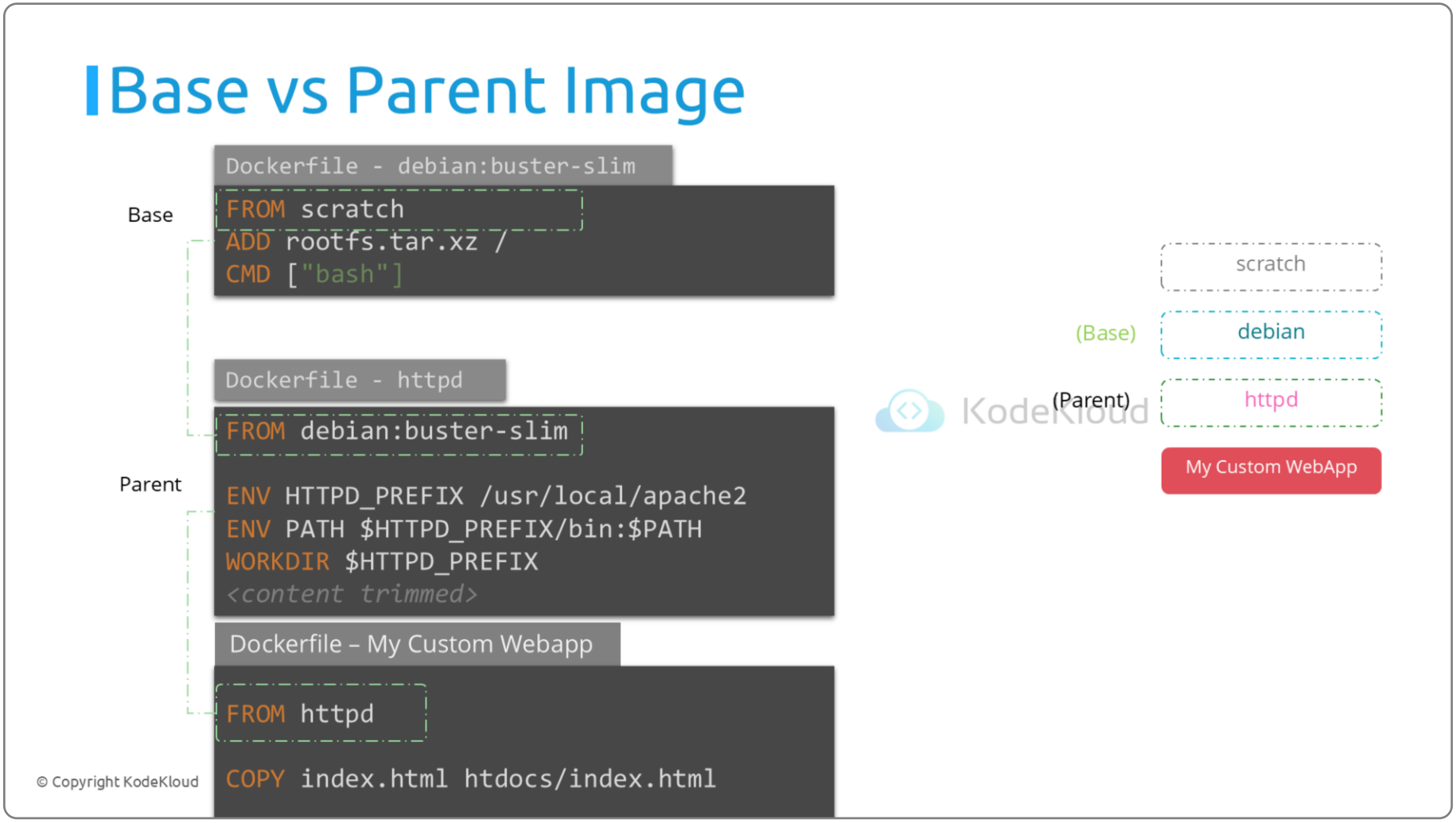


The httpd image itself is built from another image which happens to be the Debian image.

So the httpd image starts with the Debian image and then builds on to installing httpd on it.

And let’s go deeper and see what the Debian image itself is built on.

Base vs Parent Image



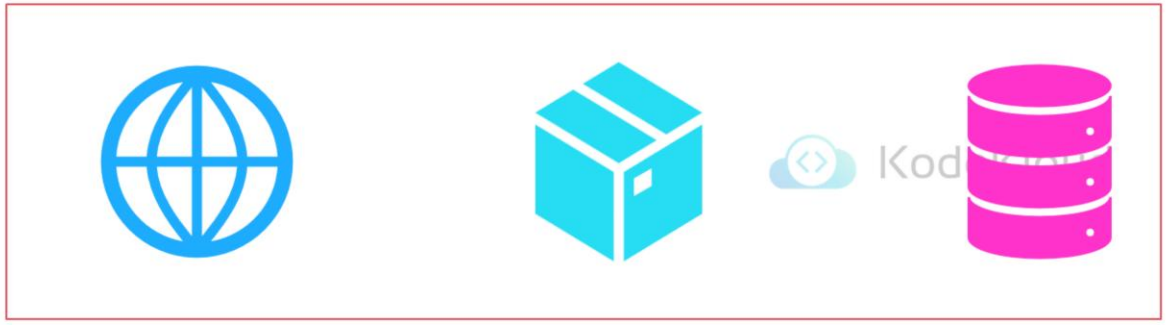
We see that the Debian image is built from SCRATCH. When an image is built from scratch it is called as the base image.

Now remember, even though they are called parent and base, you might find people using these terms interchangeably at times. So its no big deal.

For the remainder of this lecture we will refer to base image as any image from which our image is built. It could be a

parent or a base image, but we will refer to it as a base image.

Modular



Let's look at some of the best practices to be followed while building an image.
To begin with, do not build images that combine multiple applications – such as a web server, database, other services all into one image.

Modular

Instead build images that are modular – that solve one specific problem. Such as a separate image for a web server or a separate one for database. Each image can have its own libraries and dependencies and not really worry about the other application. But these different images when deployed as containers can together form a single large application that has different services. And each component can scale up or down as required without having to scale the other components.

Persist State

KodeKloud

Another best practice to be followed is not storing store data or state inside a container. This is because, containers are ephemeral in nature. We should be able to destroy them and bring them back online and not lose the data along with the ccontainer.

Persist State

KodeKloud

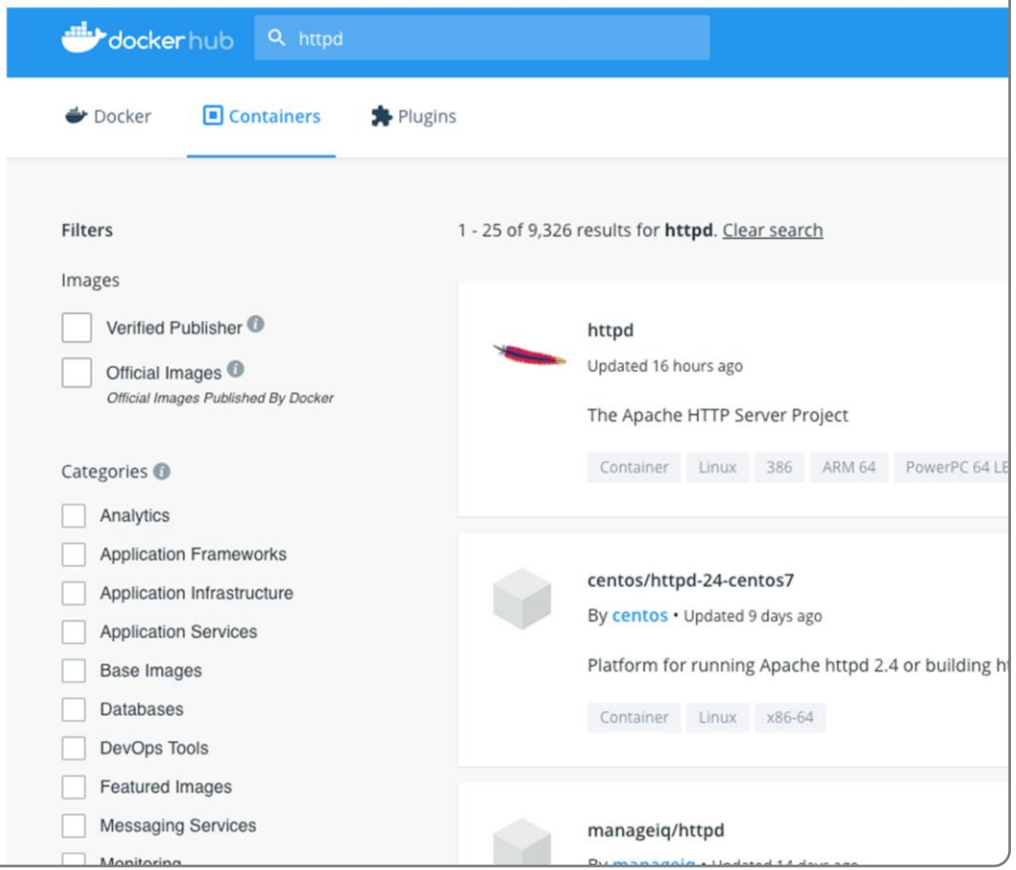
Always store data in either an external volume or some kind of caching service like redis.

Choosing a base image

Dockerfile – My Custom Webapp

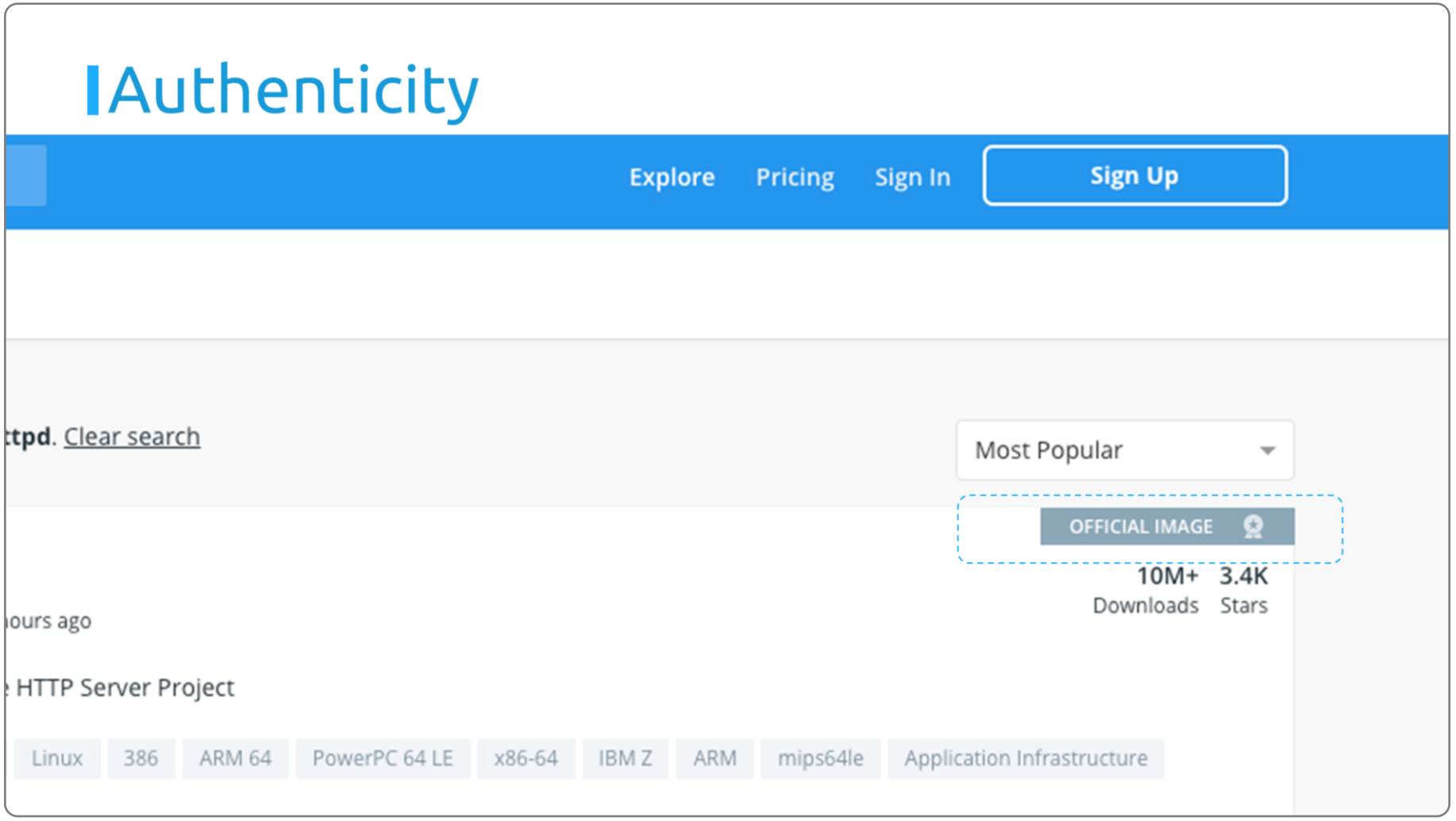
```
FROM ??????

COPY index.html htdocs/index.html
```

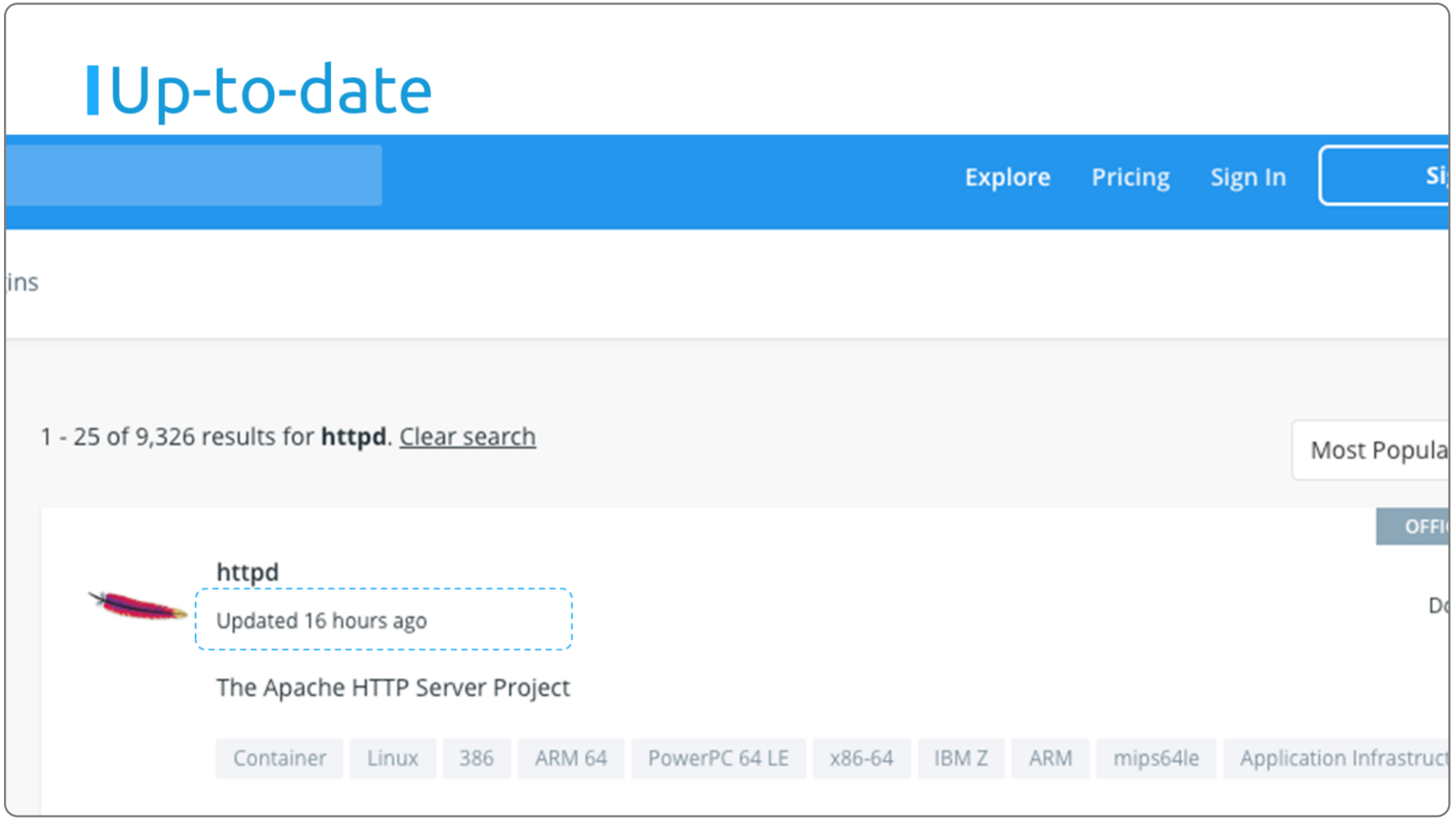


So how do you choose a base image? Say you want to build your own web application, and instead of building one from scratch you want to start from an existing base image. Which one do you choose?

Of course you start by looking at images that suit your technical needs such as say your application requires httpd then you look for httpd images or if you are building an nginx based server then you use the nginx base image. In this case my application requires an httpd server, so I look for httpd images on Dockerhub. You must look for images with authenticity.



You must look for images with authenticity. The official image or verified publisher tag indicates the image is from official sources.



Images must also be up to date. Images that are constantly updated are less likely to have vulnerabilities in them. We will talk about vulnerabilities in images in one of the upcoming lectures.

Slim/Minimal Images

1. Create slim/minimal images
2. Find an official minimal image that exists
3. Only install necessary packages
   - Remove Shells/Package Managers/Tools
4. Maintain different images for different environments:
   - Development – debug tools
   - Production - lean
5. Use multi-stage builds to create lean production ready images.

When creating images, make sure to keep the size of the image as small as possible. This will allow images to be pulled faster from remote repositories and spin up more instances easily as required. There are minimal versions of official operating systems available. Try to use them as base images.

Only install libraries and packages that are absolutely necessary for the application to run and delete and clean up as much as possible. Do not add unnecessary files or temporary files to the image. Remove tools like curl or wget that could be

used by attackers to download files if they were to gain access to the container. If you have package managers like yum or apt then those can be used to install unwanted software. So remove them.

At times development environments may have debug tools or other packages required for development. These need not be part of production. So it is good to not include these in production images. So you may consider having different images for different environments.

All of these will ensure that the images you build are minimal in size there by making it faster to build and pull as well as secure as you are not adding any packages that may be used by attackers to install malicious software and exploit vulnerabilities.

Distroless Docker Images

Contains:
- Application
- Runtime Dependencies

Does not contain:
- Package Managers
- Shells
- Network Tools
- Text editors
- Other unwanted programs

- gcr.io/distroless/static-debian10
- gcr.io/distroless/base-debian10
- gcr.io/distroless/java-debian10
- gcr.io/distroless/cc-debian10
- gcr.io/distroless/nodejs-debian10

- gcr.io/distroless/python2.7-debian10
- gcr.io/distroless/python3-debian10
- gcr.io/distroless/java/jetty-debian10
- gcr.io/distroless/dotnet

https://github.com/GoogleContainerTools/distroless

One such set of images are google's distroless docker images. They contain only the application and its runtime dependencies. And does not contain package managers, shells, network tools, text editors or other unwanted programs. More information about this can be found in the link below.

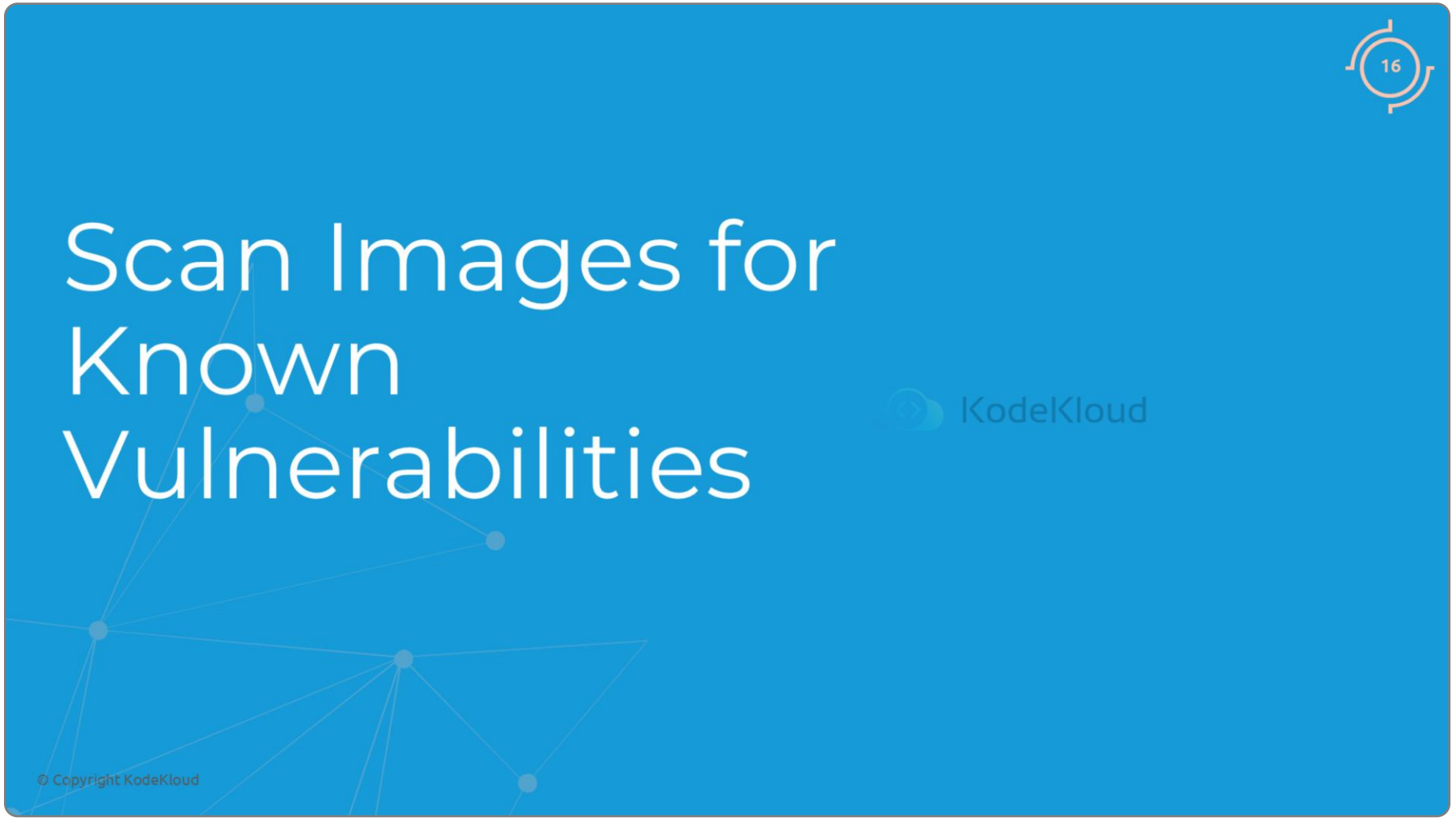
Vulnerability Scanning

```
trivy image httpd
httpd (debian 10.8)
==================
Total: 124 (UNKNOWN: 0, LOW: 88, MEDIUM: 9, HIGH: 25, CRITICAL: 2)
```



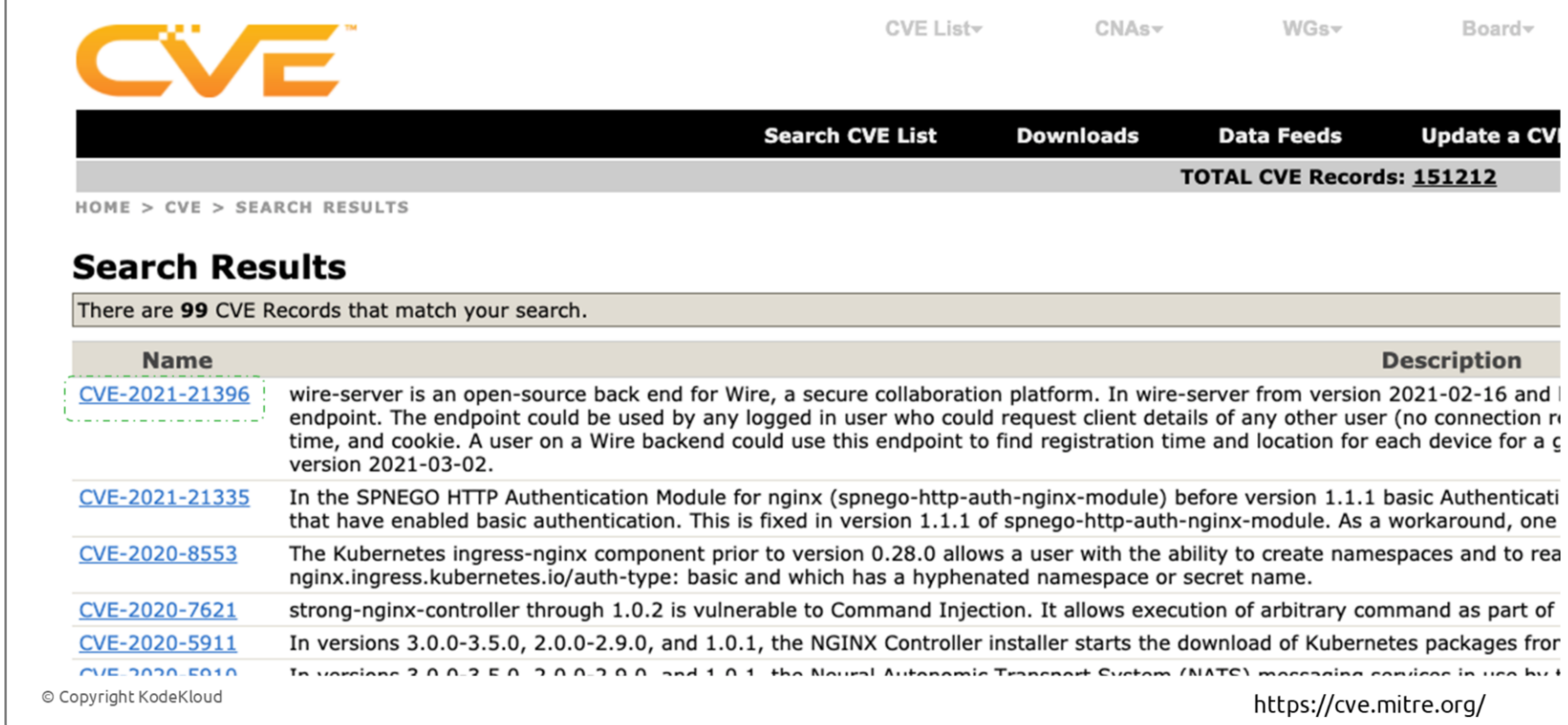
```
trivy image httpd:alpine
httpd:alpine (alpine 3.12.4)
============================
Total: 0 (UNKNOWN: 0, LOW: 0, MEDIUM: 0, HIGH: 0, CRITICAL: 0)
```


A minimal image is also less vulnerable to attack. For example a quick scan of vulnerability run on an httpd image using the trivy tool shows that there are atleast 124 known vulnerabilities in it. But when run on an httpd alpine based image, we see that there are 0 vulnerabilities. The lesser packages there are within an image, the fewer vulnerabilities are. We will talk about vulnerability scanning and the trivy tool in more detail in one of the upcoming lectures.



