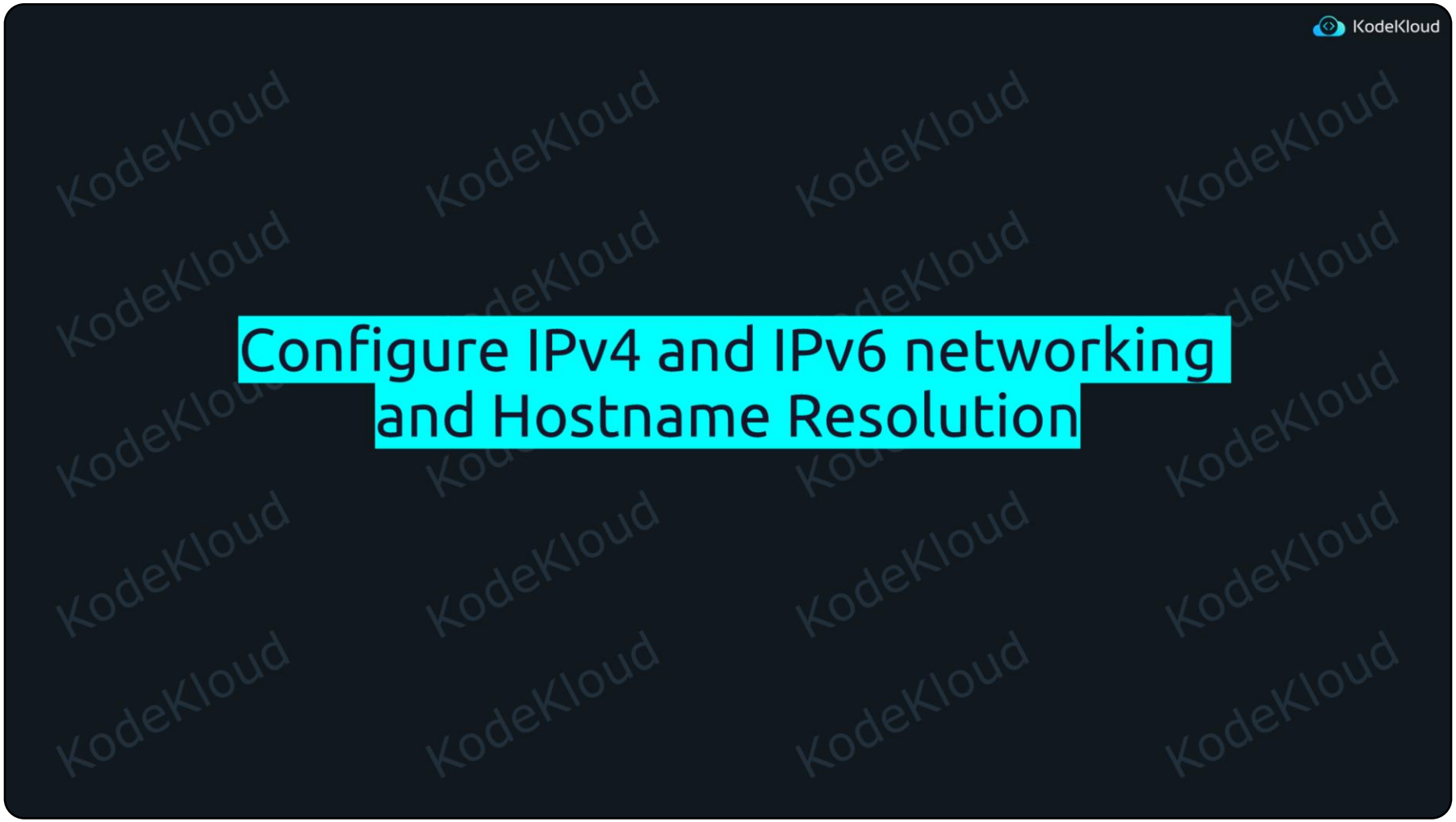
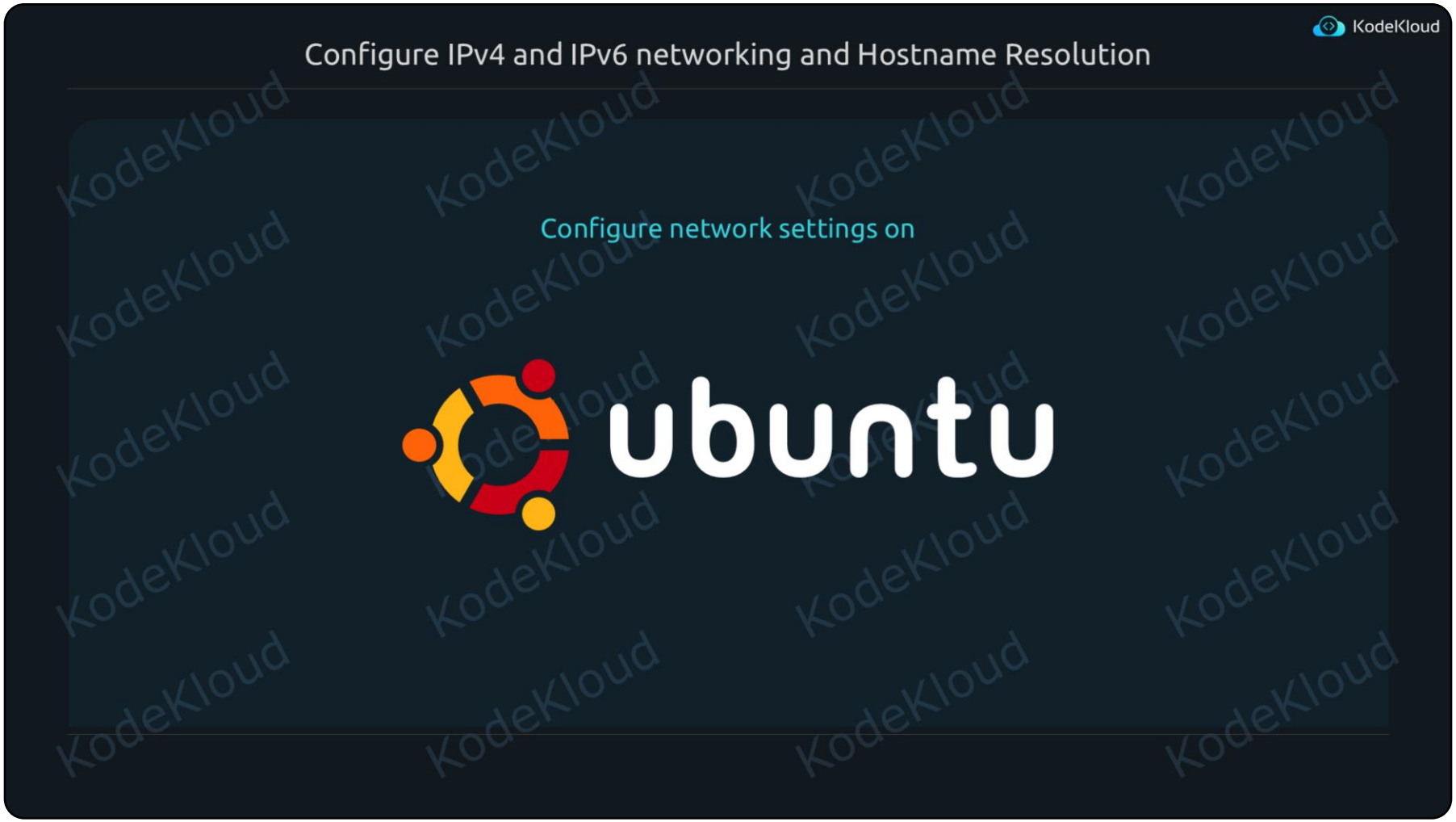
KodeKloud

Visit www.kodekloud.com to discover more.

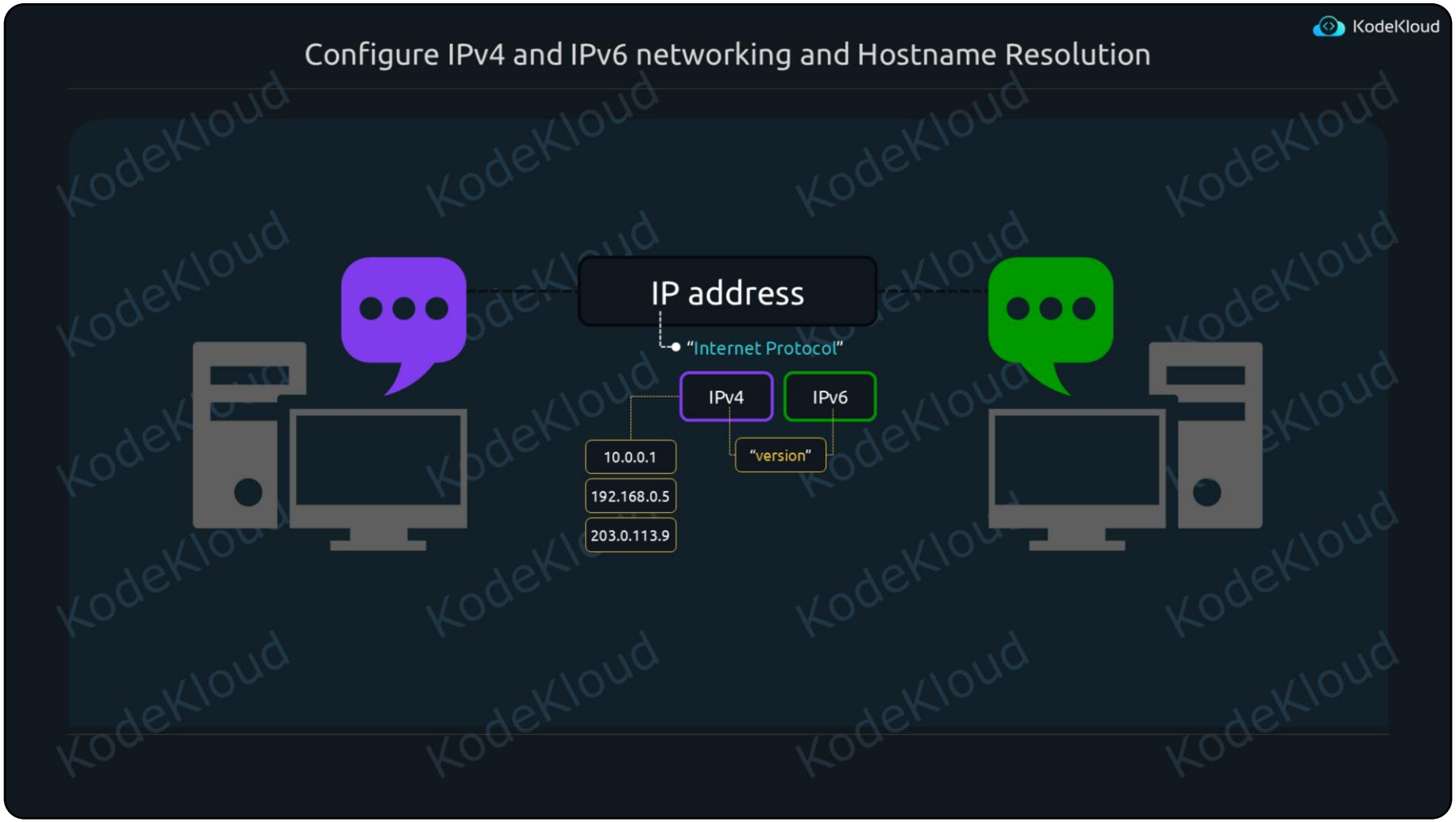


In this lesson we'll learn how to configure network settings on Ubuntu.



In this lesson we'll learn how to configure network settings on Ubuntu.

Let's start with the basic concepts first.

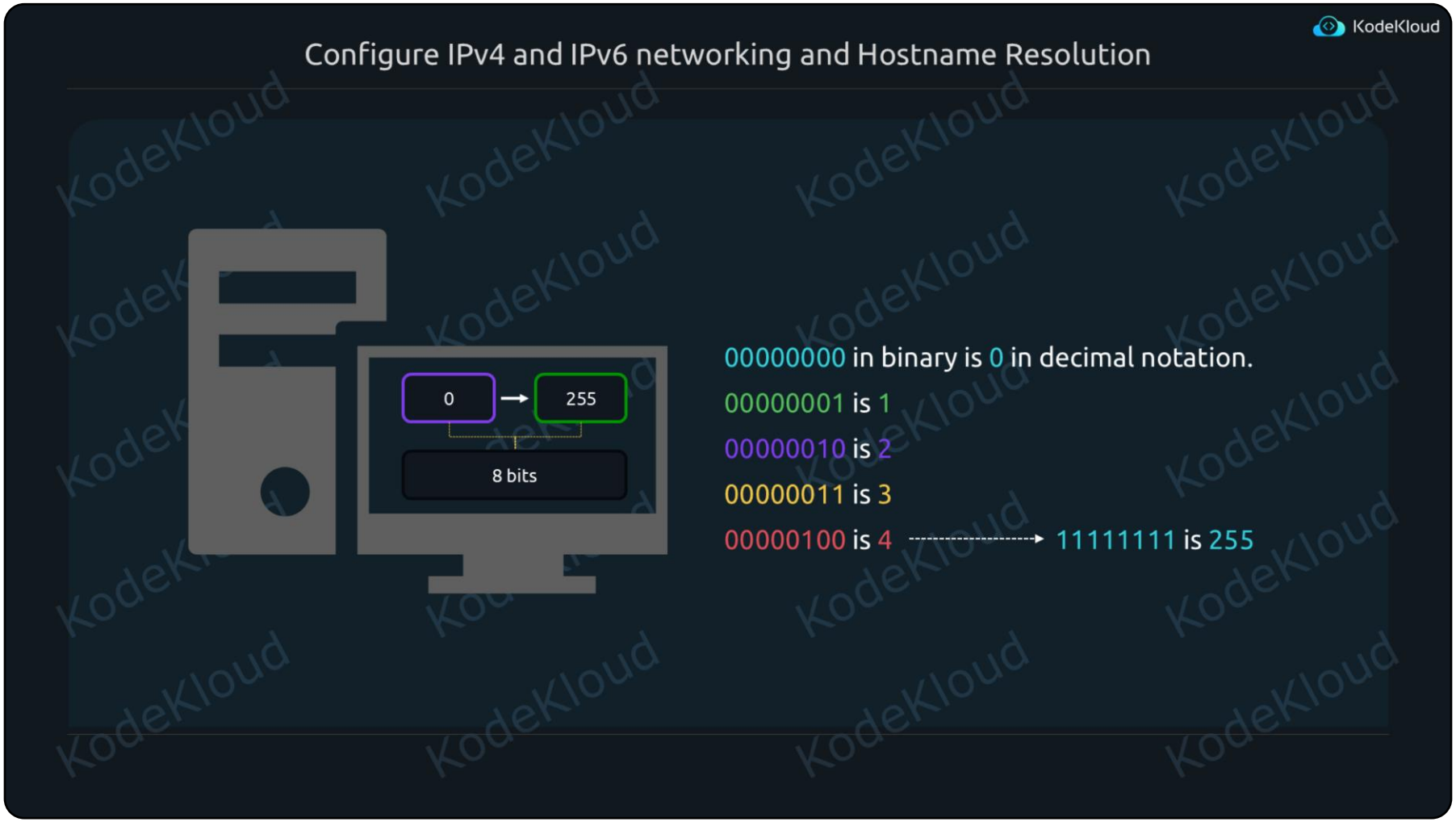


To be able to communicate across networks, a device needs an IP address. IP stands for Internet Protocol. And there are two kinds of IPs: IPv4 and IPv6. The "v" here stands for "version".

In version 4 of the Internet Protocol, addresses look like this:

10.0.0.1

192.168.0.5
203.0.113.9



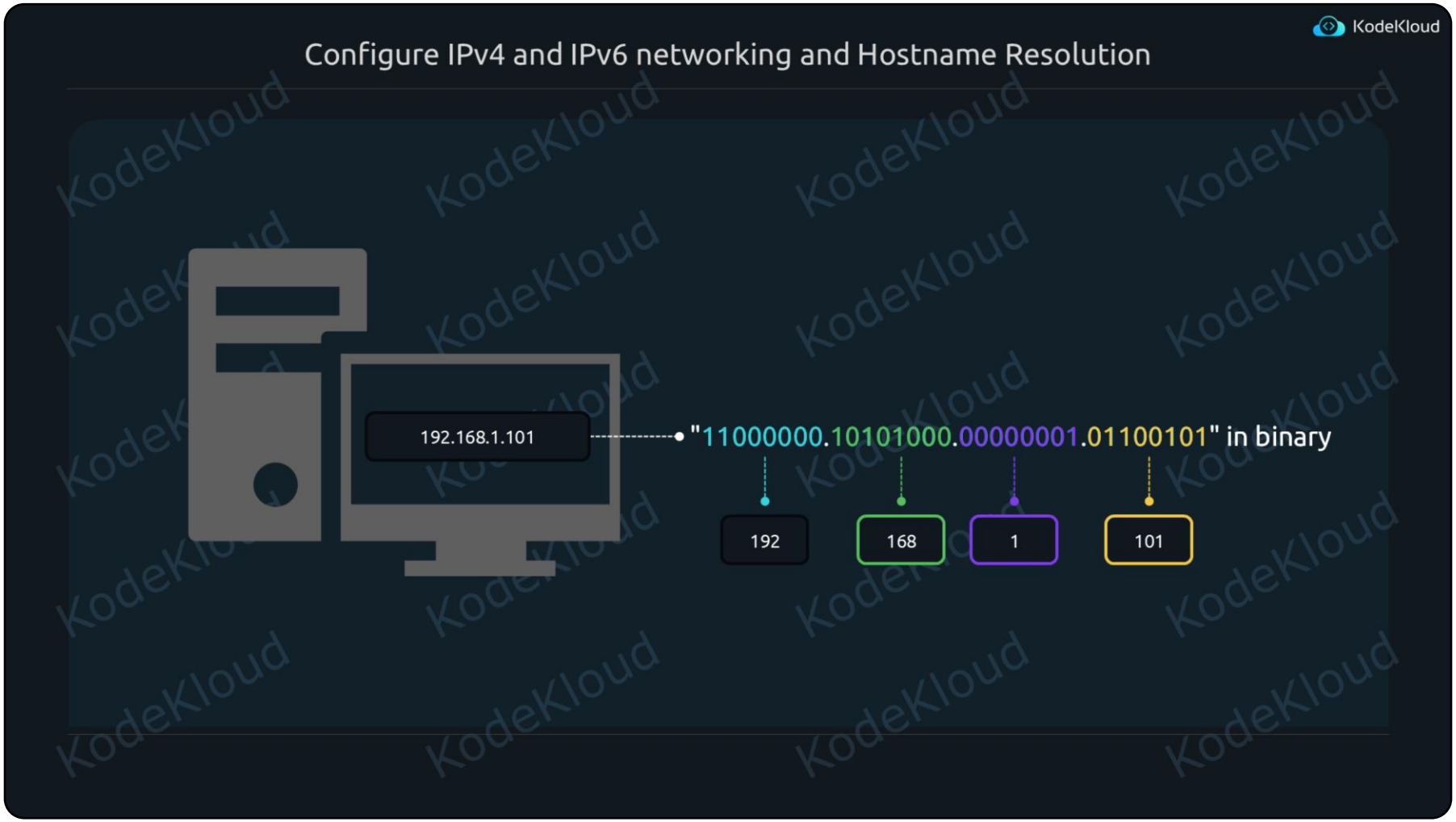
Basically, four numbers that can take values from 0 to 255. Why these specific values? Because internally, computers use 8 bits for each number here. And 8 bits can represent a number from 0 to 255. For example:

00000000 in binary is 0 in decimal notation.
00000001 is 1
00000010 is 2

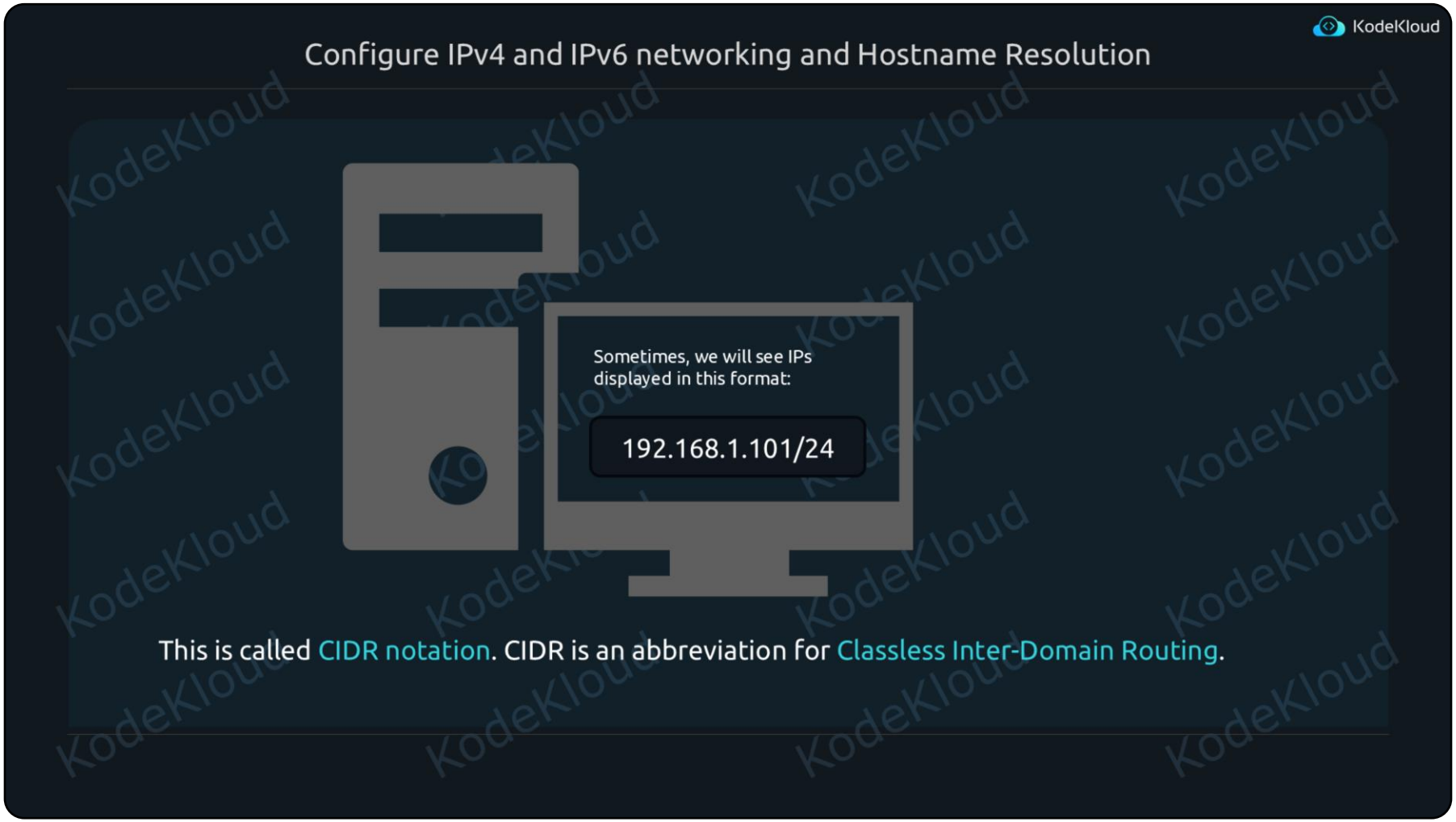
00000011 is 3
00000100 is 4

all the way to

111111111 which is 255



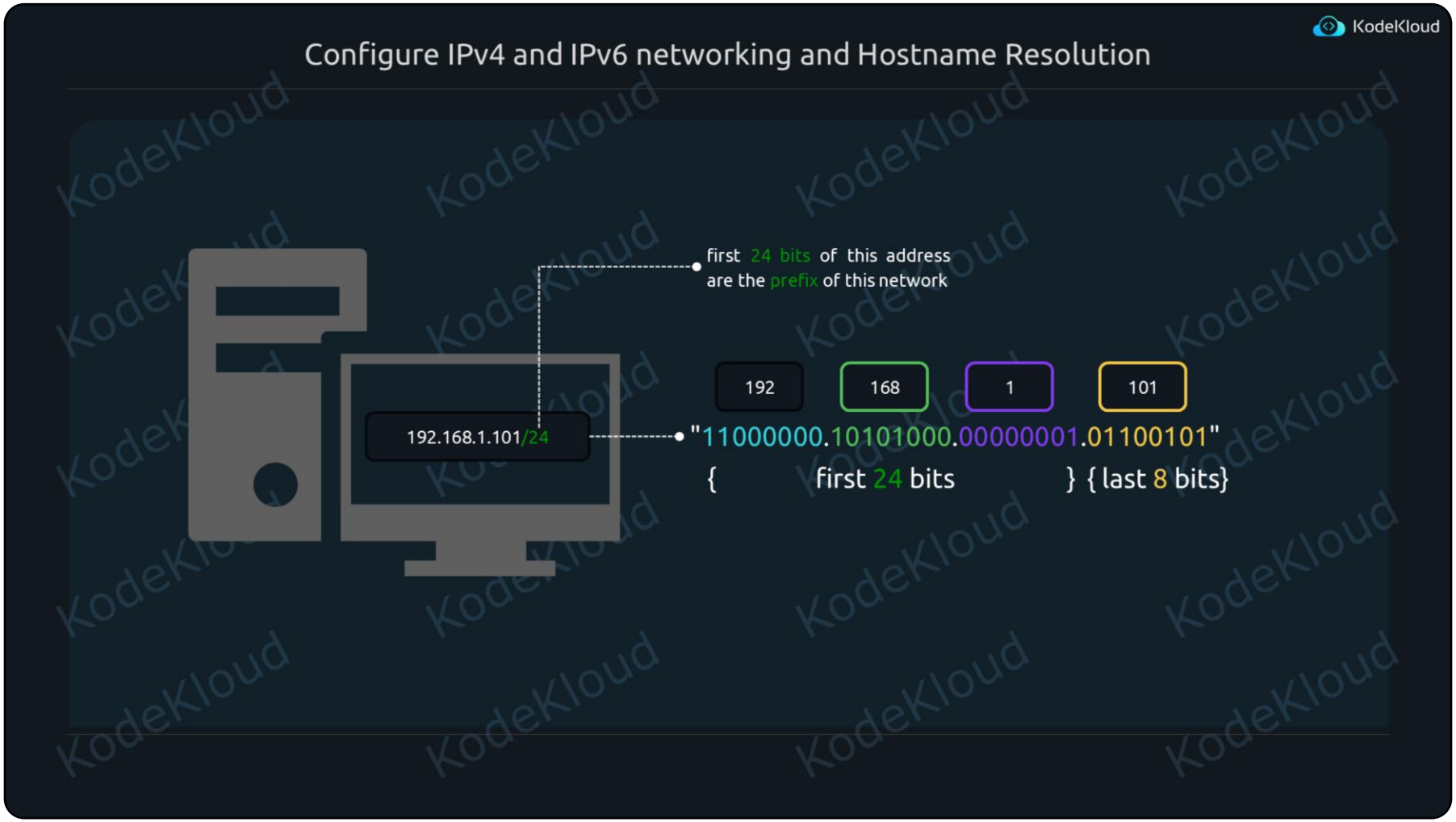
So an IP like "192.168.1.101" in decimal is actually "11000000.10101000.00000001.01100101" in binary. And this binary representation is important for the next thing we'll discuss.



Sometimes, we will see IPs displayed in this format:

192.168.1.101/24

This is called CIDR notation. CIDR is an abbreviation for Classless Inter-Domain Routing.