Hello and welcome to this lecture. In this lesson we'll learn how to scan container images, to make sure they do not have any known security vulnerabilities that attackers can easily exploit. But to understand how this kind of scanning software works, we'll first need to learn about CVEs

Common Vulnerabilities and Exposures (CVE)



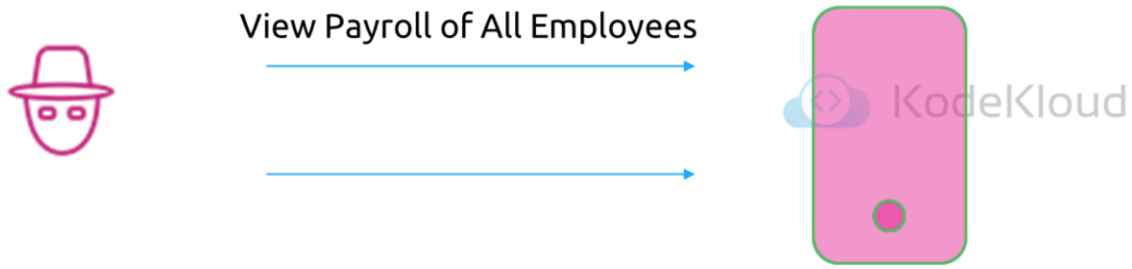
https://cve.mitre.org/

So what Is a CVE? CVE is short for Common Vulnerabilities and Exposures. Computer code is not perfect. If a bad guys finds this bug, they use it to abuse the system.
If the good guys finds these bugs, it would be nice if they can tell the whole world about it, so we can all get informed, correct the code, update our software and make our systems a little bit safer.

We need a central database so that anyone can submit this information when he/she finds something. This makes it easier

to report bugs and avoid duplicate entries. Each CVE gets a unique identifier. It also makes it easier to find information, look up all known bugs for a particular application, and so on.

Common Vulnerabilities and Exposures (CVE)



So what kind of problems are considered as CVEs and can enter a CVE database? It's usually one of two things:

Anything that allows an attacker to bypass security checks and do things he/she shouldn't be allowed to do. Say for example view the payroll details of all employees. Something that only authorized users should have access to.

The other kind of problem is anything that allows an attacker to mess up your system, seriously degrade performance,

interrupt services and so on.

CVE Severity Scores

CVSS v2.0 Ratings

Severity	Base Score Range
Low	0.0-3.9
Medium	4.0-6.9
High	7.0-10.0

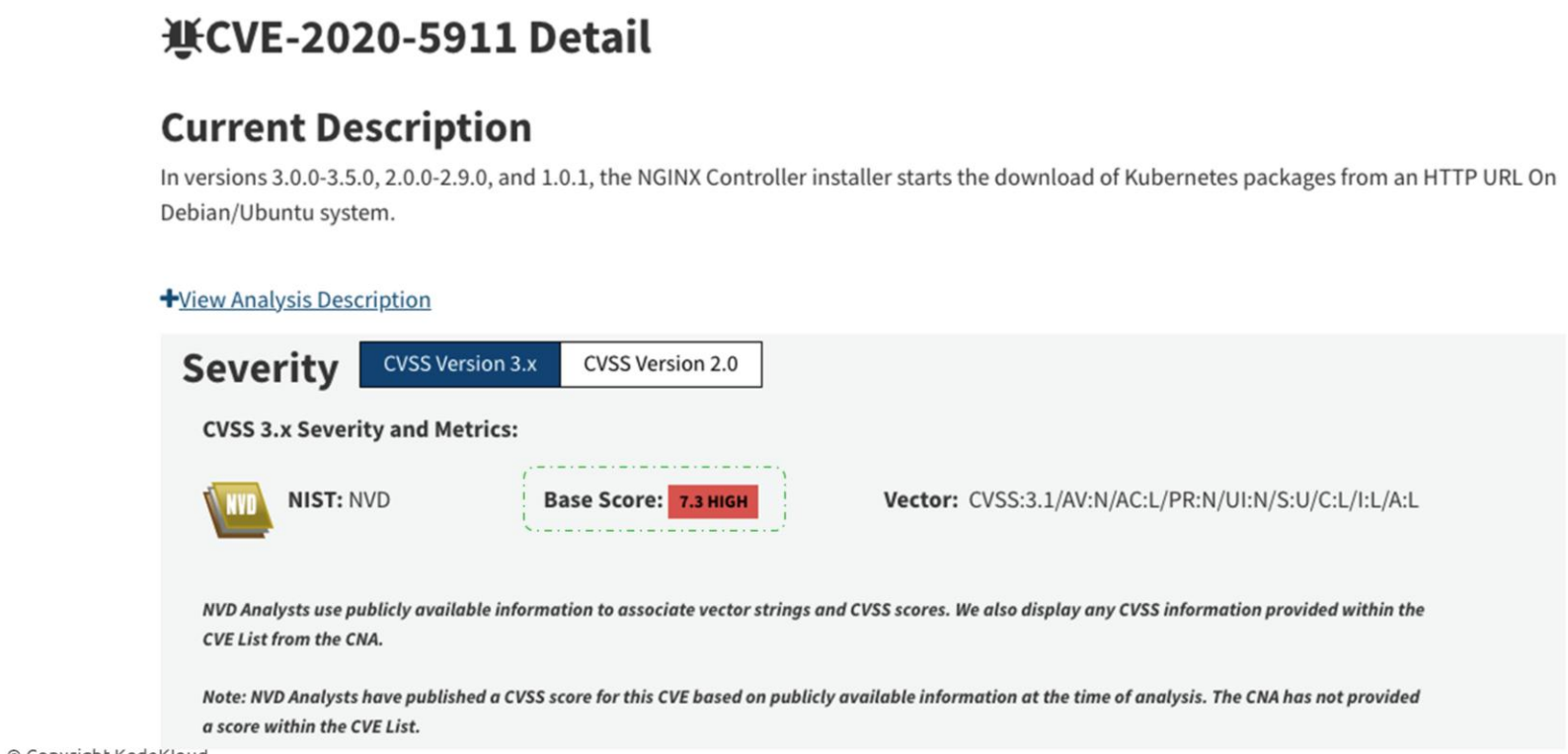
CVSS v3.0 Ratings

Severity	Base Score Range
None	0.0
Low	0.1-3.9
Medium	4.0-6.9
High	7.0-8.9
Critical	9.0-10.0

Each CVE gets what is called a severity score or severity rating (between 0 to 10 or None, Low, Medium, High, Critical). We'll often have to deal with a very high number of vulnerabilities and these scores can help us get an idea, at a glance, of what we should worry more or less about. They can help us prioritize what to take care of first. If we see a score of 9.5 or Critical, it's a good indicator that's a serious problem we should deal with immediately, and leave the ones with lower scores for later. A very high score can also be interpreted along the lines of "attacker can do a lot of damage with this" or "attacker can assume total control of your system with this". Lower scores might mean "they can do some things they

shouldn't be able to do, but nothing terribly bad".

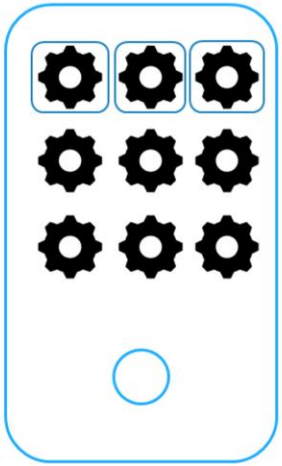
CVE Severity Scores



Here is an example of a CVE identified where the Nginx controller installer starts the download of Kubernetes packages from an HTTP URL on Debian and ubuntu systems.

The severity score for this is 7.3 which is high.

CVE Scanner



Name	
CVE-2021-21396	wire-server is an open-source b endpoint. The endpoint could be time, and cookie. A user on a W version 2021-03-02.
CVE-2021-21335	In the SPNEGO HTTP Authentica that have enabled basic authent
CVE-2020-8663	Envoy version 1.14.2, 1.13.2, 1
CVE-2020-8553	The Kubernetes ingress-nginx c nginx.ingress.kubernetes.io/aut
CVE-2020-7621	strong-nginx-controller through
CVE-2020-5911	In versions 3.0.0-3.5.0, 2.0.0-2
CVE-2020-5910	In versions 3.0.0-3.5.0, 2.0.0-2 authorized.
CVE-2020-5909	In versions 3.0.0-3.5.0, 2.0.0-2
CVE-2020-5901	In NGINX Controller 3.3.0-3.4.0
CVE-2020-5900	In versions 3.0.0-3.4.0, 2.0.0-2
CVE-2020-5899	In NGINX Controller 3.0.0-3.4.0 the database, to request a pass
CVE-2020-5895	On NGINX Controller versions 3. can make AVRD segmentation f
CVE-2020-5894	On versions 3.0.0-3.3.0, the NG

Our servers have numerous packages and services running on them. With containers and container orchestration involved there are additional processes running in the form of containers. While we discussed that we should keep these at a minimum and remove all unnecessary packages and software. How do we know how vulnerable the packages on our system are? And how vulnerable the containers are? That's where CVE Scanners can help us.

Container scanners look for vulnerabilities in the execution environment. More specifically, such a scanner looks at

applications included in that container and tells you what vulnerabilities they are known to have. So, for example, it sees that you have Nginx version 1.14.2 in there. Based on this version, it can then tell you a list of known vulnerabilities for that particular version.

Once the list of vulnerabilities are identified you may chose to upgrade to a newer version that fixes these problems. Or else implement additional security measures that prevents the problems from being exploited. Or remove that package if it is not needed. As dicussed before, the packages you have on your system it is more likely to have larger number of vulnerabilities. A solution is reduce attack surface by removing unnecessary packages.

Let’s look at one such vulnerability scanner in more detail.

Trivy

Debian/Ubuntu

Add repository to `/etc/apt/sources.list.d`.

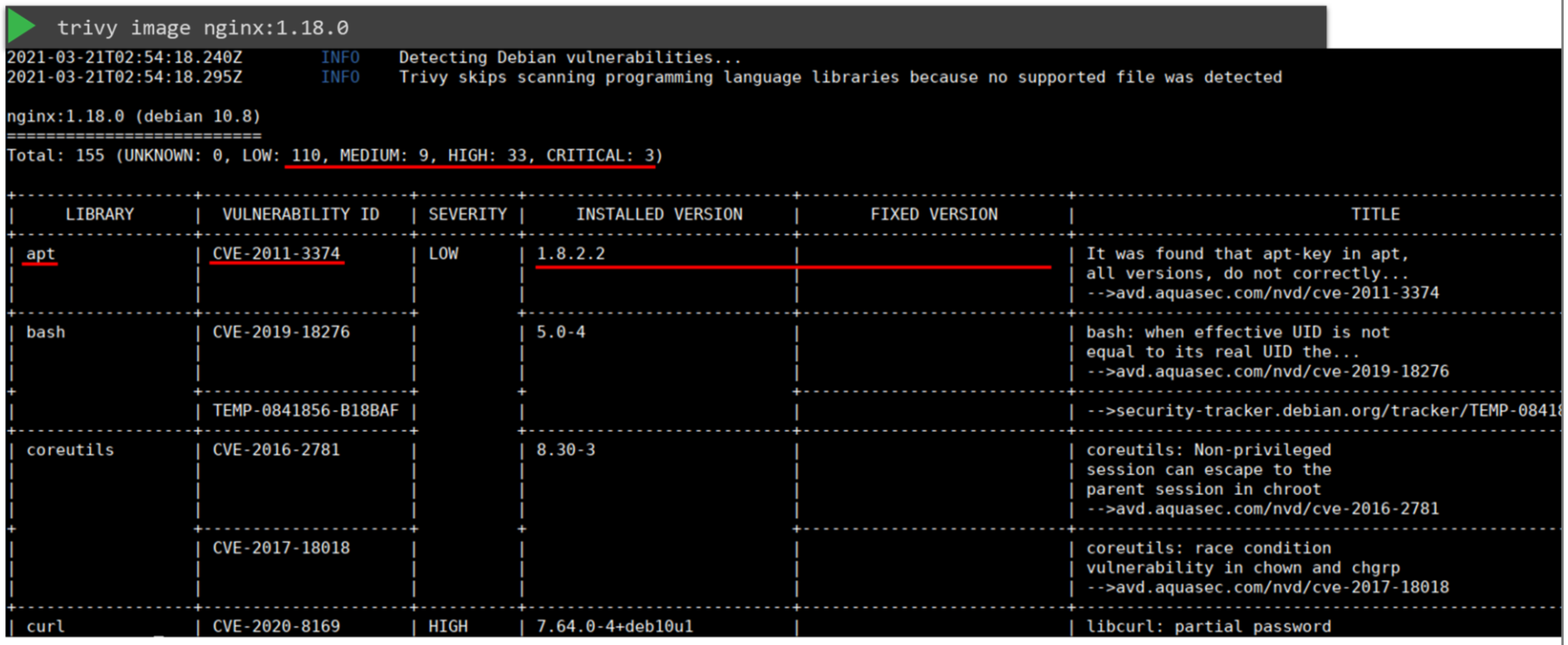
```
$ sudo apt-get install wget apt-transport-https gnupg lsb-release
$ wget -qO - https://aquasecurity.github.io/trivy-repo/deb/public.key | sudo apt-k
$ echo deb https://aquasecurity.github.io/trivy-repo/deb $(lsb_release -sc) main |
$ sudo apt-get update
$ sudo apt-get install trivy
```

https://aquasecurity.github.io/trivy/latest/installation/

Trivy is a simple and comprehensive Vulnerability Scanner for Containers and other artifacts from Aqua Security. And is suitable for integration with CI/CD Pipelines.

Installation instructions can be found in their documentation pages. Its as easy as installing the dependencies and installing the trivy package. You'll do this yourself in the upcoming labs.

Trivy



Once installed initiating a scan is as easy as running the trivy image command followed by the name of the image. Specify the name of the image as you would specify it in a docker run command.

It then runs a vulnerability scan and returns a summary of all vulnerabilities detected in that image. In this example its identified 110 Low risk vulnerability, 9 medium risk, 33 high and 3 critical ones.

It then lists details about each of these vulnerability, its ID, severity and description about the issue. Let’s pick the first vulnerability in this case the one for the apt library.

Trivy

```
trivy image --severity CRITICAL nginx:1.18.0
```

```
trivy image --severity CRITICAL,HIGH nginx:1.18.0
```

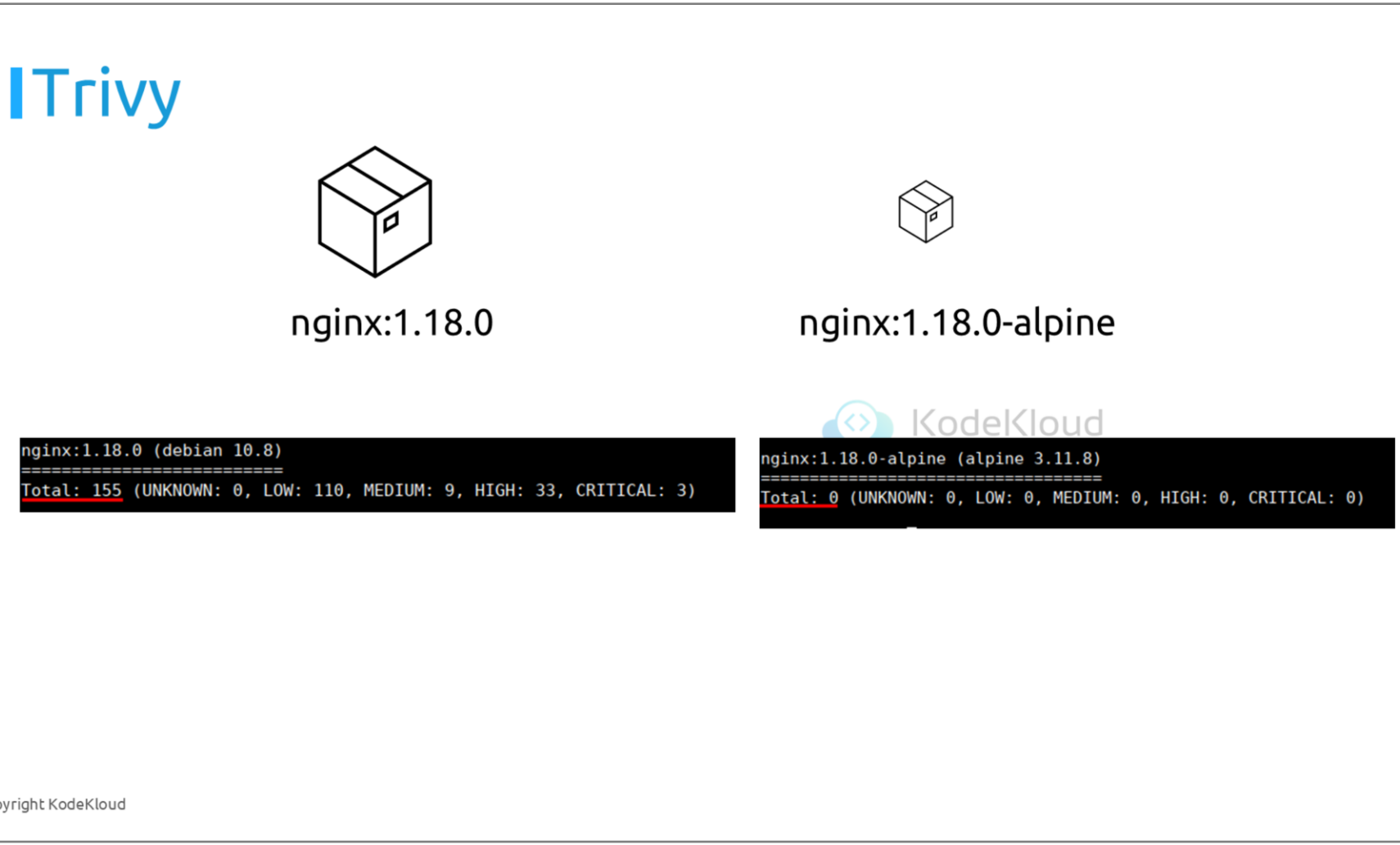
```
trivy image --ignore-unfixed nginx:1.18.0
```



```
docker save nginx:1.18.0 > nginx.tar
```

```
trivy image --input archive.tar
```


The trivy command can take additional options to filter results. Such as the severity flag takes a severity and only lists vulnerabilities of that severity level. You can specify multiple values for the same option like this.
And if we're only interested in the CVEs that we can immediately fix by upgrading vulnerable software packages, we can use the --ignore-unfixed command line option.
Say we save a docker image into a tar archived format like this with the docker save command. We can use trivy to scan the image in the archive format, using the input option and by specifying the name of the tar archive like this.



Earlier we discussed that one of the solutions to reduce vulnerabilities is by reducing the attack surface by using images with minimal packages in them. Here are 2 images. The nginx 1.18.0 and the nginx 1.18.0-alpine. The output from a trivy scan reveals that the first has 155 total vulnerabilities and the second one as 0. So you must always chose an image that is stripped down of all unnecessary packages.

Best Practices

- Continuously rescan images
- Kubernetes Admission Controllers to scan images
- Have your own repository with pre-scanned images ready to go
- Integrate scanning into your CI/CD pipeline

Let’s talk about some of the best practices. So, we just saw that we scanned that Nginx container and it reported zero CVEs. However, few months later, some new security issues might be discovered. That's why you want to periodically rescan your images, and make sure they're still safe to use.

Integrate scanning into deployment using admission controllers in Kubernetes. We discussed about admission controllers and webhooks earlier in this course. Admission controllers can be used to initiate scan of images every time before a pod

gets deployed.

As you can imagine that might delay the deployment process and so an alternate approach is to have an internal registry with pre-scanned images that are ready to go. This way you don’t have to scan the image each time it is deployed.

And finally integrate scanning into your CI/CD pipeline for your custom applications. This way every time code is pushed and a new image of the application is built, a scan of the image is automatically performed and vulnerabilities are reported at the source.

Well that’s it for now. Head over to the labs and practice working with Image scanners.

In this lecture we will talk about securing images.

Image

nginx-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
  - name: nginx
    image: nginx
```


We will start with basics of image names and then work our way towards secure images repositories and how to configure your pods to use images from secure repositories. We deployed a lot of different kinds of pods hosting different kinds of applications throughout this course, like webapps, databases, redis cache etc. Let’s look at a pod definition file for instance.

Here we have used the nginx image to deploy an nginx container.

Image

Let’s take a closer look at this image name. The name is nginx. But what is this image and where is this image pulled from? This name follows Docker’s image naming convention. NGINX here is the image or the repository name.

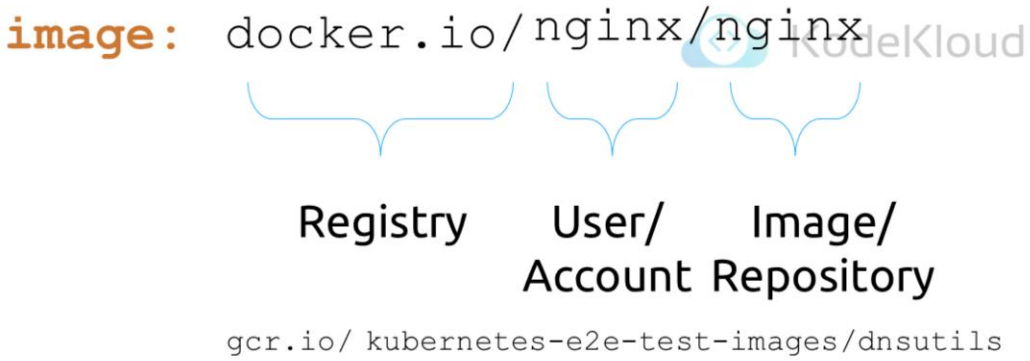
Image

When you say nginx, its actually nginx/nginx. The first part stands for the user or account name. So if you don't provide a repository name, it assumes it to be the same as the repository name which in this case is nginx.

If you where to create your own account and create your own repositories or images under it, then you would use a similar pattern.

Now where are these images stored and pulled from?

Image



Since we have not specified the location where these images are to be pulled from, it is assumed to be on docker's default registry – dockerhub. The dns name for which is docker.io. The registry is where all the images are stored. Whenever you create a new image or update an image, you push it to the registry and every time anyone deploys this application it is pulled from the registry. There are many other popular registries as well.

Google's registry is at gcr.io, where a lot of Kubernetes related images are stored. Like the ones used for performing end

to end tests on the cluster.

These are all publicly accessible images, that anyone can download and access. When you have applications built in-house that shouldn’t be made available to the public, hosting an internal private registry may be a good solution. Many cloud service providers such as AWS, Azure or GCP provide a private registry by default.

Private Repository

```
docker login private-registry.io
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to
https://hub.docker.com to create one.
Username: registry-user
Password:
WARNING! Your password will be stored unencrypted in /home/vagrant/.docker/config.json.

Login Succeeded
```



```
docker run private-registry.io/apps/internal-app
```


On any of these solutions, be it on Docker hub or googles registry, or your internal private registry, you may chose to make a repository private, so that it can be accessed using a set of credentials.

From a Docker perspective to run a container using a private image, you first login to your private-registry using the docker login command. Input your credentials. Once successful run the application using private-registry.

Private Repository

```
docker login private-registry.io
```

```
docker run private-registry.io/apps/internal-app
```

nginx-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
  - name: nginx
    image:
  imagePullSecrets:
  - name: regcred
```

```
kubectl create secret docker-registry regcred \
    --docker-server= private-registry.io    \
    --docker-username= registry-user  \
    --docker-password= registry-password  \
    --docker-email= registry-user@org.com
```


Going back to our pod definition file, to use an image from our private registry we replace the image name with the full path to the one in the private registry. But how do we implement the login? How does Kubernetes get the credentials to access the private registry?

We first create a secret with the credentials. The secret is of type docker-registry and we name it regcred. We then specify the register server name , the username to access the registry, the password and the email address of the user.

We then specify the secret inside our pod definition file under the imagePullSecrets section. When the pod is created kubernetes uses the credentials from this secret to pull images.

Well that’s it for this lecture.

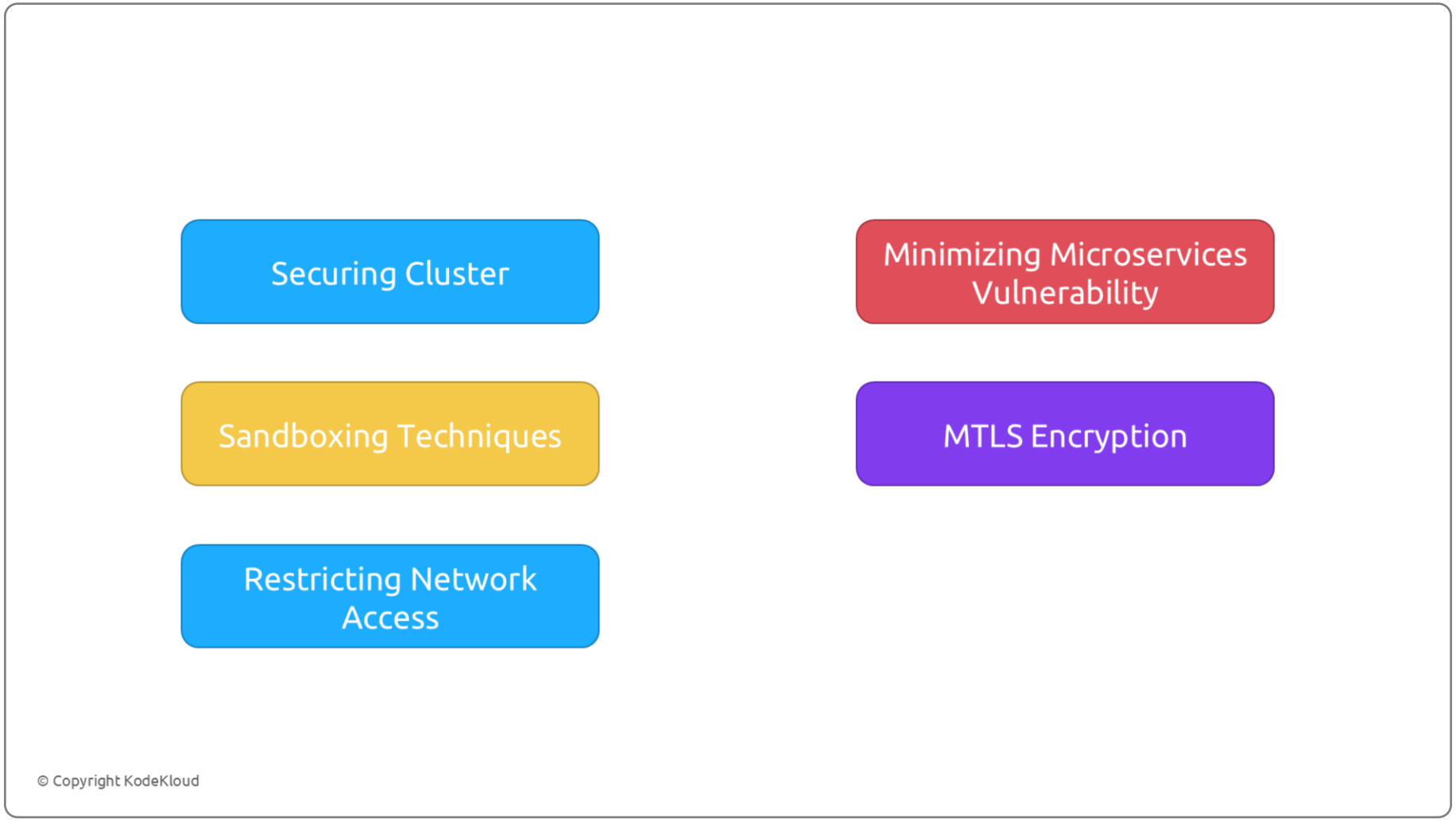
References

https://kubernetes.io/docs/tasks/configure-pod-container/pull-image-private-registry/

Perform Behaviour Analytics of syscalls

KodeKloud

In this lecture, we will learn how to monitor our Kubernetes Clusters against abnormal behavior, possible ongoing cyber attacks and security breaches.

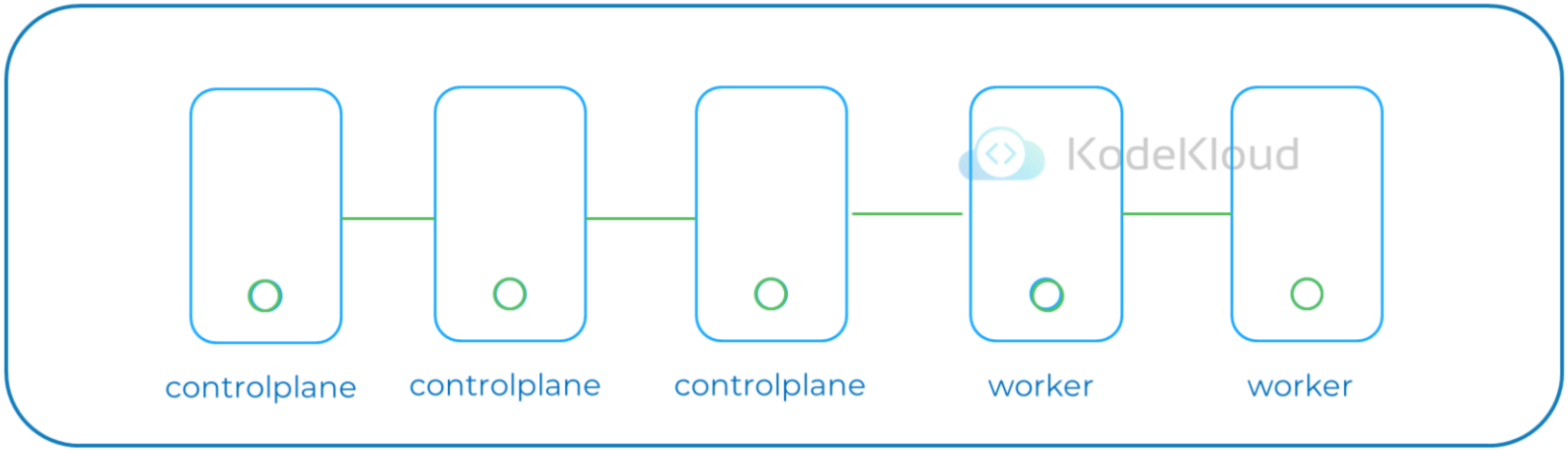


Throughout the course, we have seen different ways of securing our kubernetes infrastructure – such as securing the controlplane components, the workloads, limiting access to containers using sandboxing techniques, using mTLS, restricting network access e.t.c.

Even if we use all of these techniques to secure our infrastructure, there is no absolute guarantee that an attack will not happen in the future.

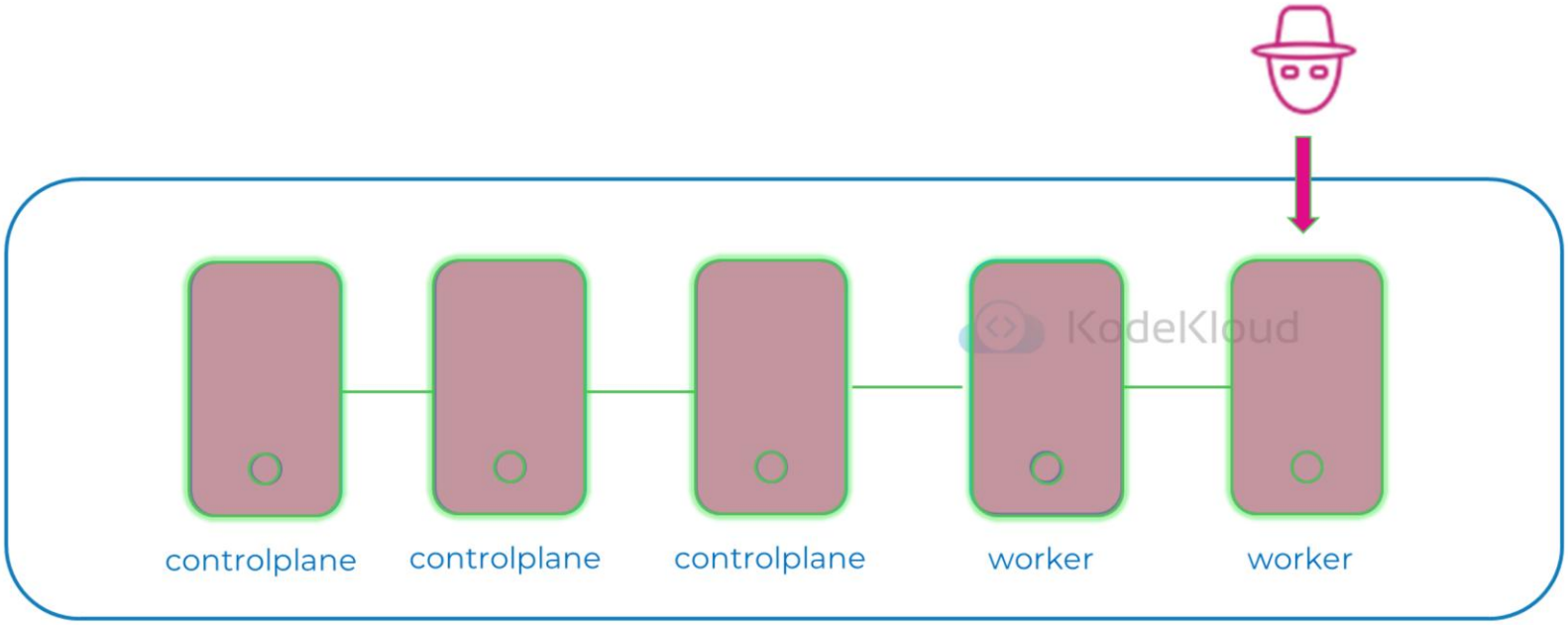
Theoretically, there will always be a way for an attacker to get inside.

So we still have to prepare for scenarios where something goes wrong and the containers become compromised.



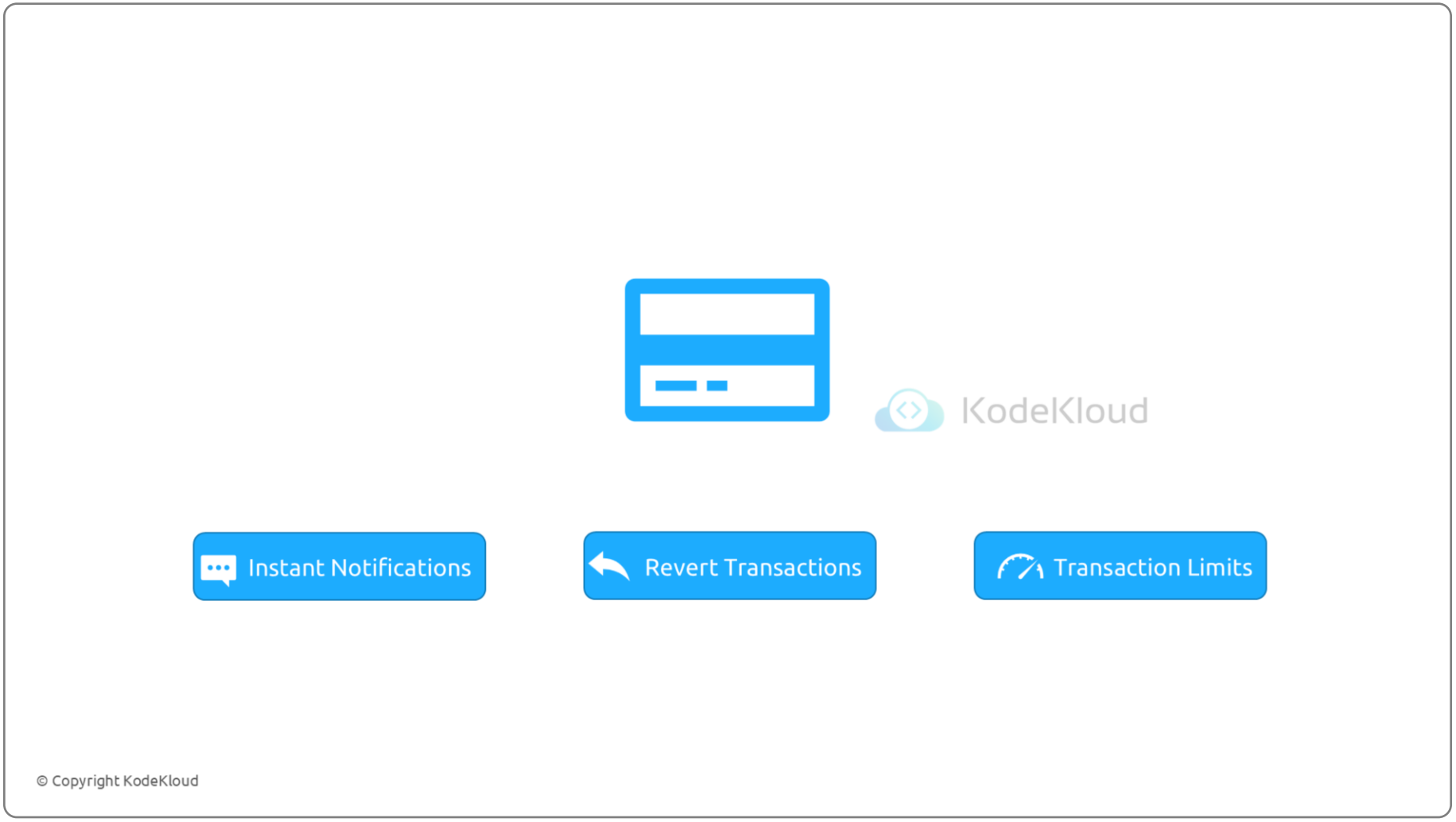
It might seem that if an attacker already got into our systems, it's already too late and all is lost. However, this may not necessarily be true.

The sooner we find out about something that went wrong, the better it is.



It might seem that if an attacker already got into our systems, it's already too late and all is lost. However, this may not necessarily be true.

The sooner we find out about something that went wrong, the better it is.



To understand this, let’s use an analogy of using credit or debit cards.<c>

Security has improved a lot in recent years. Smart chips have been added to cards to make cloning them much harder and make authentication with the ATMs much more reliable.

But even with all the added security measures, someone can still physically steal your card, and, if they know your PIN, they

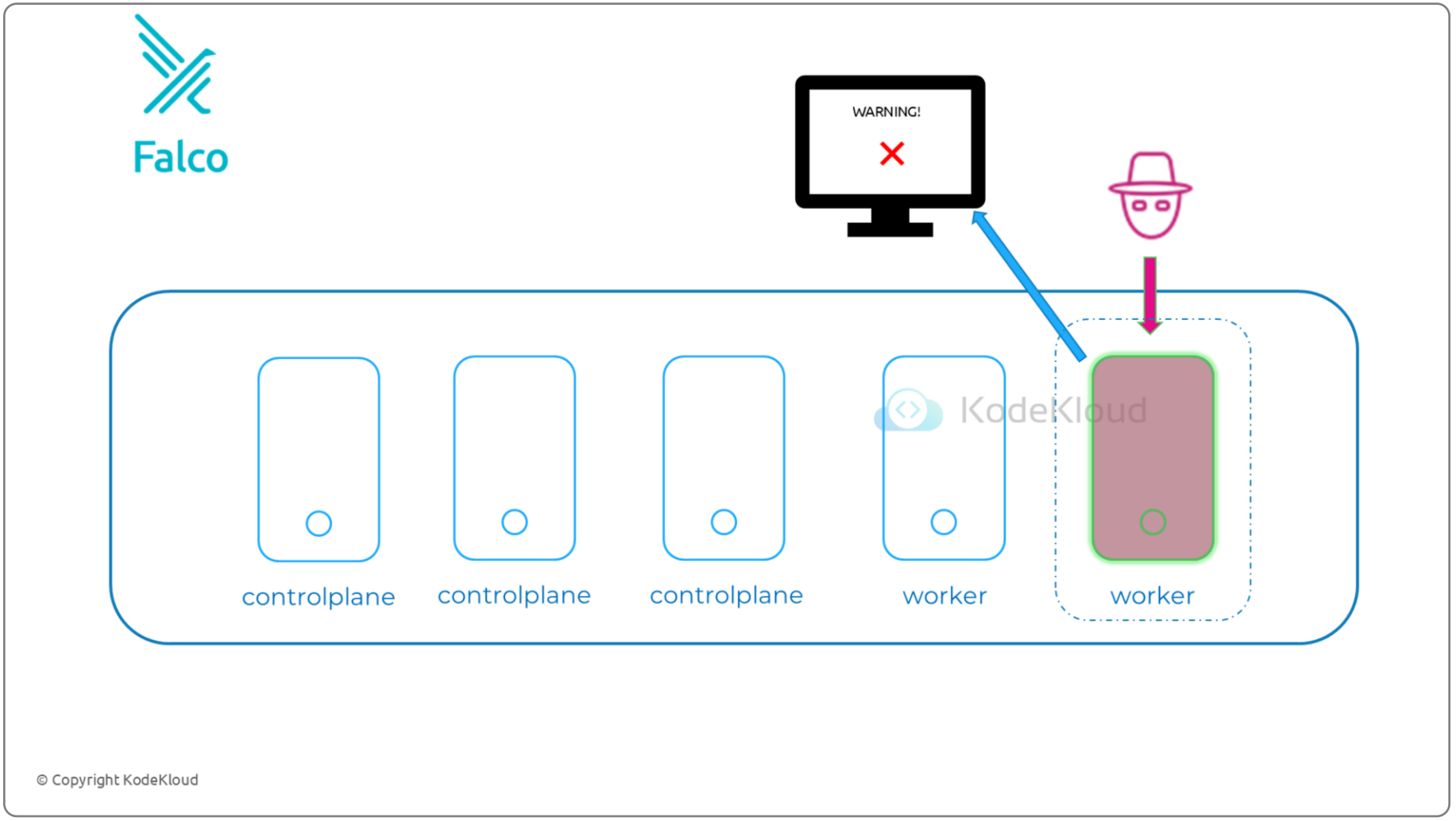
can withdraw money. Now they can use contact less methods to withdraw money or make a purchase.

In the past, without smartphones, if someone stole money from you today, you may only find out about it days or weeks later when you check your bank balance.
By then, it could have been too late to reverse the transaction(s).

But now imagine that you get a notification on your smartphone<c> every time your card is used. As soon as someone steals from you, you get notified. Since you instantly find out about this, you can quickly call your bank and tell them about the problem. They can track what happened and you get your money back, since transactions are still fresh they can quite easily revert them.<c> Even if the attack happened, the damage can be minimized and it can be repaired if you react quickly.

To limit the damage further,<c> you can set up daily transaction limits so in case of a compromise the card can only be used for withdrawing limited funds which can again be reversed if identified in time.

The same is applicable for computer systems that have been breached.

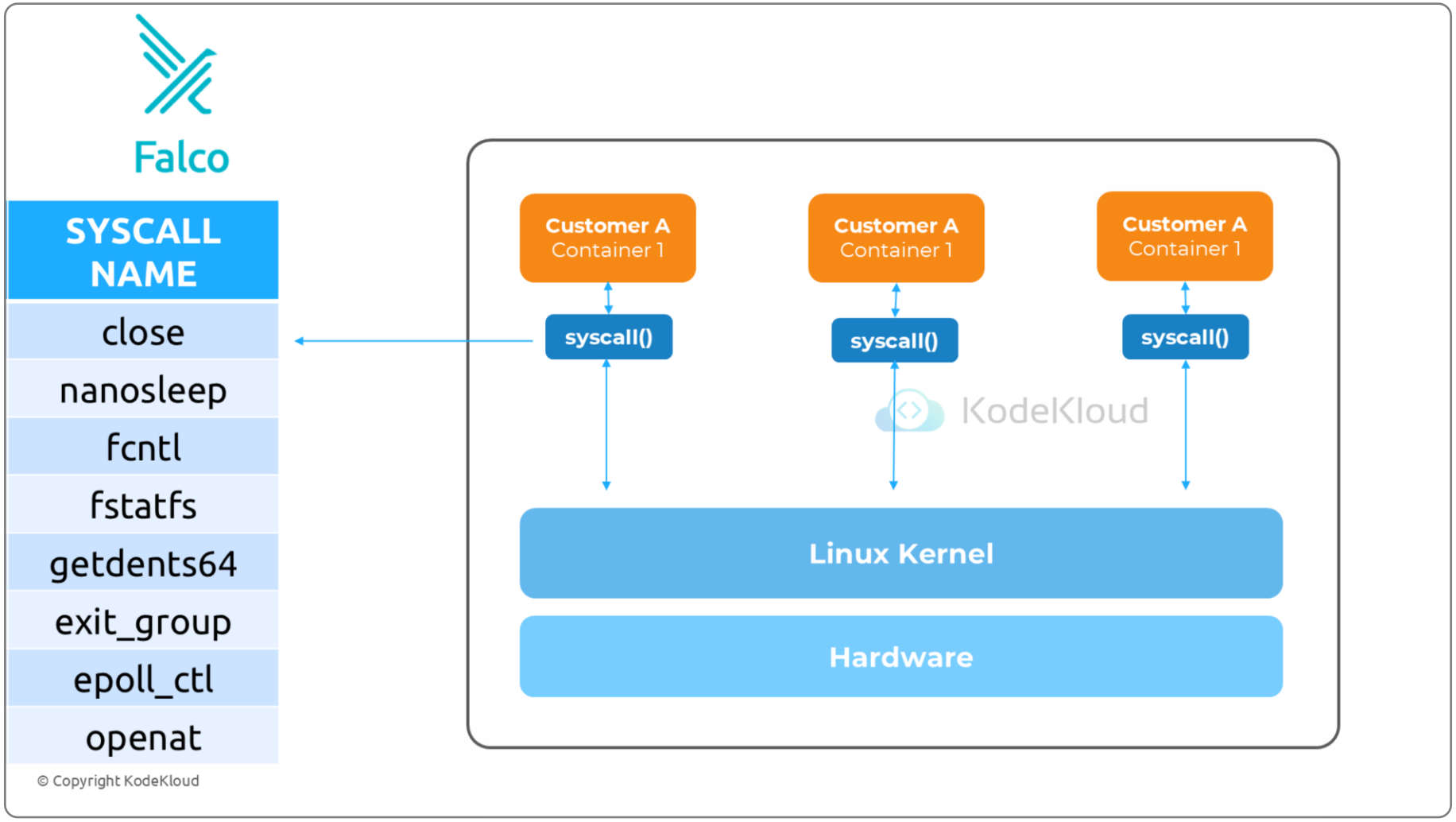


If a breach does occur, it is crucial that we identify it as soon as possible. By reacting quickly, we can prevent the damage from spreading to other systems and reduce the blast radius.

This would then allow us to replace the compromised nodes/pods and patch whatever security hole was used to get in, so it can never be used again.

So how do we identify breaches that have already occurred in our Kubernetes Cluster?

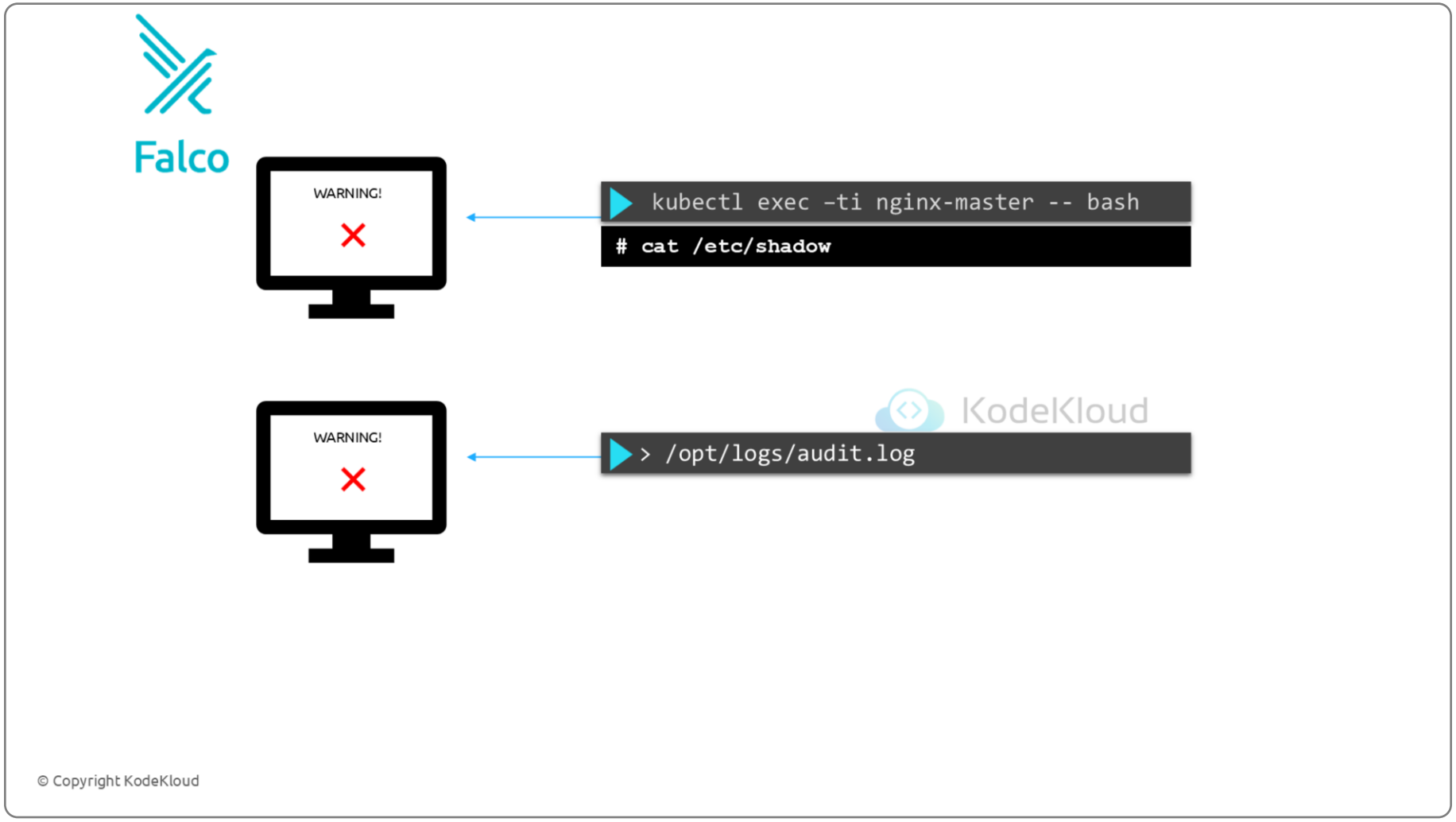
<c>For this we can make use of tools such as Falco from sysdig.