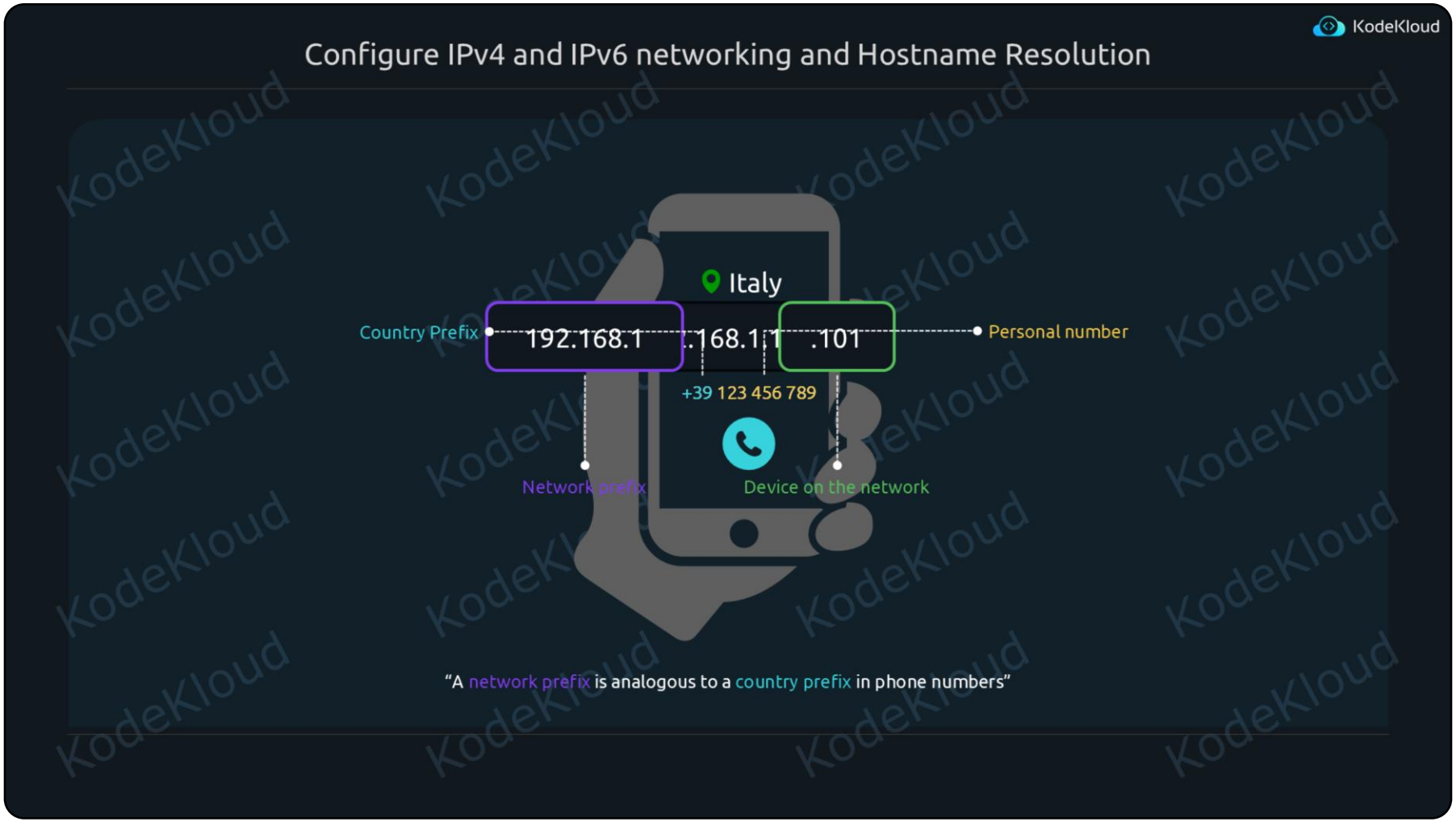
We can see an extra /24 at the end, next to the IP address. This tells us that the first 24 bits of this address are the prefix of this network.

If we take another look at the binary notation, this will become easier to understand.

192.168.1.101 in binary is:

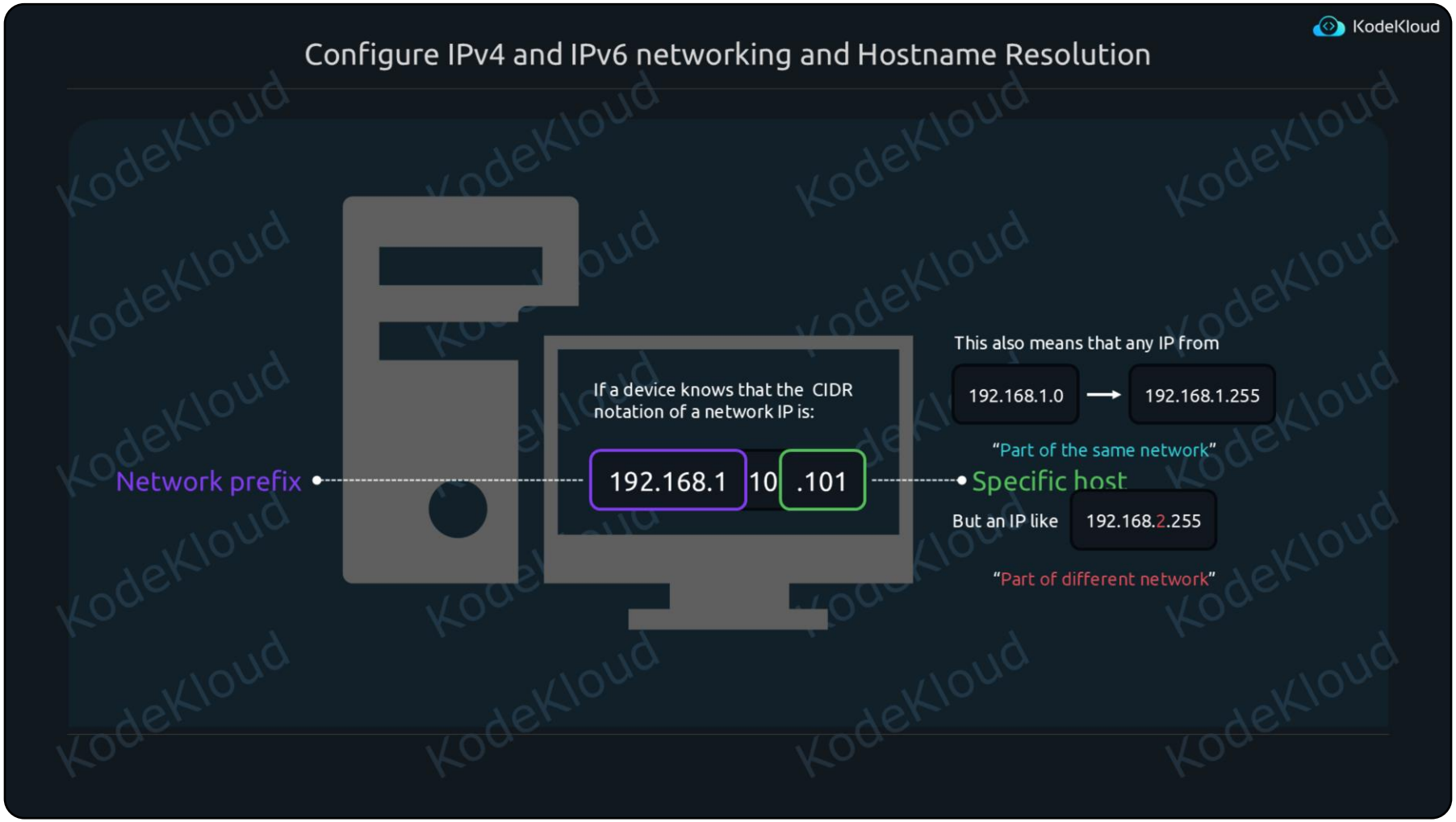
192 168 1 101
11000000.10101000.00000001.01100101
{ first 24 bits }{last 8 bits}

We can see that the first 24 bits align with the first three numbers we have in this IP address.



Which means that 192.168.1 is the network prefix. Basically, the address of the network itself. And the last part, .101 refers to a specific device on that network.

It's somewhat similar to phone numbers. For example, if you call someone from Italy, you need to use the +39 prefix. When you call +39 123 456 789, the first part, +39, is the country prefix, and 123 456 789 is the personal number of the one you are calling. So a network prefix is analogous to a country prefix in phone numbers.

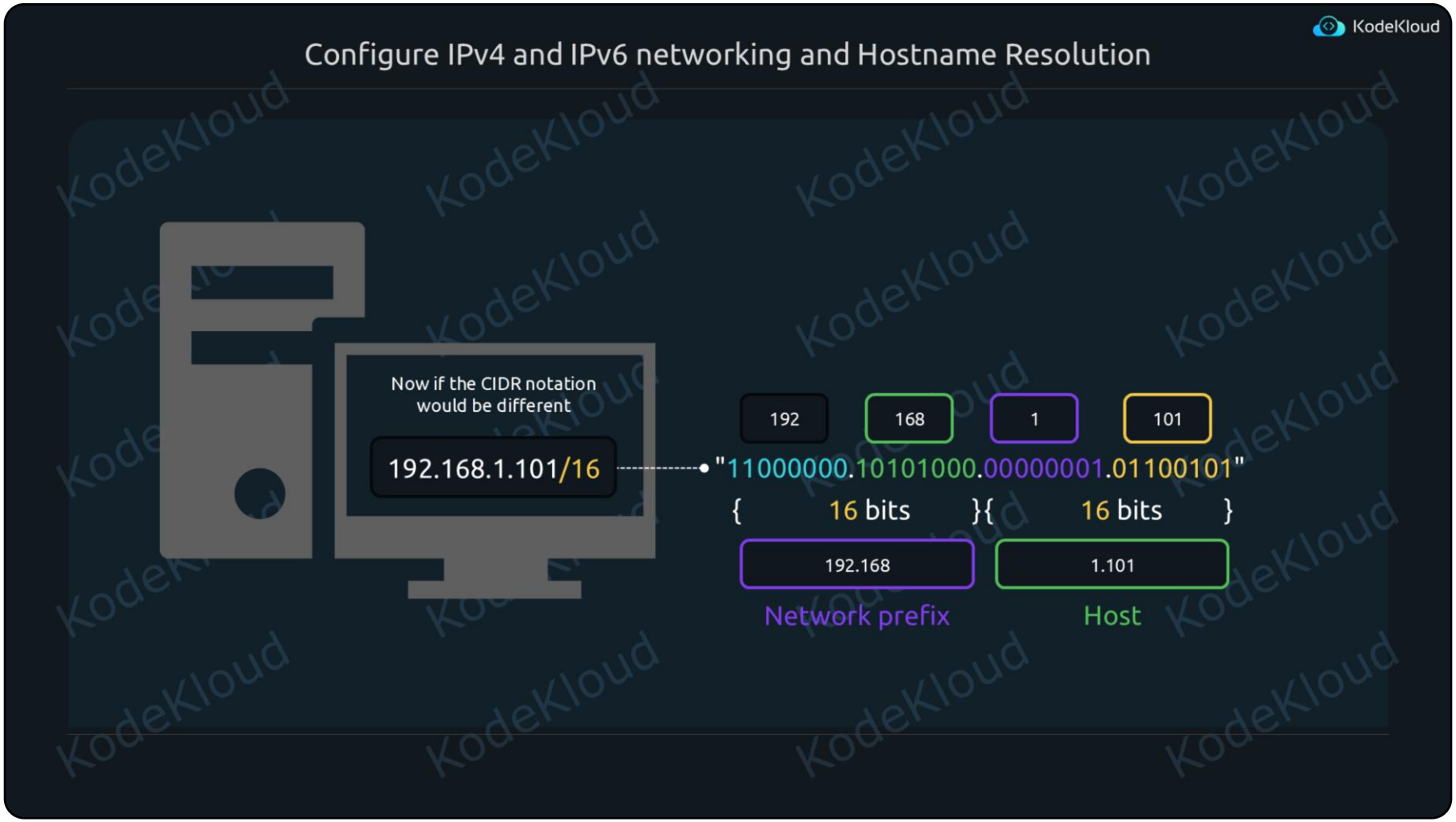


If a device knows that the CIDR notation of a network IP is:

192.168.1.101/24

then it understands that 192.168.1 refers to the generic network prefix, and .101 to a specific host on this network.

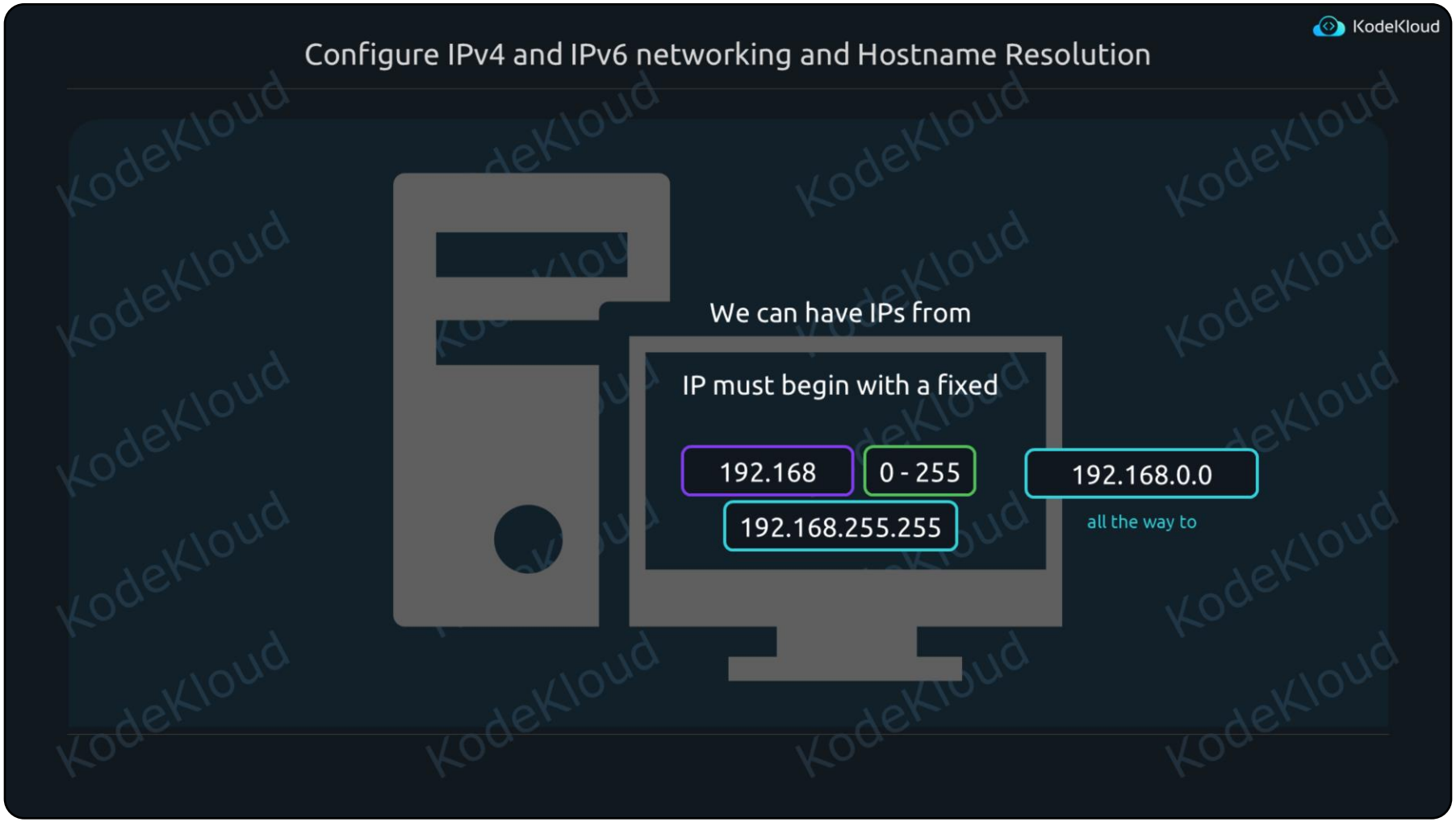
This also means that any IP from 192.168.1.0 all the way to 192.168.1.255 is part of this same network. But an IP like 192.168.2.101 is part of a different network, because the prefix does not have the same initial 3 numbers. Or, if we think in binary, it does not have the same first 24 bits, as the /24 CIDR notation indicates it should have.



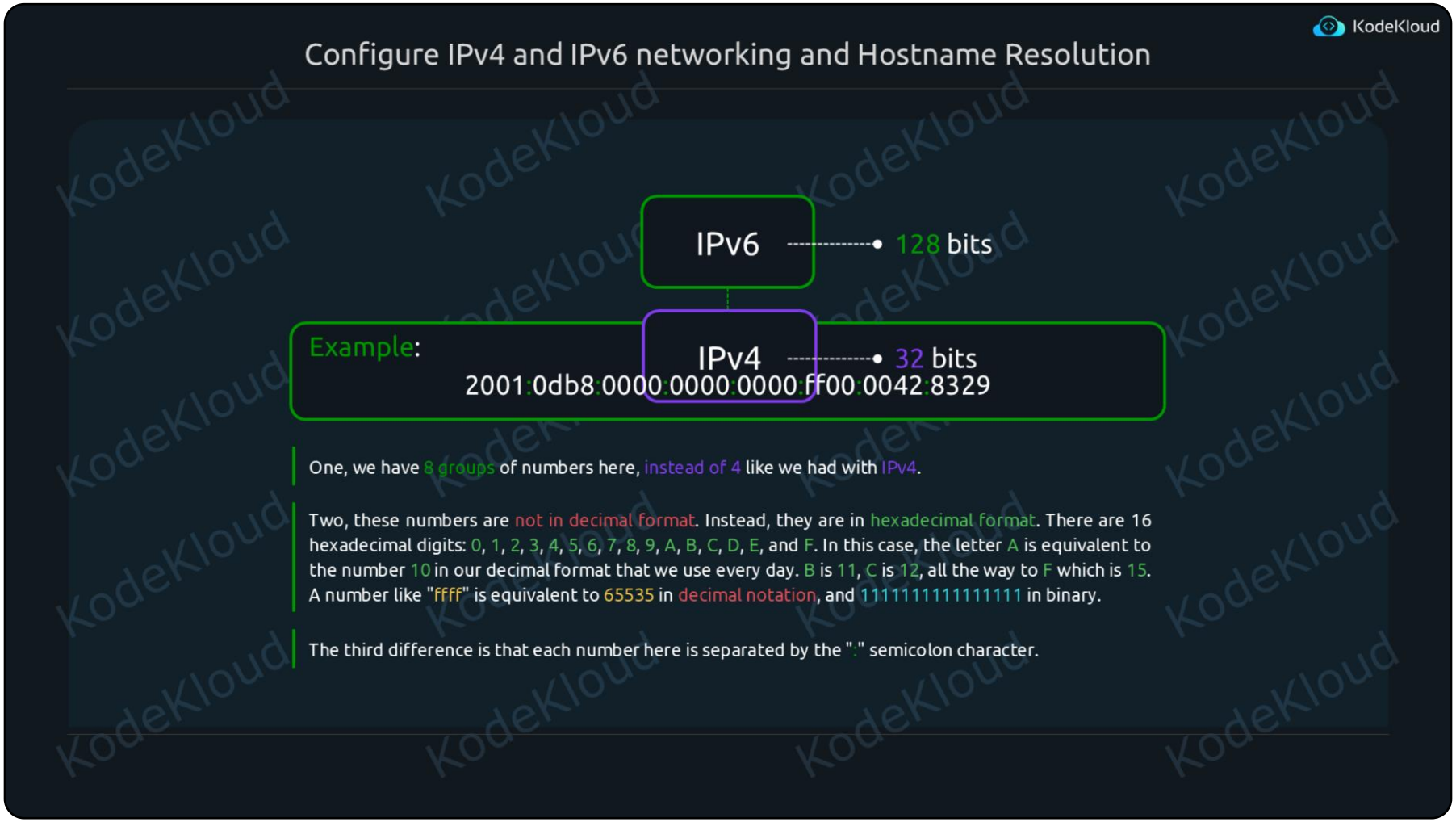
Now if the CIDR notation would be different, for example 192.168.1.101/16, we'd get something along these lines:

192 168 1 101
11000000.10101000.00000001.01100101
{ 16 bits }{ last 16 bits }
Network prefix Host

So in this case just the first 16 bits are the network prefix. Which coincide with the first two numbers: 192.168. This means that the last two numbers now, 1.101 refer to a host/device connected to this this network.



Otherwise said, for our current CIDR notation, any IP in this network must begin with a fixed "192.168." while the last two numbers can be anything between 0 and 255. So we can have IPs from: "192.168.0.0" all the way to "192.168.255.255".

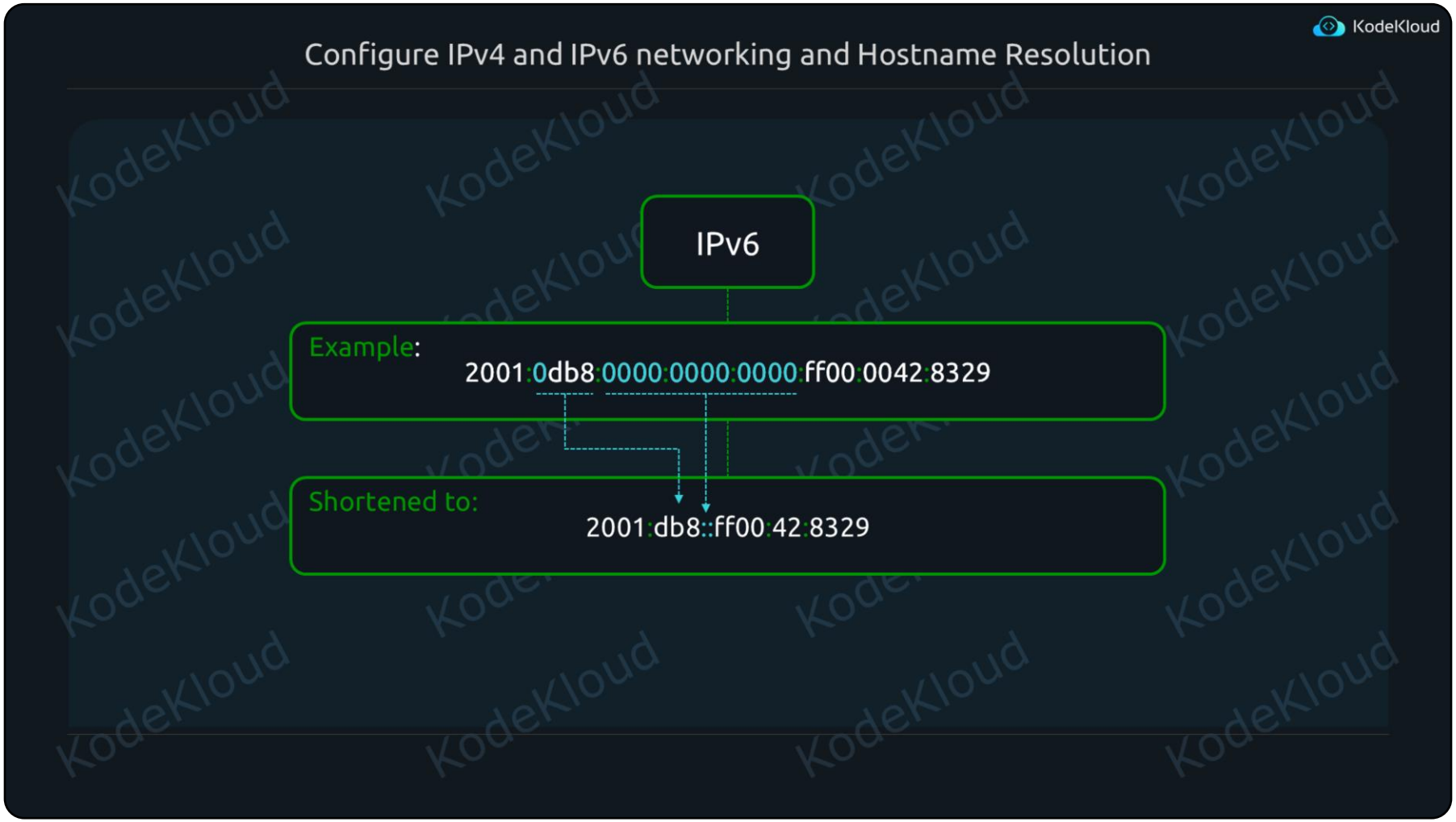


Now let's look at IPv6. These addresses have 128 bits, instead of 32 bits, as is the case for IPv4. Here is an example of an IPv6 address:

2001:0db8:0000:0000:0000:ff00:0042:8329

We can notice that this is a bit different in three key areas:

One, we have 8 groups of numbers here, instead of 4 like we had with IPv4.
Two, these numbers are not in decimal format. Instead, they are in hexadecimal format. There are 16 hexadecimal digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. In this case, the letter A is equivalent to the number 10 in our decimal format that we use every day. B is 11, C is 12, all the way to F which is 15. A number like "ffff" is equivalent to 65535 in decimal notation, and 1111111111111111 in binary.
The third difference is that each number here is separated by the ":" semicolon character.



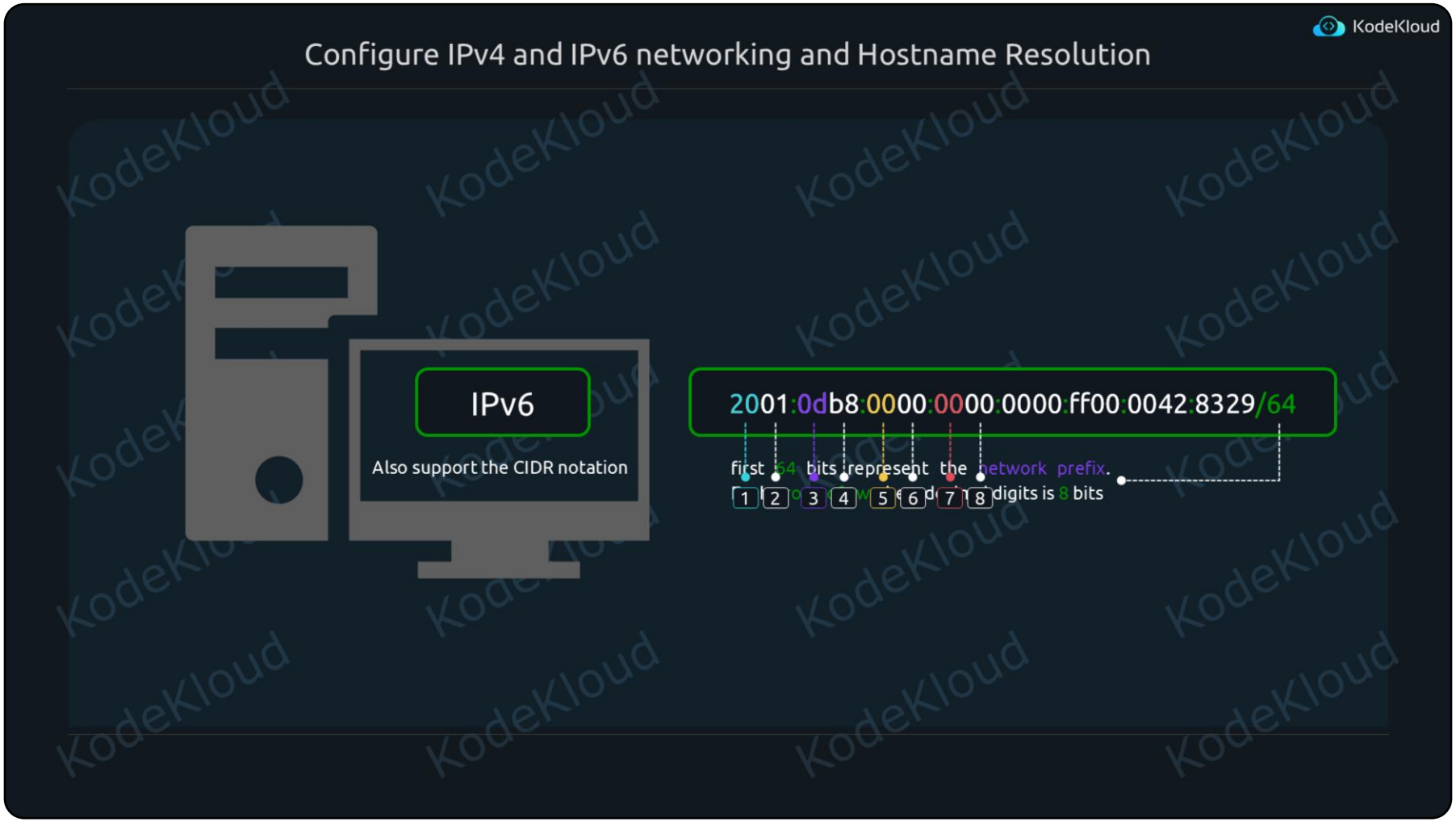
Since these IPv6 addresses are quite long, we'll often see them shortened. So an IP like:

2001:0db8:0000:0000:0000:ff00:0042:8329

will usually be shortened to

2001:db8::ff00:42:8329

This is done by removing all leading zeroes first. For example, we can see that "0db8" became "db8". And the consecutive group of "0000:0000:0000" was abbreviated to two consecutive semicolon characters "::".