In the Earlier sections of the course, we learned about syscalls in detail and saw how tools like strace and aquasec tracee can be used to analyze the syscalls that are used by the application inside a pod.

When we have hundreds of applications running on several hundred pods generating thousands of syscalls it is quite meaning less to just monitor the syscalls.

What we need is a way to analyze the syscalls and filter events that are suspicious.



For example, some one accessing the bash shell of a container or a program trying to access the /etc/shadow file where the password data is stored can be considered to be suspicious activity.

Lets take a look at some more examples - Attackers often want to erase tracks that they compromised your system (so they can keep it compromised for a longer time). So they often delete some parts of logs that tracked how they got into the system or how they changed something. Normally, an administrator rarely has reason to delete (recent) logs so this

activity is can be considered anomalous and can be used as an early sign of intrusion.

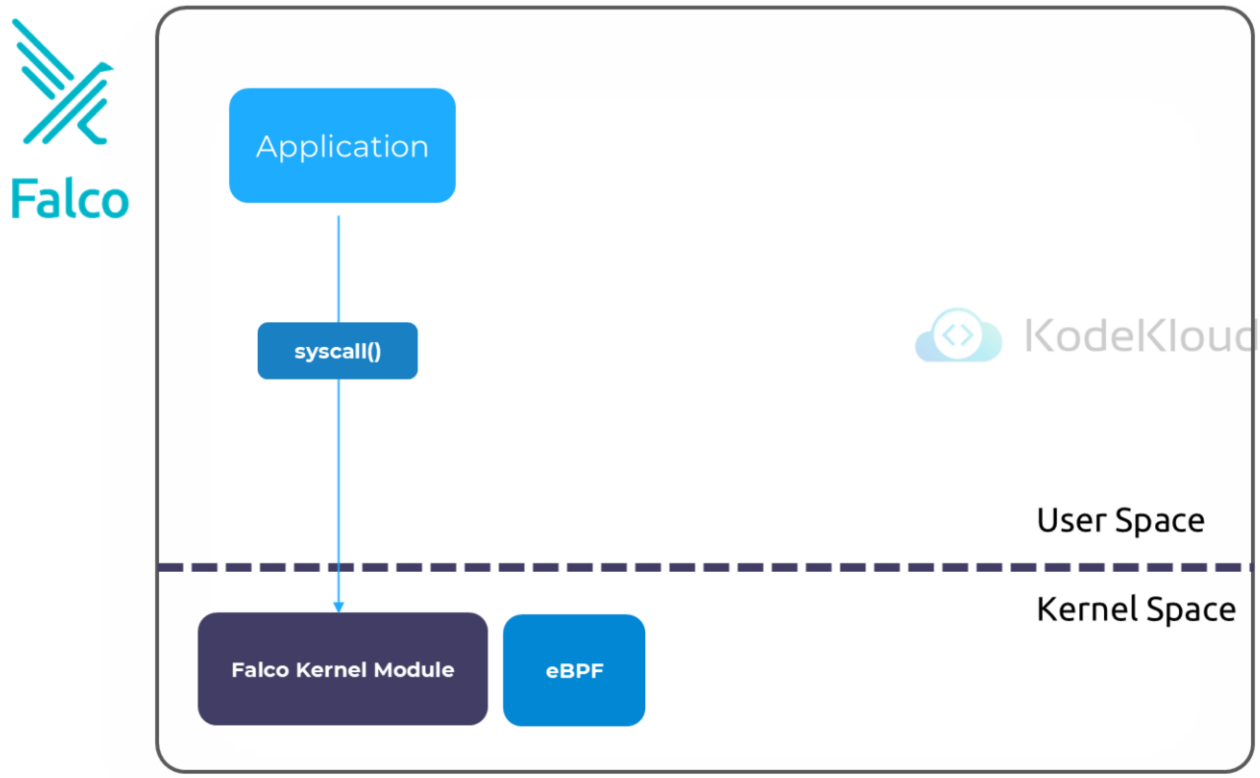
While it may have been an administrator who had legitimately accessed the containers shell and the program accessing files under /etc/ for non-malicious reasons, falco can capture this event and then send alerts using various notification channels.

Well, that’s it for this lecture. In the upcoming lecture, we will see how to install Falco in our Kubernetes Cluster and use it to detect and analyze threats.

Falco Overview and Installation

In this lecture, we will install Falco in our Kubernetes cluster and then use it to detect and analyze threats.

Falco Architecture



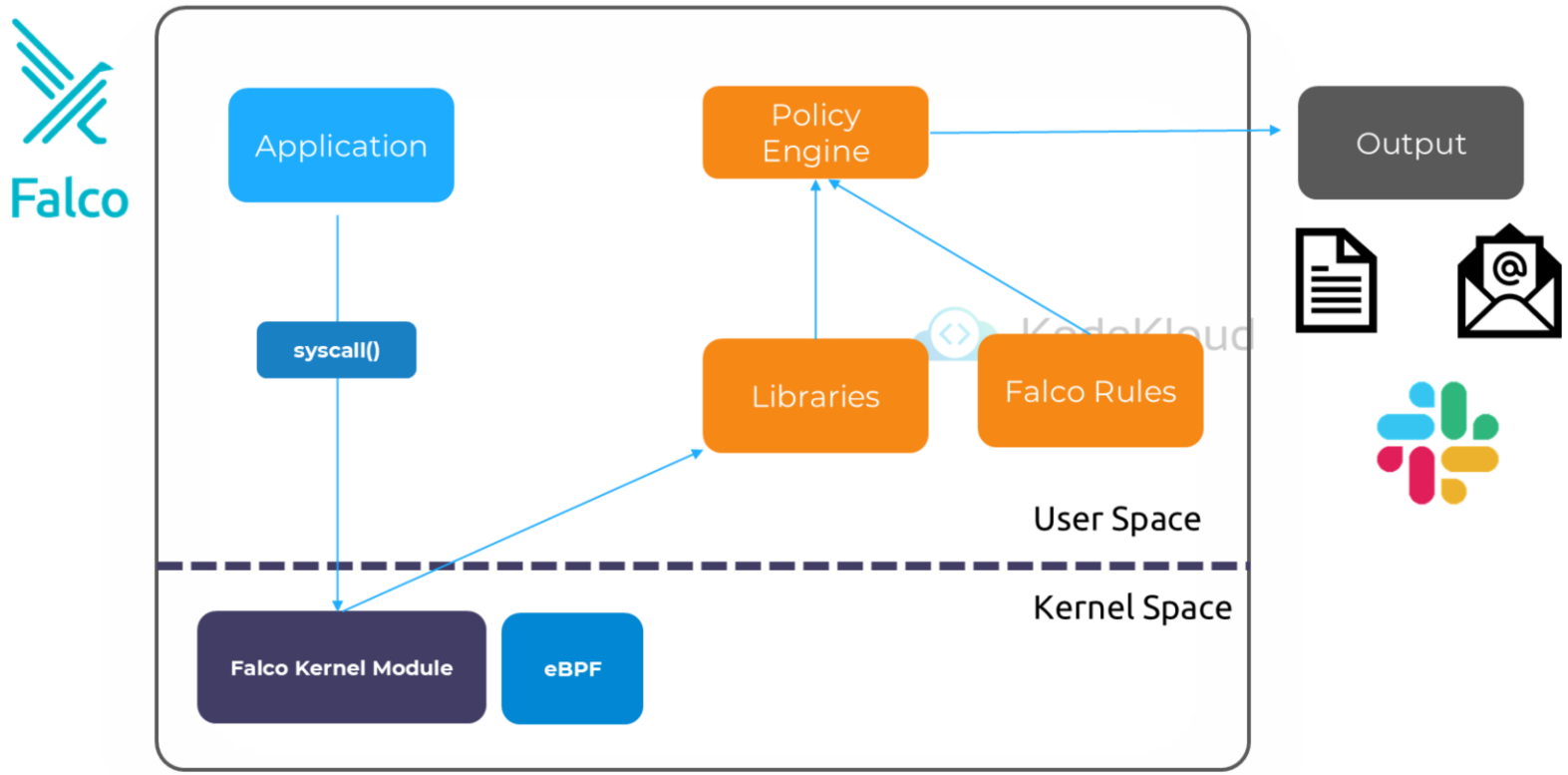
Before we install Falco and start using it. Lets take a quick look at the high level overview of how Falco works.

<c>Falco needs to see what system calls are coming though from the applications in the user space into the Linux kernel. That means it somehow has to insert itself into the kernel, to "sit in the middle" and see what's coming in. One way Falco does this is by making use of a Kernel Module.<c>

However, this basically means that we insert additional code, right inside the Linux code, so it's pretty intrusive. As such, some managed Kubernetes service providers do not allow us to do this.

But Falco can also interact with the kernel through what is called eBPF<c> or the extended berkely packet filter. We saw this briefly when we got introduced to AquaSec Tracee which also uses eBPF.
eBPF is somewhat less intrusive and safer, so some providers are more inclined to allow this method.

Falco Architecture



The system calls are then analyzed by the sysdig libraries in the userspace.

<c>The events are then filtered by the Falco Policy engine by making use of pre-defined rules that can detect whether the event was suspicious or not.

<c>Such an event is then alerted via outputs such as syslog files, standard output or alerts to other programs email alerts.

Slack channel alerts e.t.c

Install as a Package

```
curl -s https://falco.org/repo/falcosecurity-3672BA8F.asc | apt-key add -
```

```
echo "deb https://download.falco.org/packages/deb stable main" | tee -a /etc/apt/sources.list.d/falcosecurity.list
```

```
apt update -y
```

```
apt get install -y linux-headers-$(uname -r)
```

```
apt install -y falco
```

```
systemctl start falco
```


https://falco.org/docs/getting-started/installation/

As mentioned before, Falco needs to be able to interact with the kernel. If we install Falco as a regular software package on the Linux Operating system, this will install the Falco Kernel Module to inspect the syscalls.

To do this, use the steps provided in the falco installation documentation.

Once installed, enable and start the falco service. <c>

One major advantage of running falco as a service on the node directly is that incase of a compromise, Falco is isolated from Kubernetes and can continue to detect and alert suspicious behaviour.

Install as a DaemonSet

```
helm repo add falcosecurity https://falcosecurity.github.io/charts
```

```
helm repo update
```

```
helm install falco falcosecurity/falco
```

```
NAME: falco
LAST DEPLOYED: Wed Mar  7 20:19:25 2021
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
Falco agents are spinning up on each node in your cluster. After a few
seconds, they are going to start monitoring your containers looking for
security issues.


No further action should be required.
```


https://github.com/falcosecurity/charts/tree/master/falco

Alternatively, if installing directly on the node is not possible, we can also run Falco as a DaemonSet on the nodes of the cluster.

The easiest way to deploy this DaemonSet is by using the helm chart. For the detailed steps, please checkout the reference section.

Install as a DaemonSet

```
kubectl get pods
```

```
NAME          READY   STATUS    RESTARTS   AGE
falco-7grdt   1/1     Running   0          2m21s
falco-tmq28   1/1     Running   0          2m21s
```


https://github.com/falcosecurity/charts/tree/master/falco

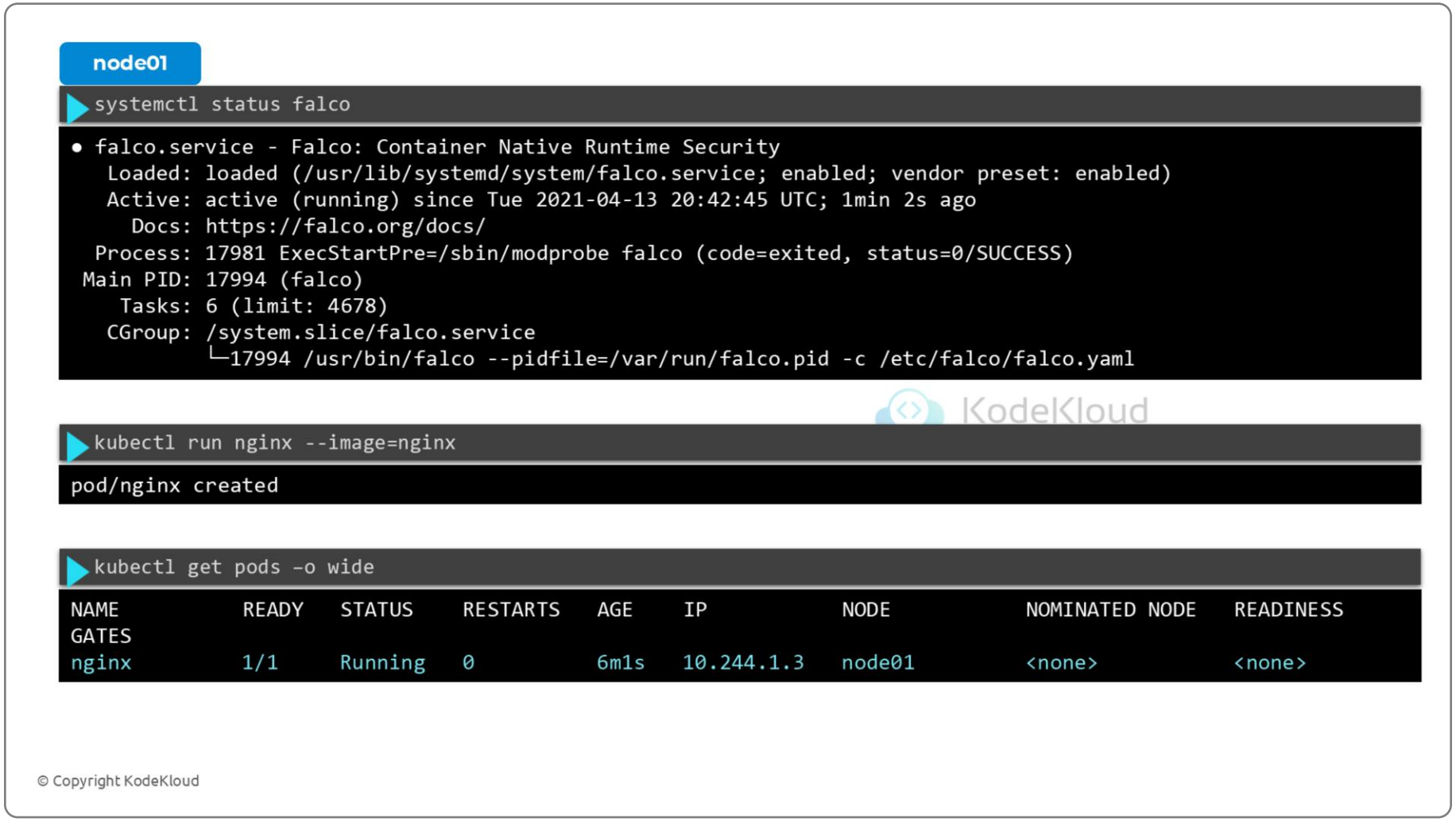
Once installed, you should see the falco pods running on all the nodes of the Cluster.

And that’s it. We are now ready to use Falco Rules to detect and alert anomalous behaviour.

Use Falco to Detect Threats

KodeKloud

Now that we have installed Falco in our cluster, lets use it to detect and alert suspicious behavior.

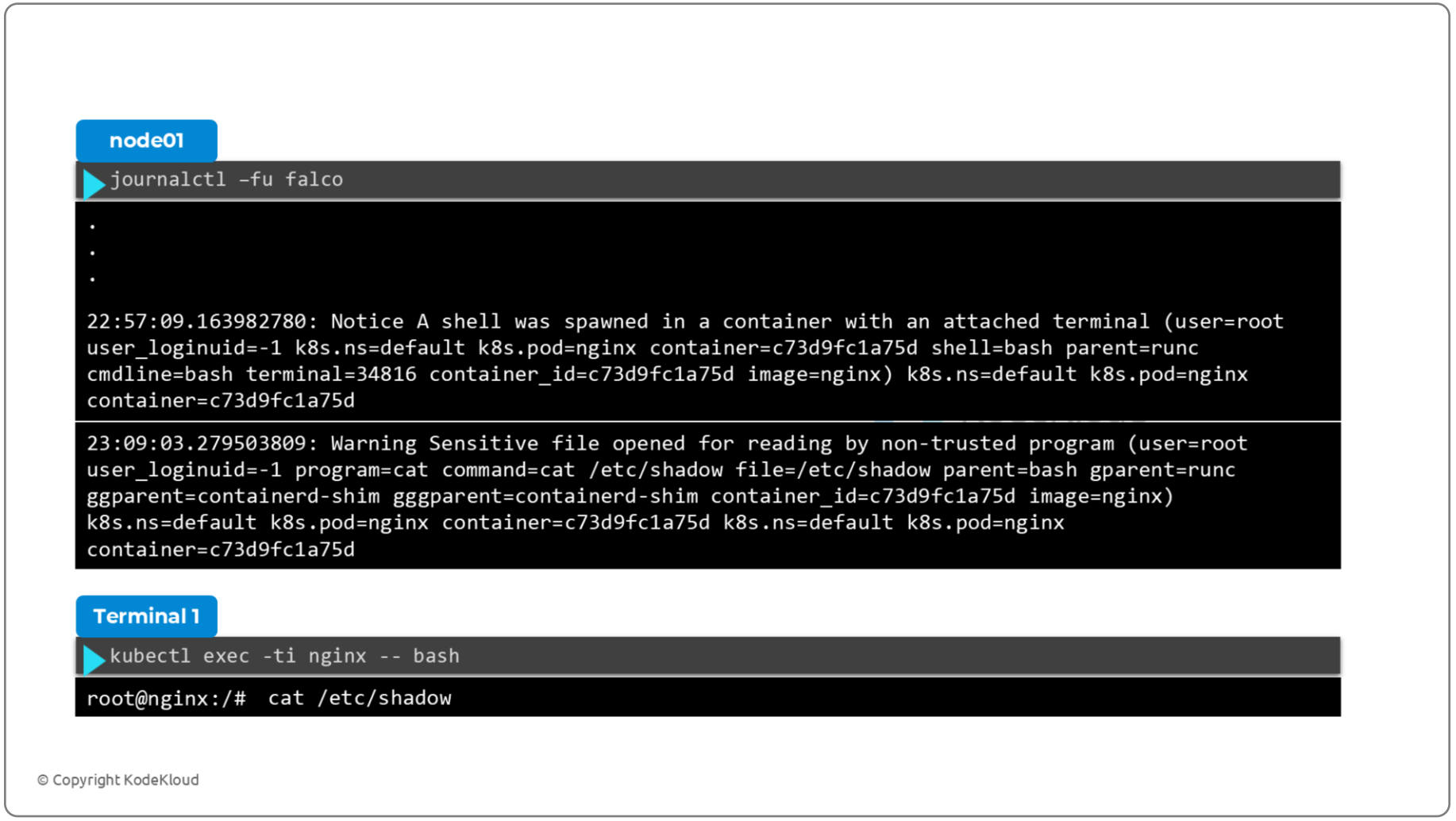


Before we begin, let us make sure that Falco is running as expected on the nodes of the cluster.

Let us assume that Falco was directly installed on the host as a package. <c>So, we can make use of “systemctl status falco” to check if it is running on the host.

Next, let us create an nginx pod.

In this case, we can see that it is running on node01.



On a separate terminal, SSH to node01, and run journalctl –fu falco to inspect the events generate by falco.

The –f flag will make sure that the new events are continuously printed to the screen as soon as they are created.

You may notice a lot of existing logs. Let’s ignore those for now.

Next, back on Terminal1, open a shell on the nginx container by using the kubectl exec command.

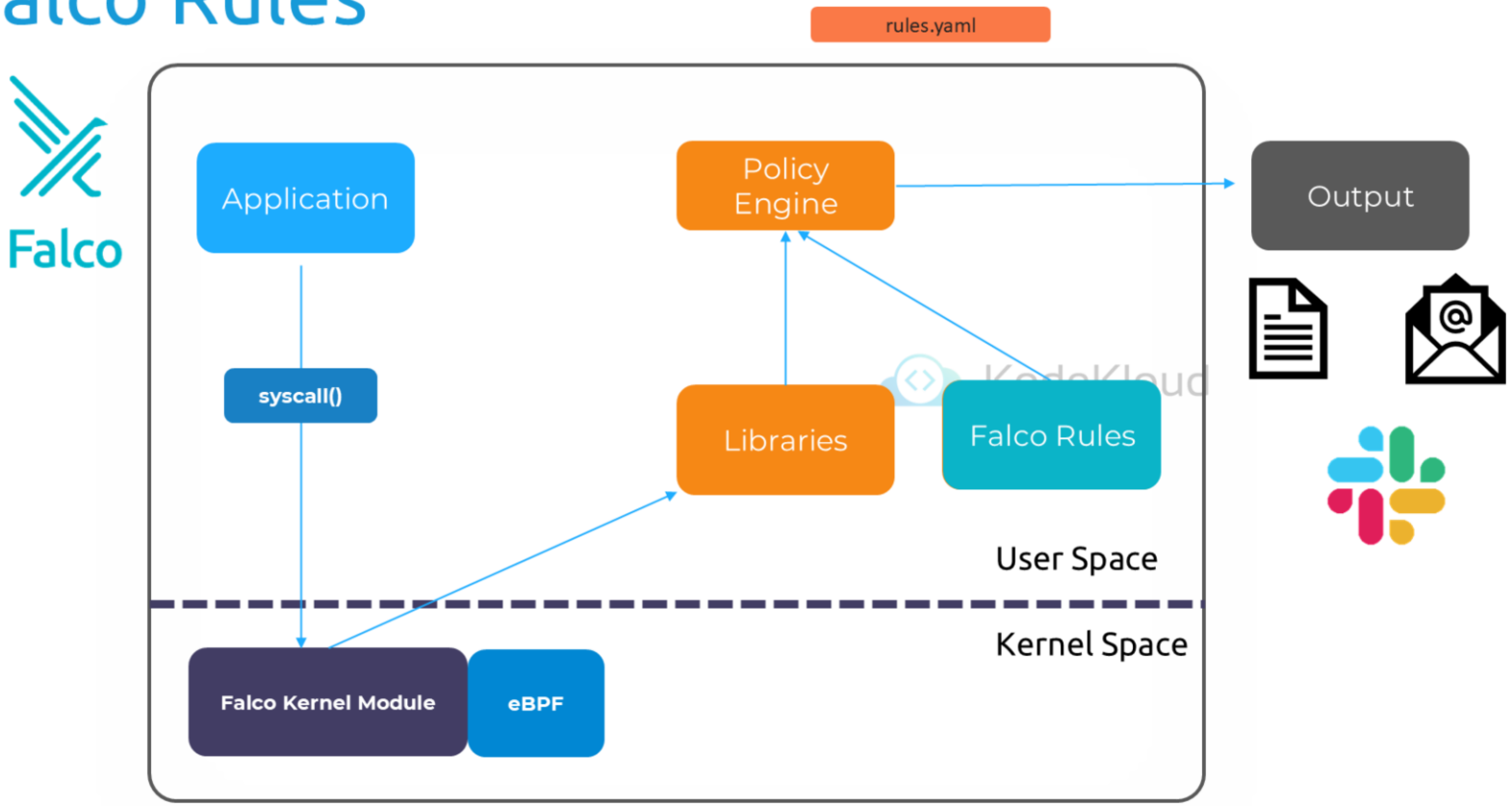
Immediately an alert should be visible in the journal logs that A Shell was spawned inside the container of the Nginx Pod. It will also provide you other outputs such as the container ID, Image used for the NGINX pod, the namespace e.t.c.

Next, let us read the contents of the /etc/shadow file inside the container on terminal 1. Another warning is immediately displayed – notifying that a sensitive file was opened.

Falco knew that some one opened a shell inside the Nginx container and opened the shadow file.

So how did that happen?

Falco Rules



In the previous lecture, we had a quick overview of how Falco works. To define which events should be considered as an anomaly and sent as an alert, we use Falco rules

Falco implements several rules by default - one of which alerted us that a shell was opened inside a container and another one that warned us that sensitive files read inside the container.

These default rules are defined within what is known as the Rules file.

Falco Rules

rules.yaml

```
- rule: <Name of the Rule>
  desc: <Detailed Description of the Rule>
  condition:   <When to filter events matching the rule>
  output:  <Output to be generated for the Event>
  priority:  <Severity of the event>
```

KodeKloud

A Falco rules file is a YAML file just like definition for Kubernetes Objects. And this file contains three types of elements - rules, lists and macros .

<c>The first element we are going to look at is is the rules as it is the fundamental element of a rules file.

The rules element defines all the conditions under which an alert should be triggered and consists of 5 mandatory keys.

<c>The first key is rule which accepts a unique name for the rule.

<c>We have the description key which is the long description of what the rule detects.

<c>We then have the condition key which is a filtering expression that is applied against events to match the rule . We will see this in more detail soon.

<c>Then we have the output key which is the output message to be logged if an event matches the rule.

<c>And finally we have the priority which is the severity of the alert.

Using this file as the template, let us now create our own custom rule from scratch. This rule should will will alerts us if a shell is opened in the container.

Falco Rules

rules.yaml

```
- rule: Detect Shell inside a container
  desc: Alert if a shell such as bash is open inside the container
  condition: container.id != host and proc.name = bash
  output: Bash Shell Opened (user=%user.name %container.id)
  priority: WARNING
```

Lets use "Detect Shell inside a container" as our rule name.
And let us also add an appropriate description for this rule.
For the condition, lets use a logical expression like this.Here we want an event to be filtered if the bash process was run but only when it is run inside a container. If it was opened within the host, the event should be ignored.

Next lets define a custom output message to be logged – One which reads - "Bash Shell Opened and include the user that

opened the bash shell as well as the containerID.

And finally, we can set priority of this event. Let us set it to Warning, for this example.

Let us now try to understand how this rule is configured. The first two fields – rule and description is self explanatory. So - lets take a closer look at the condition, output and priority.

Falco Rules

rules.yaml

```
- rule: Detect Shell inside a container
  desc: Alert if a shell such as bash is open inside the container
  condition: container.id != host and proc.name = bash
  output: Bash Opened (user=%user.name container=%container.id)
  priority: WARNING
```

KodeKloud

container.id

proc.name

fd.name

evt.type

user.name

container.image.repository

https://falco.org/docs/rules/supported-fields/

Let’s look at the condition first. Here we want the event to be matched only if the bash process was run inside the container. For that we make use of a filter <c> container.id which should not be equal to host and another filter <c>proc.name equal to bash.

So what is container.id and proc.name here?

These are known as Sysdig filters and they are extensively used by Falco. The Falco Policy engine makes use if these filters to extract information about an event such as the container ID, process name and several other.

<c>Container.id filters the name of the container.

<c>Proc.name – the name of the process.

Similarly, some of the other commonly used filters are, <c>fd.name which is the name of the file descriptor. This is used to match events against a specific file such as reading or writing to the file.

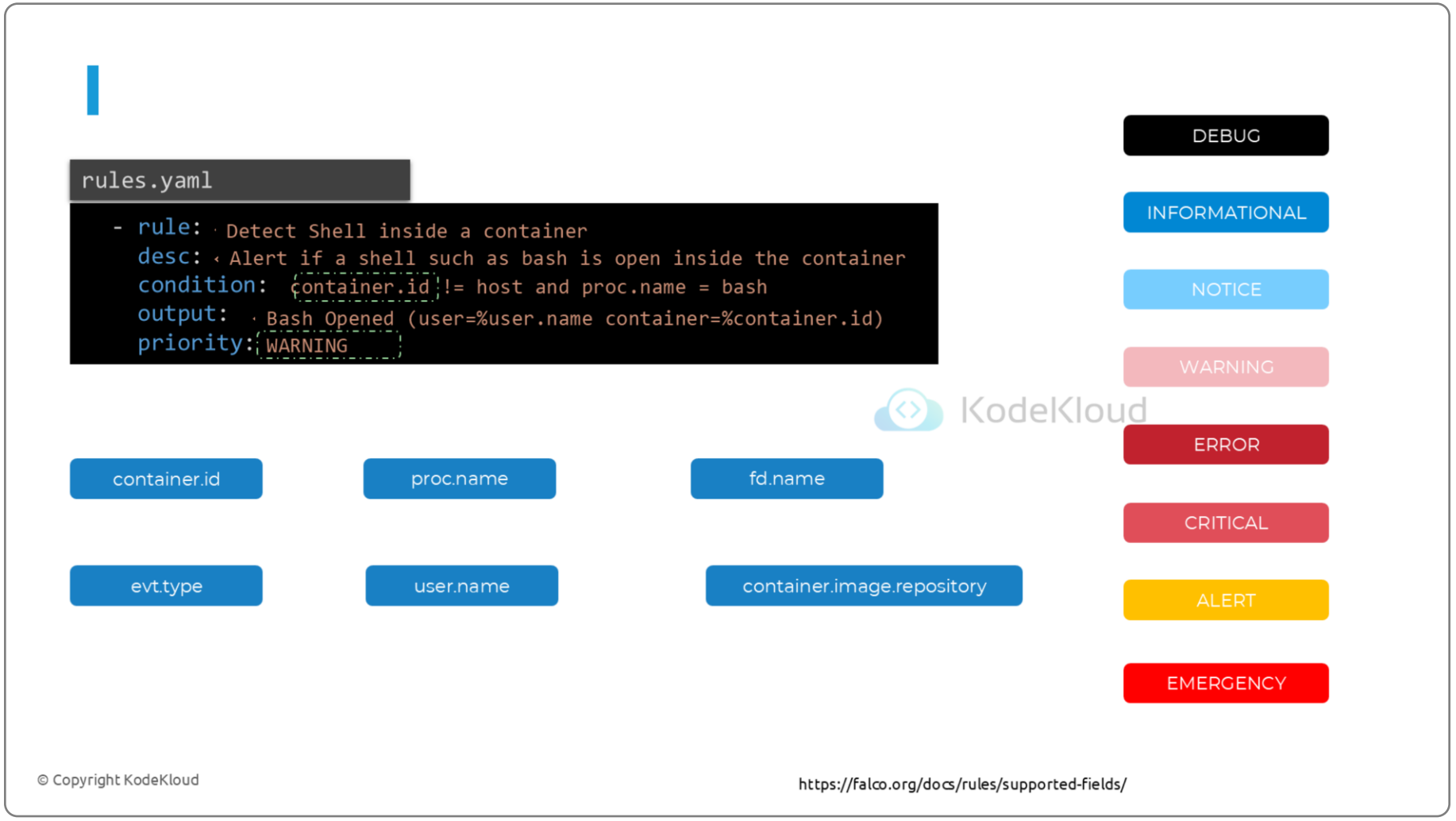
Then we have the <c>evt.type which is used to filter system calls by name as execve, open, accept , connect e.t.c

<c>The user.name is used to filter the user whose actions generated the event.

<c>The container.image.repository filters specific images by name.

Let’s turn our attention to output now. We have again used filters here to log the name of the <c>user that opened shell in to the container and the ID of this container.

We have only used a handful of filters here for our Falco rule. For more information on sysdig filters please refer the reference documentation.



Finally, we can set severity to this by making use of the priority field.

In this example we have used the value as <c>WARNING but depending on the severity we can change them to any of– <c>debug, informational, notice, warning, error, Critical alert, or emergency. Here debug is the lowest severity and Emergency is the highest severity

rules.yaml

```
- rule: Detect Shell inside a container
  desc: Alert if a shell such as bash is open inside the container
  condition: container.id != host and proc.name in (linux_shells)
  output: Bash Opened (user=%user.name container=%container.id)
  priority: WARNING
- list: linux_shells
  items: [bash, zsh, ksh, sh, csh]
```


https://falco.org/docs/rules/supported-fields/

So we now have created a rule that will alert us when some one opens the bash shell in a container.

But what about other shells? Such as – bourne shell, ZSH, korn shell e.t.c?

Instead of creating individual rules for each shell, we can make use of a list – like this<c>
Let's call the list linux_shells with an array of items that consists of different possible shells that a container may have such

as bash, zsh, sh and csh.

Now, we can update the condition to use the list – like this.

Now let's try to simplify this rule even further. In this condition, we have used an expression <c> container.id != host to make sure that the even happens inside the container and not on the host.

Instead of using this expression, we can make use of a shortcut like this.<c>

This shortcut is called a macro. <c>Here we have used a default macro called container which does the same thing as the

expression we used before. By using this micro the condition is now easier to write and understand.

Similarly, there are several macros that can be used when writing custom rules.

For the complete list, <c> checkout the reference documentation in this link.

Well, that’s it for this lecture. In the next lecture, we will get introduced to the configuration files used by Falco and learn how we can apply the custom rule that we just created.

MONOLITHS & MICROSERVICES

Before diving into service mesh and Istio, I'd like us to have a look at the evolutionary changes in software design in the last two decades.

Software Development until the 2000s

Animated:
The 2000s started with a breakthrough proposition that would completely change our view of software development.
At that time, the software development had become very process oriented and slow. For some industries like defense and aviation for example, it took almost 20 years for a project to finish. There was a huge time gap between the software need arising and the software being delivered. So businesses changed before even the software projects got delivered.
Many projects had been laid off before even it was finished, so much money was wasted business owners

and software professionals both were frustrated.

Agile Manifesto

Individuals & Interactions — Processes and Tools

Working Software — Comprehensive Documentation

Customer Collaboration — Contract Negotiation

Responding to Change — Following a plan

So in 2001, 17 independent-minded software practitioners got together and published Agile Manifesto. They were telling the world that the way people created software had major flaws and it needed to change. From the experience they had, they decided to value…
Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation

Responding to change over following a plan.

That is, while there is value in the items on the right, we value the items on the left more.”

Based on Agile practice not only do we collaborate with our customers more but also we evolve our business models and software based on true experimentation.

Agile Manifesto

“We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:”

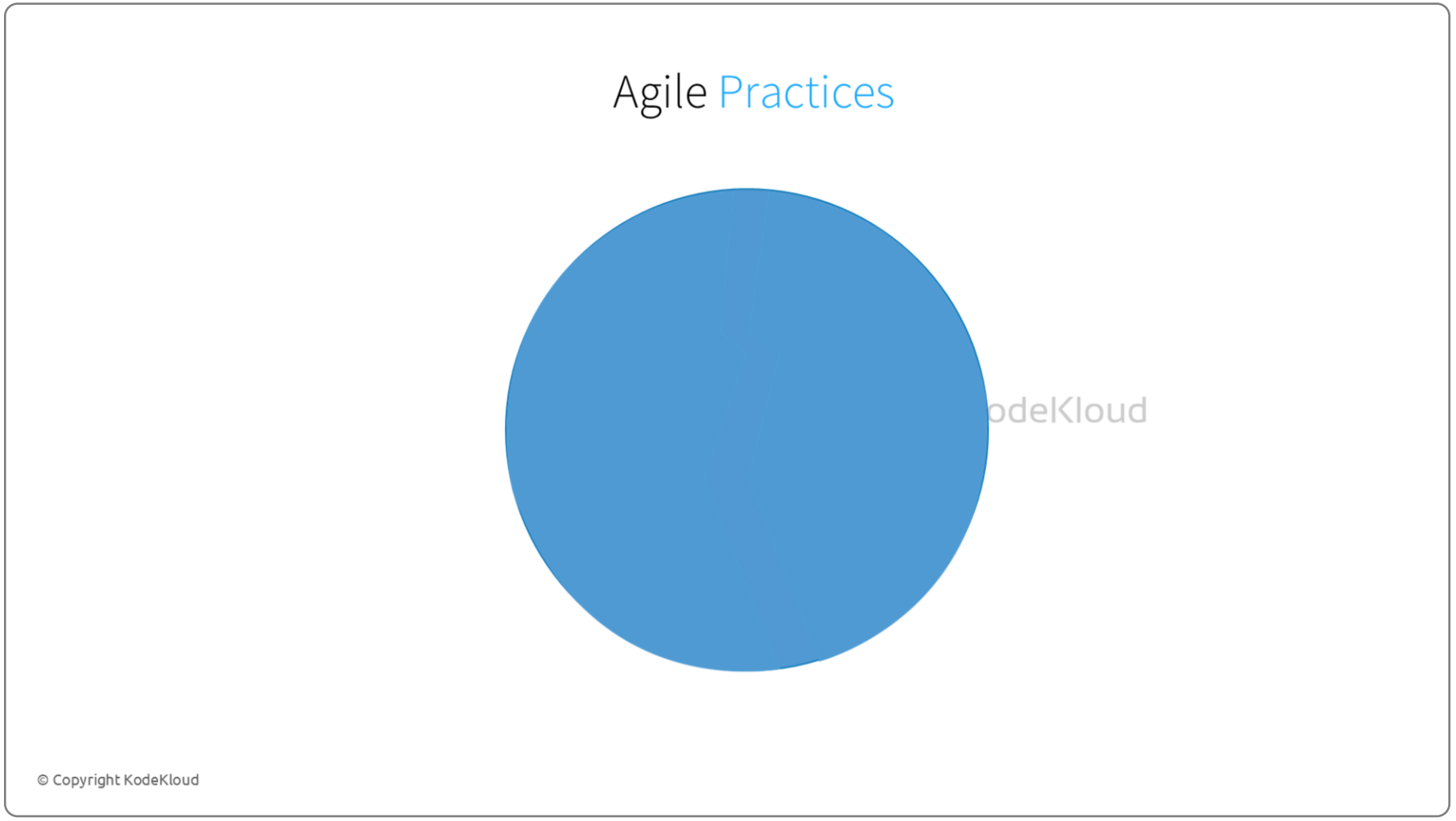
“That is, while there is value in the items on the right, we value the items on the left more.”

So in 2001, 17 independent-minded software practitioners got together and published Agile Manifesto. They were telling the world that the way people created software had major flaws and it needed to change. From the experience they had, they decided to value…
Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation

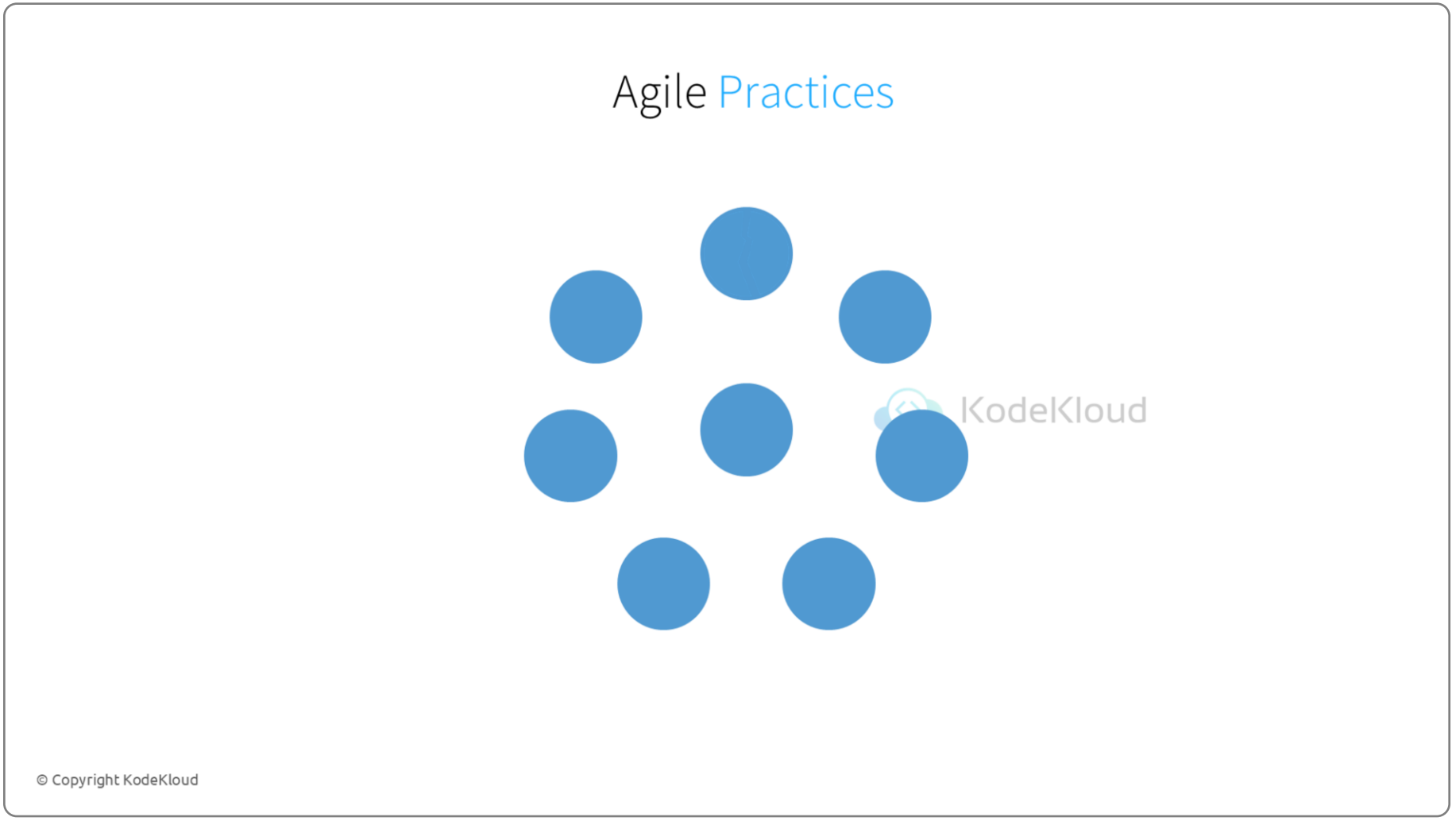
Responding to change over following a plan.

That is, while there is value in the items on the right, we value the items on the left more.”

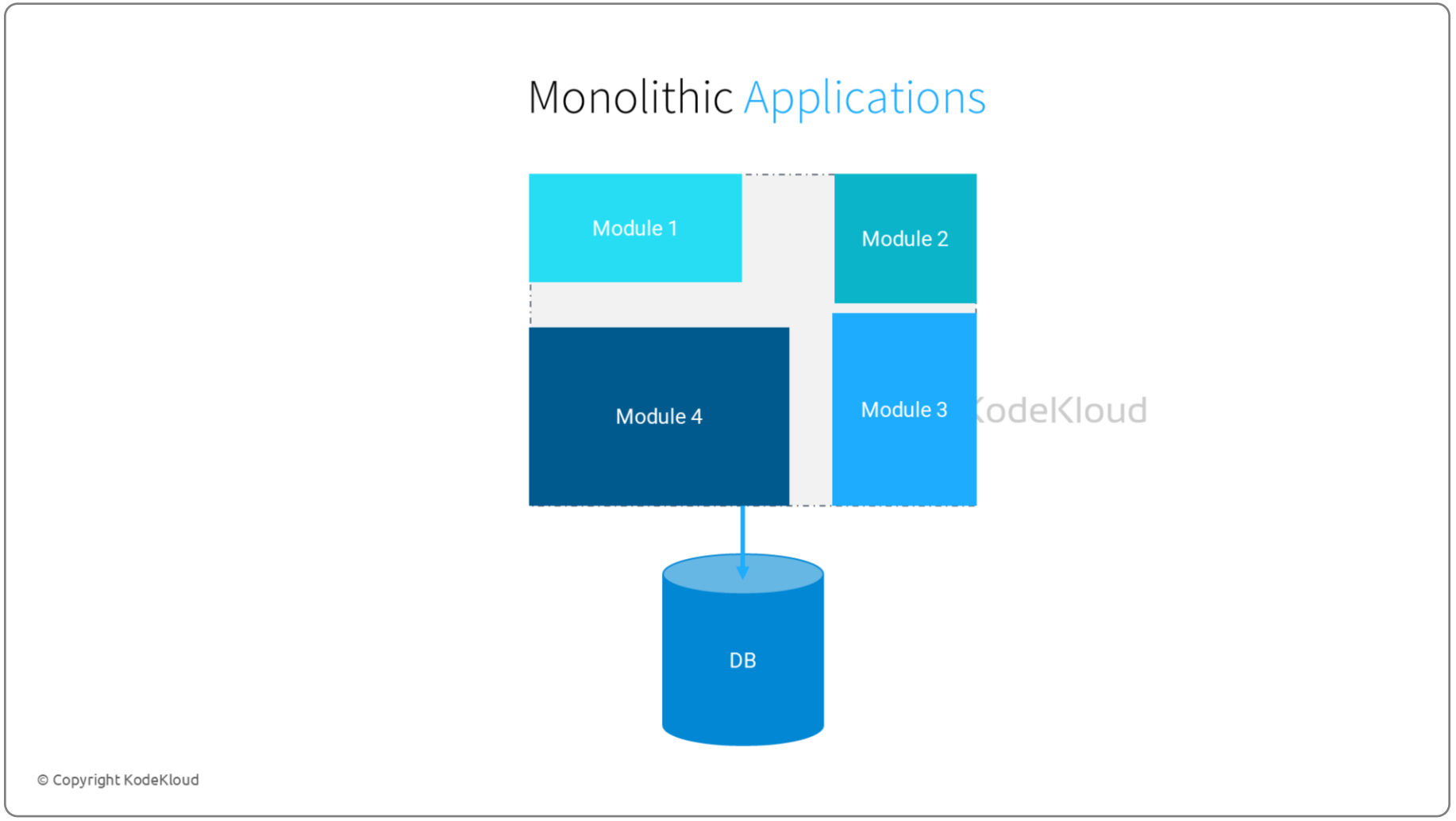
Based on Agile practice not only do we collaborate with our customers more but also we evolve our business models and software based on true experimentation.



When you are working on one large app and if something breaks, it breaks entirely.

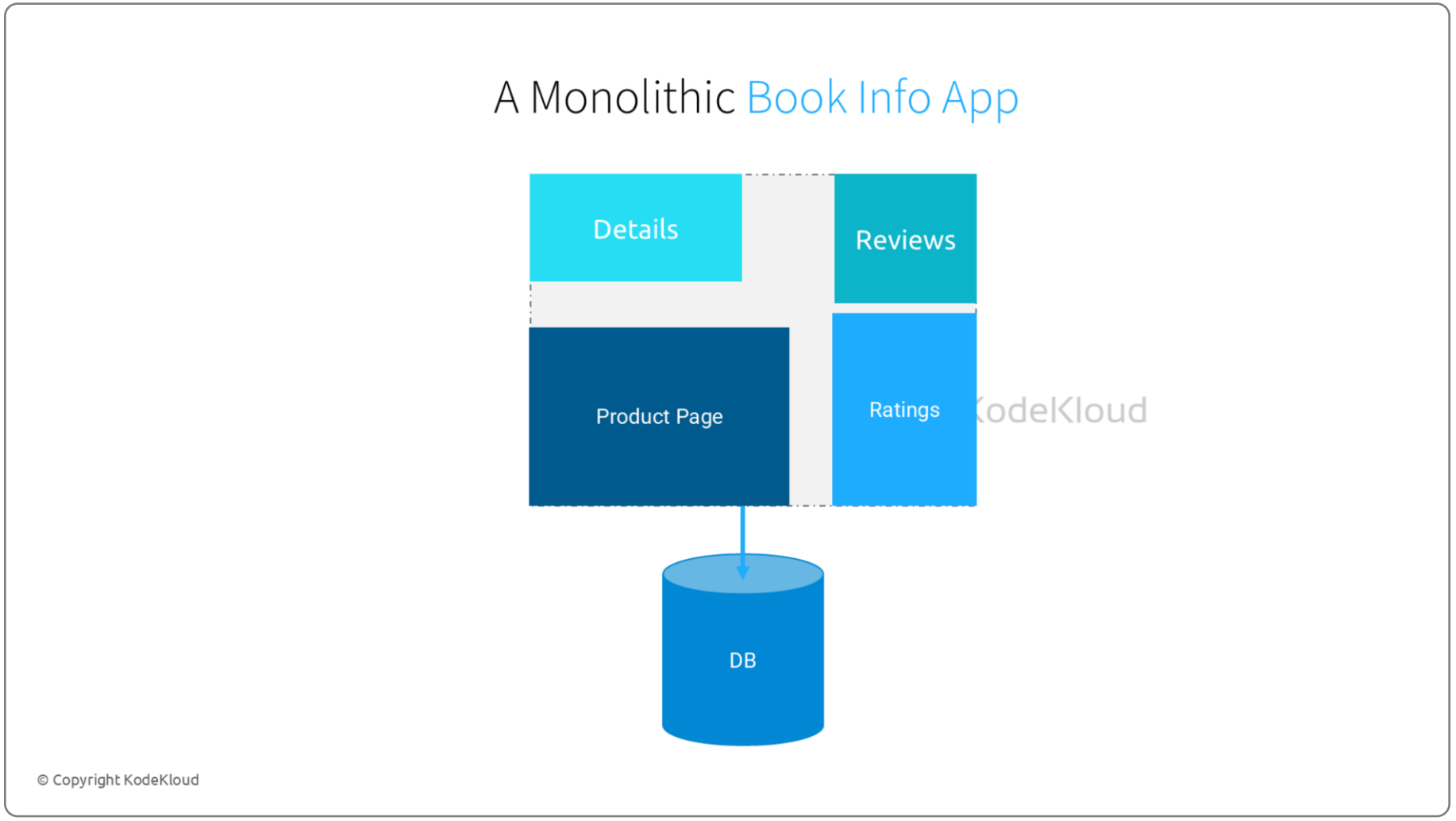


However ,if you're changing small portions of your app, things would be more under control. The piece that you are experimenting with might be affected but the risk will be less than before. So we started designing our applications to be smaller and smaller, so that we can isolate risks during experimentation, at the same time deploy faster and more frequently.



So the giant systems we have been designing, have been becoming one big problem standing in the way of innovation and agility. We had to take a look at our traditional software architectures and redesign them.
1. The term monolithic is used when all the functionality in an app needs to be deployed at the same time and there is a unified approach to all the separate functionalities within the monolithic boundaries.
2. All of this functionality almost always share the same code base and it has no clear boundaries between them. These pieces are tightly coupled. All the code could even be working as a single process. There is usually a single database for

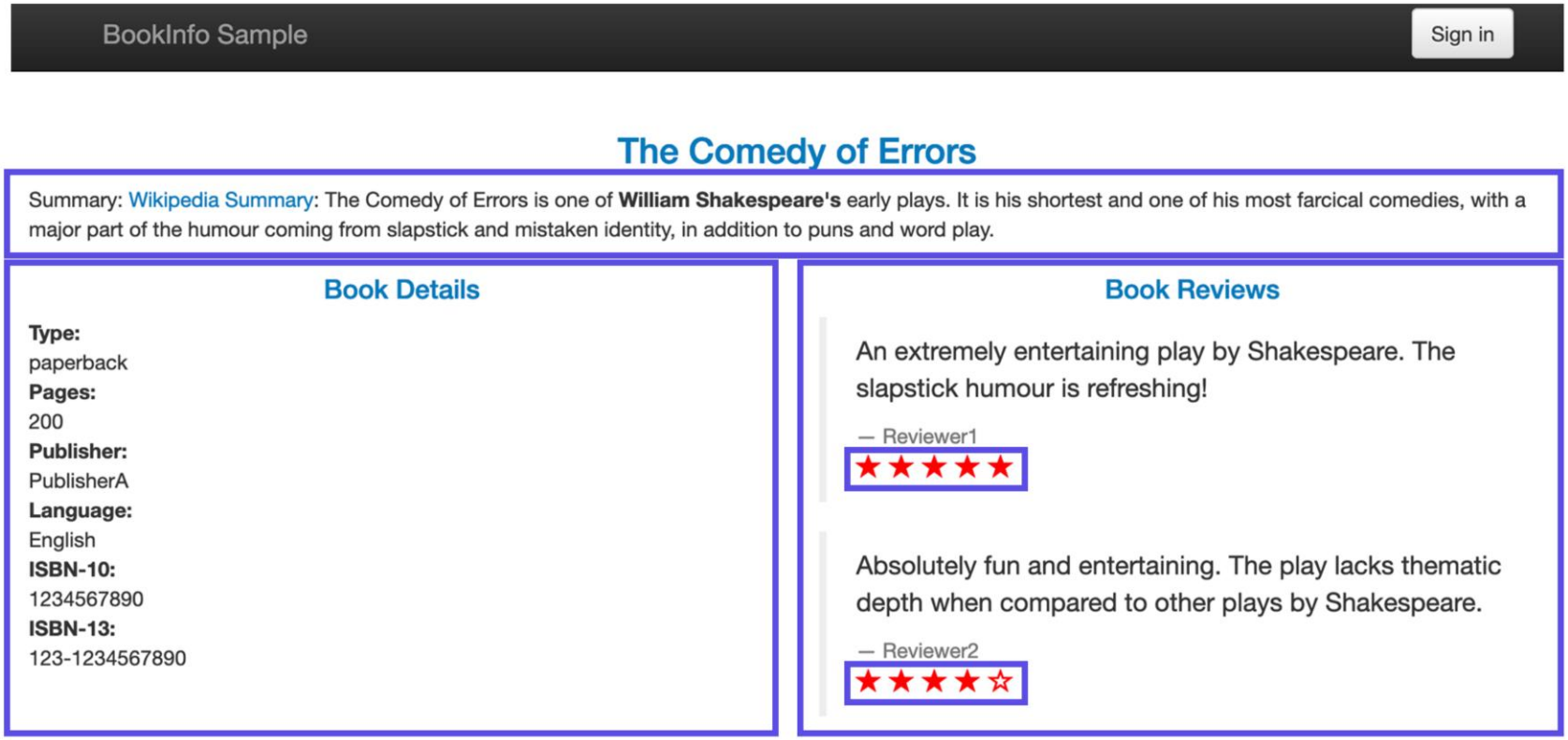
persistency in this model which becomes a huge bottleneck at one point.