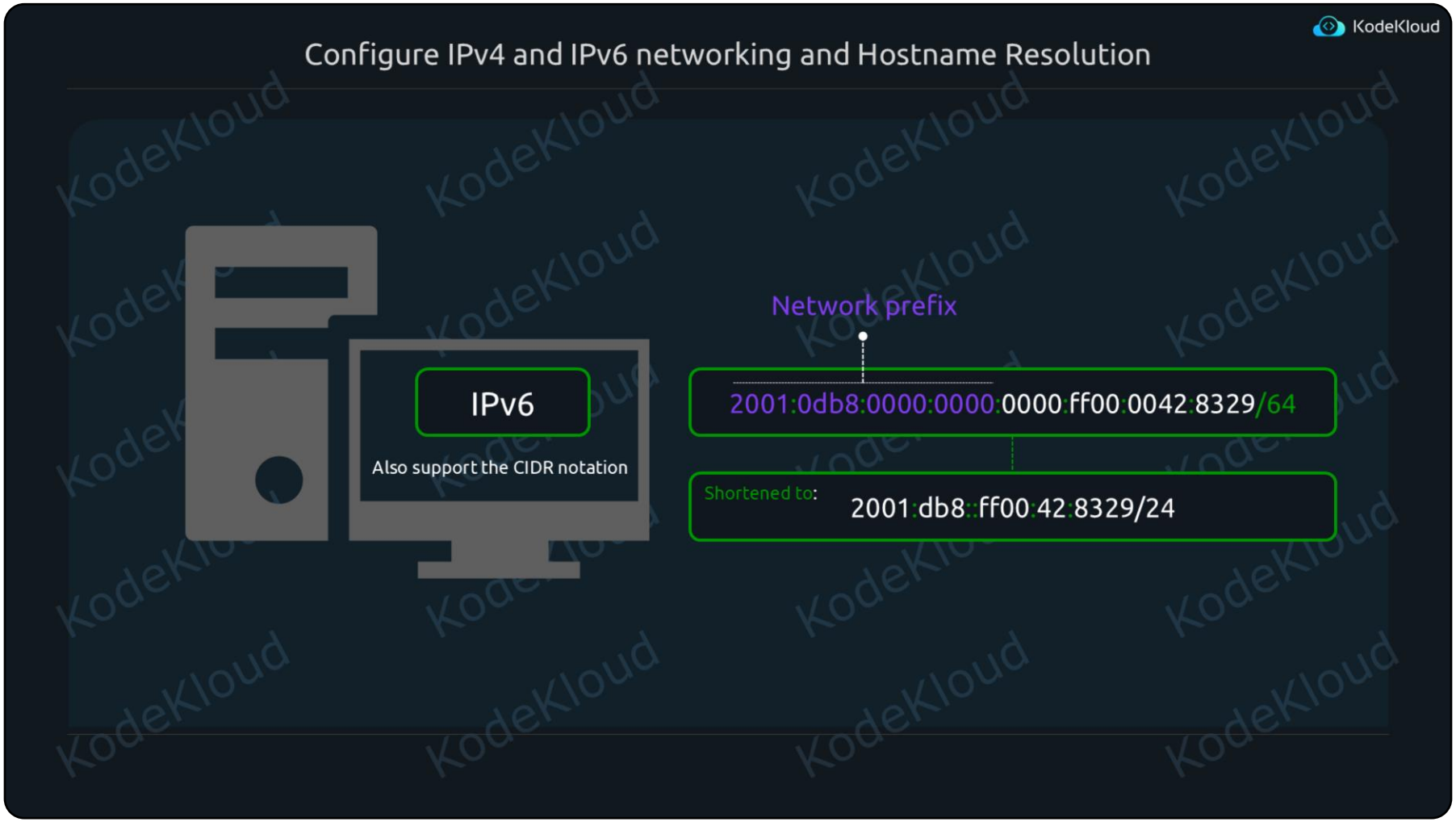
IPv6 addresses also support the CIDR notation. So an address like:

2001:0db8:0000:0000:0000:ff00:0042:8329/64

signals that the first 64 bits represent the network prefix. And each group of two hexadecimal digits is 8 bits.

2001:0db8:0000:0000:0000:ff00:0042:8329/64
1 2   3 4   5 6   7 8

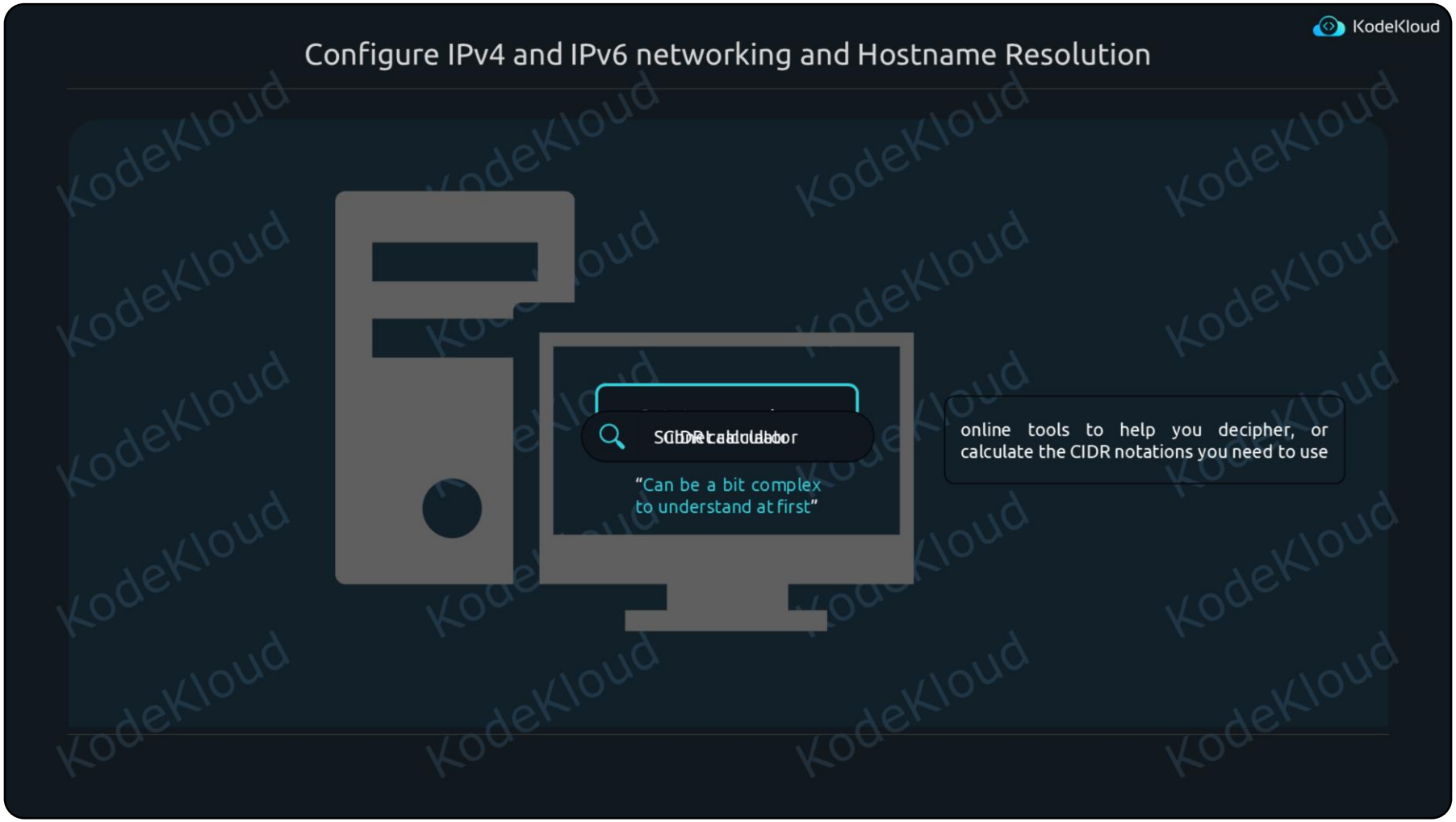
So 64 bits would mean 8 such groups.



In this case, 2001:0db8:0000:0000 is the network prefix of this IP, for a /64 CIDR notation.

Of course if the IP is shortened, you'll usually see this noted as:

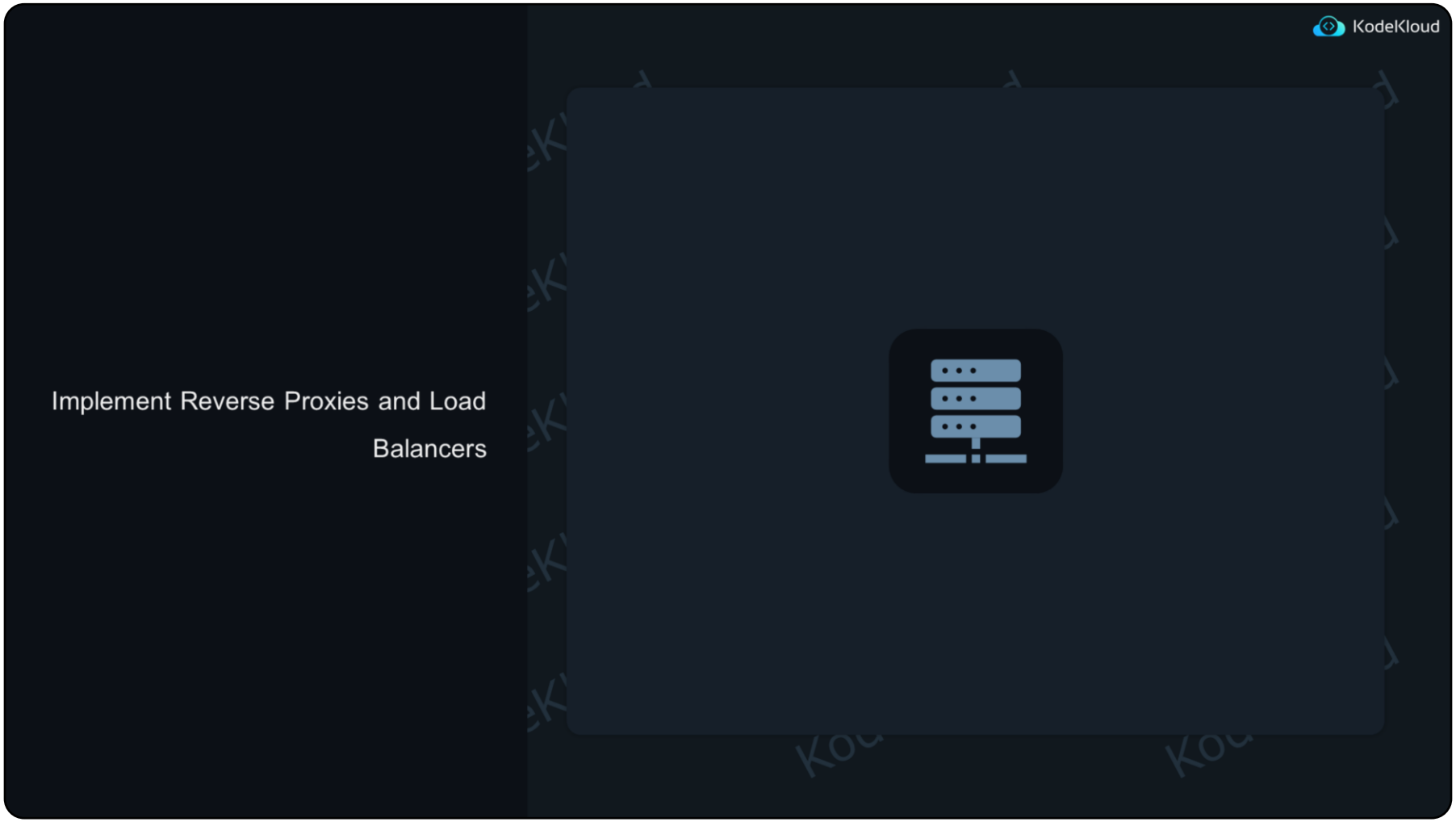
2001:db8::ff00:42:8329/24



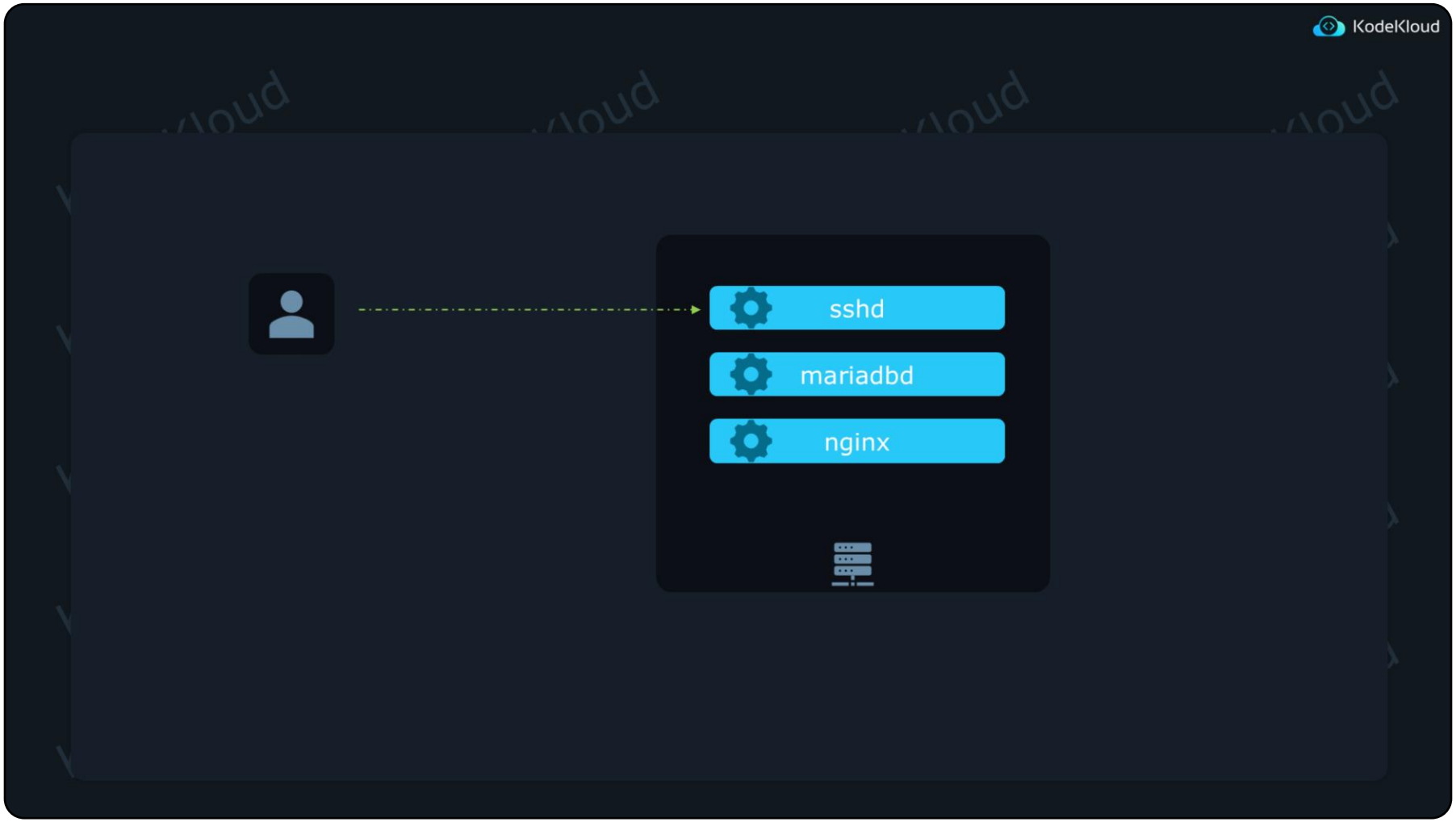
As we can see, for prefixes that are multiple of the number 8, things are straightforward enough. Unfortunately, some CIDR notations can be a bit harder to understand. If the prefix is 20 bits, for example, then this does not coincide with the first 3 numbers in an IPv4 address anymore. But there are online tools to help you decipher, or calculate the CIDR notations you need to use. Just look up "CIDR calculator" or "subnet calculator" on the Internet.

KodeKloud

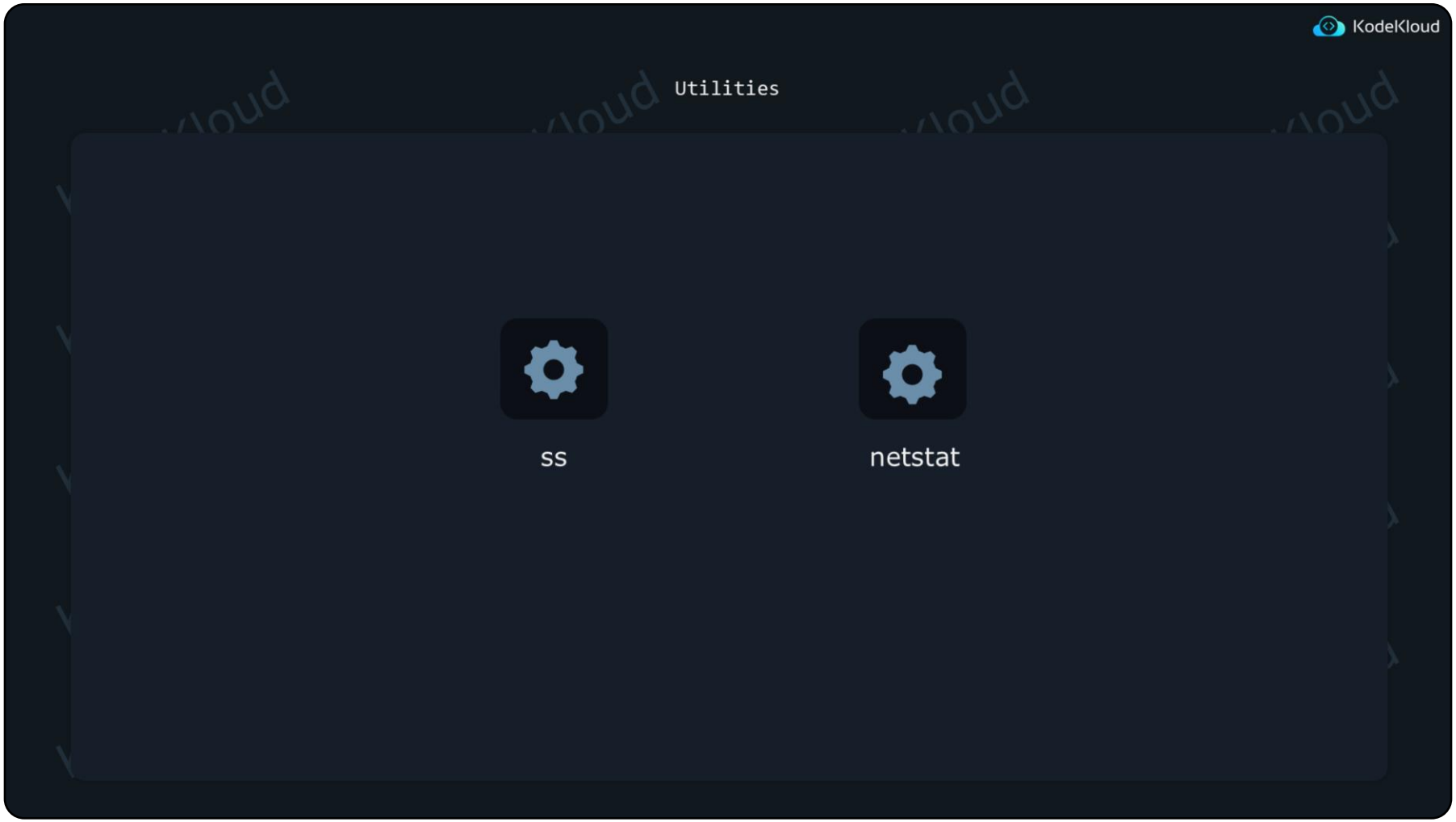
Visit www.kodekloud.com to discover more.



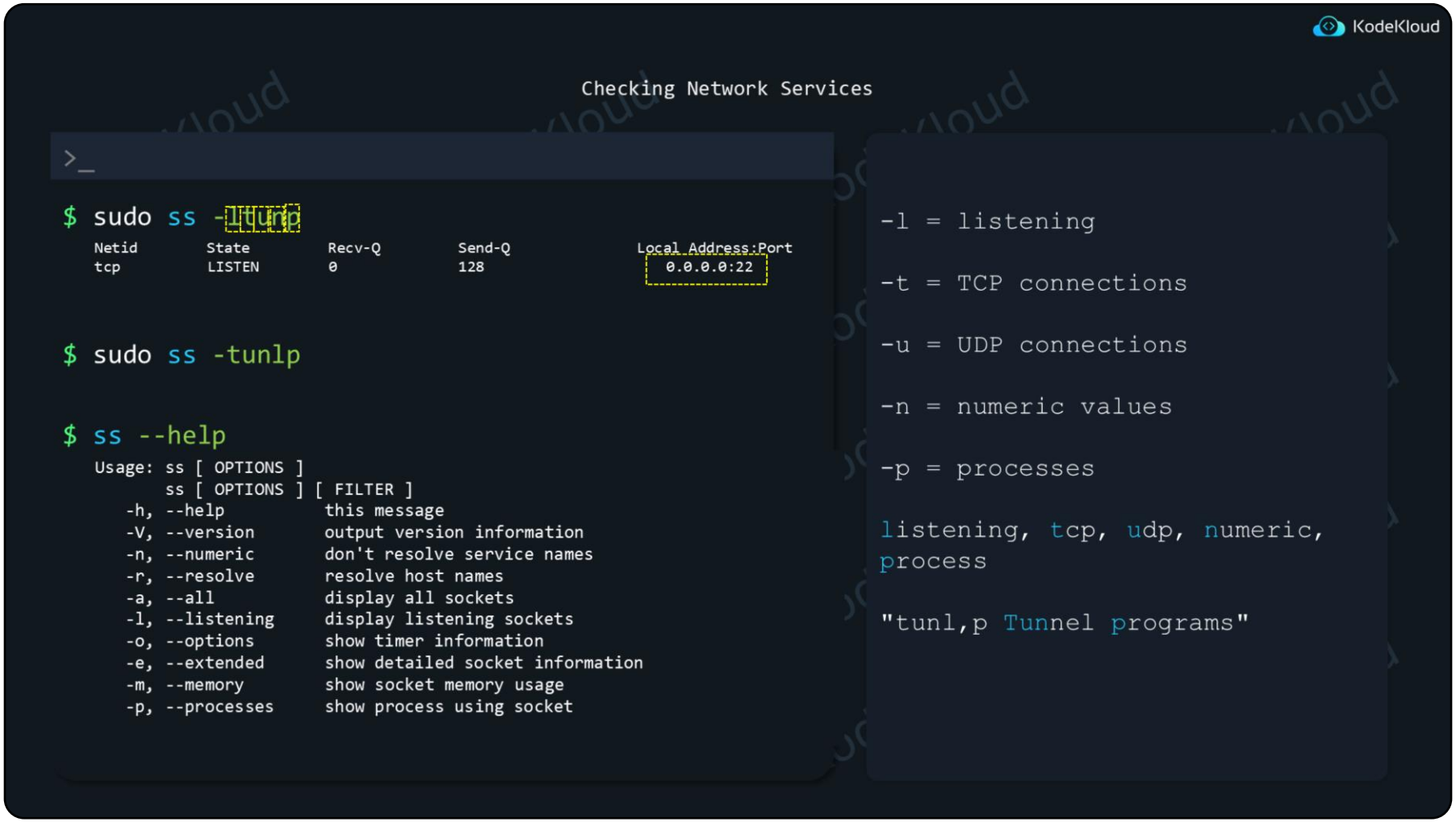
Now, let's look at starting, stopping, and checking the status of network services in Linux.



Most servers have at least a few services running on them that deal with network data in one way or another. One example is the SSH daemon, running in the background and listening for incoming network connections, to let people log in remotely. Let's see how we can inspect what programs are currently running and waiting for incoming network connections.



There are two utilities we can use: ss and netstat. ss is the more modern tool, netstat is older and might get removed in future versions of Linux-based operating systems.



To see programs that are ready to accept incoming network connections, we can use this command:

sudo ss –ltunp

The output here is truncated to fit on our screen, but in the next screen we'll show output for the ss utility with all fields displayed.

-l lists what is currently "listening" for connections. "listen" in this context means that something is scanning for incoming network data and is ready to accept an incoming connection.

-t lists TCP connections

-u lists UDP connections

-n lists numeric values. A command like "ss -ltup", without the -n option will show us output with the service name instead of the port number. With the -n option we get port numbers, so we can see the exact port number, 22, where the SSH daemon is listening for incoming connections.

-p shows which process is involved with every entry of our output

Normally, ss does not require root privileges. But in this case, because we use -p, we'll need to add sudo in front of our command. That's because we need root privileges to explore processes owned by root that are listening for incoming connections.

To more easily remember all these 5 options, -ltunp, you can think of this:

listening, tcp, udp, numeric, process

Another trick to remember these options is to reorganize them and write in a different order:

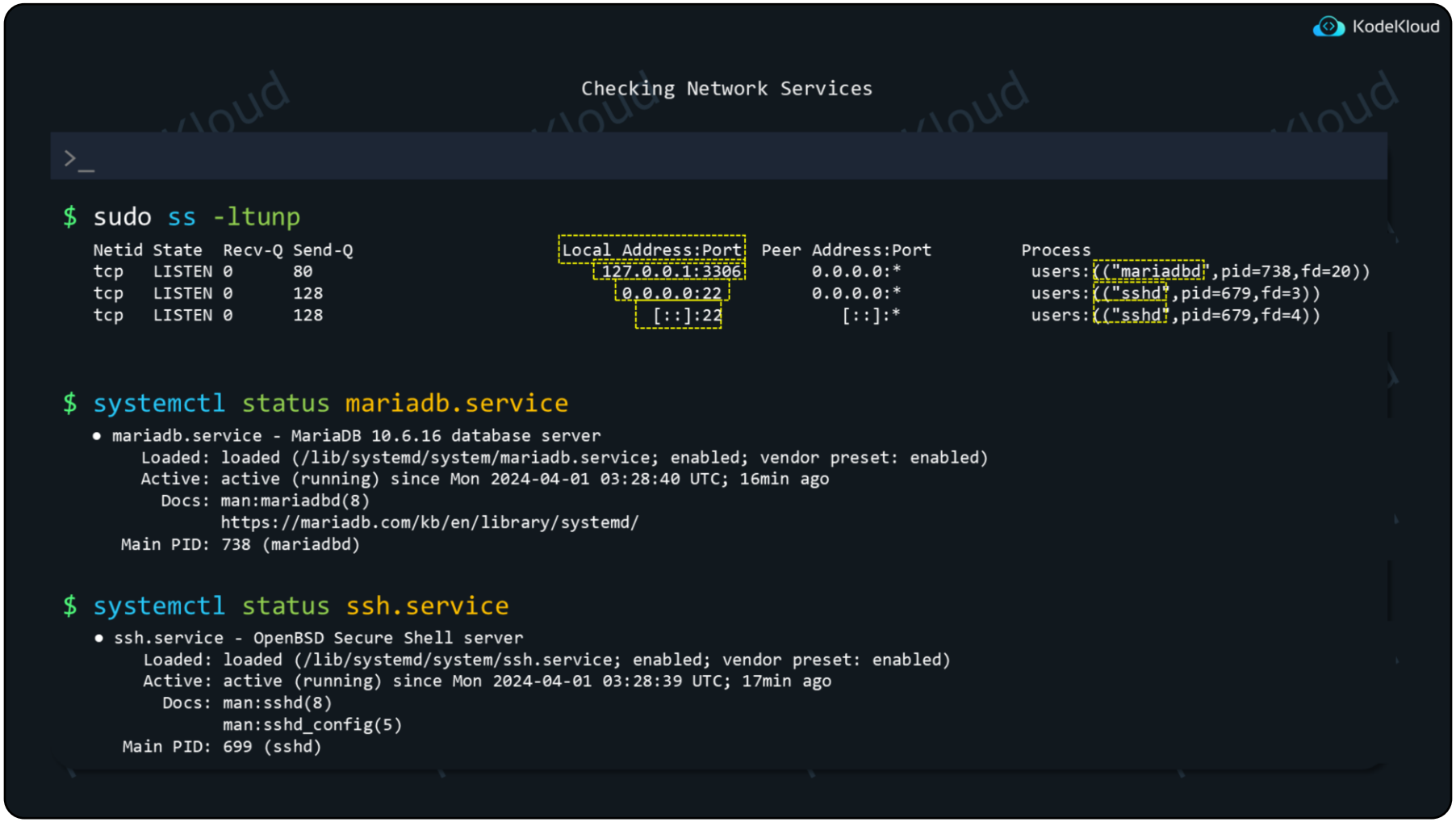
sudo ss -tunlp

You can think of "tunl,p Tunnel programs" as a quick way to memorize the 5 letters.