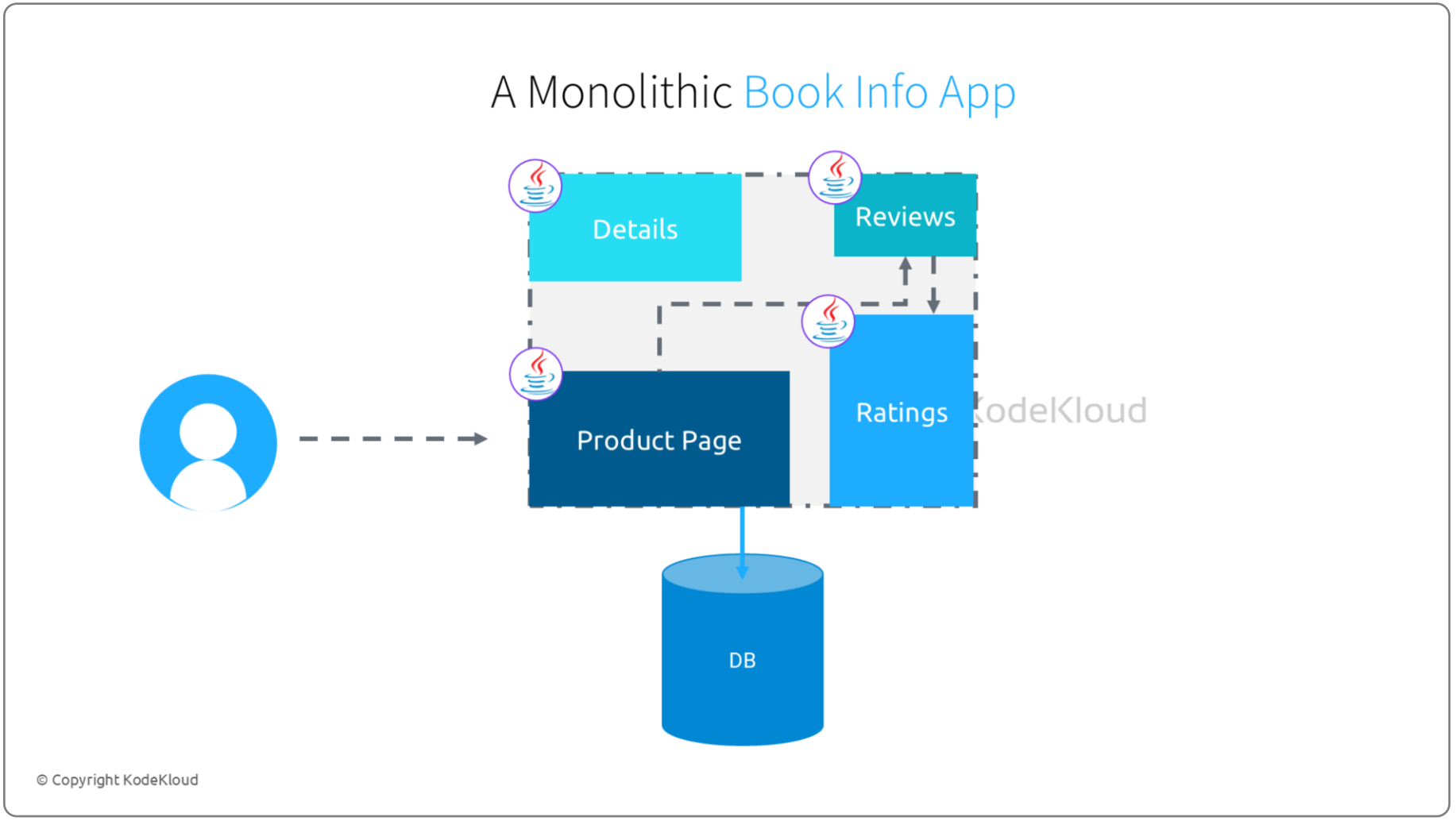
Let's see how a monolith works in a real life application: Here's our Book Info Application. We will use this as our example application throughout this cours It consists of 4 different modules. Details, Reviews, Ratings, Product Page.

It is a modular application but still a monolith, all the services depend on a specific version of the other one. You need to deploy the whole package and possibly send some scripts to the database.

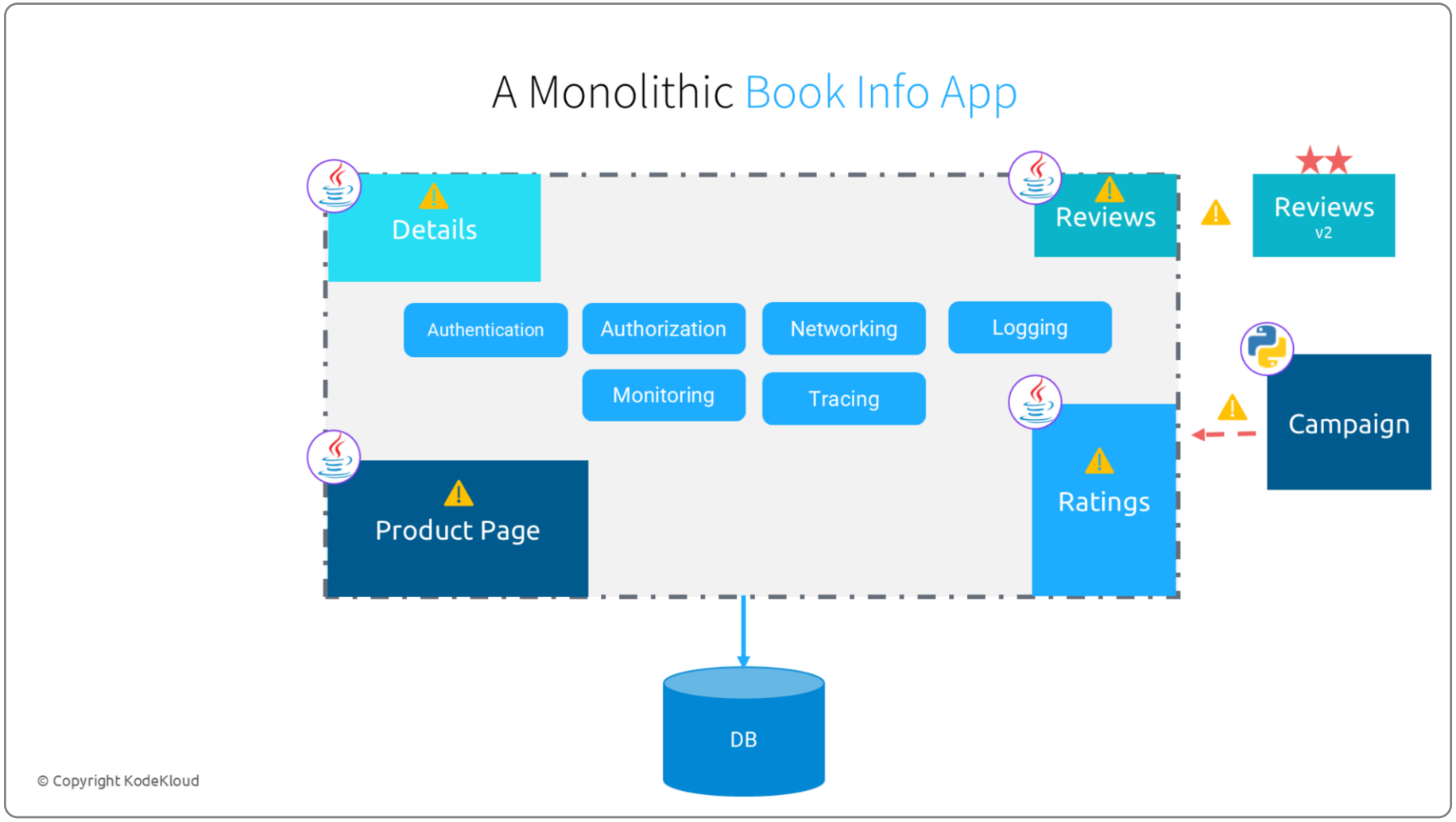
A Monolithic Book Info App



Product Page shows the book's information, reviews and ratings.
All the data of this application comes from different modules. But they are not separately designed and can not be scaled



Let’s try to understand the dependencies of the modules and the problems of The Monolithic BookInfo.
The customer lands on the product page.
Product page gets these information from Reviews and Details modules.
3. Reviews service gathers the number of ratings from Ratings service.They are all written in the same language, Java.



Apart from these modules, the application also takes care of networking, authentication, authorization rules and how data is transferred between modules, logging, monitoring, tracing etc.

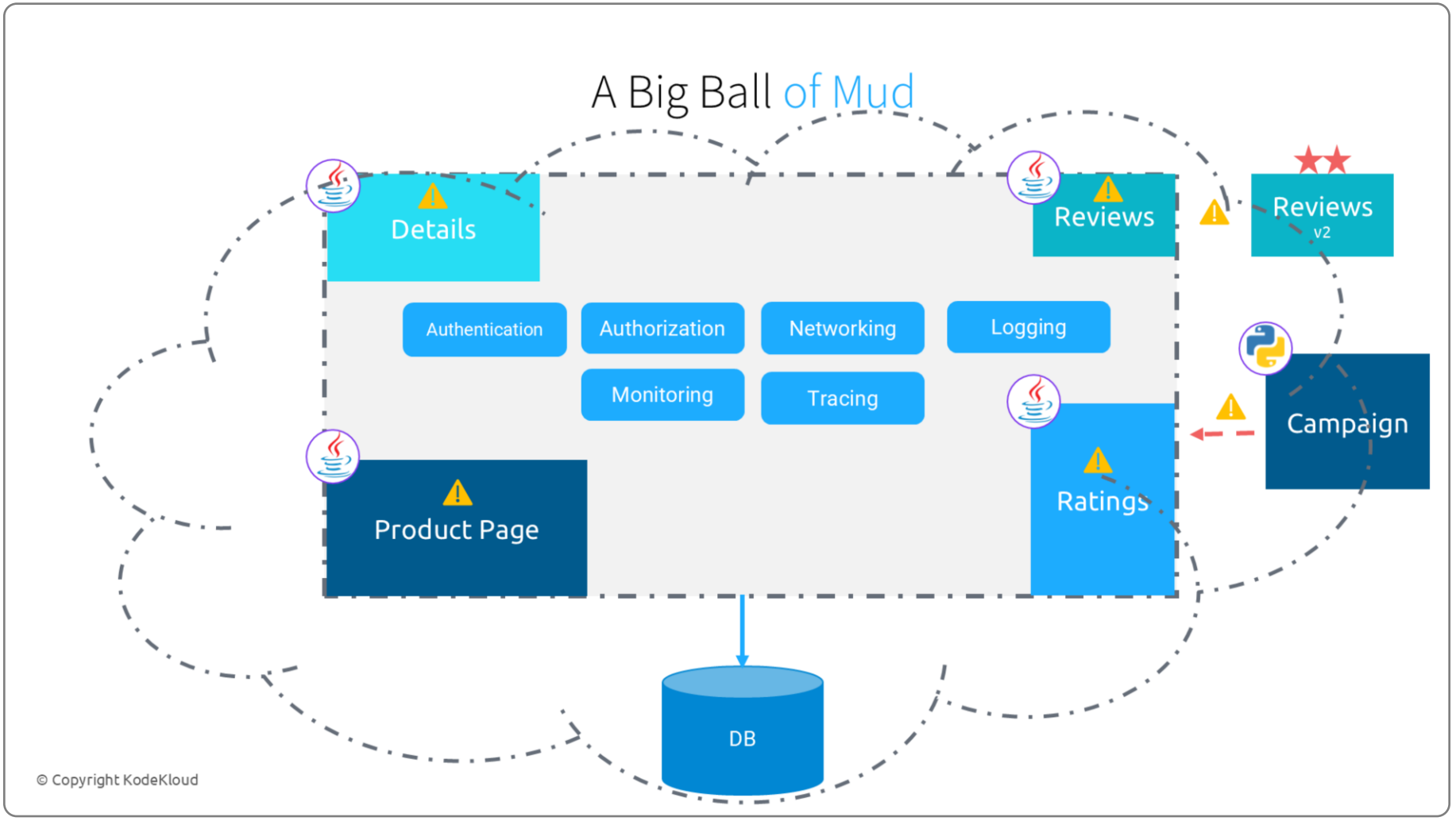
5. Every once in a while the ratings module, because of the amount of the data it holds, have problems and
6. this affects the whole system. It's not possible to just scale ratings or leaving it out of the system without touching the code and re-deploying it.

2. Now a new campaign module is going to be developed,

3. But this time a new team is formed for this and they want to use a different language. Since everything is unified, and all the important functionality like authorization lies within the monolith, they have a hard time designing the architecture.

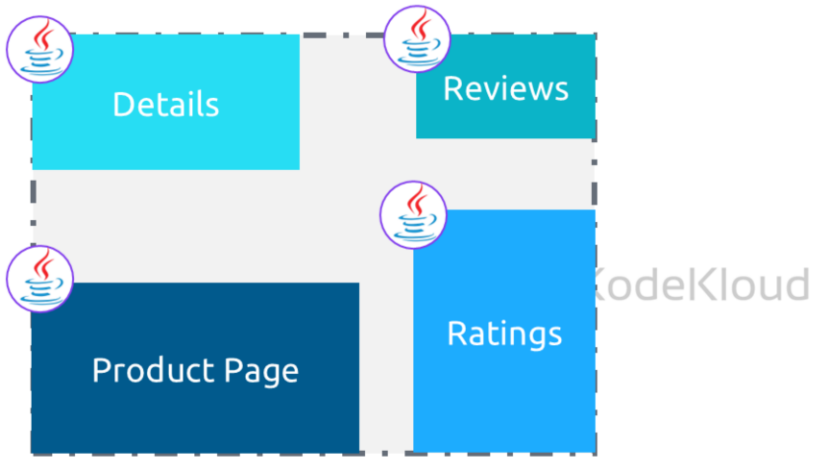
4. And also our product owners want to try a new version of reviews, with red stars for Xmas. They want to test this functionality on a segment of users and if they like it, we can use it for the whole system. This does not look possible without deploying the entire application in 2 different versions.

This is a very simple app with very little functionality and you can see what problems we started tackling already. Now think of huge applications that has been around for decades and more. There may be hundreds or more developers worked on these systems and starting with some loose rules on architecture, these systems might become a big ball of mud.

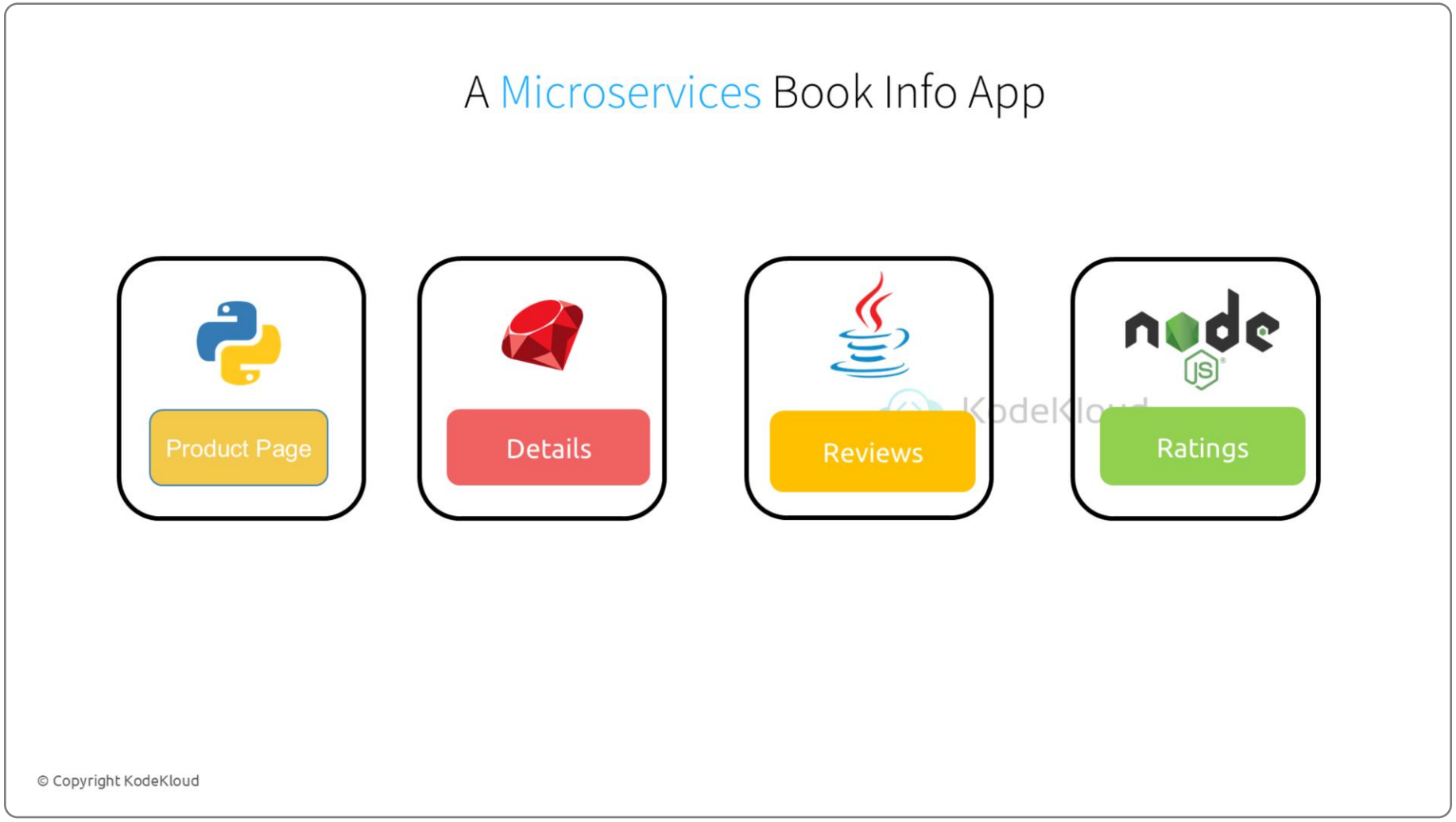


Big ball of mud is a famous idiom for these types of software systems. Without noticing, your monolith application can go out of hand and become one of these.

A Monolithic Book Info App



Now let's have a look at how this Book Info monolithic application could …

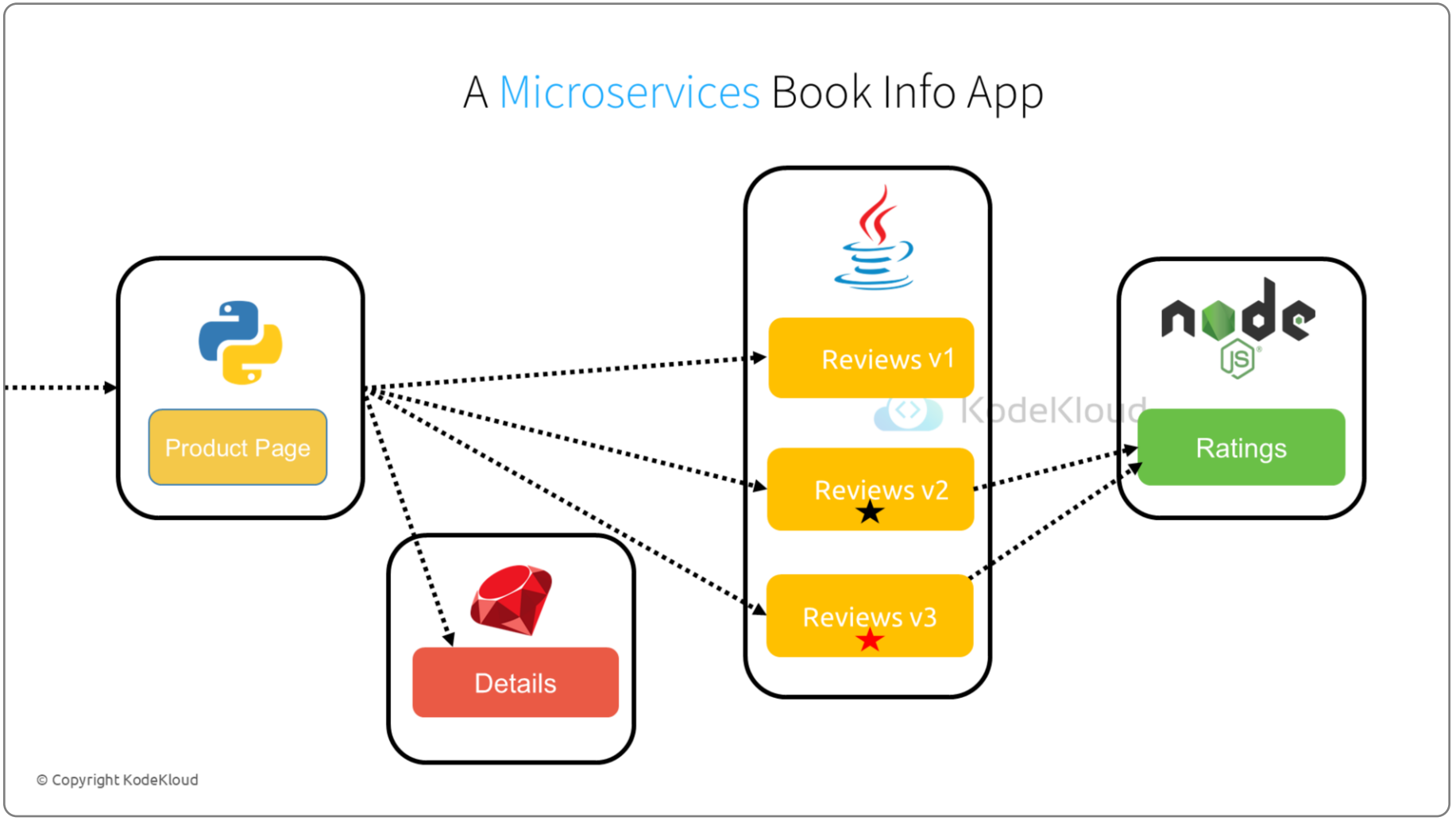


magically and miraculously turn into microservices.

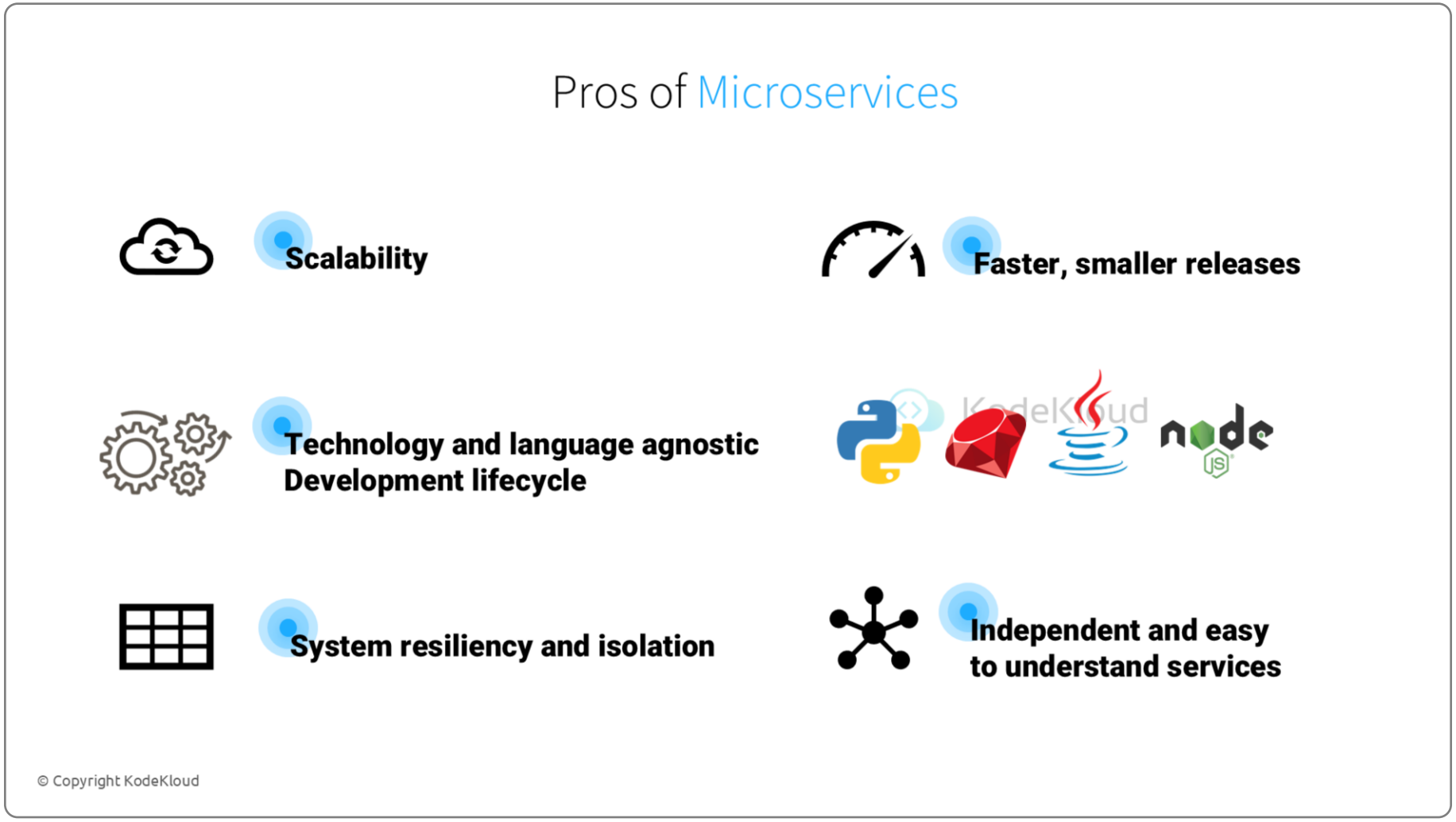
Well, to be honest, it is not an easy task to refactor monoliths and it's not an overnight transformation. It's a cultural, technical and organizational effort which paves way to being (agile?)cloud-native.

With the new microservices architecture, each module is now its own independent and separate application.

Our product page has been now transformed into a python app. It still functions as our landing page.
Book details have been refactored into a Ruby application.
Reviews app has been transformed into a java app.
The ratings module has now been re-designed and implemented in nodejs.

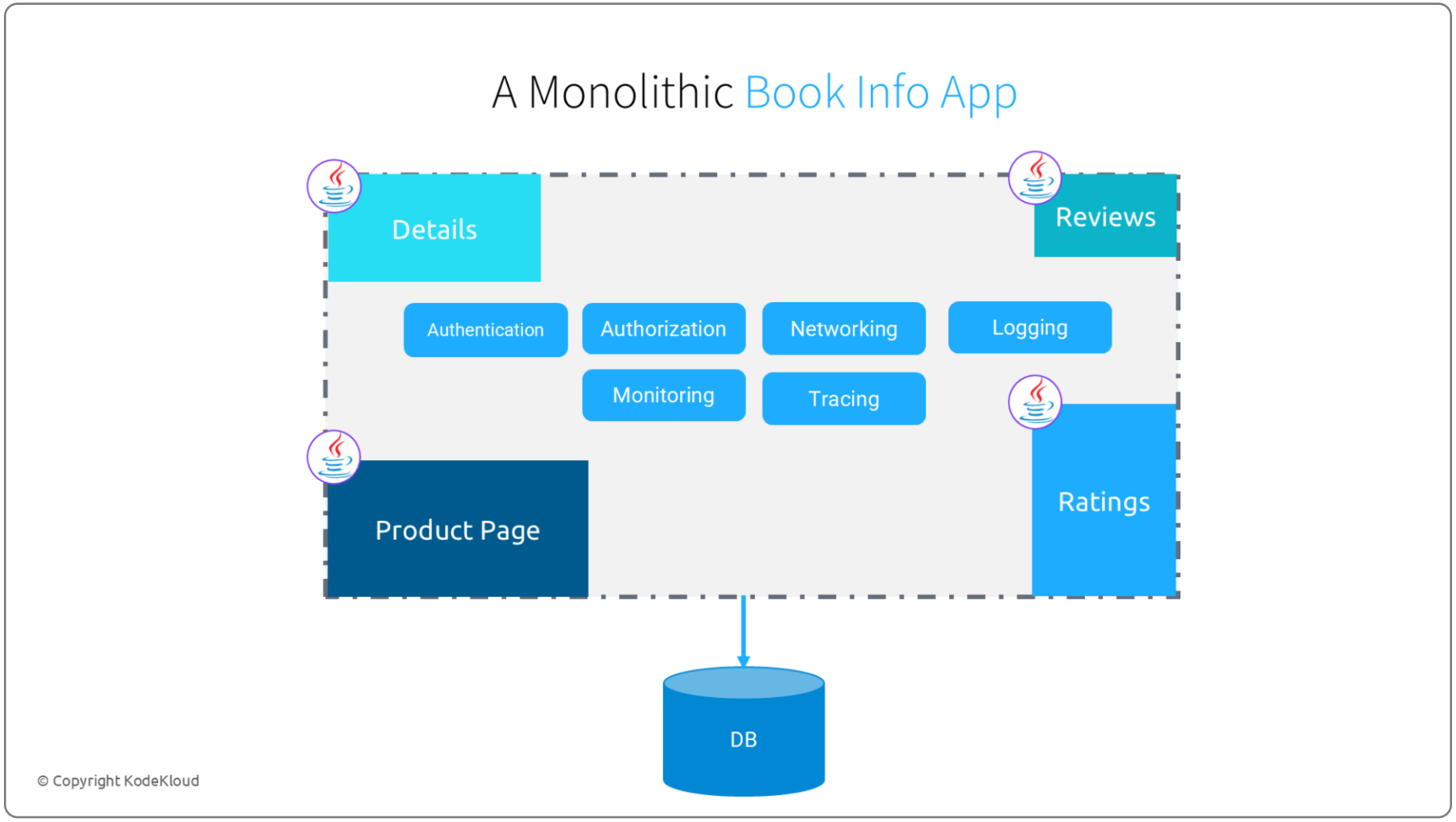


Moreover the reviews app now has multiple versions – v1, v2 and v3 to test different ideas. v2 is a no star version and v3 is a red star version.
As before users land on the product page which contacts the details and reviews services to show information regarding various products.



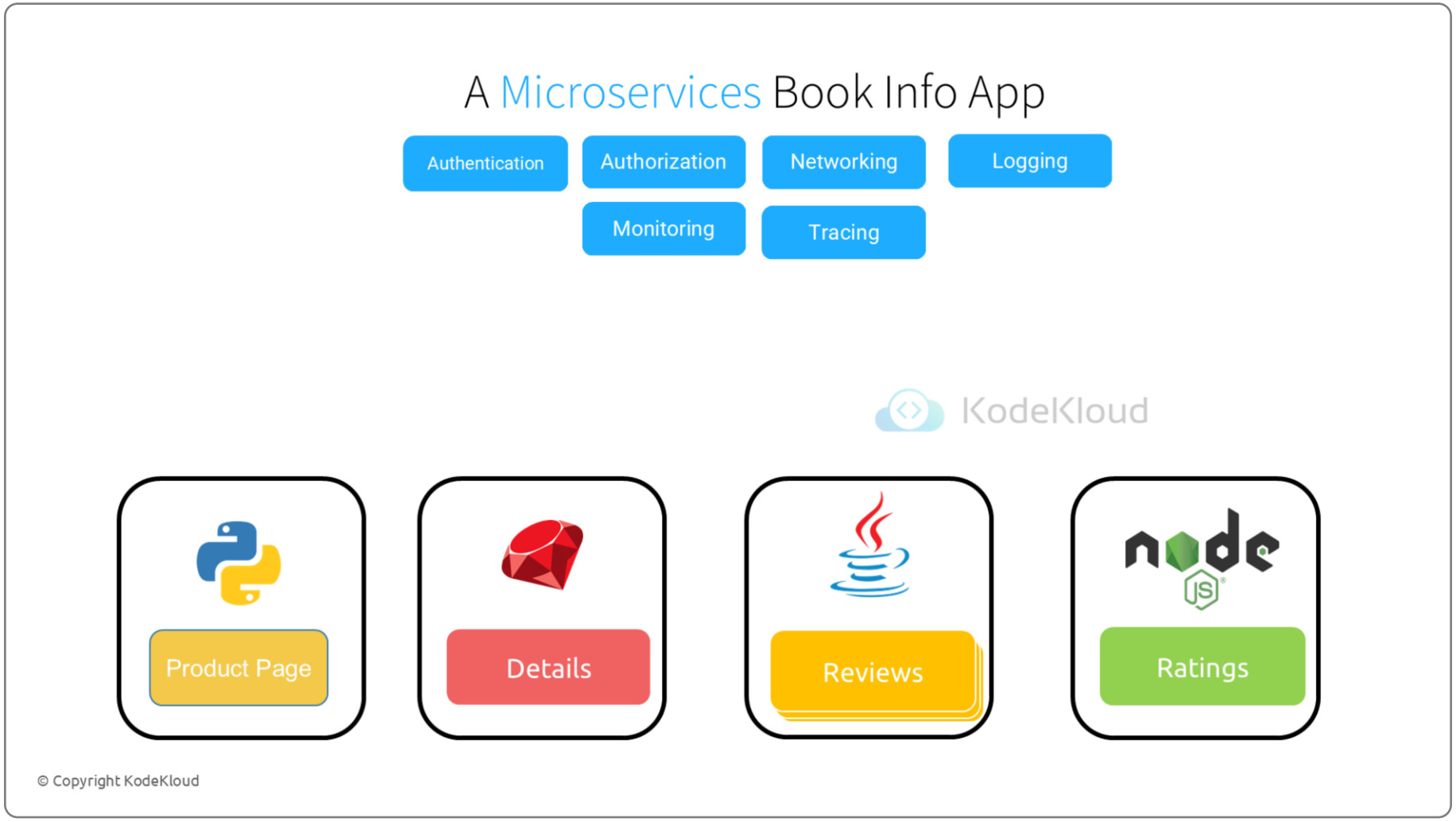
0. So now that we have seen how Book info transformed into microservices, let's talk about the improvements and wins:
Ratings module will not be a problem anymore. Now that it is fully independent, we can scale it out or down depending on the load from our customers
We can now deploy every piece of book info without interfering with others. It will make our releases smaller, faster and less risky
With microservices architecture, each service can be written with different languages. So here in our application, we have

now 4 different languages and each team has autonomy.
Our services now are isolated from failure of other services because of loosely coupling. (And the end-to-end application will have more resiliency since it’s different parts can be monitored, changed or rolled back easily.??)
Instead of having 1 big application, we now have 6 smaller applications. So hopefully you won’t have to deal with big ball of mud anymore. In an ideal scenario, a microservice should have a single responsibility.



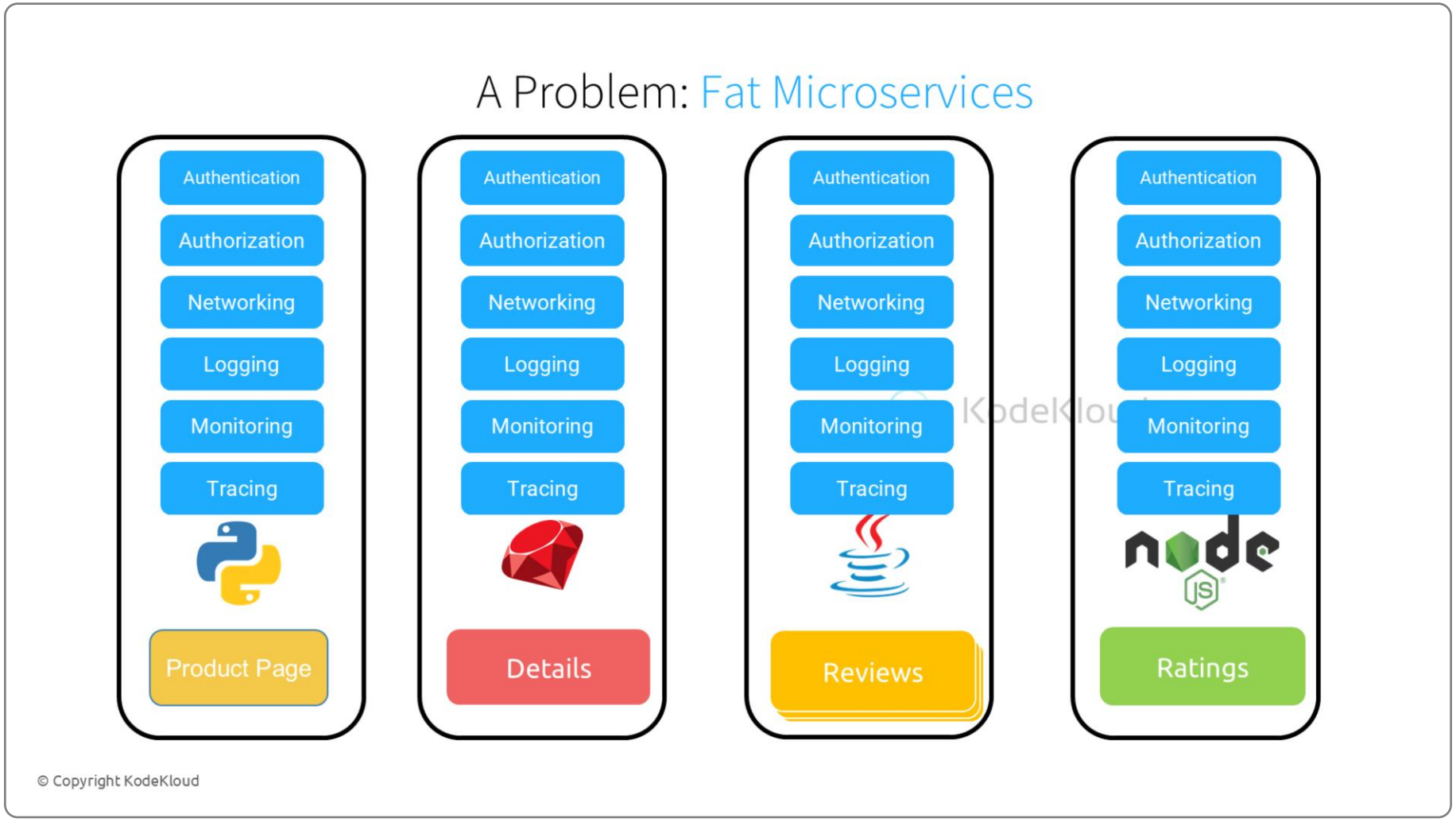
Earlier when we discussed about the monolithic book application we said that apart from the 4 different modules, the application also takes care of networking, authentication, authorization rules and how data is transferred between modules, as well as logging, monitoring, tracing etc.

But what happened to these when we moved ….



.. to the microservices model? Well they are not there. Our microservices currently do not implement any of these.

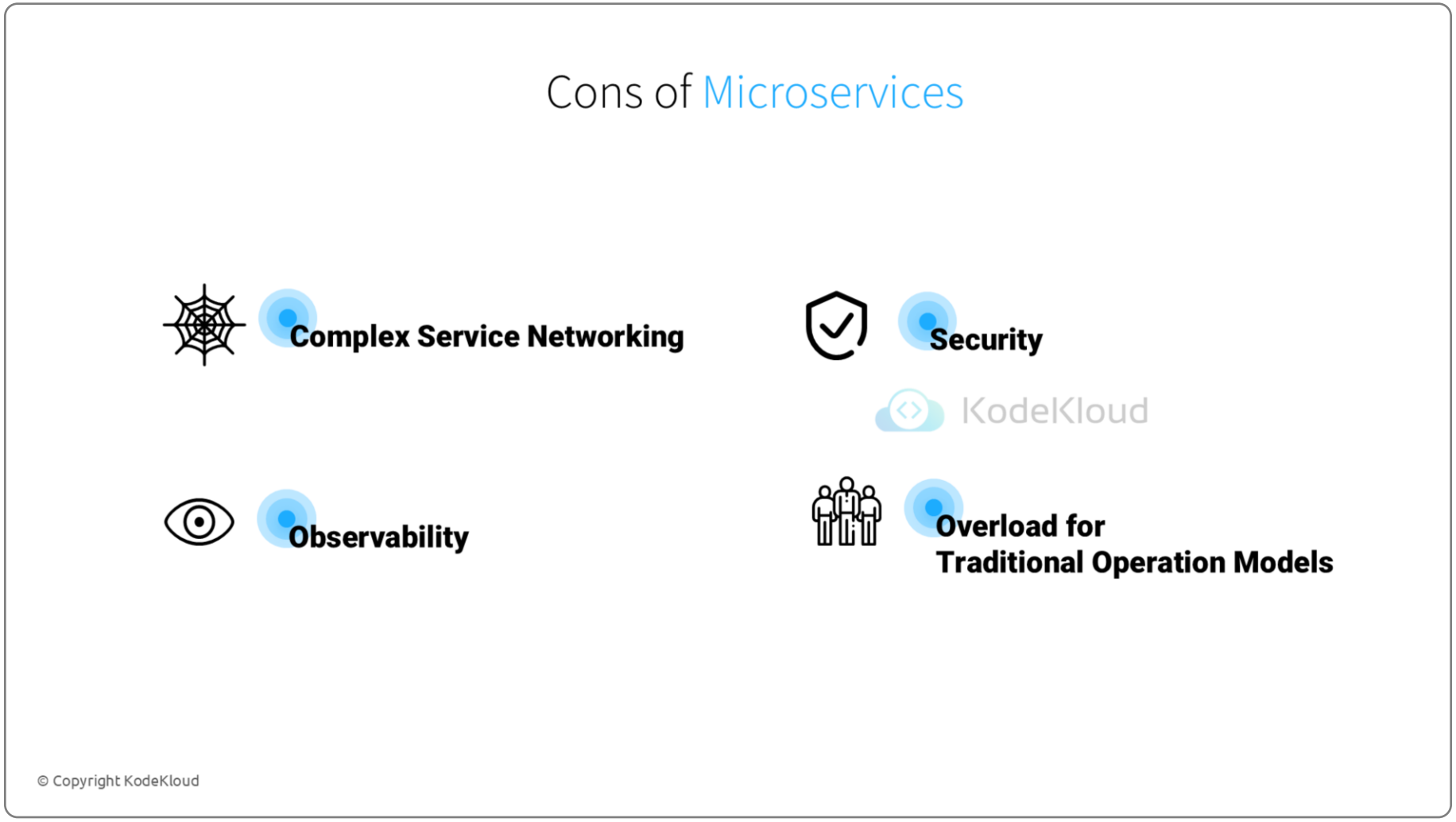
We could move these into each of our services. So let's add all these functionalities in our microservices…



You've probably seen where this is going,right?. Every microservice boundary has the same functionality coded again and again into them. Every team has to solve the same issues over and over again, and probably they'll solve them differently. Look at the code duplication here, and how do you go tell these teams to change a certificate or a monitoring agent version let's say?

These issues we have to deal with in every microservice are called Cross Cutting Concerns. When coded into the

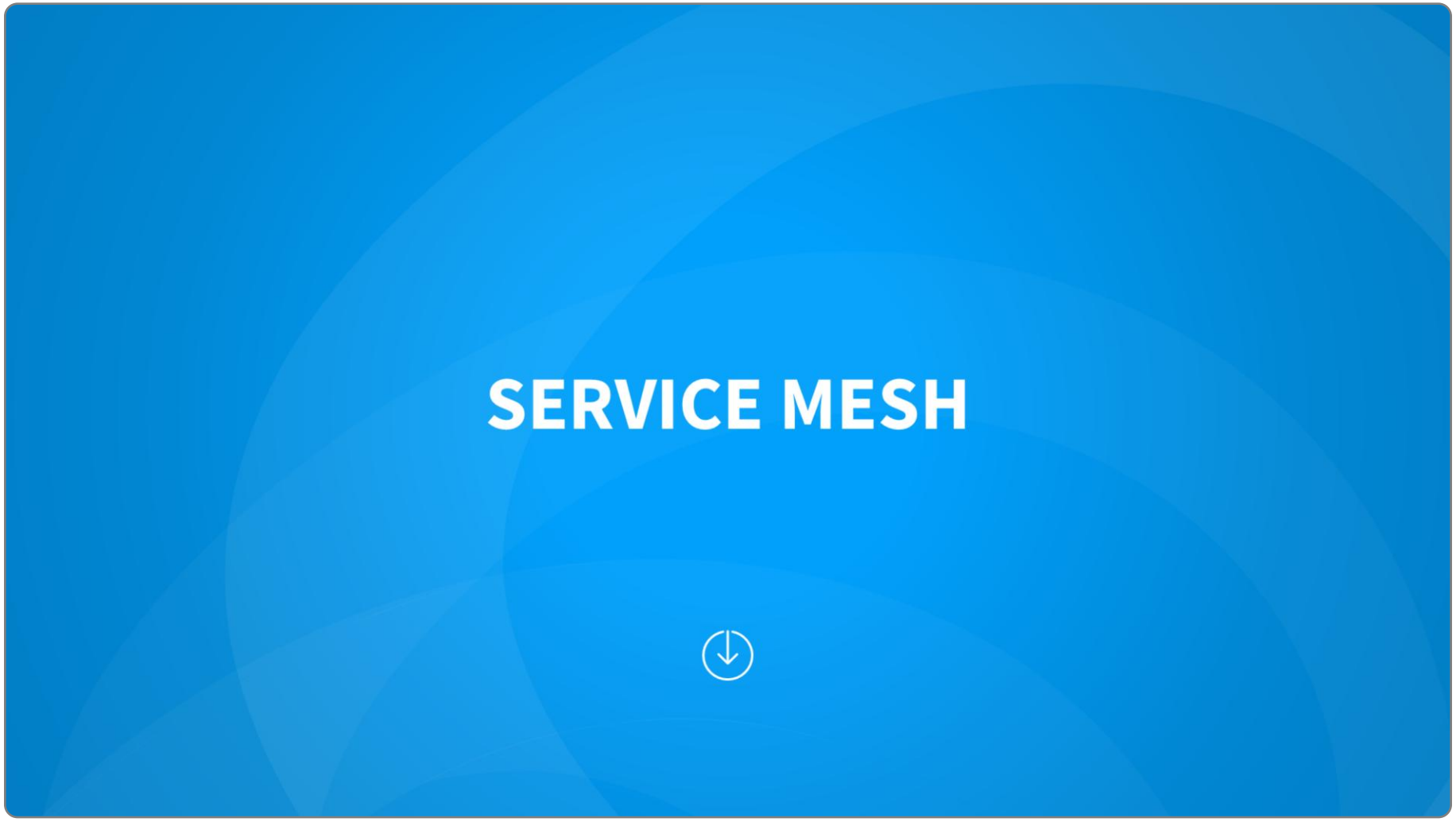
microservice, they distrupt the main reason we designed the microservices : To be able to have smaller, more independent pieces. This is known as the problem of fat microservices.



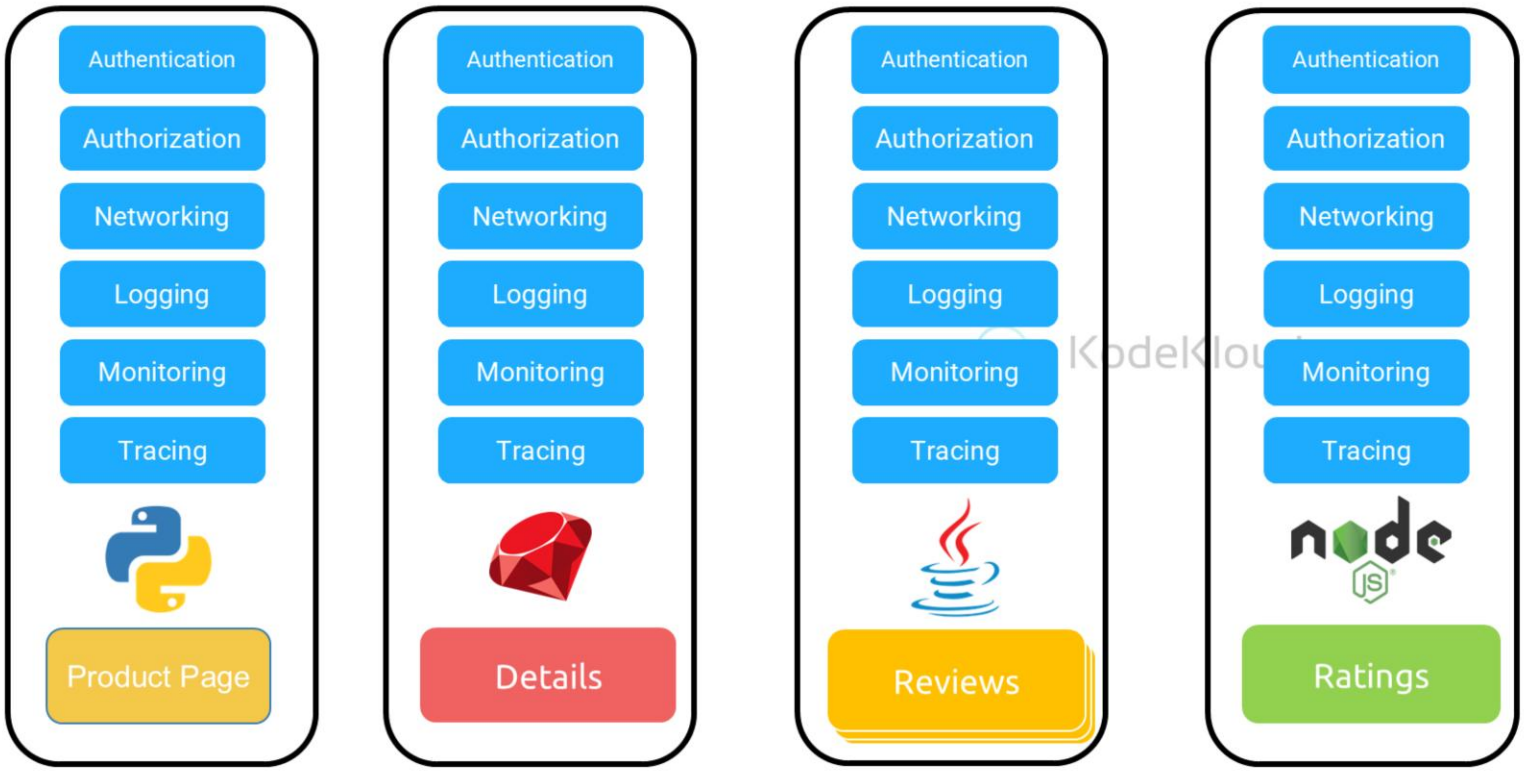
0. Microservices are not a piece of cake, they have their own challenges and they tend to get real complicated.
As we just saw in the monolithic version of our application, all aspect such as networking and security was directly coded into the application. But now all those gray areas fabricated into our monolithic application are exposed and we have to find a way to cover there. How will product page know which version of Reviews to go o? So how will one service know how to find the other one? What are the traffic rules, what are the timeouts? In a very short time, you'll have too many tiny bits of services spread out and these questions will get much harder to answer.