If you ever forget the right options, just enter:

ss --help

for a quick refresh.



In the output of our "sudo ss -ltunp" command, we can first take a look at the Local Address:Port column. An entry like this

127.0.0.1:3306

means that the program is listening for incoming connections from the system itself, but not from other computers.

127.0.0.1 is the IP address for the so-called localhost. Basically, the same computer this program is running on This way, a web server application like Nginx can connect to a database server like MariaDB, if they're both running on the same computer. Nginx can connect to 127.0.0.1, port 3306 and talk to MariaDB. In a way, MariaDB is in a protective bubble. It accepts connections only from programs running on our server but ignores connections coming from external devices.

To see what is listening for connections coming from the "outside" world, any computer on the network or the Internet, we can look for entries like this:

0.0.0.0:22

0.0.0.0 means this will accept connections from any external IP address.

[::]:22 means the same thing as 0.0.0.0:22. It just indicates that this is listening for incoming IPv6 connections, instead of IPv4. So the SSH daemon accepts connections on both the IPv4 and IPv6 protocols in this case.

Finally, we can see that connections on port 3306 are accepted by a process called mariadbd. That is the MariaDB daemon, our database server aplication. Also, connections on port 22 are accepted by our SSH daemon, running under a process called sshd. These are good hints on what we can explore next. To check the status of these network services, we can use the systemctl command:

systemctl status mariadb.service

and

systemctl status ssh.service

Note that we omit the "d" letter at the end, which stands for daemon. The service names on Ubuntu just use the name of the program itself. But on other operating systems, like Red Hat, services do have the that "d"

letter at the end. So what is ssh.service on Ubuntu is sshd.service on Red Hat.

To stop the mariadb service, we can use the usual:

sudo systemctl stop mariadb.service

If we check with

sudo ss -ltunp

again, we'll see connections are no longer accepted on port 3306 as we just closed the database server.

And of course, we could also disable the service from autostarting at boot with:

sudo systemctl disable mariadb.service

And we can enable it to autostart with:

sudo systemctl enable mariadb.service

And start it up again with:

sudo systemctl start mariadb.service

Notice that:

sudo ss -ltunp

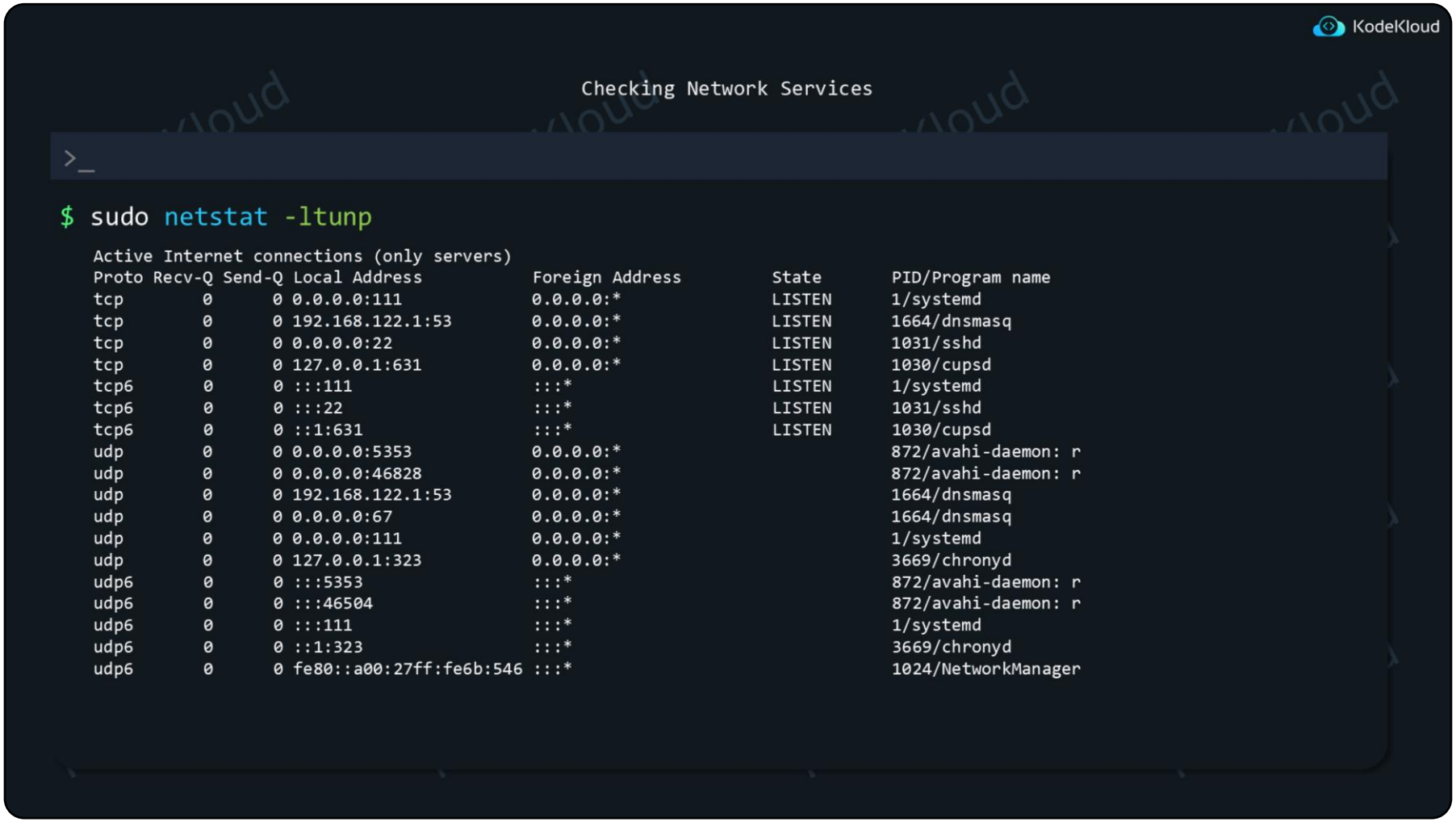
also shows us the PID of each process, in case we need to use it with other commands. For example, after we see that the SSH daemon is running under PID 679, we can explore it with a ps command:

ps 679

Or check out the files opened by this process with

sudo lsof -p 679

The PID can also be useful when checking logs.



It's worth mentioning at the end of our lesson that netstat uses almost the same options as ss, so a command like this will do the same thing as ss:

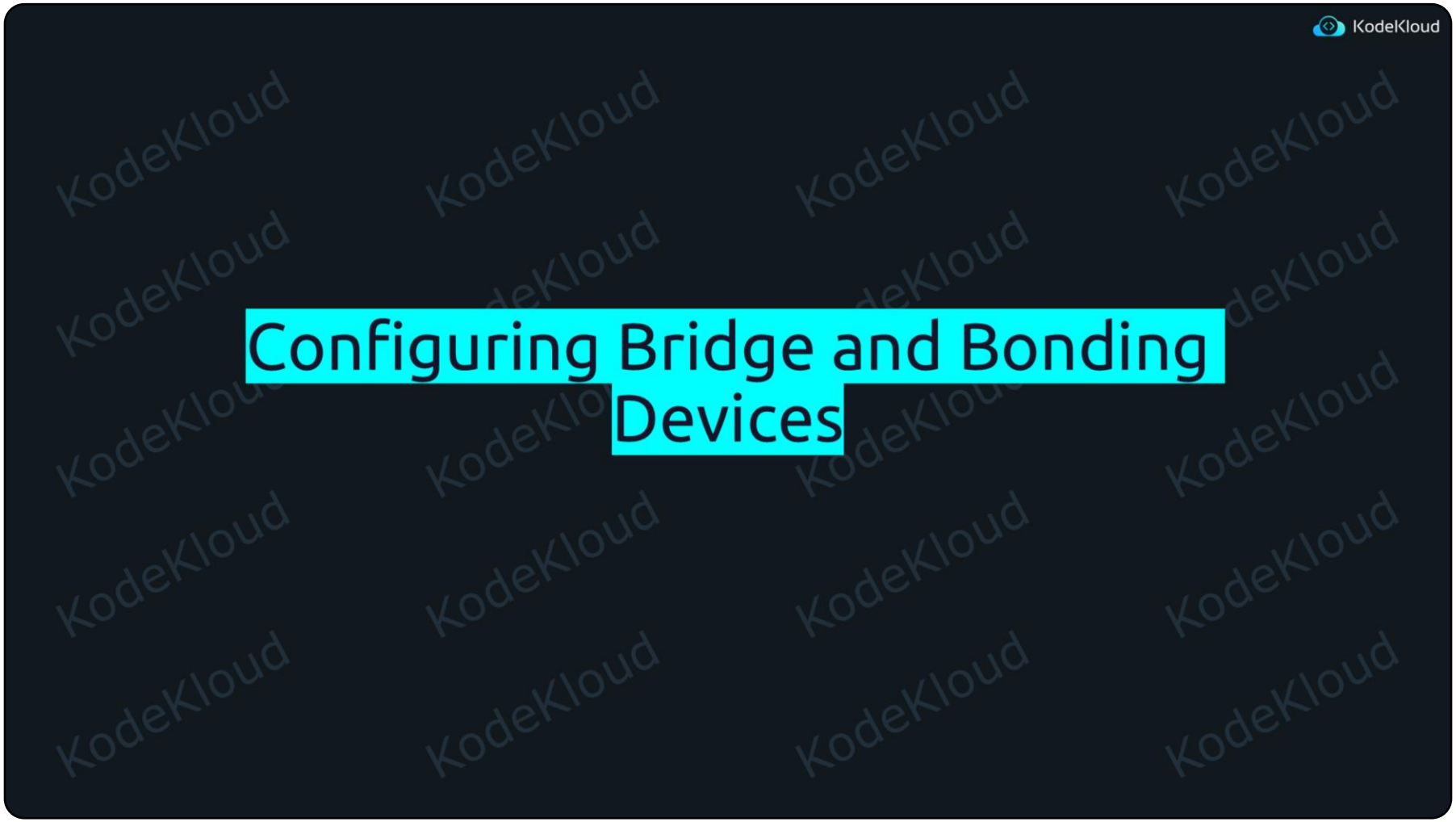
sudo netstat -ltunp

netstat's output is a bit more nicely formatted so you might prefer the cleaner output. Just remember that netstat

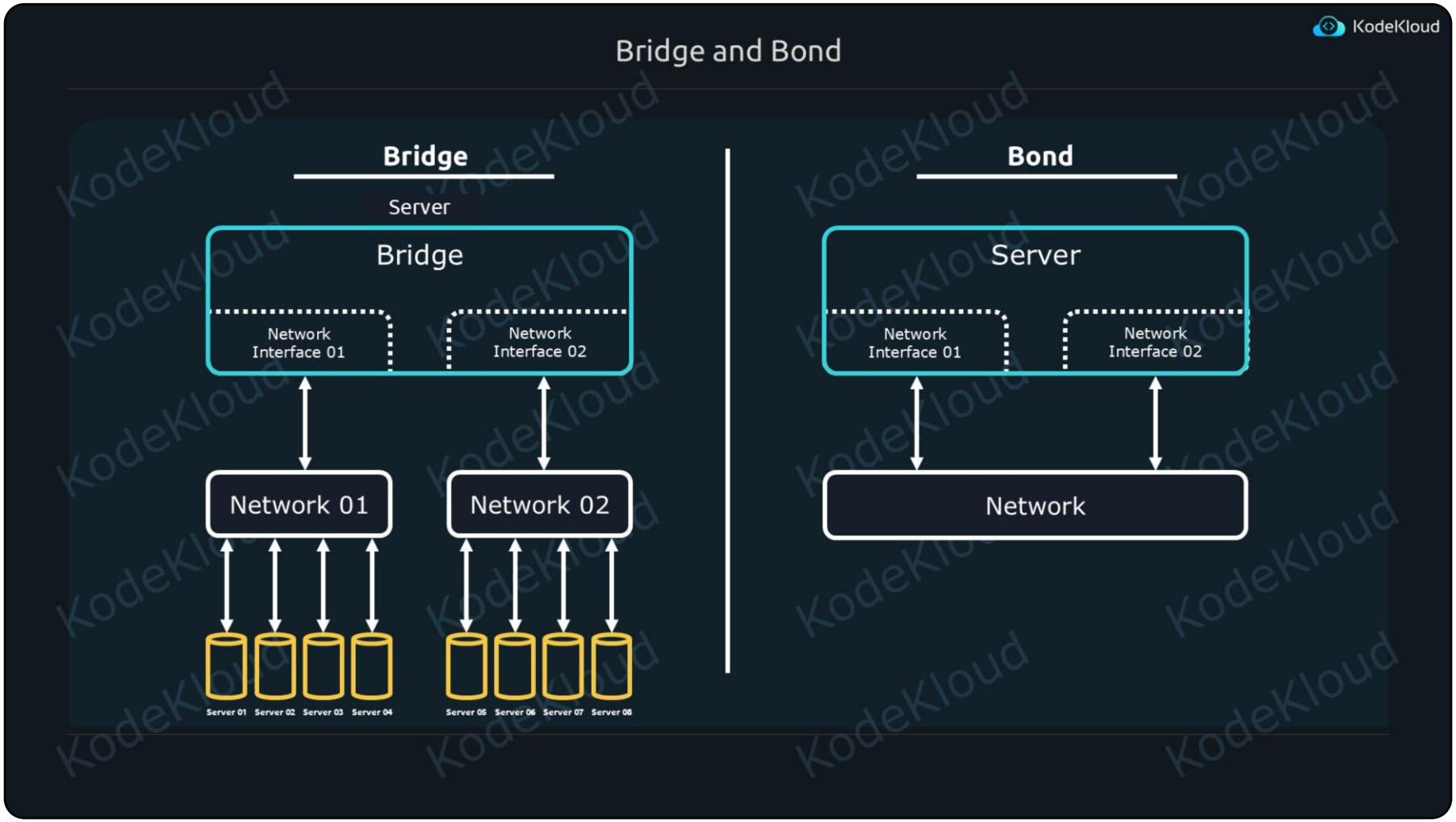
might not be available on all systems by default.

KodeKloud

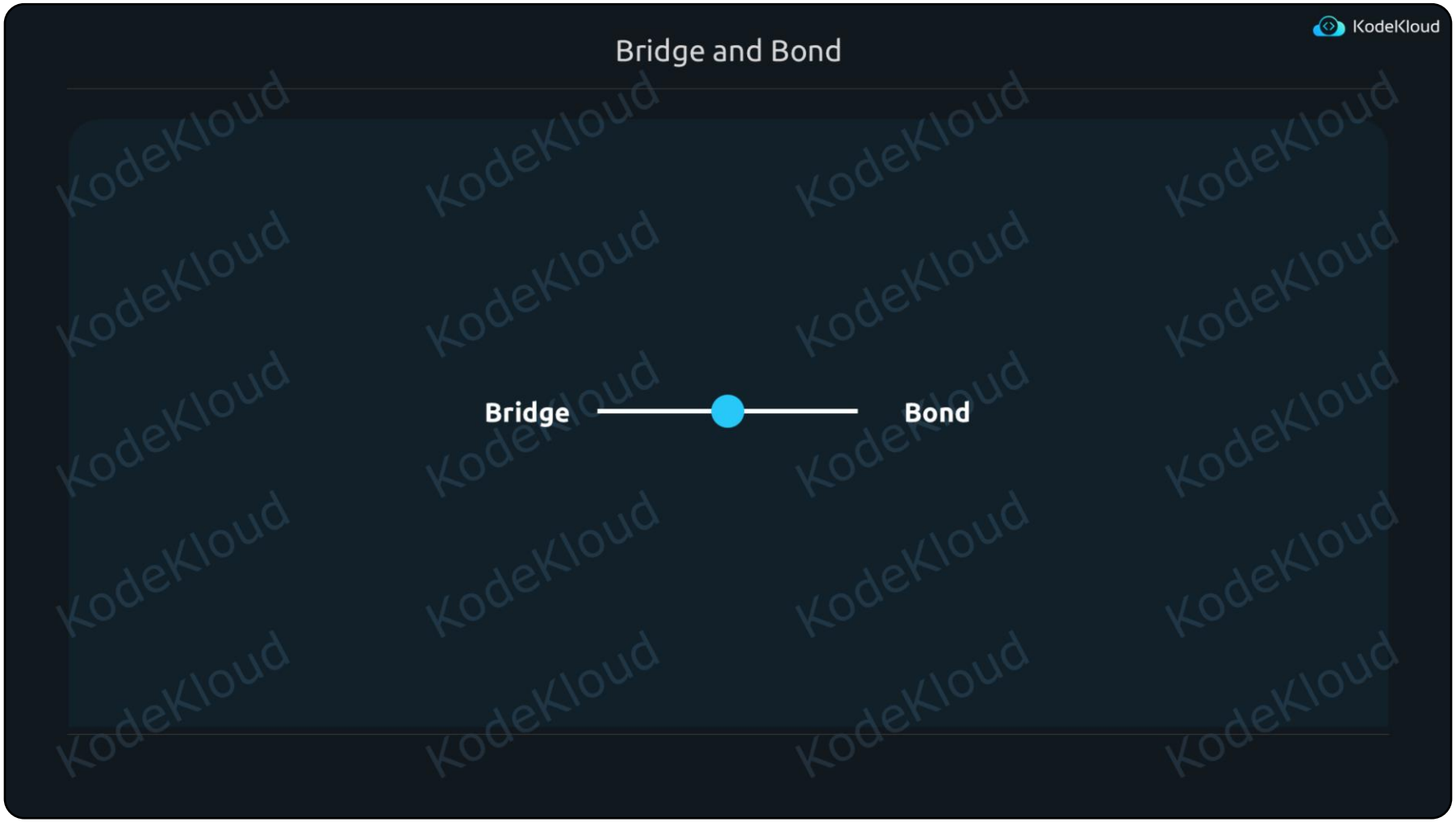
Visit www.kodekloud.com to discover more.



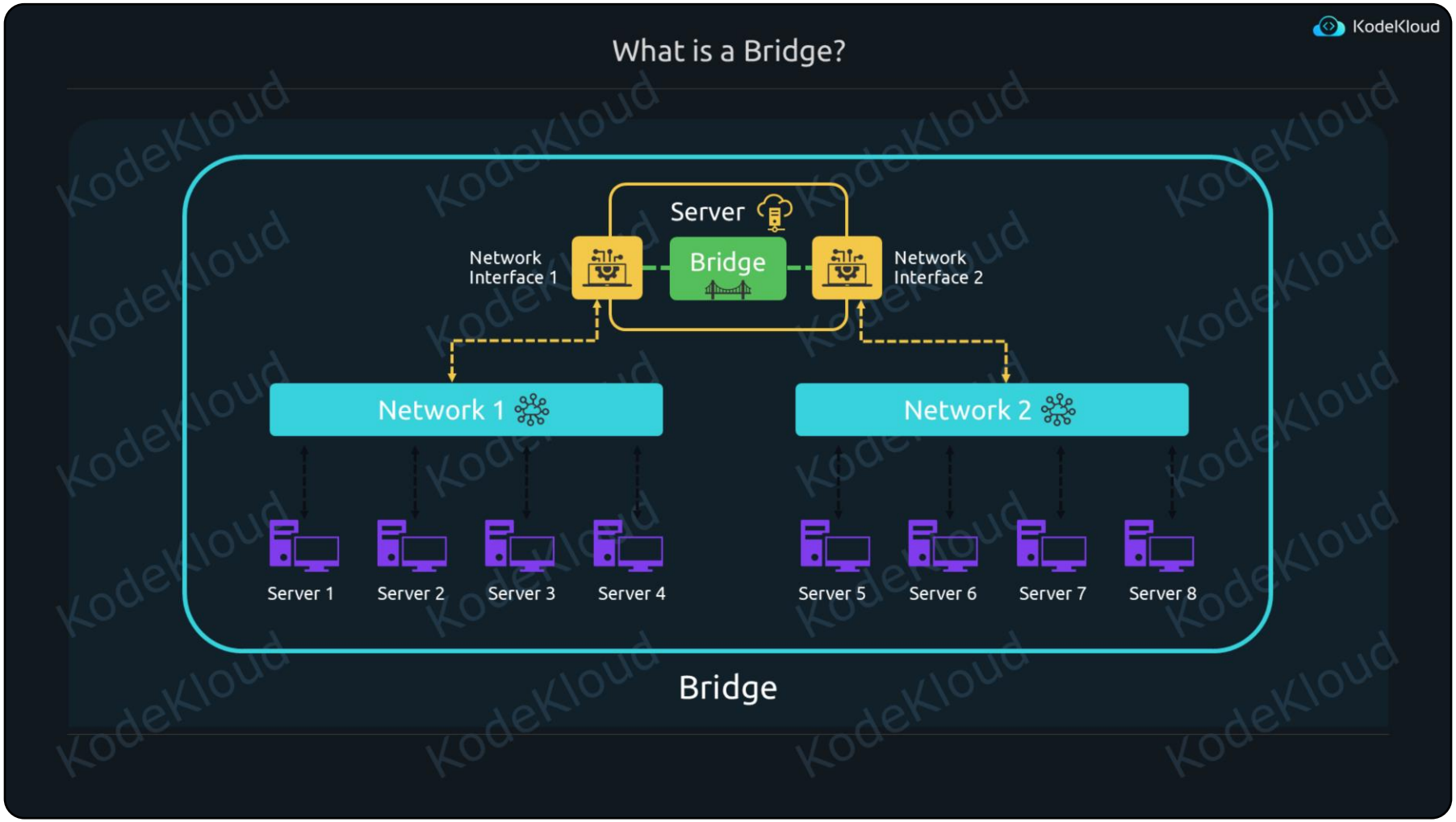
In this lesson we will learn about two concepts:
How to bridge network devices.
And how to bond network devices.



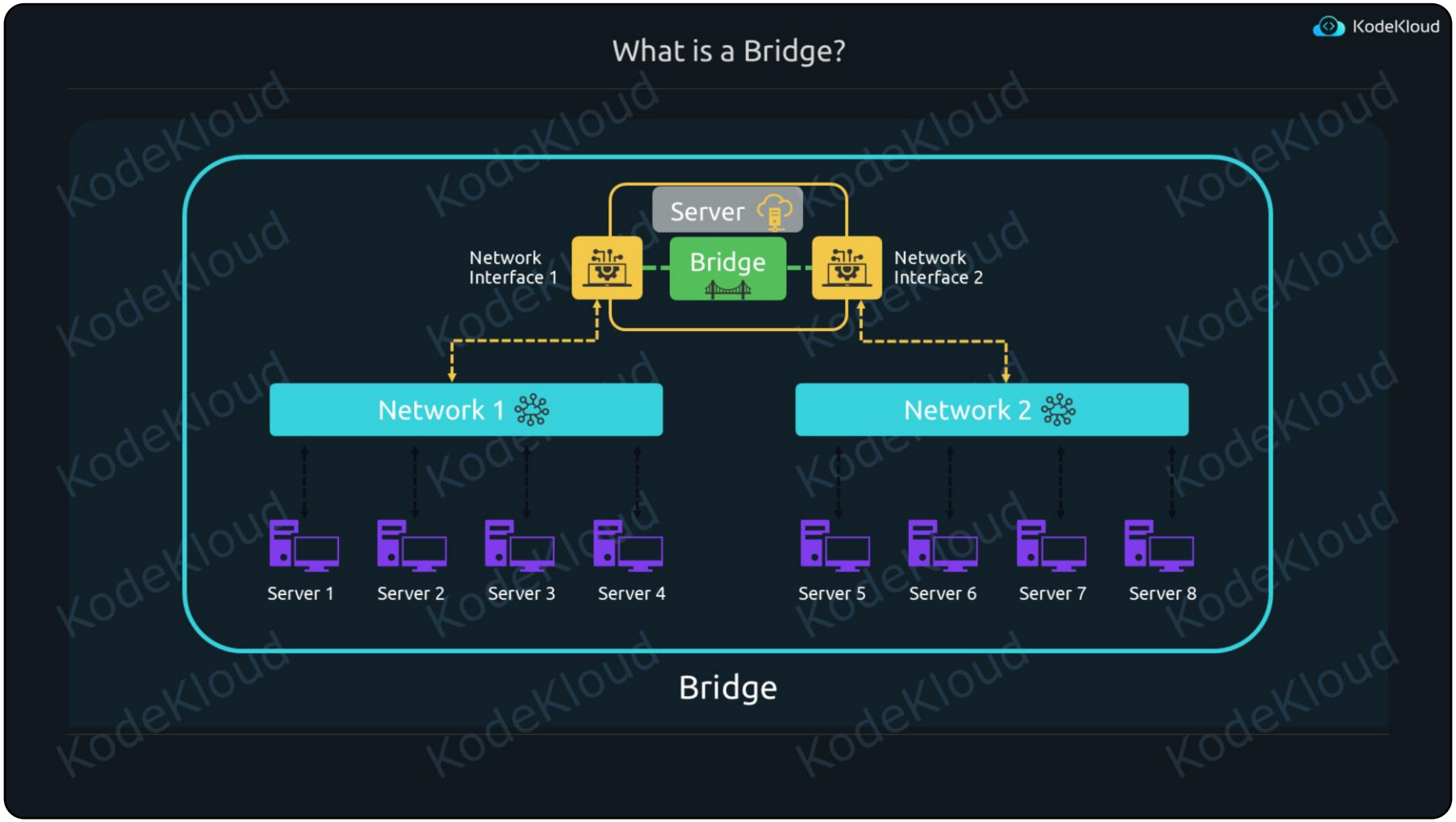
What does it mean to bridge or bond? It means that we can take two, or more network devices, and glue them together under the operating system. And this creates a virtual network device. In one case it's called a bridge, in the other case it's called a bond.



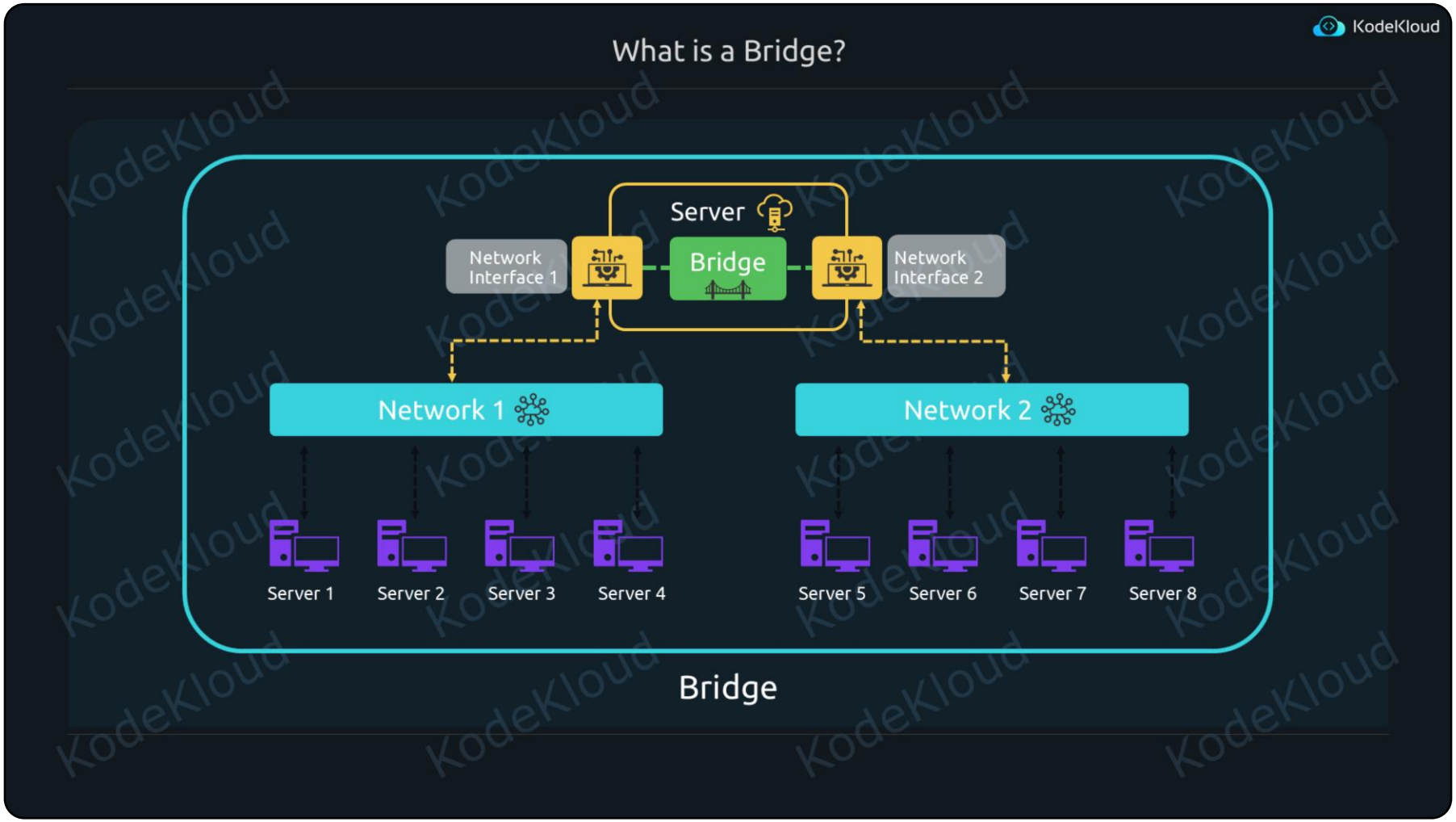
So if both bridging and bonding glue together multiple network devices into a single virtual one, how are these methods different? The clues are in the words themselves: bridge, and bond.