It was much easier to handle the security in our monolith, because it provided the proper scheme for us. In our microservices, securing service-to-service communication and end-user to service communication can become a problem itself.
Now that you have loosely coupled tiny pieces and many abstraction layers, it gets harder to pinpoint a problem on your application. For that you will need an observability strategy.
Just for a small application, we used 4 different languages. With all those different technologies of microservices, it gets very demanding to do traditional operations. Actually, operations might become a bottleneck for organizations who take on microservices. There is a new approach for that called DevOps, where the development teams work closely with operations and together they take the responsibility of their microservices for deploying, monitoring and fixing.

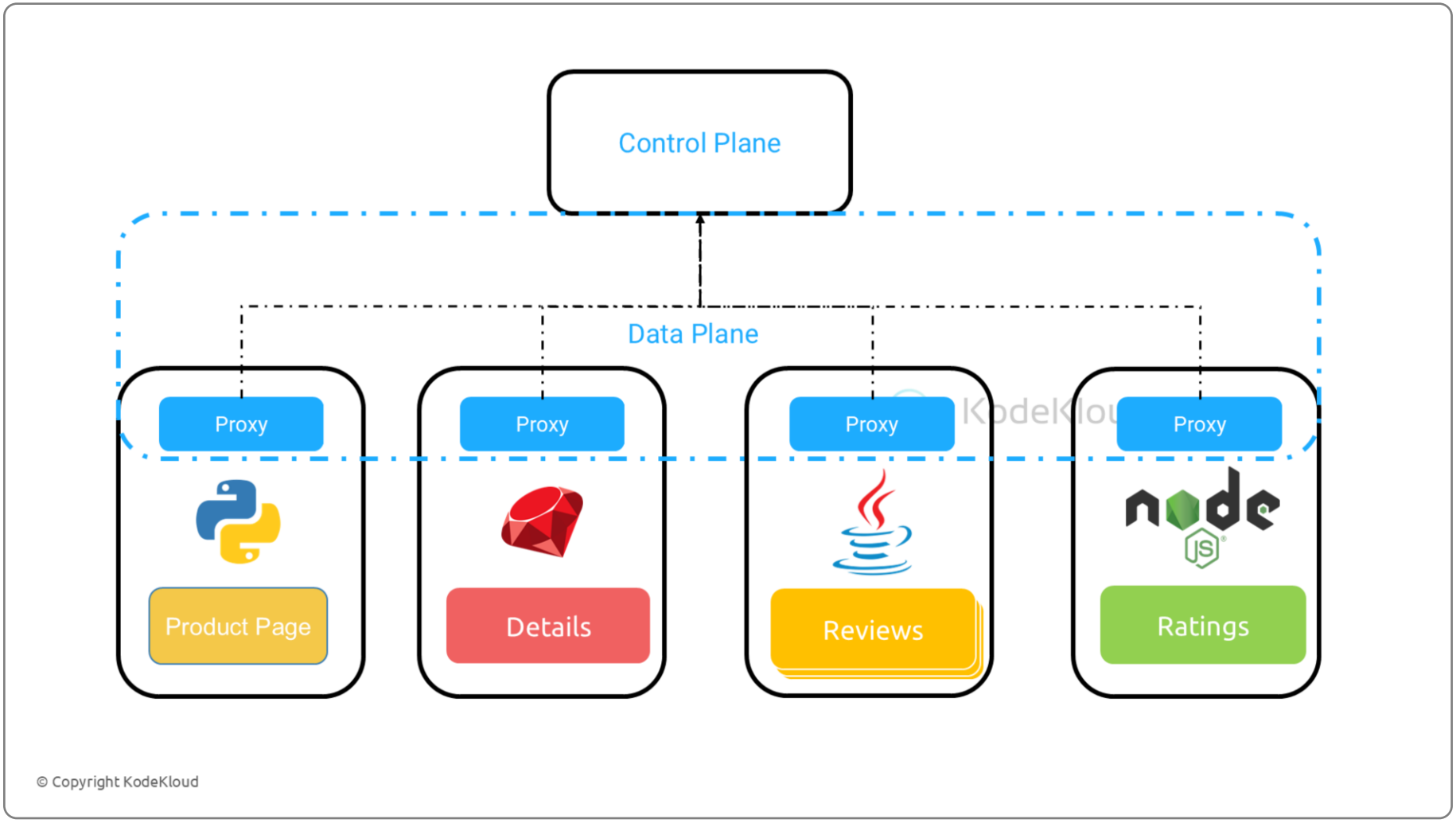
In the upcoming lesson we will look at how service meshes can help



So what is a service mesh?

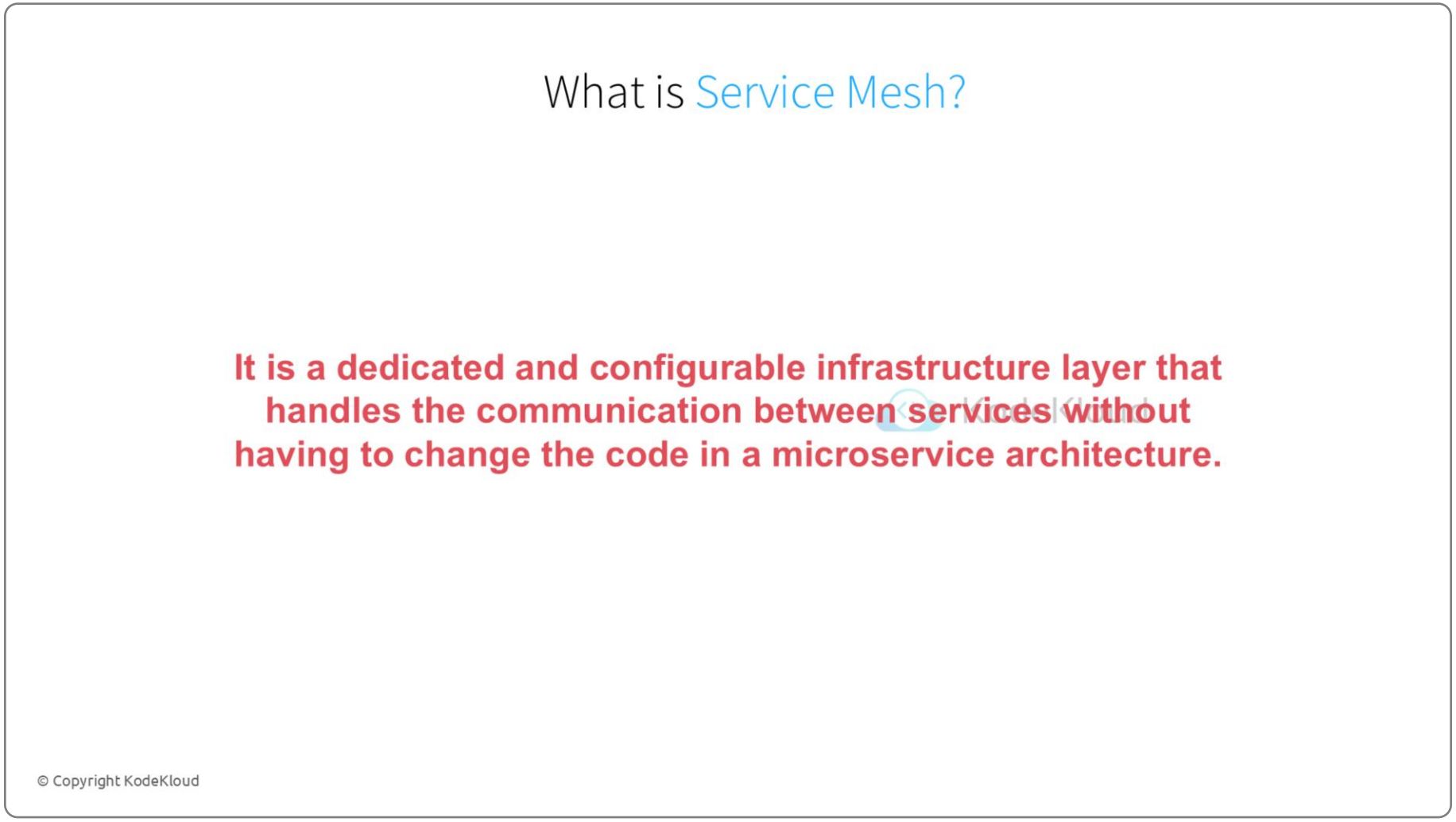


So this is where we were. Instead of embedding all the different requirements into each microservice..

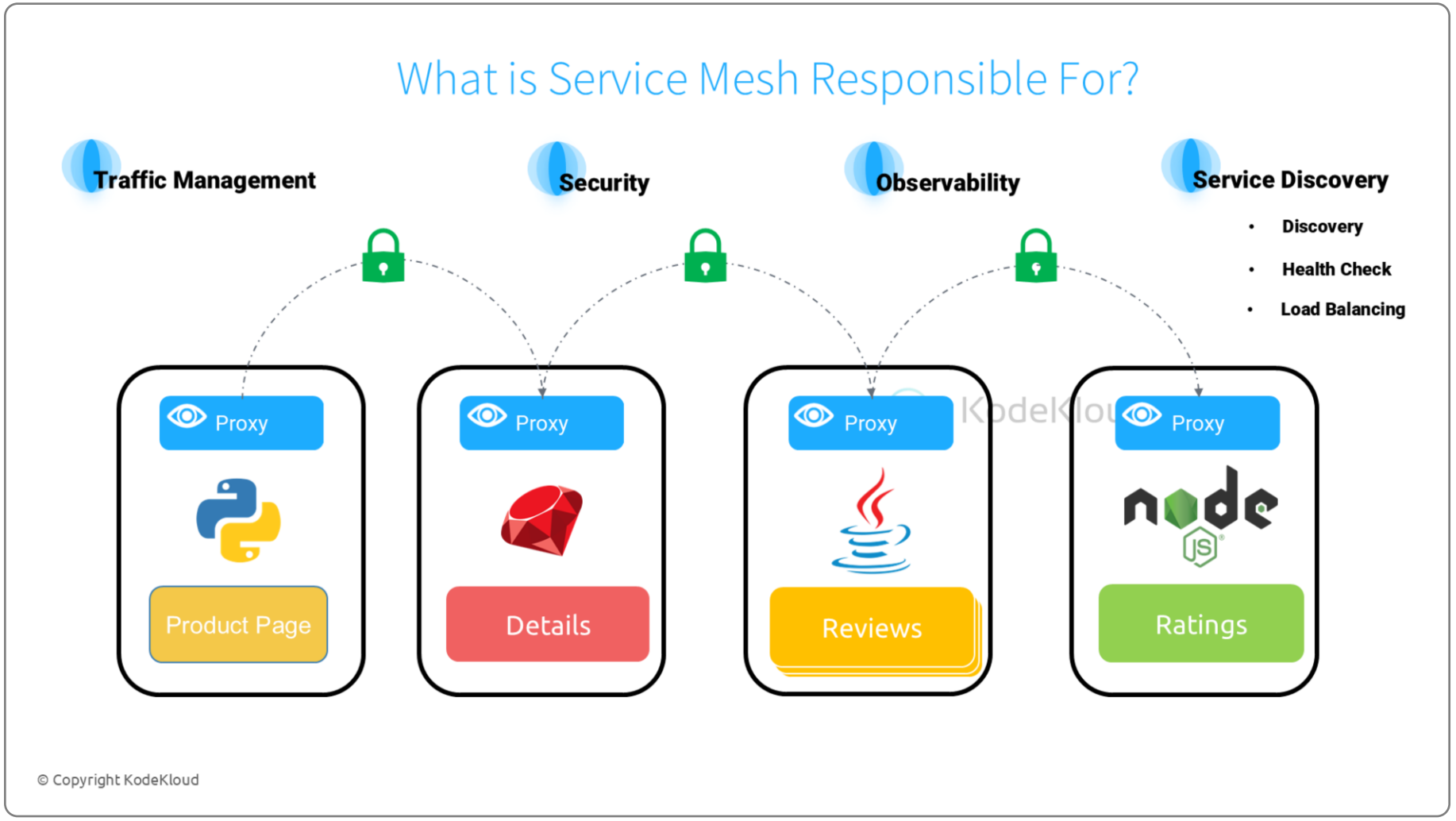


.. we replace them with a single proxy in the form of a side car container.

The proxy communicates with each other through what is known as a data plane.
And they communicate to a server side component called Control Plane. Control plane manages all the traffic into and out of your services via Proxies, so all the networking logic is abstracted from your business code. And this approach is known as a Service Mesh.



It is a dedicated and configurable infrastructure layer that handles the communication between services without having to change the code in a microservice architecture.

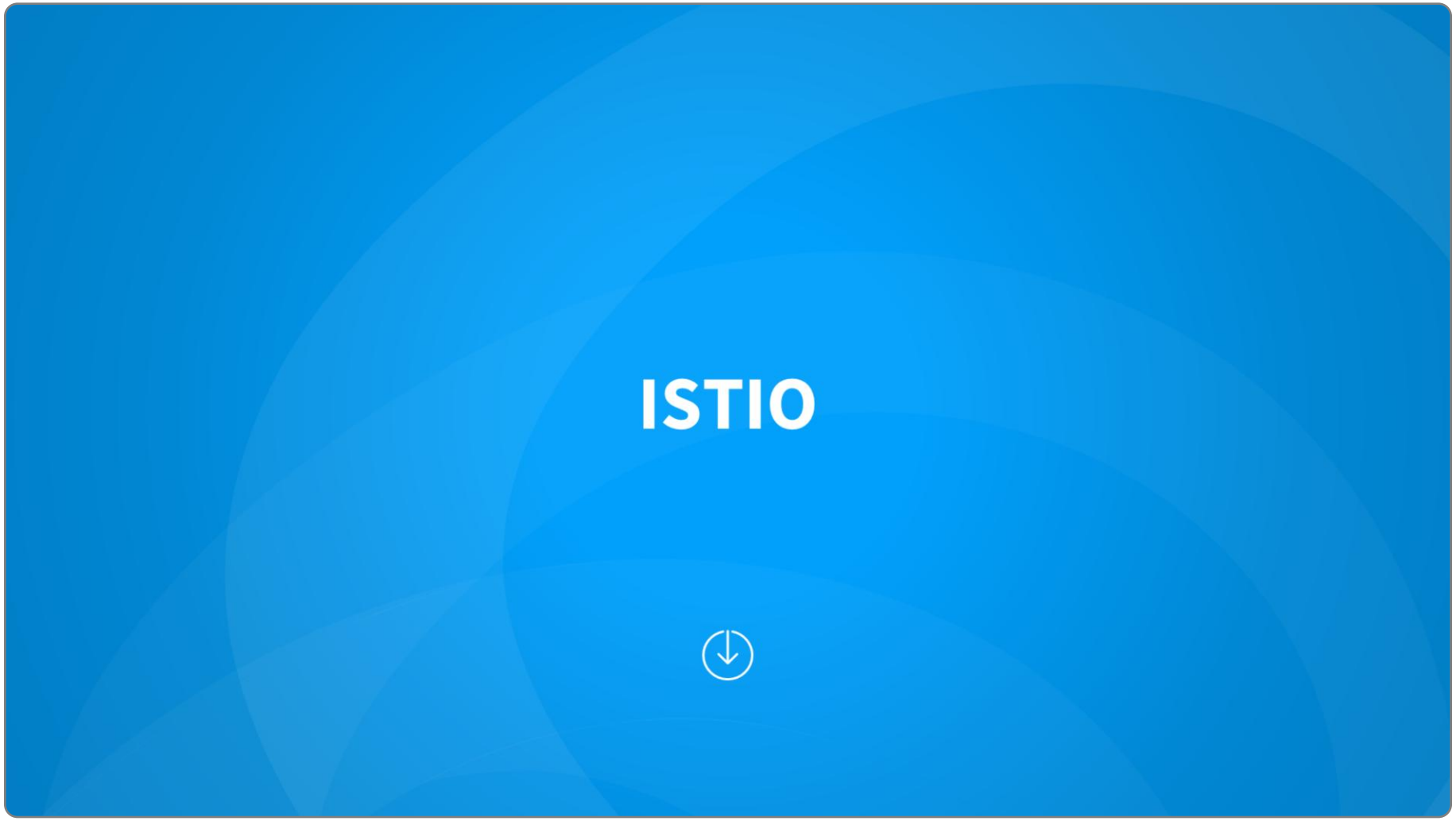


With a service mesh
You can dynamically configure how services talk to each other.
When services talk to one another, you'll have mutual TLS so your workloads can be secure.
You can see things better - How the application is doing end to end, where it is having issues and bottlenecks
And Service Discovery which covers 3 main topics:

In a dynamic cluster, We will need to know at which IP and port services are exposed so that they can find each

other.
Health check helps you dynamically keep services that are up in the mesh while services that are down are left out.
Load balancing routes the traffic to healthy instances and cuts it off from the ones who have been failing.

We will look into each of these in more detail throughout the rest of this course.

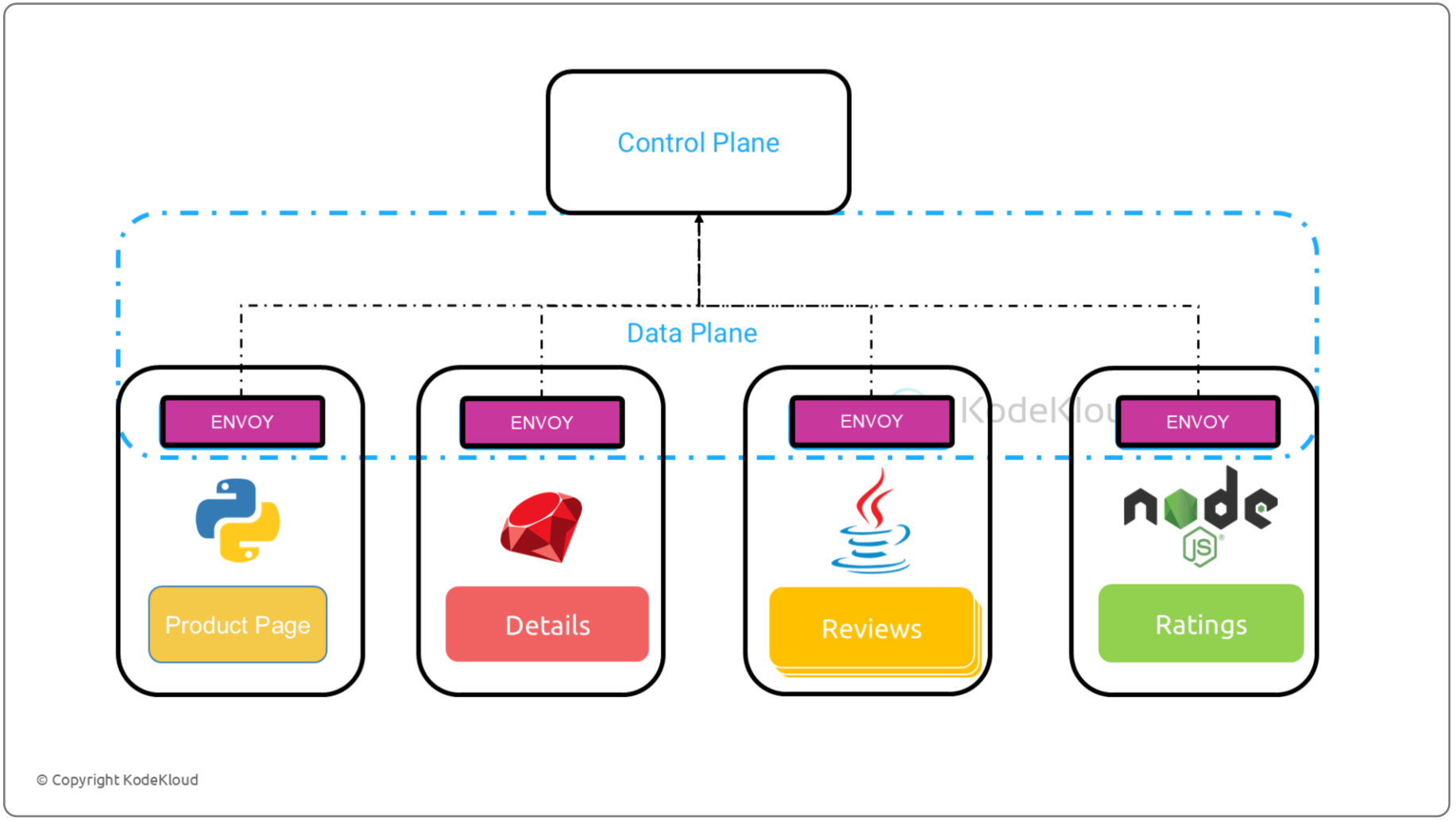


Now we'll have a look at Istio, how it works, its architecture and its core components now.

Istio is a free and open-source Service Mesh that provides an efficient way to secure, connect and monitor services.

Istio works with Kubernetes and traditional workloads thereby bringing universal traffic management, telemetry and security to complex deployments.

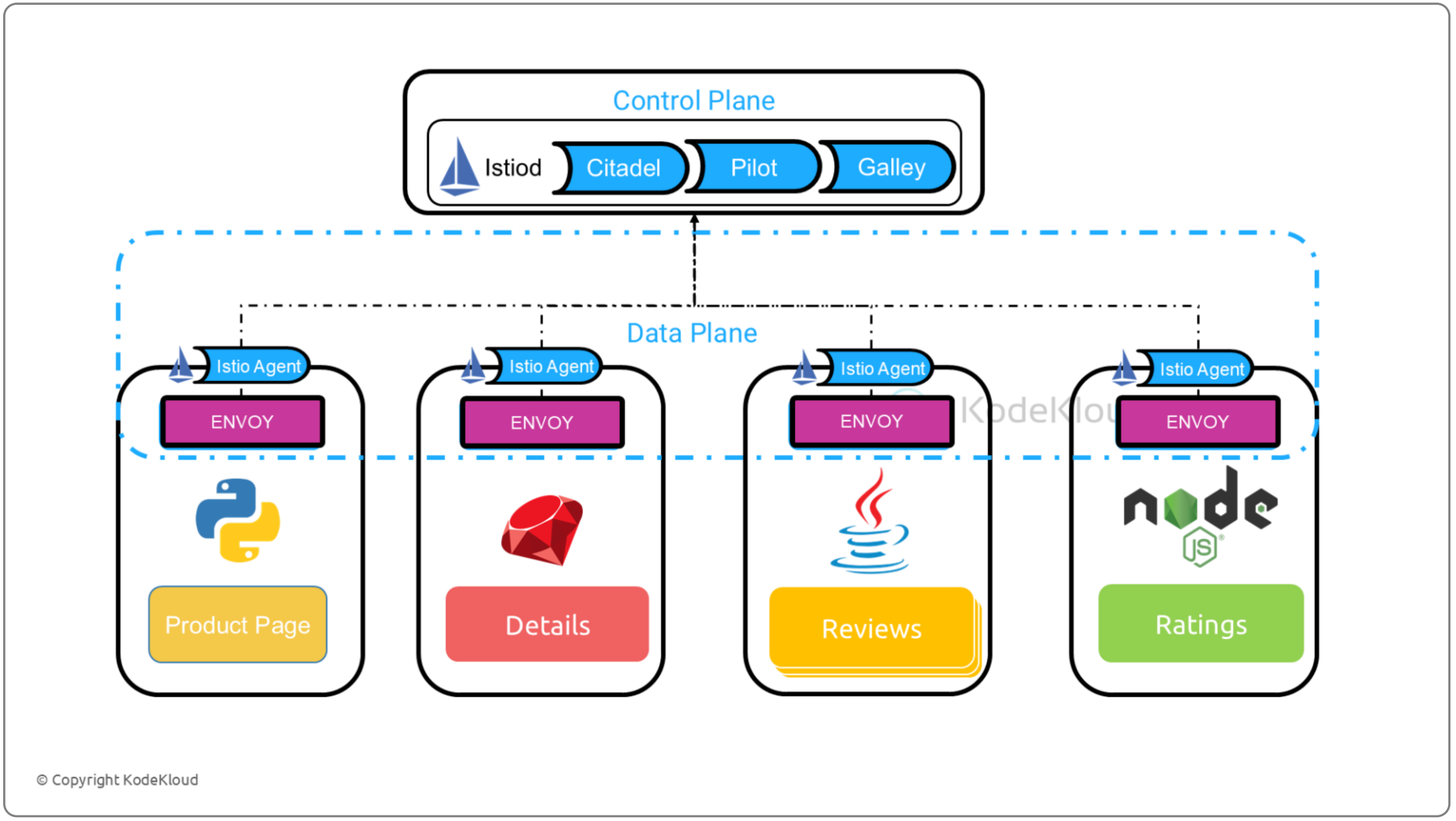
Istio is supported and implemented by leading cloud providers and consultants.



Earlier we talked about the proxy service that takes care of all the task that should be outsourced from the microservice. These proxies and the communication between them form the data plane.

2. Istio implements these proxies using an open-source high performance proxy known as Envoy.

The proxies talk to a server-side component known as the controlplane. Let's look at these in a bit more detail.



1. Originally the control plane consisted of 3 components – named Citadel, Pilot and Galley.

Citadel managed certificate generation.

Pilot helped with service discovery

And Galley helped in validating configuration files.

2. The 3 components were later combined into a single daemon called Istiod.

3. Each service or pod has a separate component in it along with the envoy proxy called as the Istio Agent. The Istio agent is responsible for passing configuration secrets to the envoy proxies.

So that’s a high level overview of Istio. Next we will get started with Istio by installing it and later we will dig deeper into Istio’s features and functionalities.