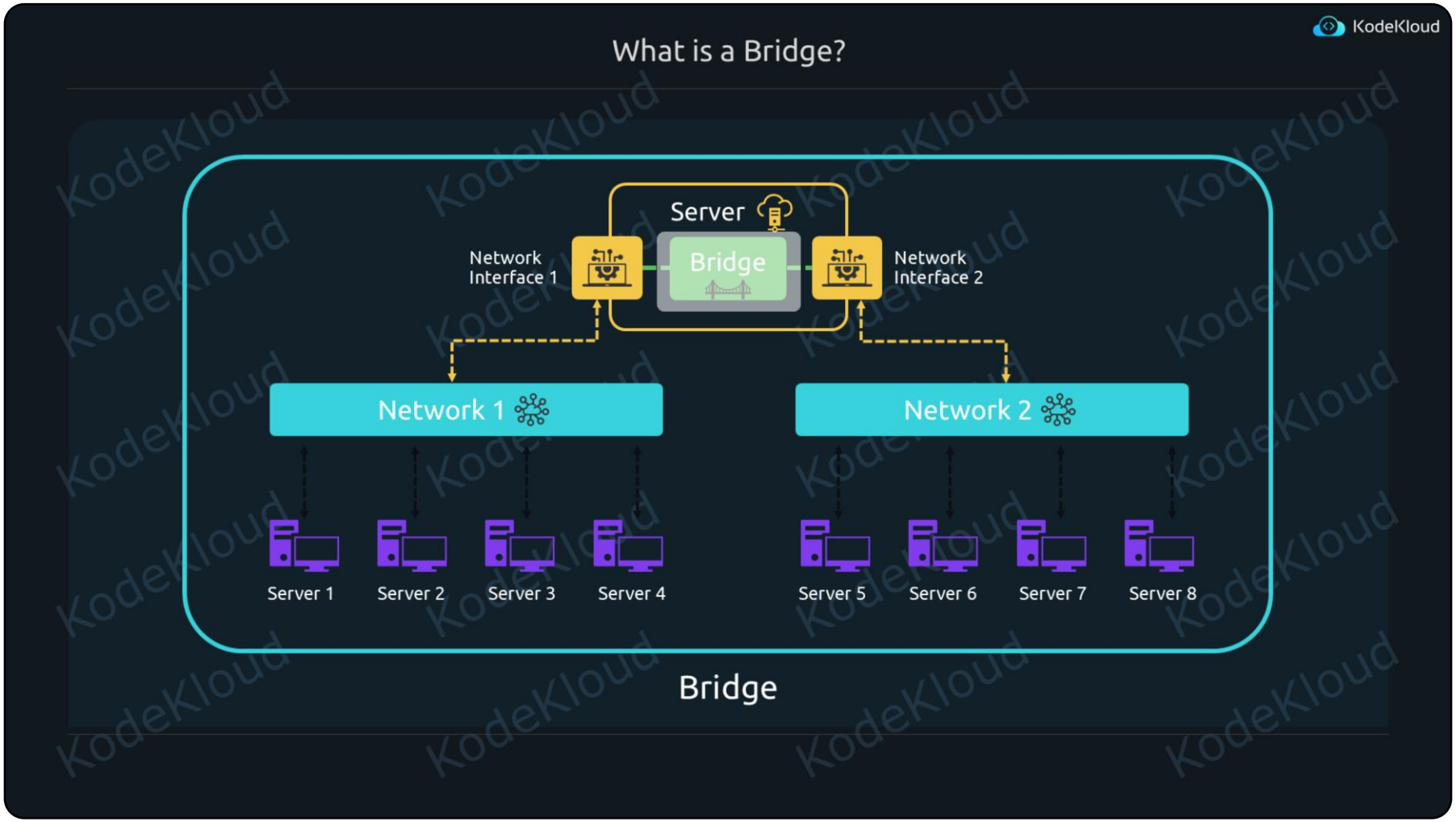
When we bridge two network devices, we are actually doing exactly what the word implies. That is, we build a bridge between Network 1 and Network 2. Such a bridge allows computers on two separate networks to talk to each other as if they are part of the same network.



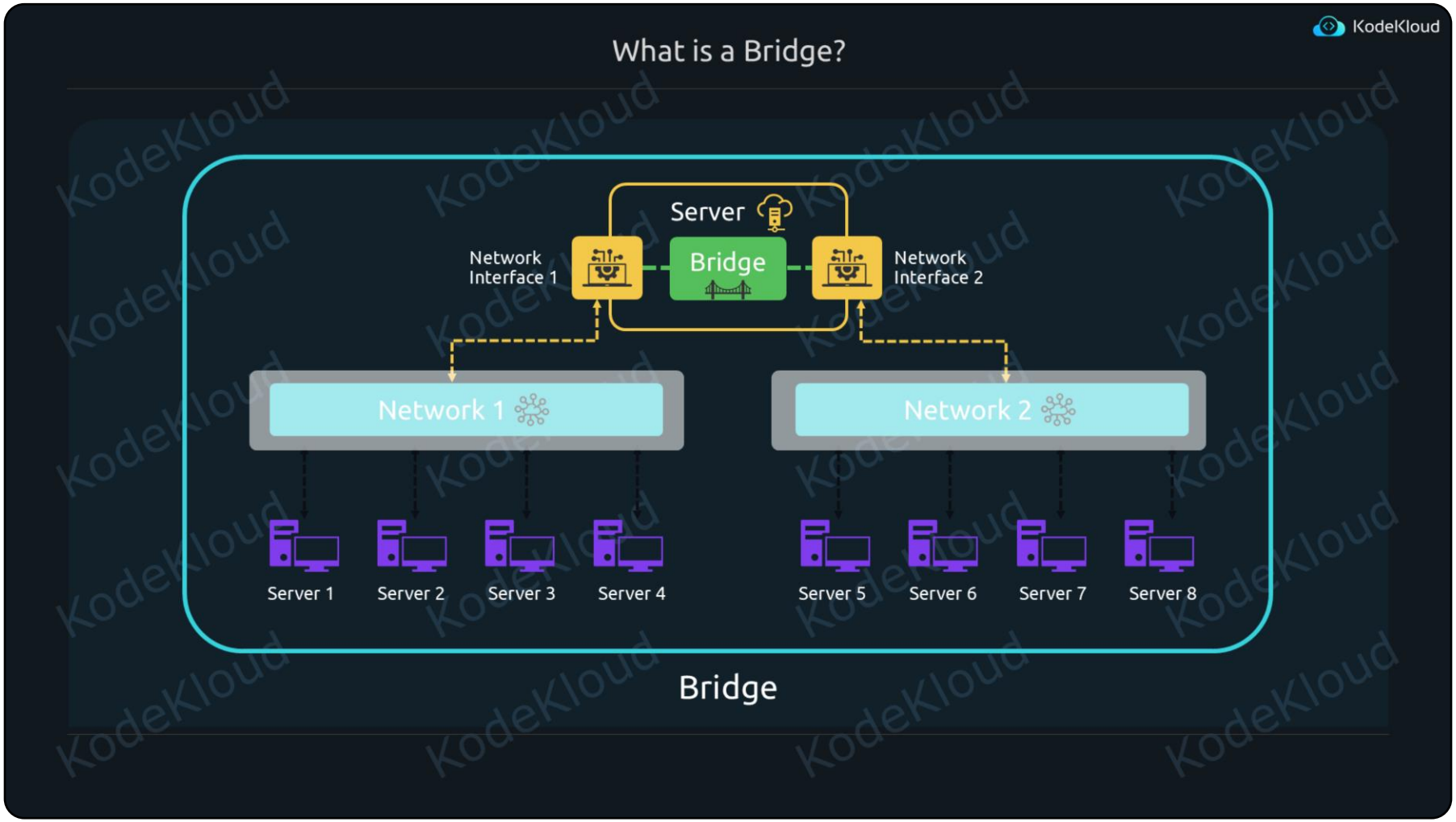
Here, we have an imaginary server at the top.



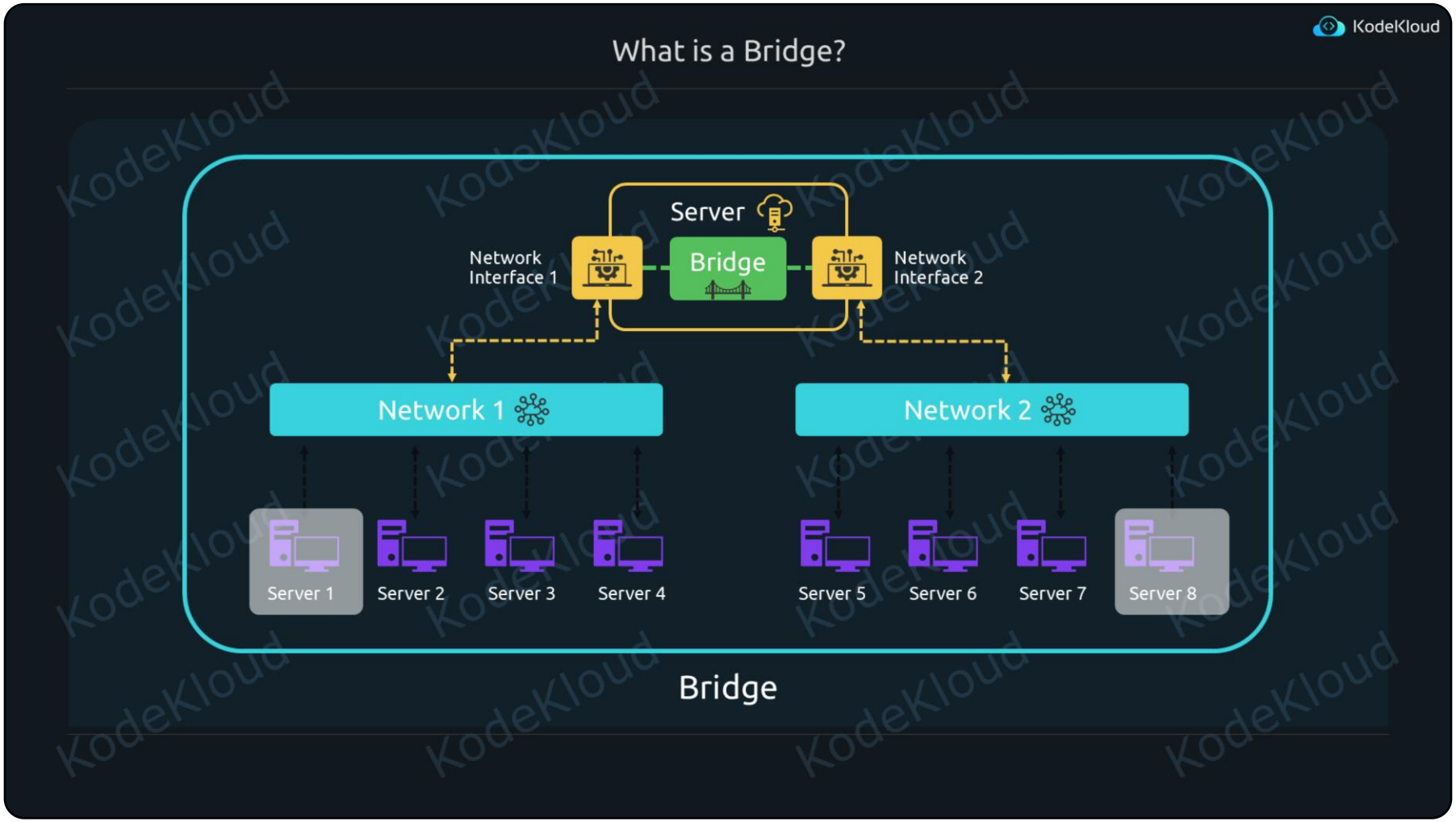
And this server has two network cards, where we insert our networking cables. Any network device is called a "network interface" in Linux terms. So Network Interface 1 and Network Interface 2 are simply our network cards.



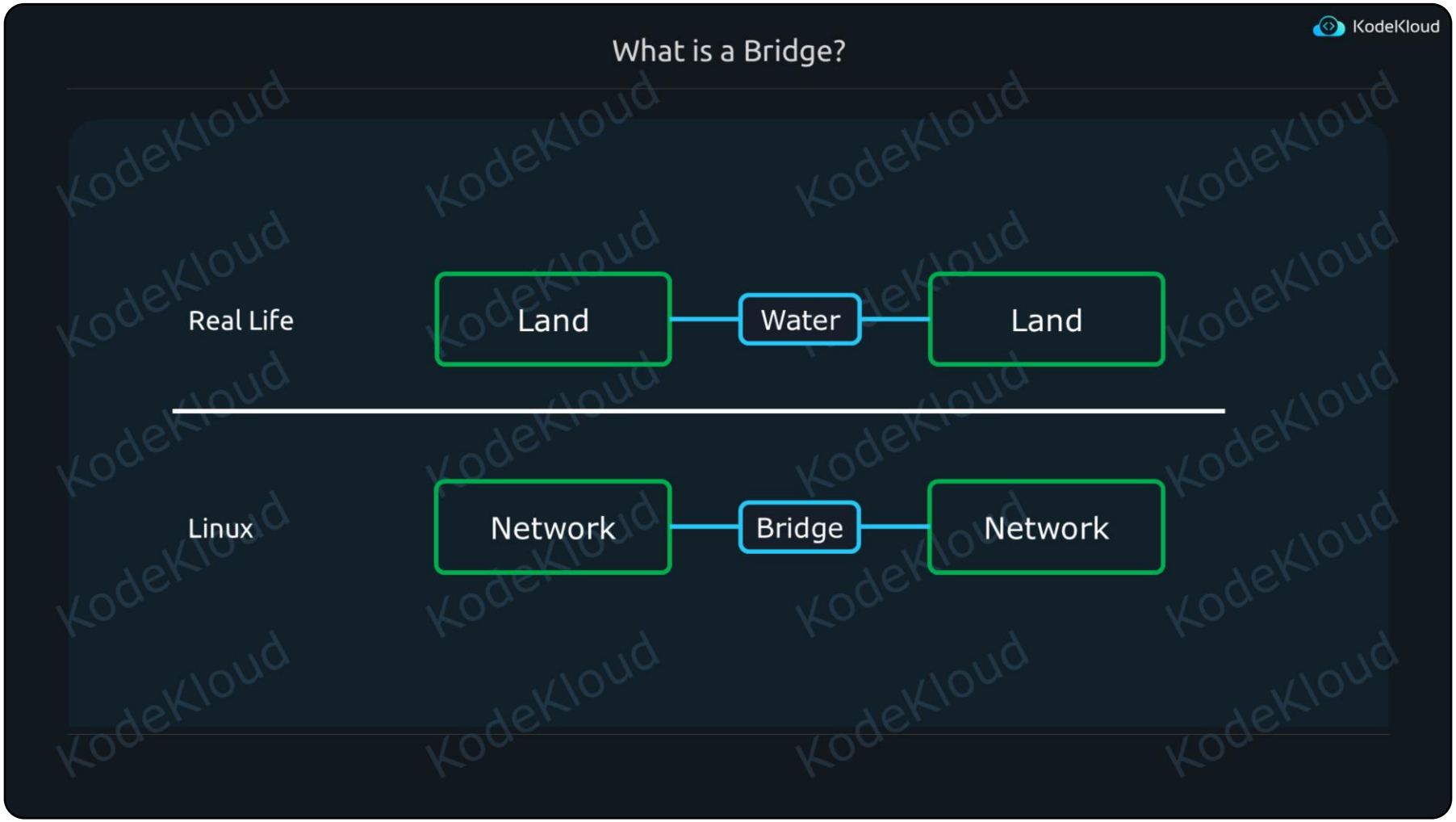
We can add these two network interfaces to a bridge.



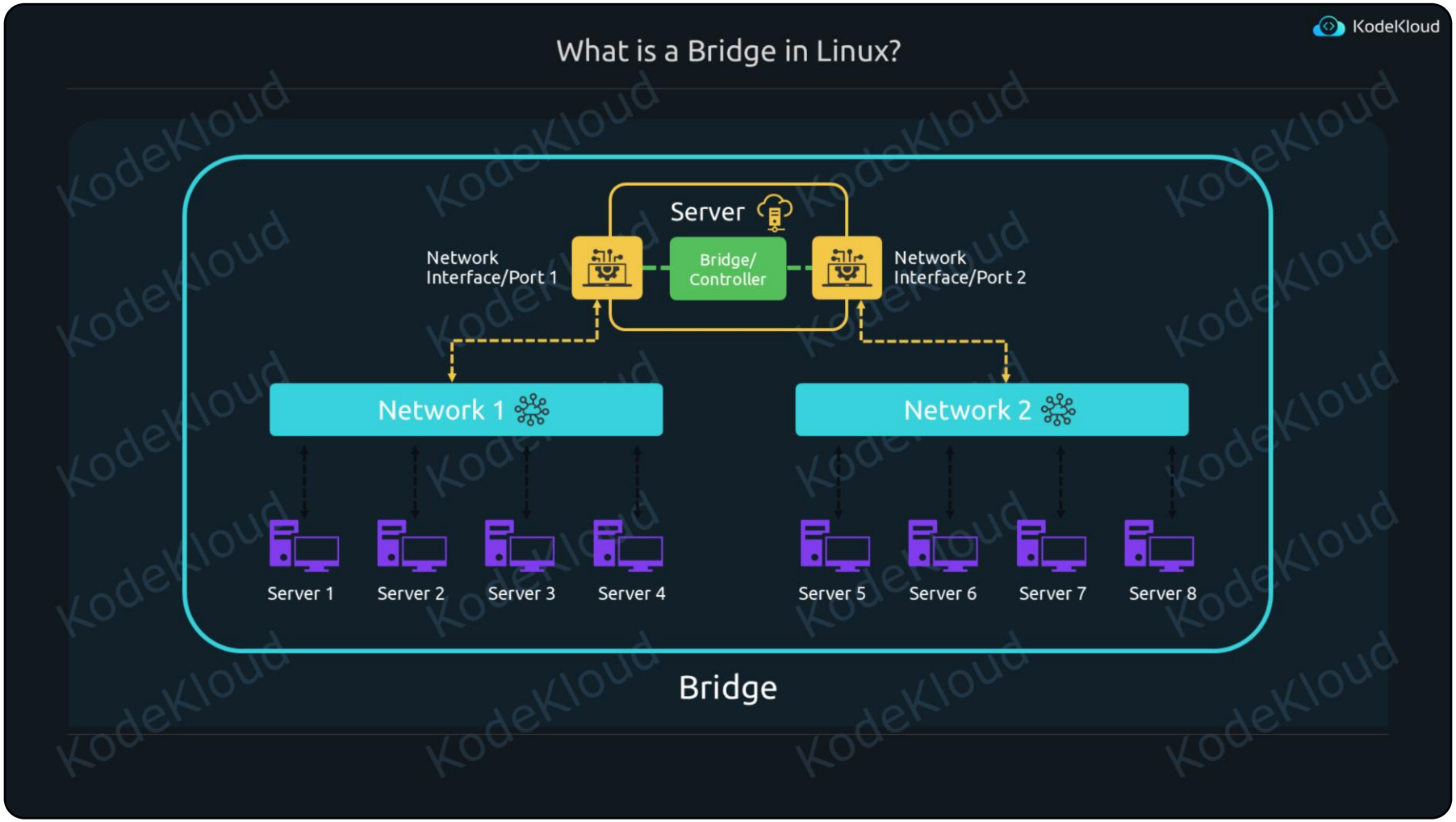
And now, all of the servers below can talk to each other, even though they are on separate networks.



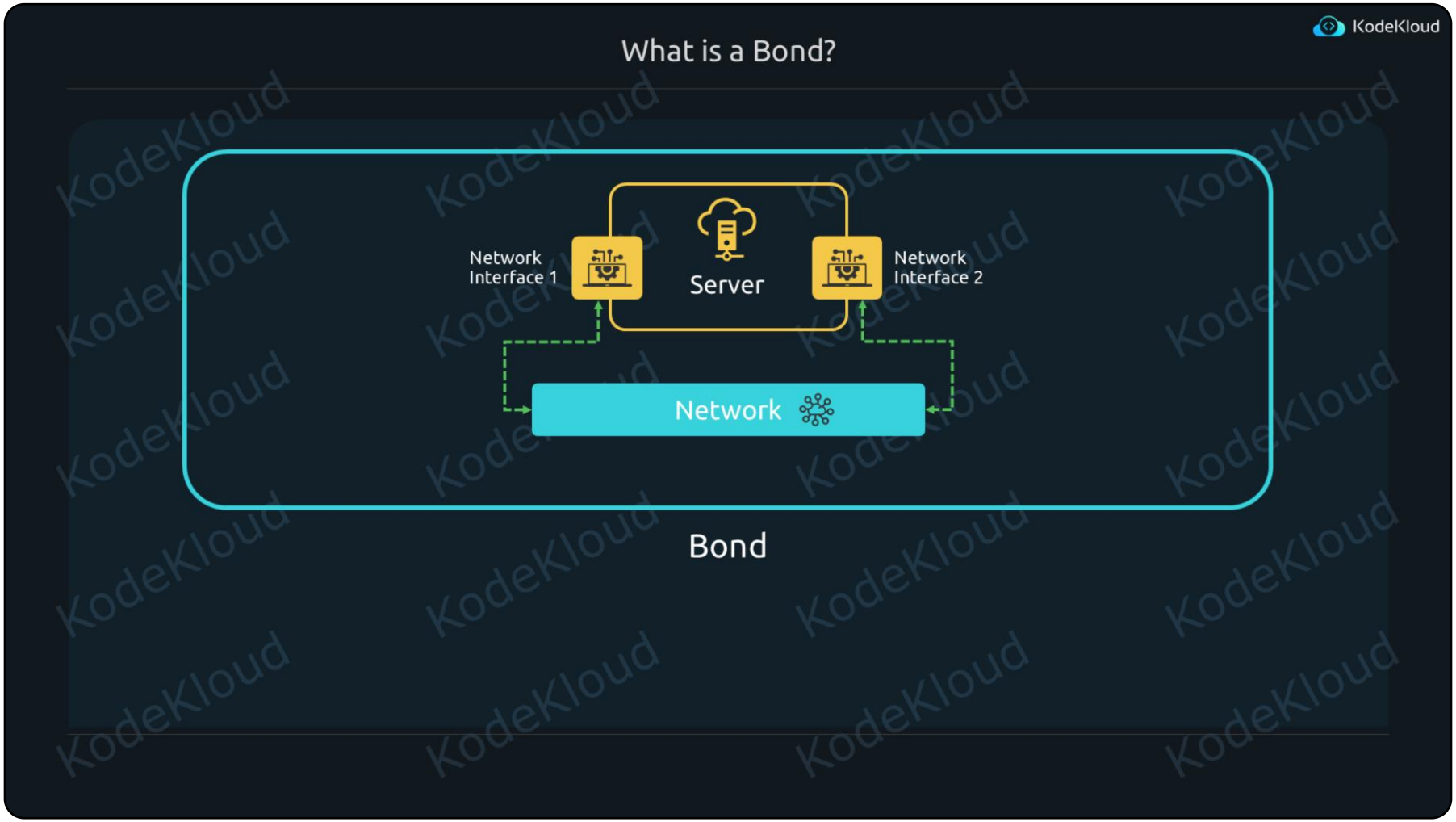
For example, Server 1 would be able to contact Server 8, through our bridge.



Just like bridges in real life connect two pieces of land separated by water, so does a bridge in Linux connect two networks. In fact, even more than two networks can be bridged, if needed.

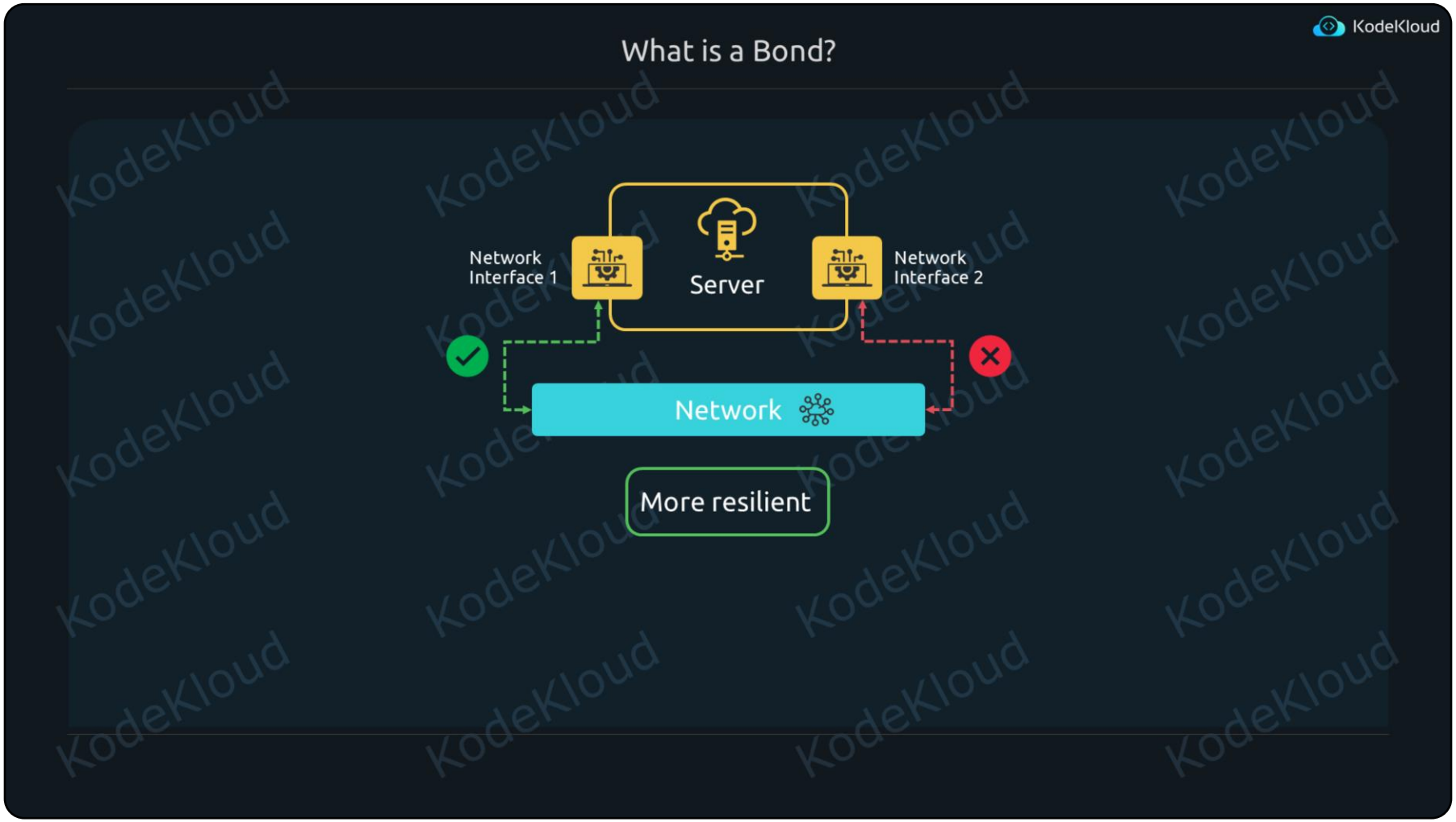


In Linux networking terms the bridge is also called the "controller" and the network devices that are part of the bridge are called "ports".

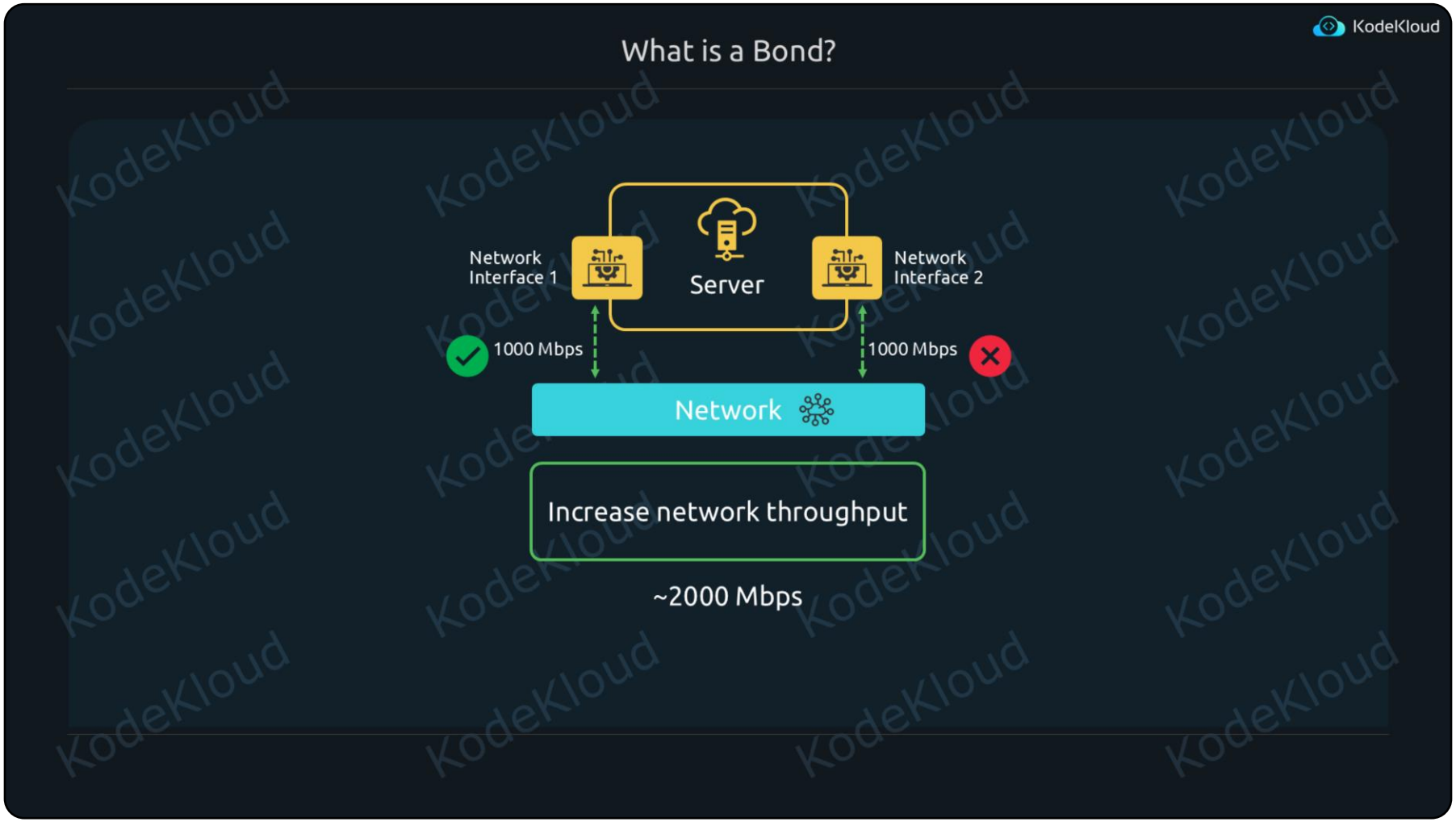


Now let's discuss our next topic: bonding.

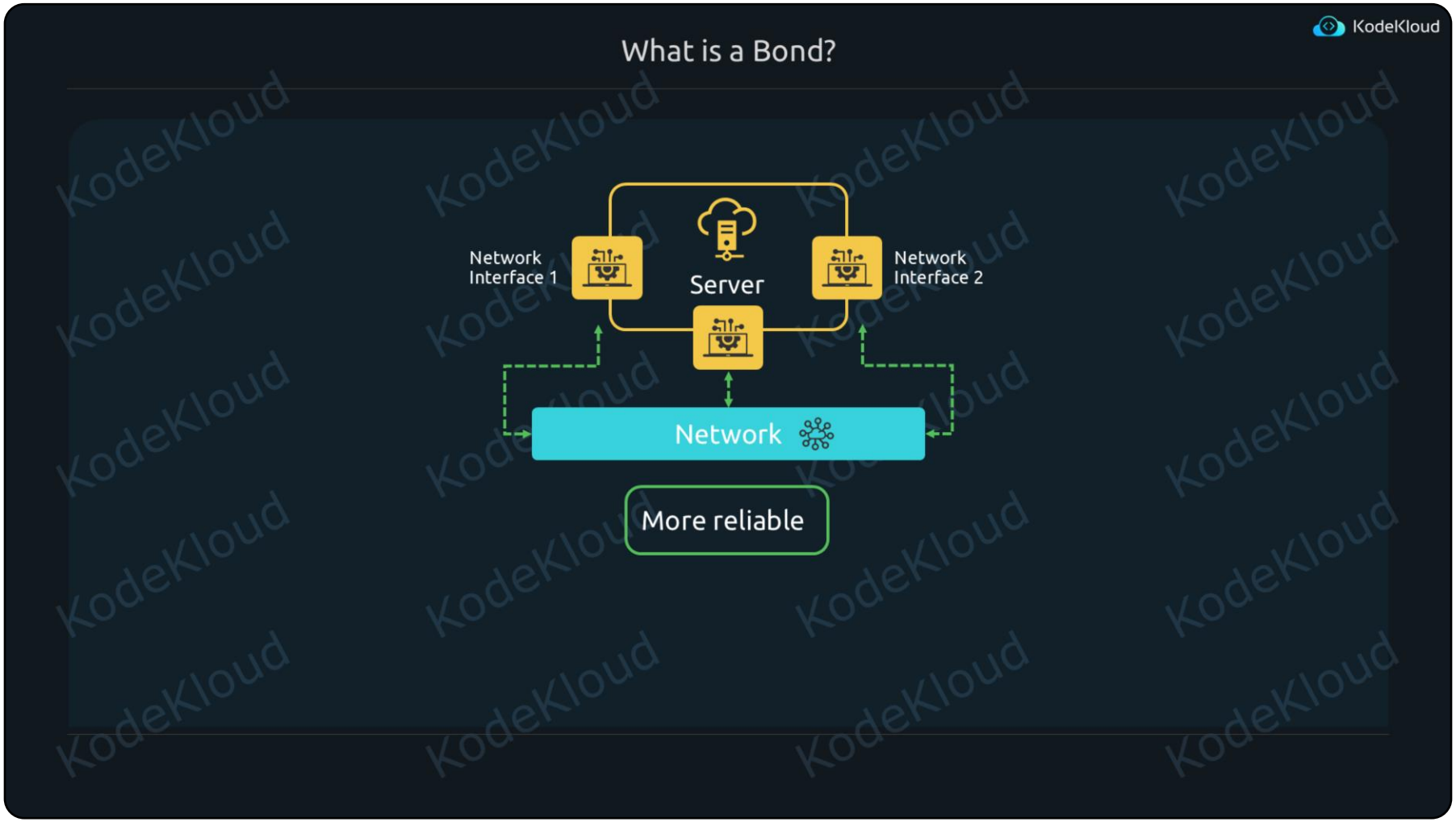
We can take two, or more network devices, and bond them together. Depending on the bond type, this can help in different ways.



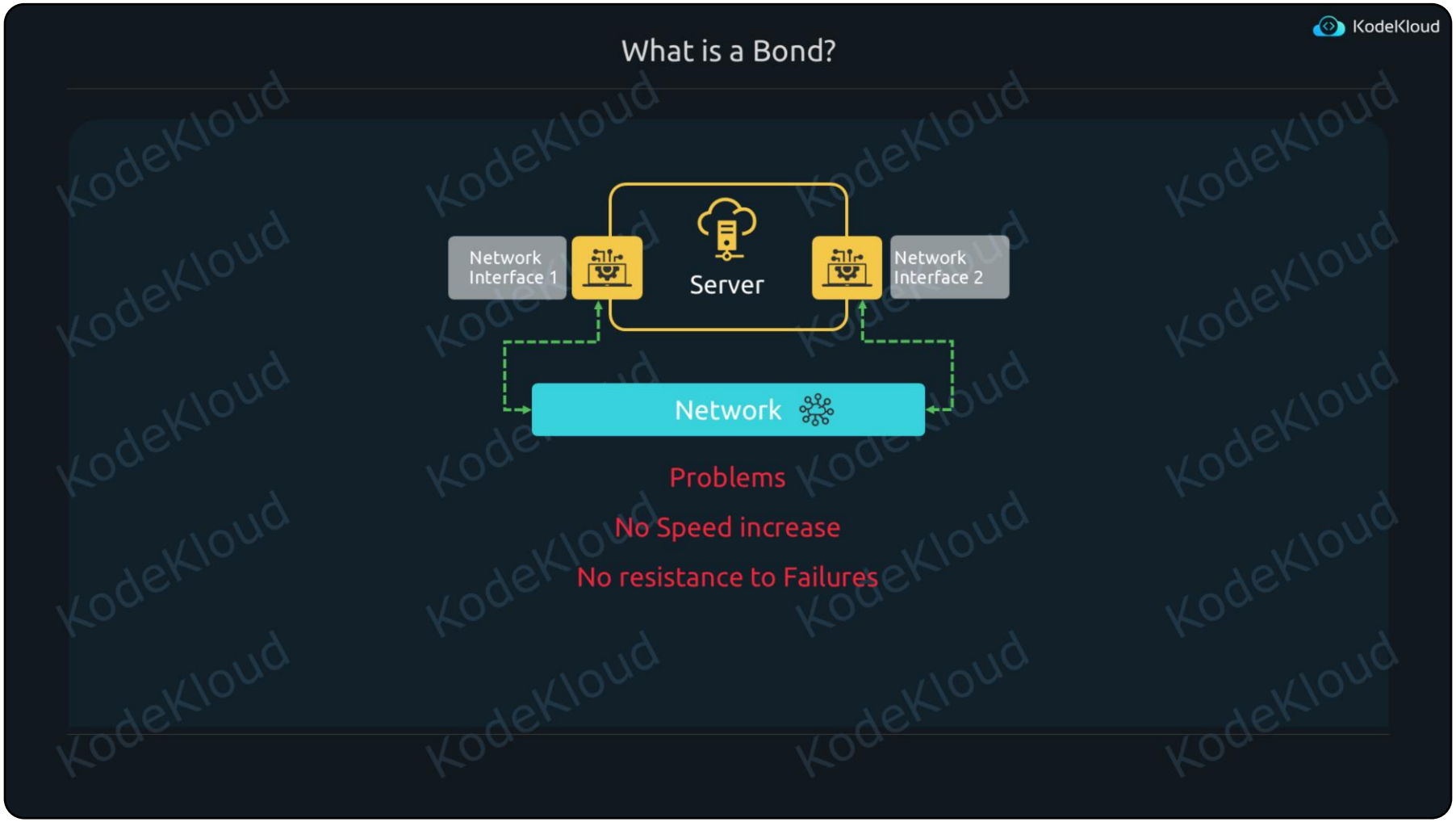
Here are just three examples:
One, it can make the connection to some network more resilient, able to keep working even if a network card goes down or gets disconnected.



Two, it can increase network throughput. For example, if one network card supports a 1000Mbps connection, two such cards can be combined to reach much higher speeds. It wouldn't get to 2000Mbps exactly, but it will be much higher than a single card supports.



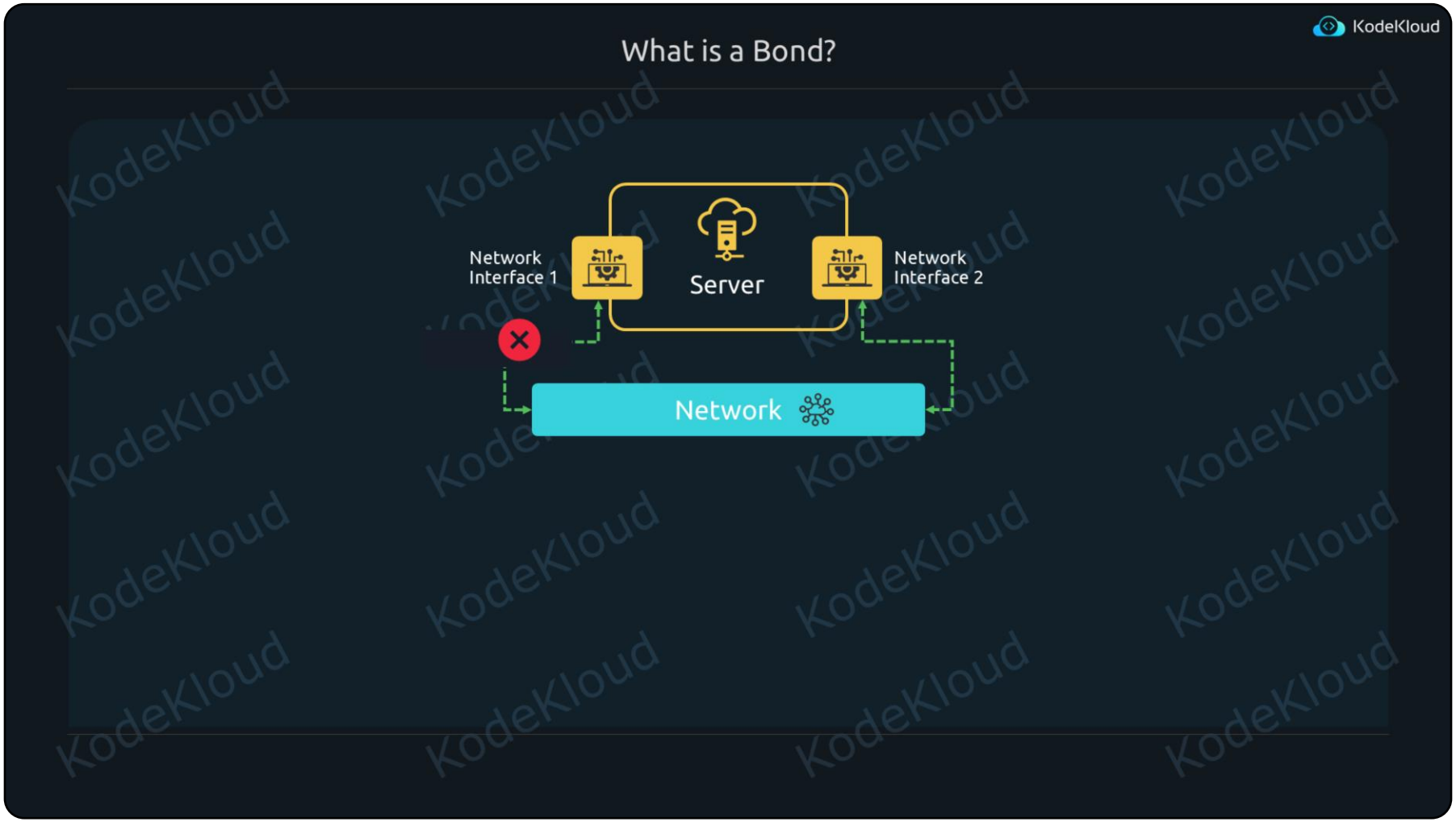
It can make a connection more reliable. For example, a single network device might experience occasional slow downs. But if we bond three network devices, we might be able to eliminate those slow downs and increase stability.



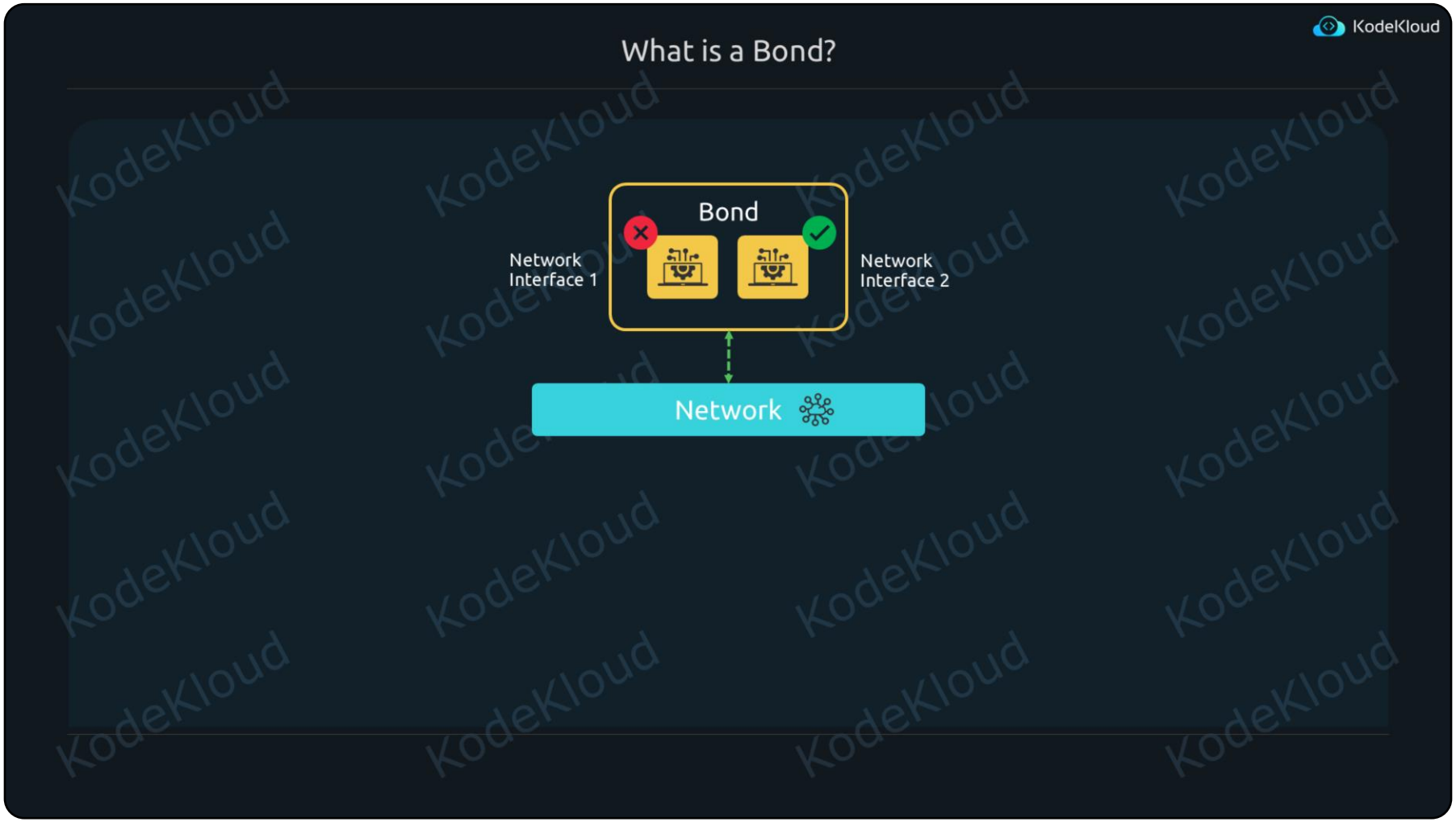
Let's see how bonding works. First, imagine this setup:

We have a server with two network cards (called interfaces under Linux terminology). There's a wire connecting the first network card to some network in the office. And we also have a wire connecting the second network card to the same network in the office. The problem with this setup is that we'd normally define two separate connections to this network under Linux. Even though both connections go to the same place. And this wouldn't provide any of the benefits

mentioned earlier. No speed increase, and no resistance to failures.



A program could use either the first connection, or the second connection to communicate with the network on the bottom. It would have to pick just one network path that it wants to use. And let's say it picks the first network card, and goes through the connection on the left. If the wire on the left is cut, then the program suddenly gets disconnected. And it fails the data transfer. Now let's see how bonding can prevent this.



If we bond these network interfaces, from a hardware perspective we still have the same setup. Two network cards are connected to some network, with the use of two separate cables.

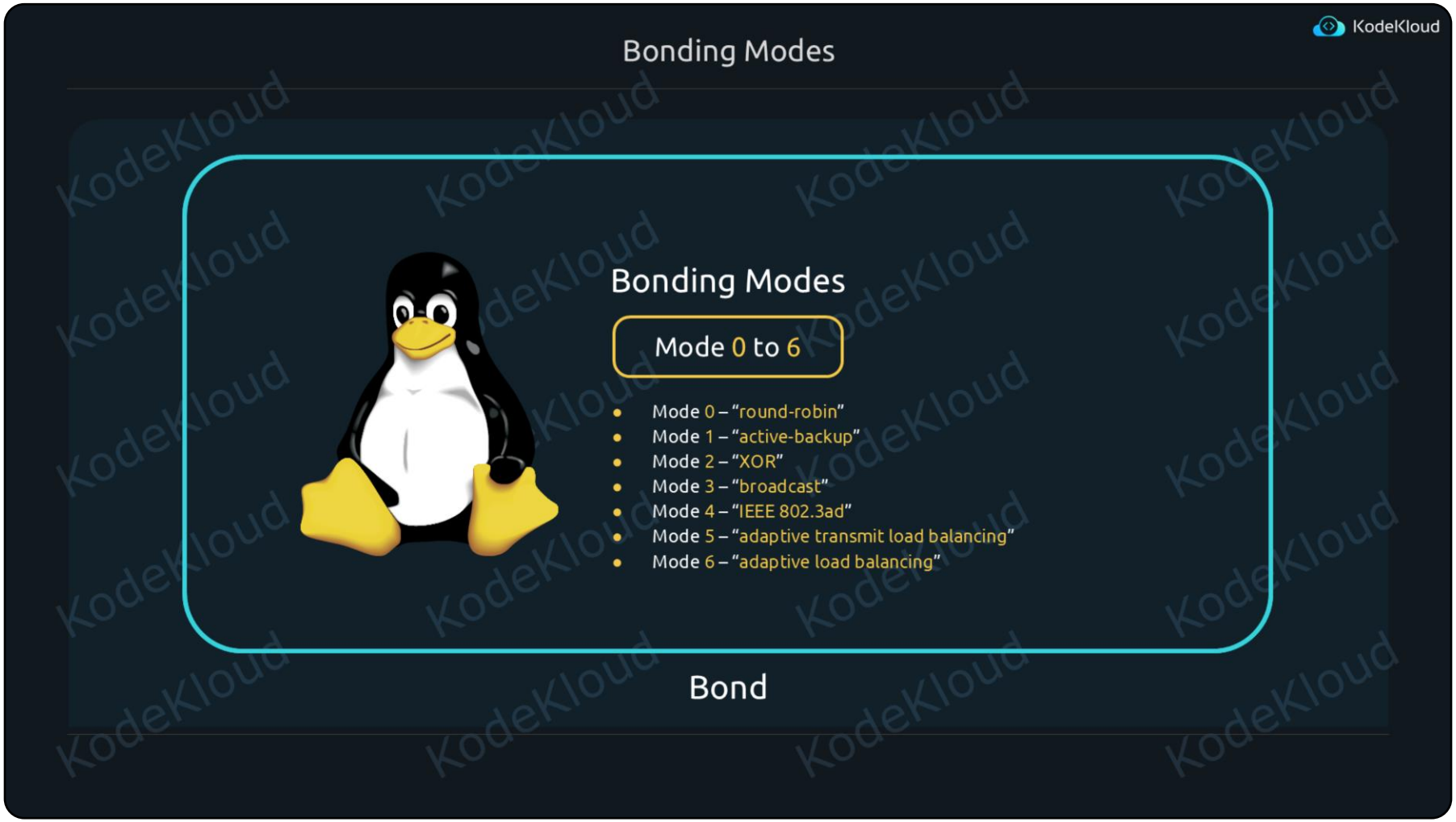
But from a software perspective, things now look like this:

To any program that wants to reach the network below, it looks like there's a single network interface: the bond we just

created. As far as the program is concerned, there's also a single wire connecting it to the network it wants to reach.

Now the negative scenario described earlier is eliminated. Let's say some application is exchanging data with the network below. Suddenly, the wire from Network Interface 1 is cut. Well, now there's no issue. Linux can simply switch over to using Network Interface 2, which still has a functional wire. As far as the application is concerned, the connection from the bond it's using, to the network below, is still working.

In other words, the operating system pretends there's a single wire between our server and the network below. And that simplifies things for us. We don't have to configure applications to use various network cards and various connections and IP addresses to reach the same destination through different paths. We just tell our applications to use the bond interface.



Bonding Modes

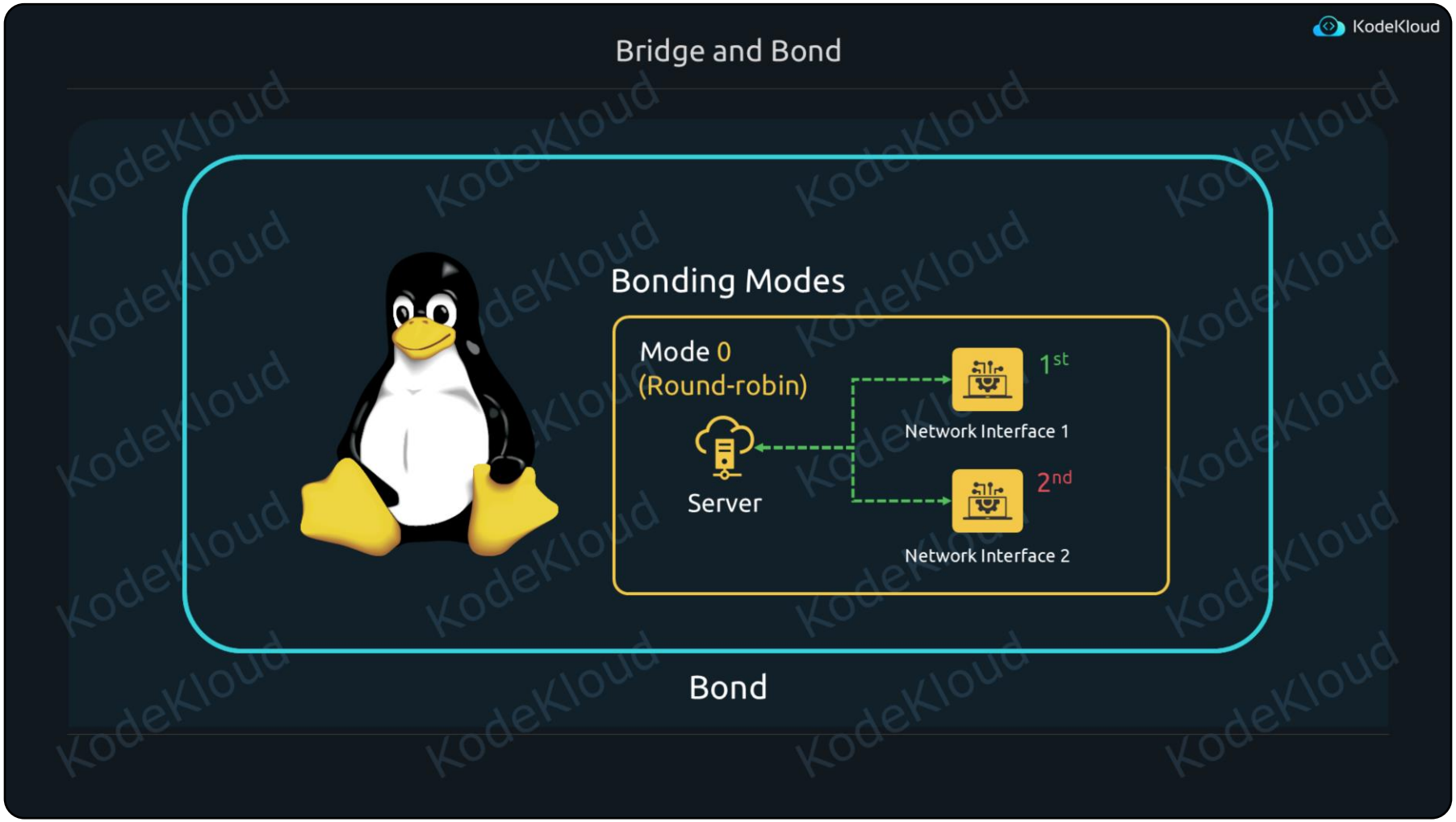
Linux can handle network traffic through these bonds in many ways. To decide how it should handle it, we have to choose what is called a bonding mode. Currently, most Linux operating systems support seven bonding modes. Their codes range from mode 0 to mode 6. If no mode is specified, some operating systems will pick mode 0 by default, also called "round-robin".

Not all modes increase network throughput above what a single card can achieve. One mode that can increase network throughput above what a single card can achieve on its own is mode 4, also called "IEEE 802.3ad".

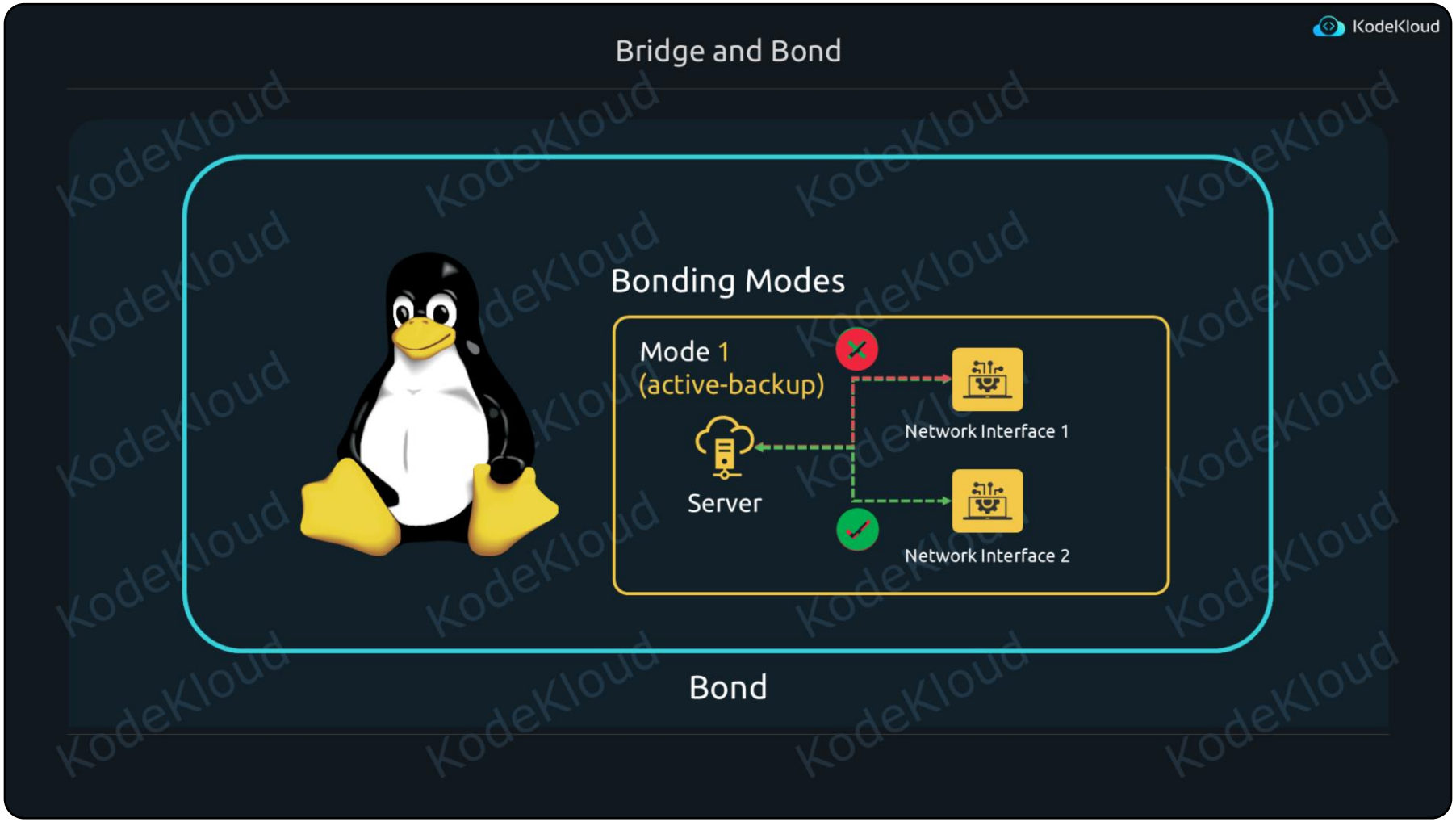
Here's the list of modes and what each number corresponds to:

Mode 0 - round-robin
Mode 1 - active-backup
Mode 2 - XOR
Mode 3 - broadcast
Mode 4 - IEEE 802.3ad
Mode 5 - adaptive transmit load balancing
Mode 6 - adaptive load balancing

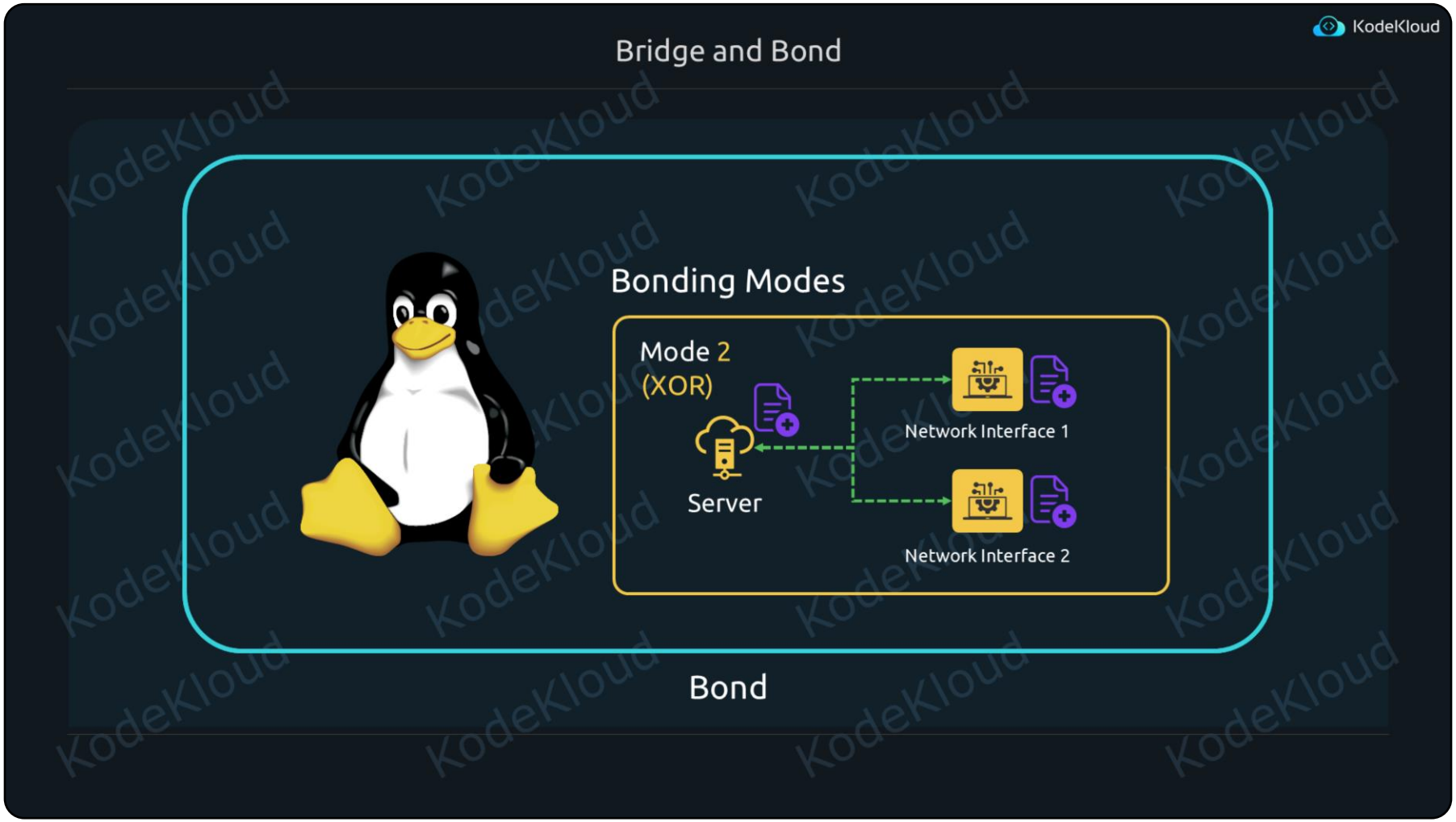
Each has its own set of advantages, disadvantages, and limitations. Making the choice is a complex task that requires thorough knowledge of the entire network in your organization. Diving into details here would take hours to explain. But here's a short summary about the modes at least:



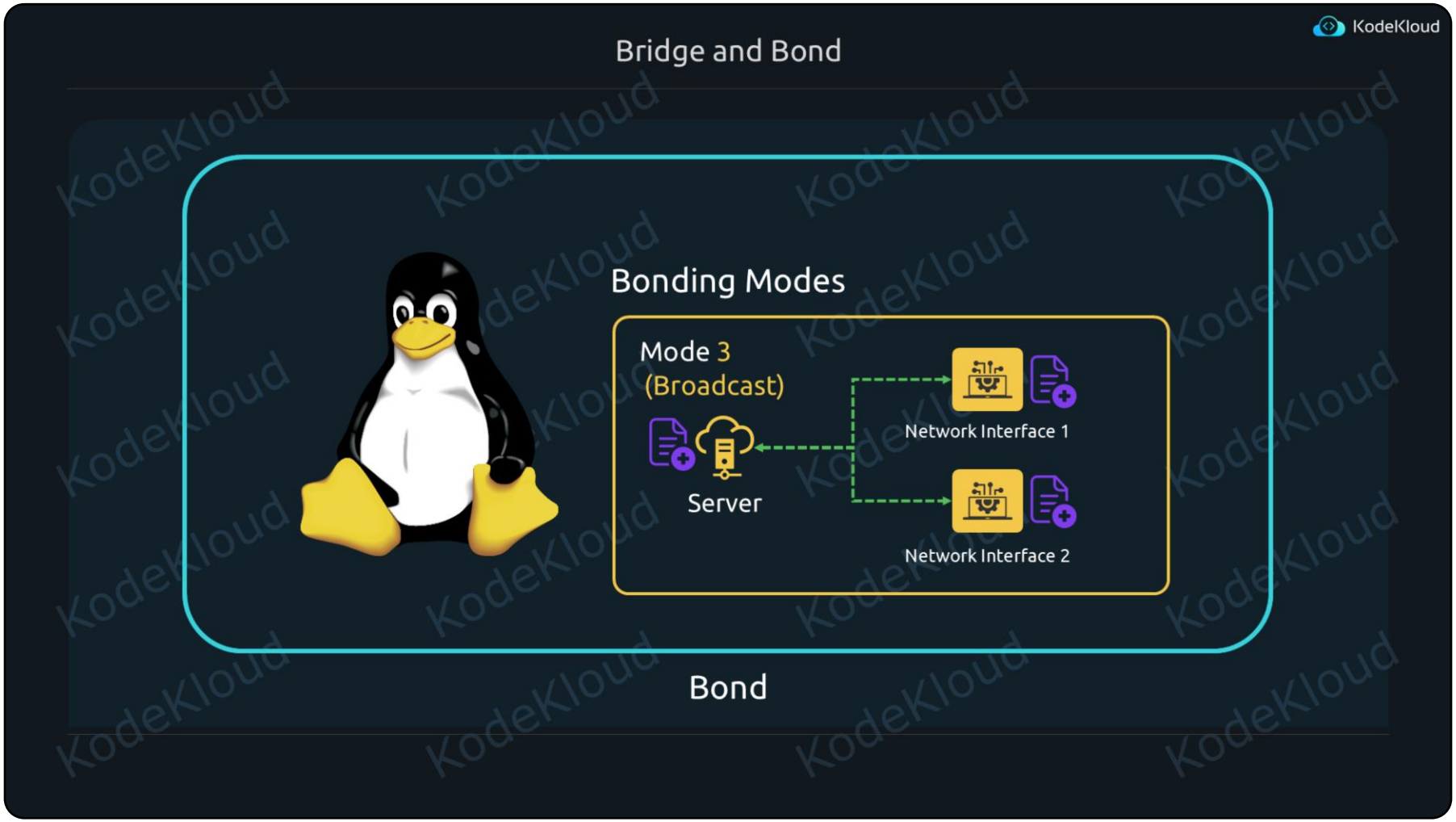
Mode 0 (round-robin) uses network interfaces in sequential order. For a certain amount of data, Network Interface 1 may be used. Then for another amount of data, Network Interface 2, and so on.



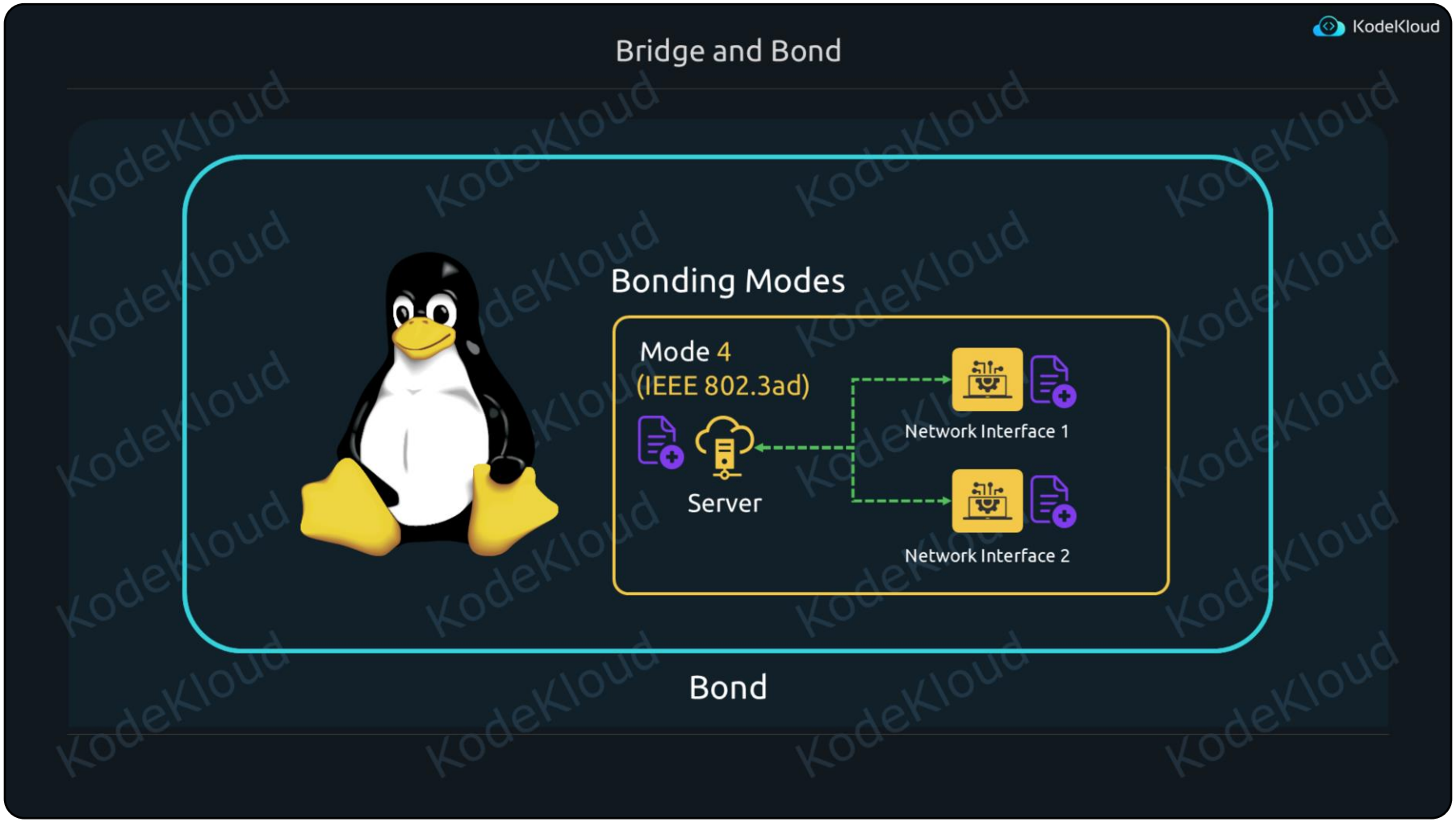
Mode 1 (active-backup) uses a single interface, and keeps all others inactive, as backups. If the active interface goes down, then a backup becomes the new active one.



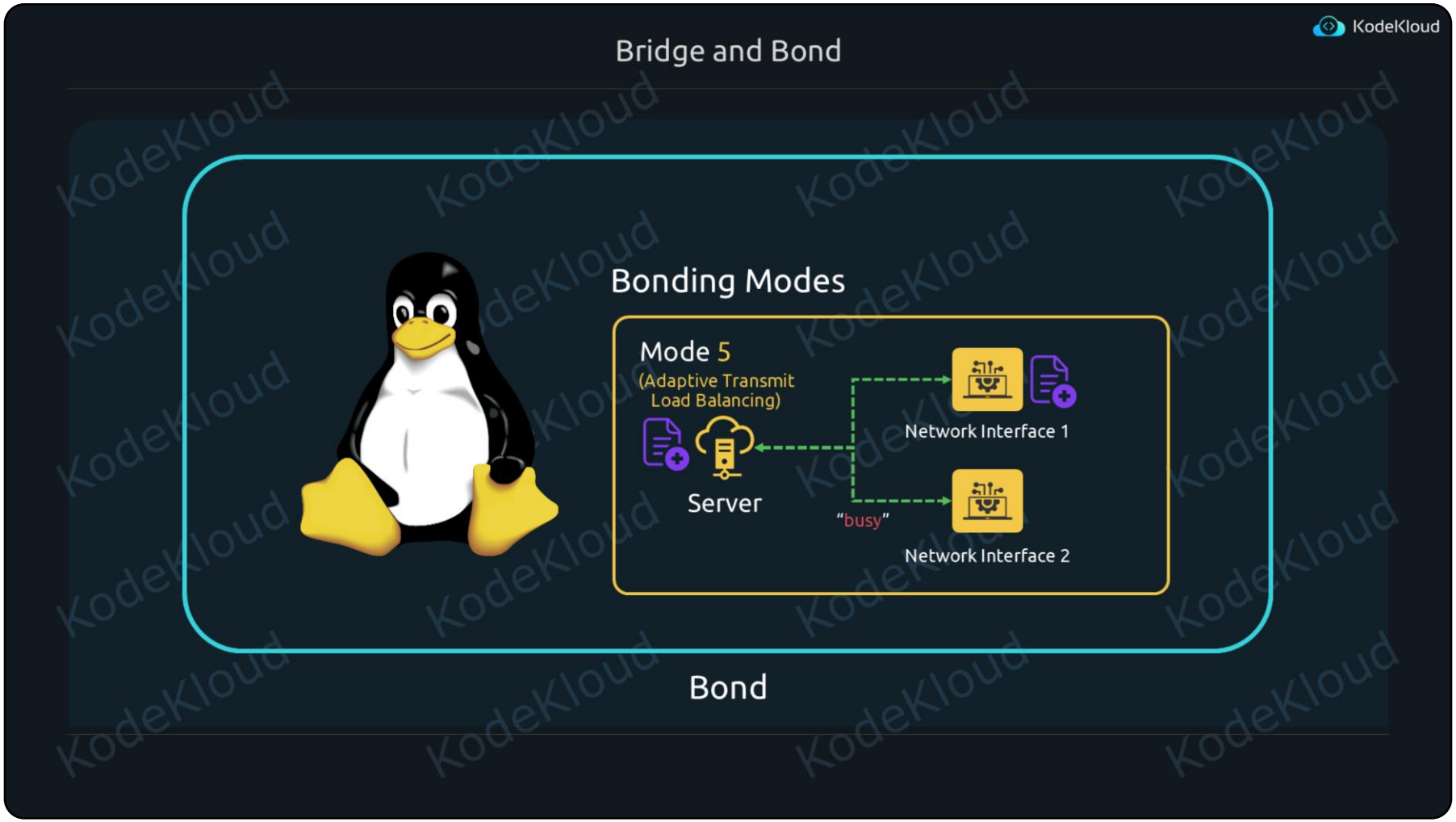
Mode 2 (XOR) picks the interface to use based on the source and destination of a data packet. This ensures that if Device 1 wants to reach Device 2 through this bond, it will always go through the same interface. And if Device 1 wants to reach Device 3, it might go through a different interface, but it will be the same one for every transfer.



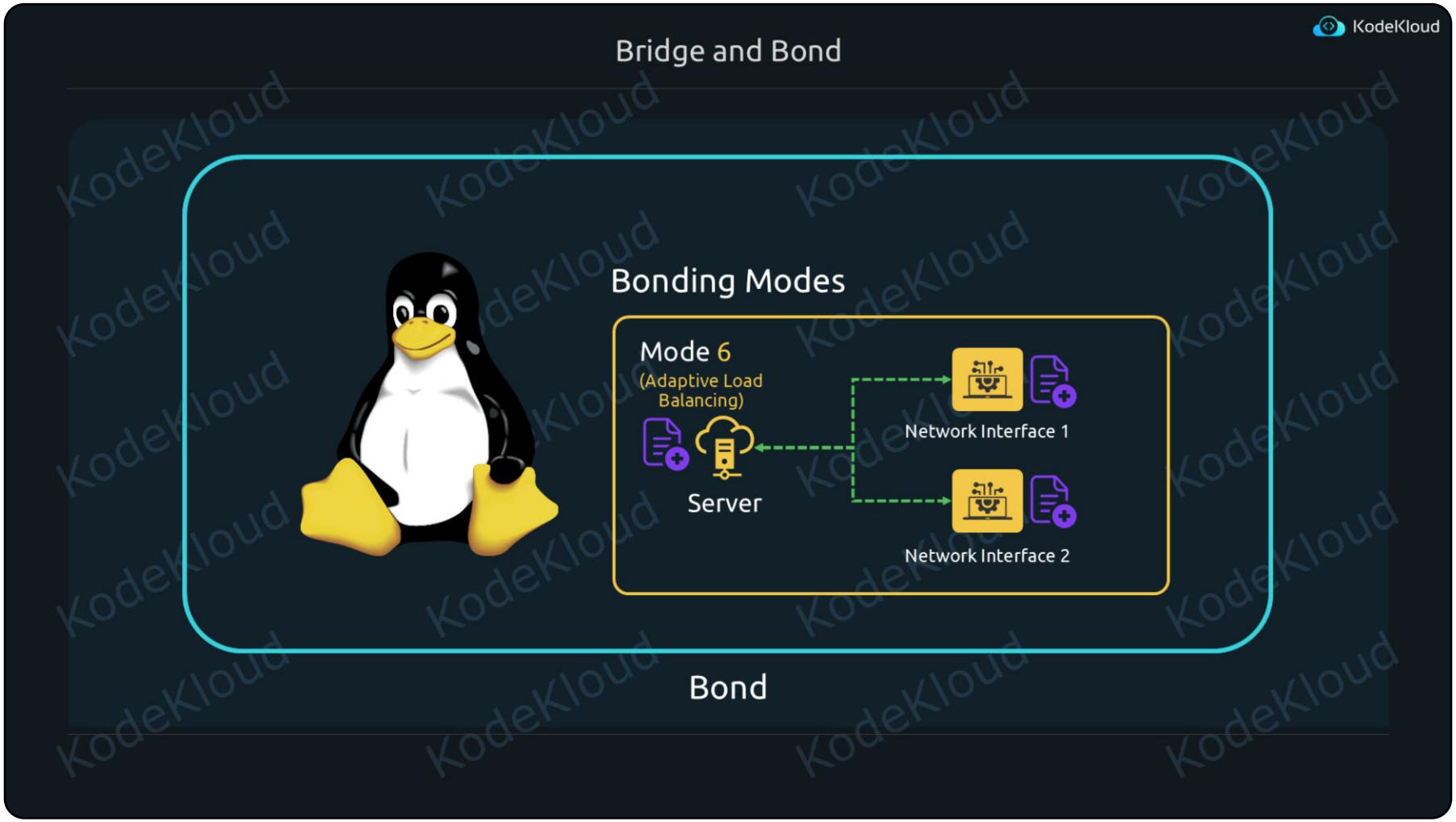
Mode 3 (broadcast) means all data is sent through all interfaces at once.



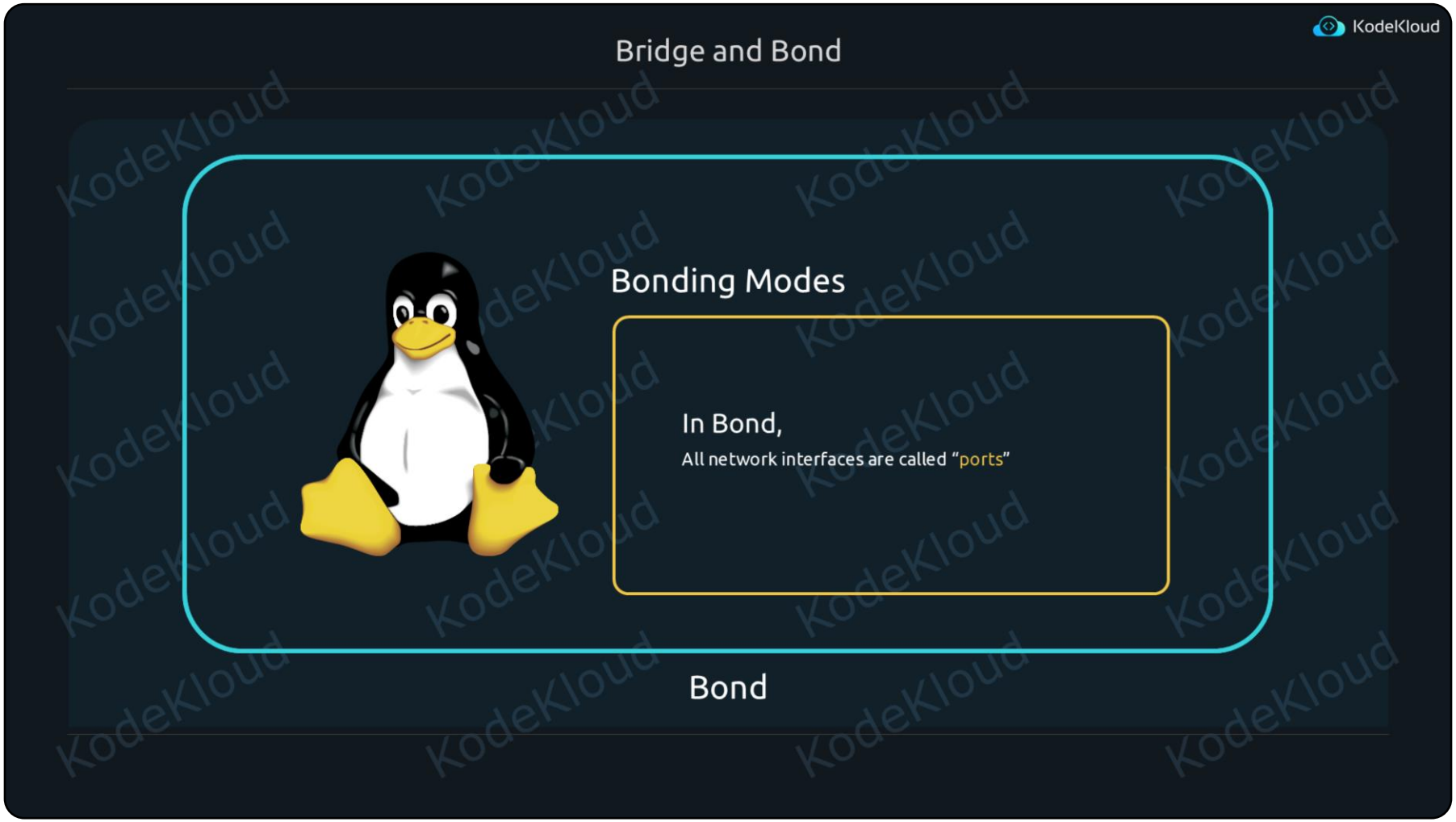
Mode 4 (IEEE 802.3ad) can help increase network transfer rates, above what a single interface supports.



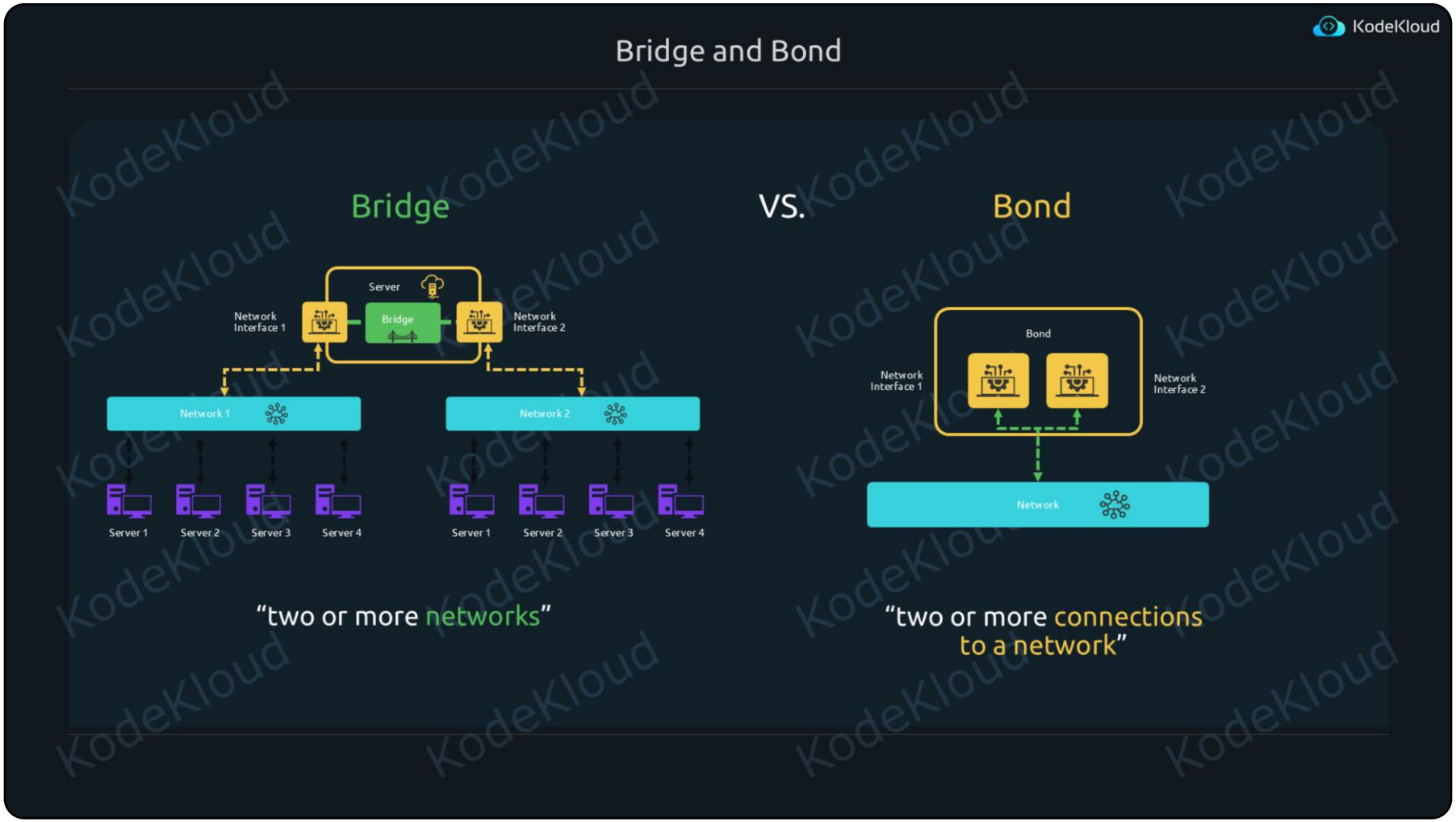
Mode 5 (adaptive transmit load balancing) load balances outgoing traffic from the bond. That is, it will try to send data from the interface that is the least busy.



Mode 6 (adaptive load balancing) tries to load balance both outgoing AND incoming traffic.



In the context of a bond, all network interfaces that are added to it are called "ports".



If we oversimplify, we can now differentiate between bridging and bonding:

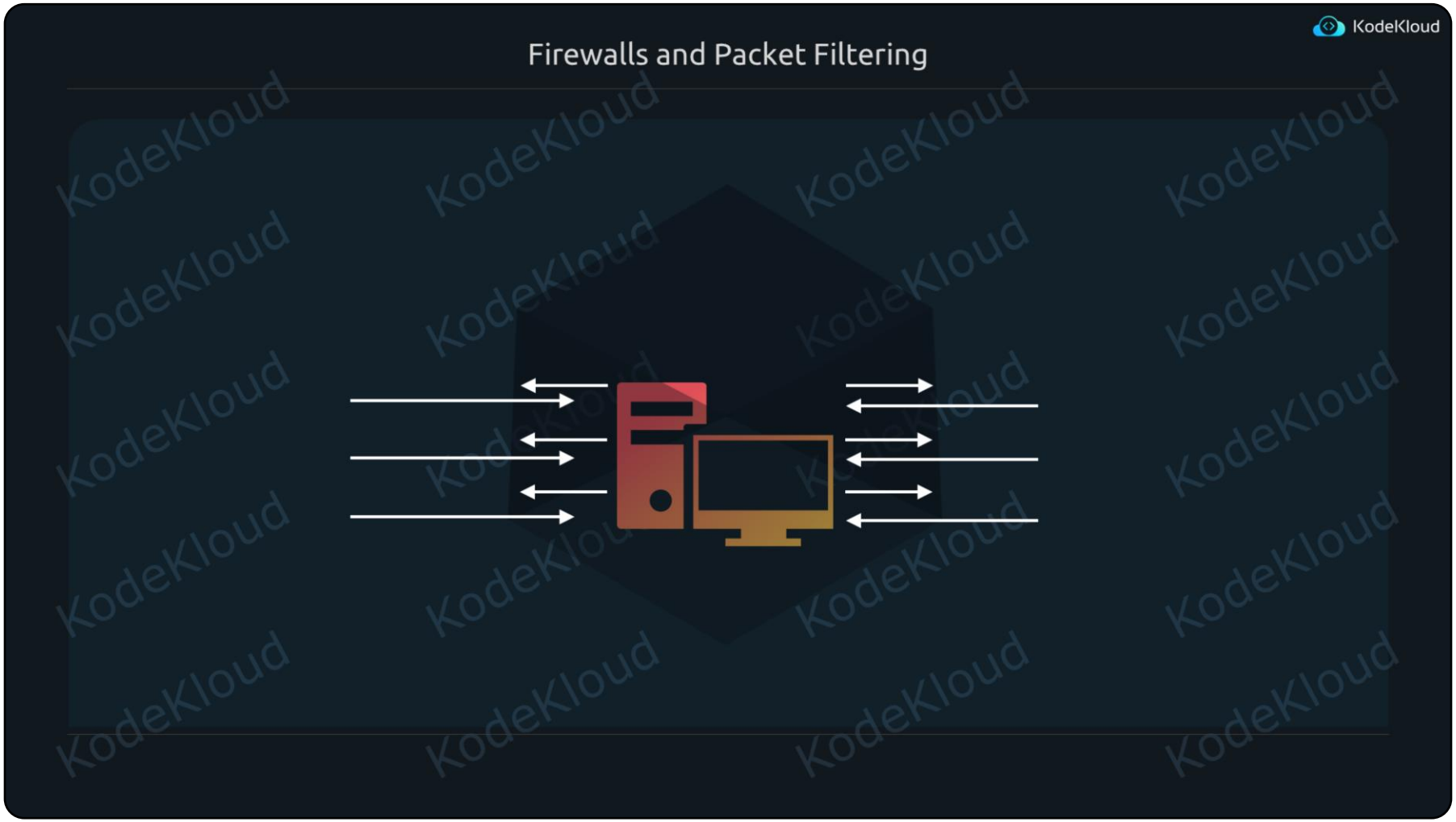
- A bridge glues together two or more **networks**, helping the devices on them talk to each other as if they are on a single network.
- A bond glues together two or more ***connections* to a network**. That is, two or more **roads** to a network are presented as a single road.

These are the typical use cases at least. Talented network administrators can find many more varied and creative uses.

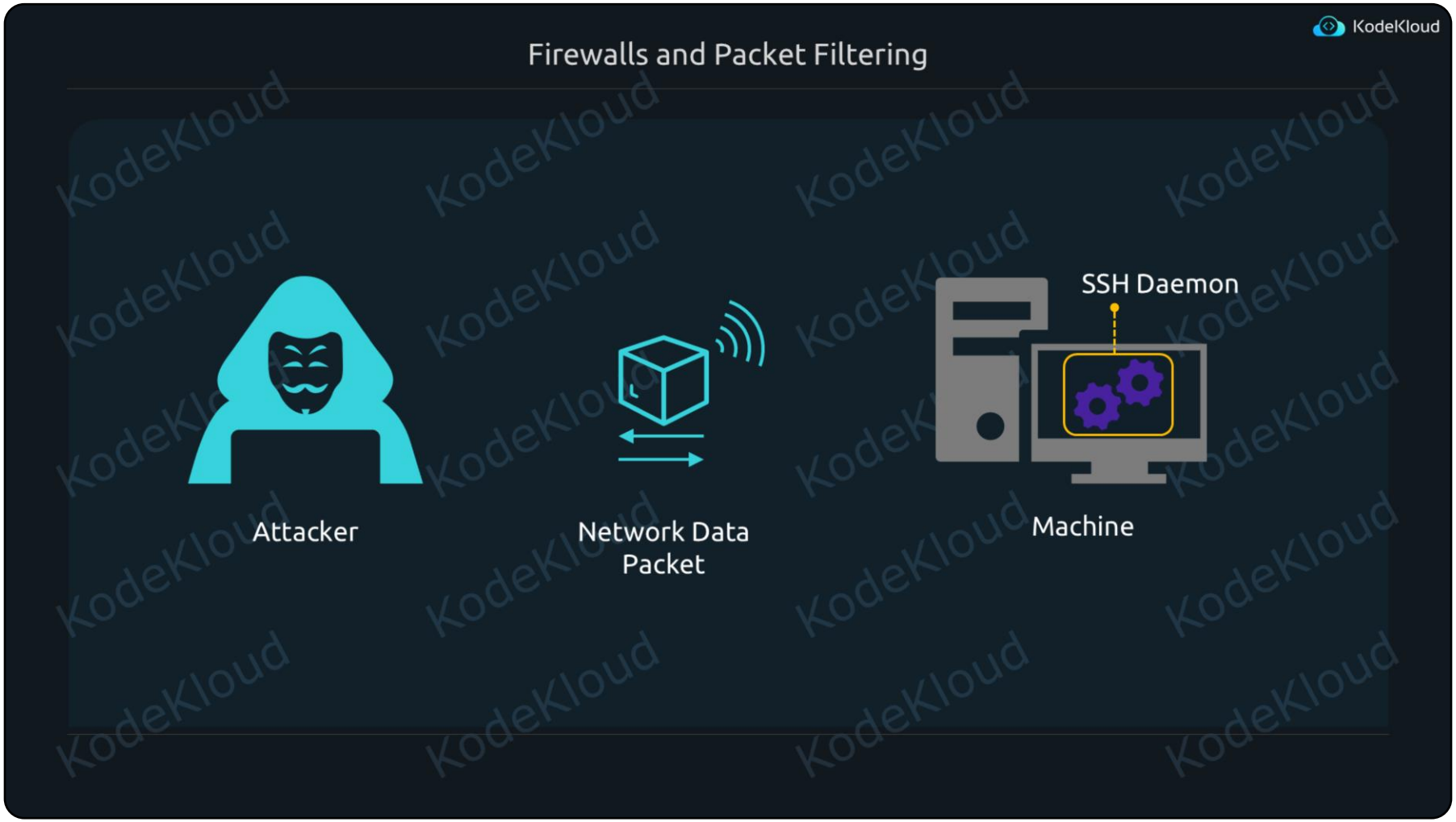
KodeKloud

Visit www.kodekloud.com to discover more.

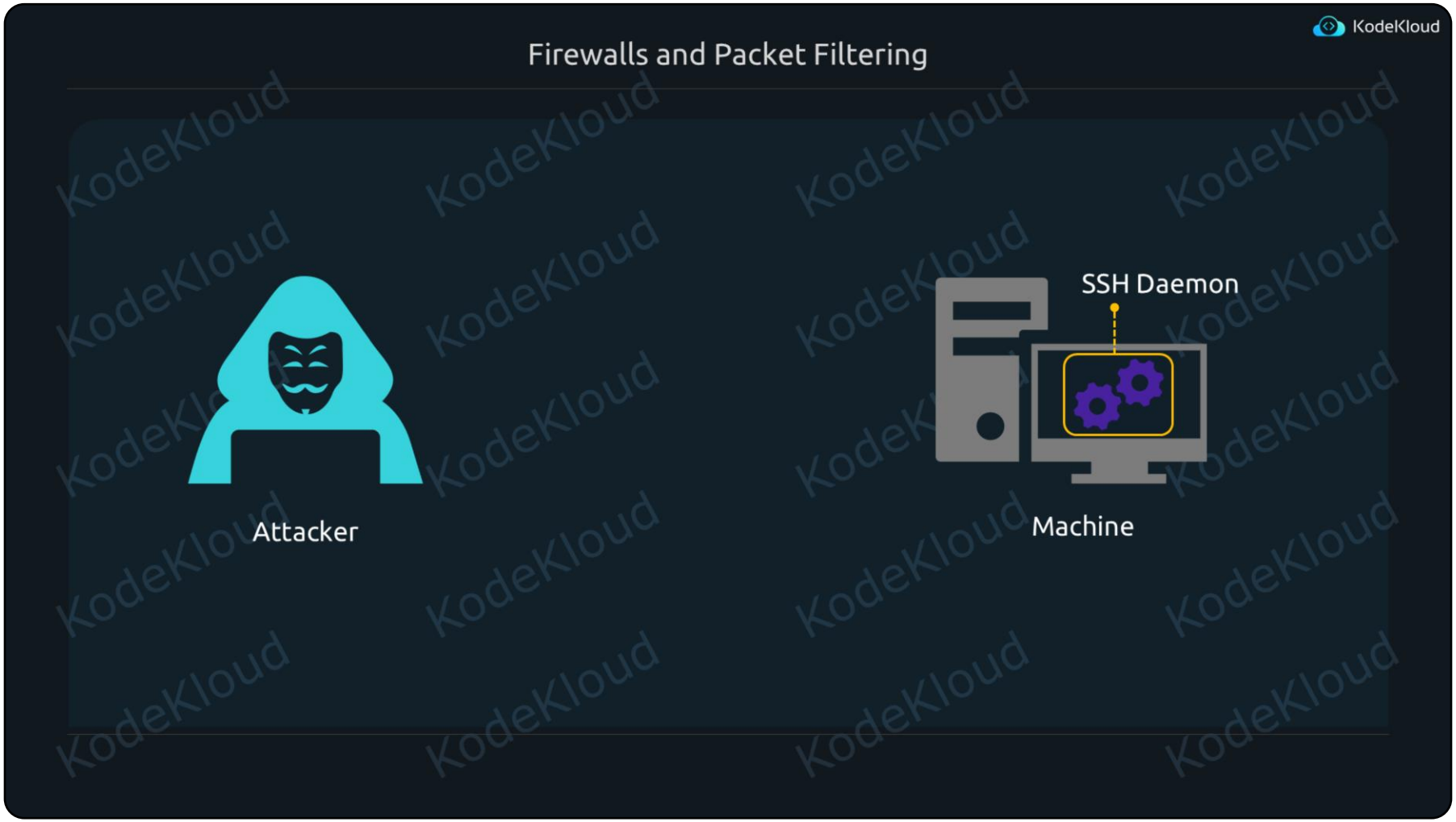
In this lesson we'll learn how to configure a firewall on Ubuntu machines.



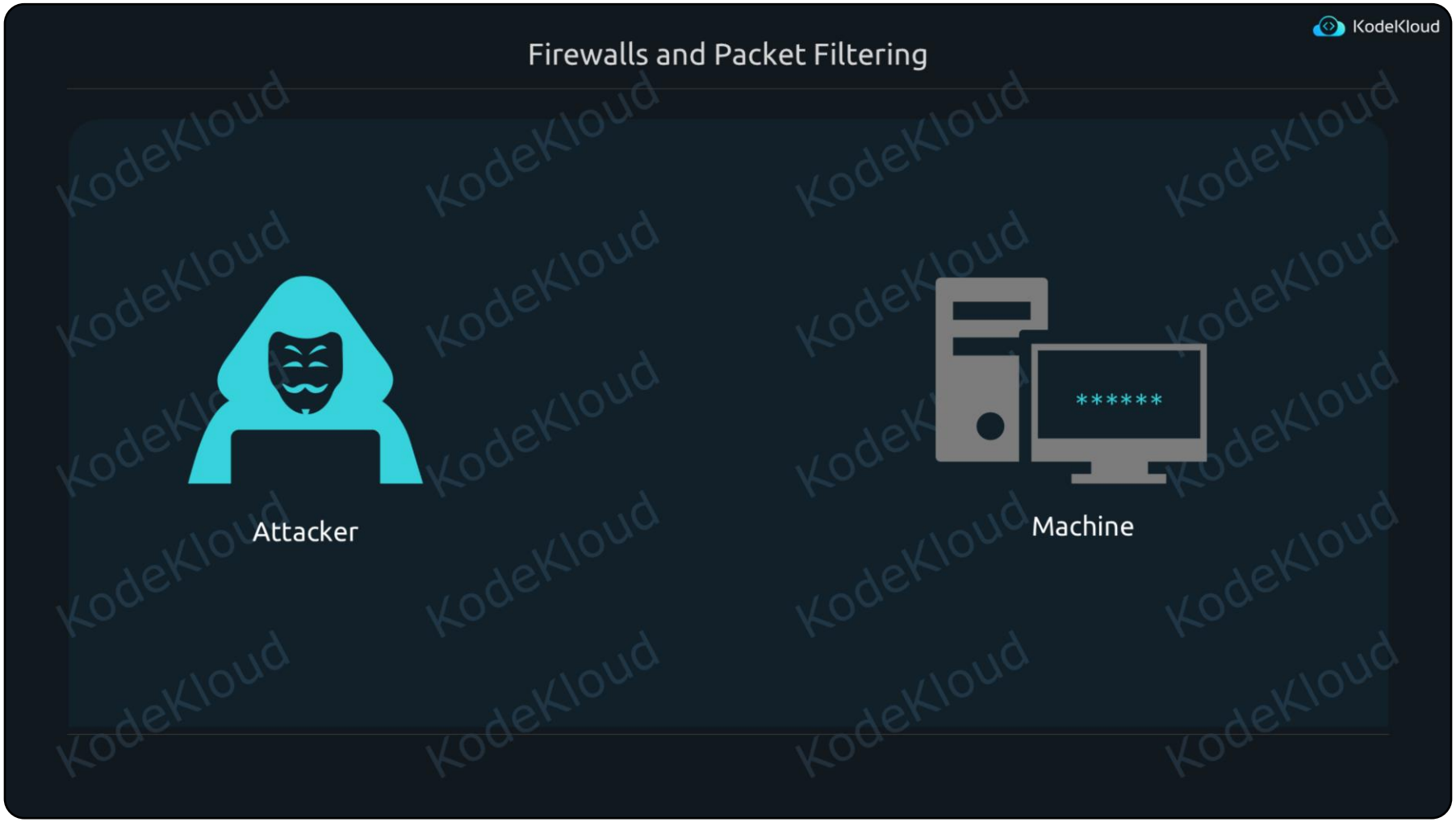
Like the name suggests, this helps us set up a line of defense against potential attackers. It lets us build a wall and block the network data that the server shouldn't process. This wall can defend both against network data that is coming in, but also for network data that is sent out.



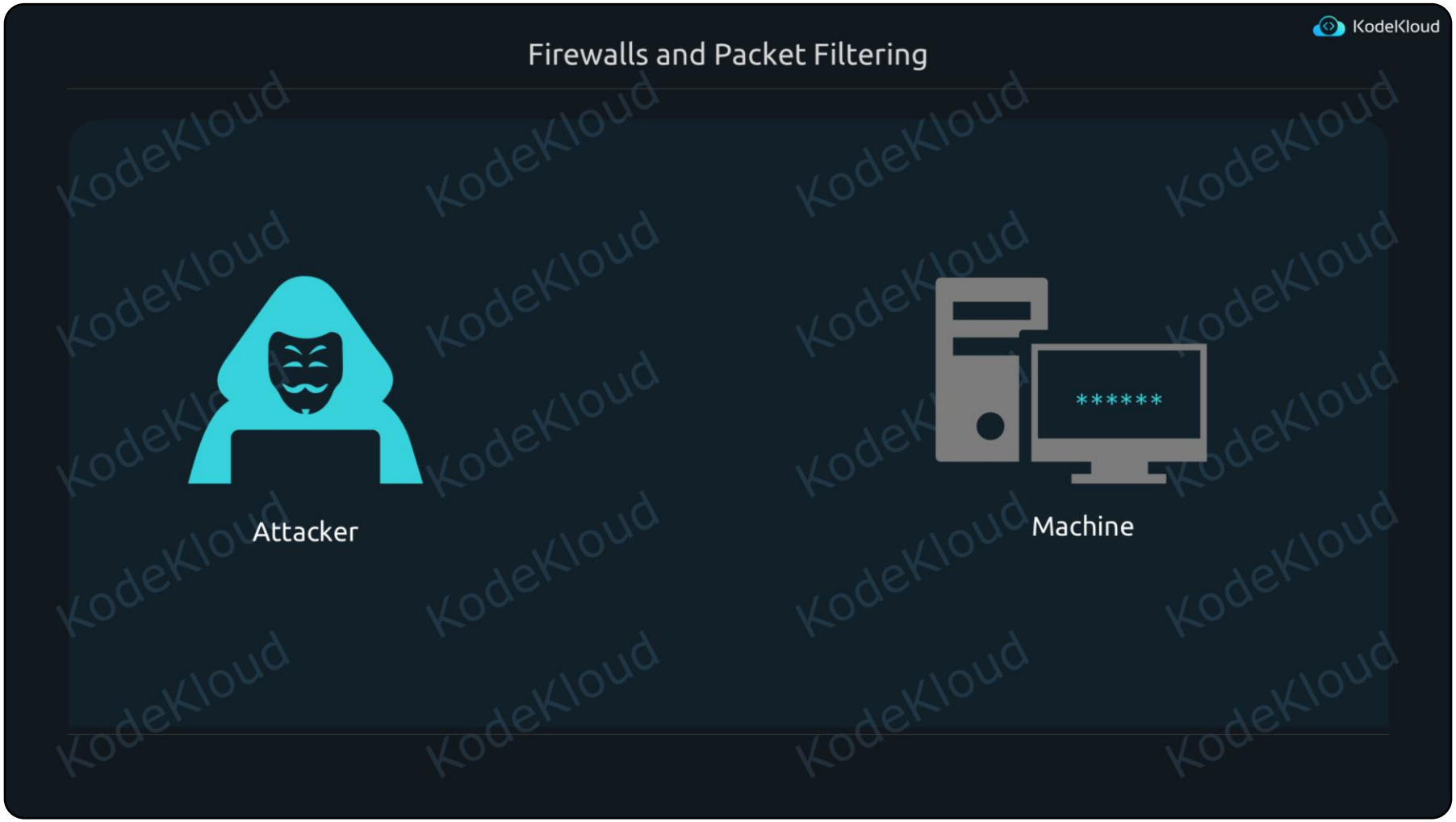
When can a firewall save the day? Here's an example. An attacker might discover that they can create a special packet of network data which can exploit the SSH daemon running on our machine. If our SSH daemon receives that packet, it might let the attacker break into our machine, log in, and do some damage.



When can a firewall save the day? Here's an example. An attacker might discover that they can create a special packet of network data which can exploit the SSH daemon running on our machine. If our SSH daemon receives that packet, it might let the attacker break into our machine, log in, and do some damage.

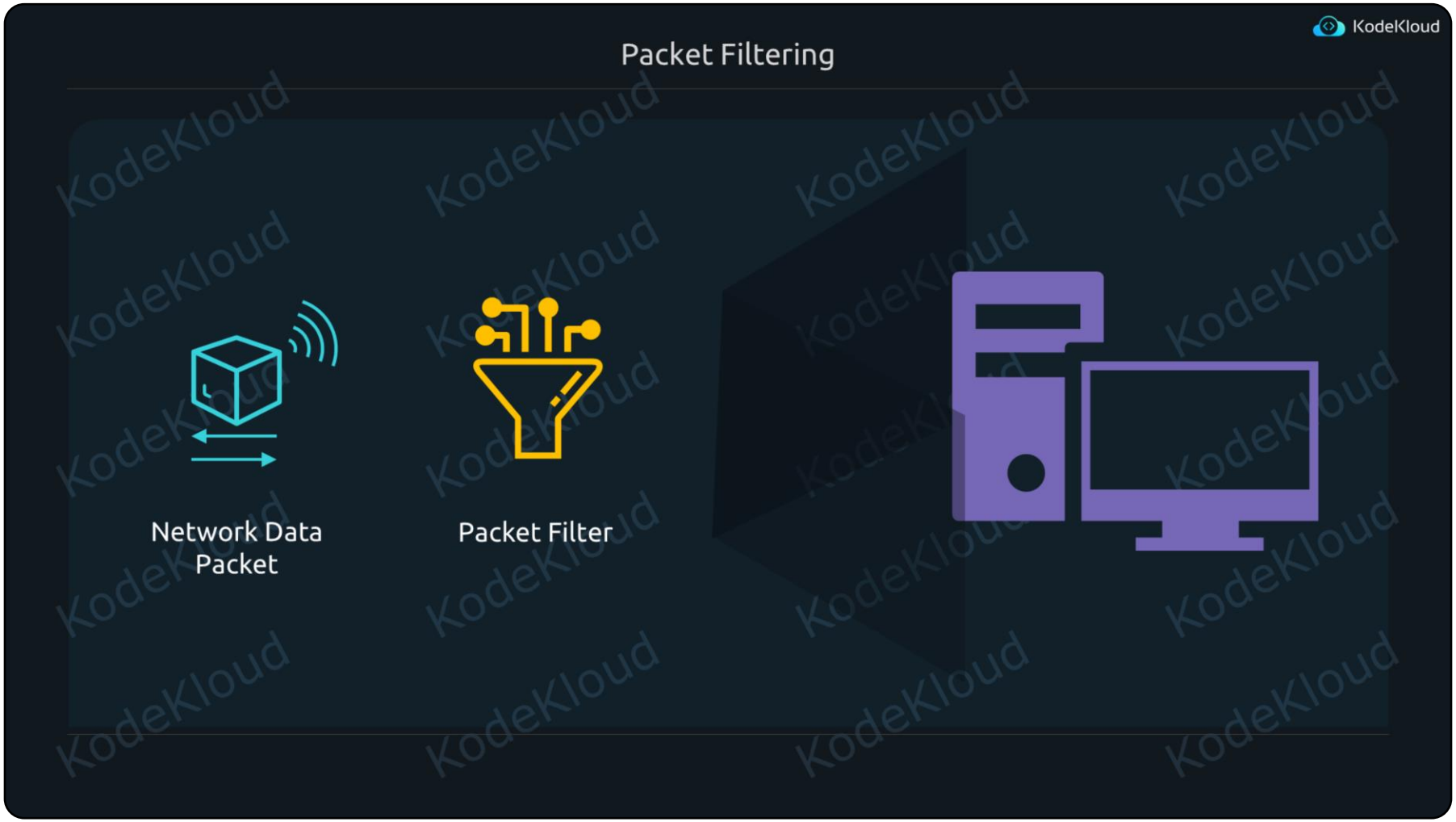


When can a firewall save the day? Here's an example. An attacker might discover that they can create a special packet of network data which can exploit the SSH daemon running on our machine. If our SSH daemon receives that packet, it might let the attacker break into our machine, log in, and do some damage.

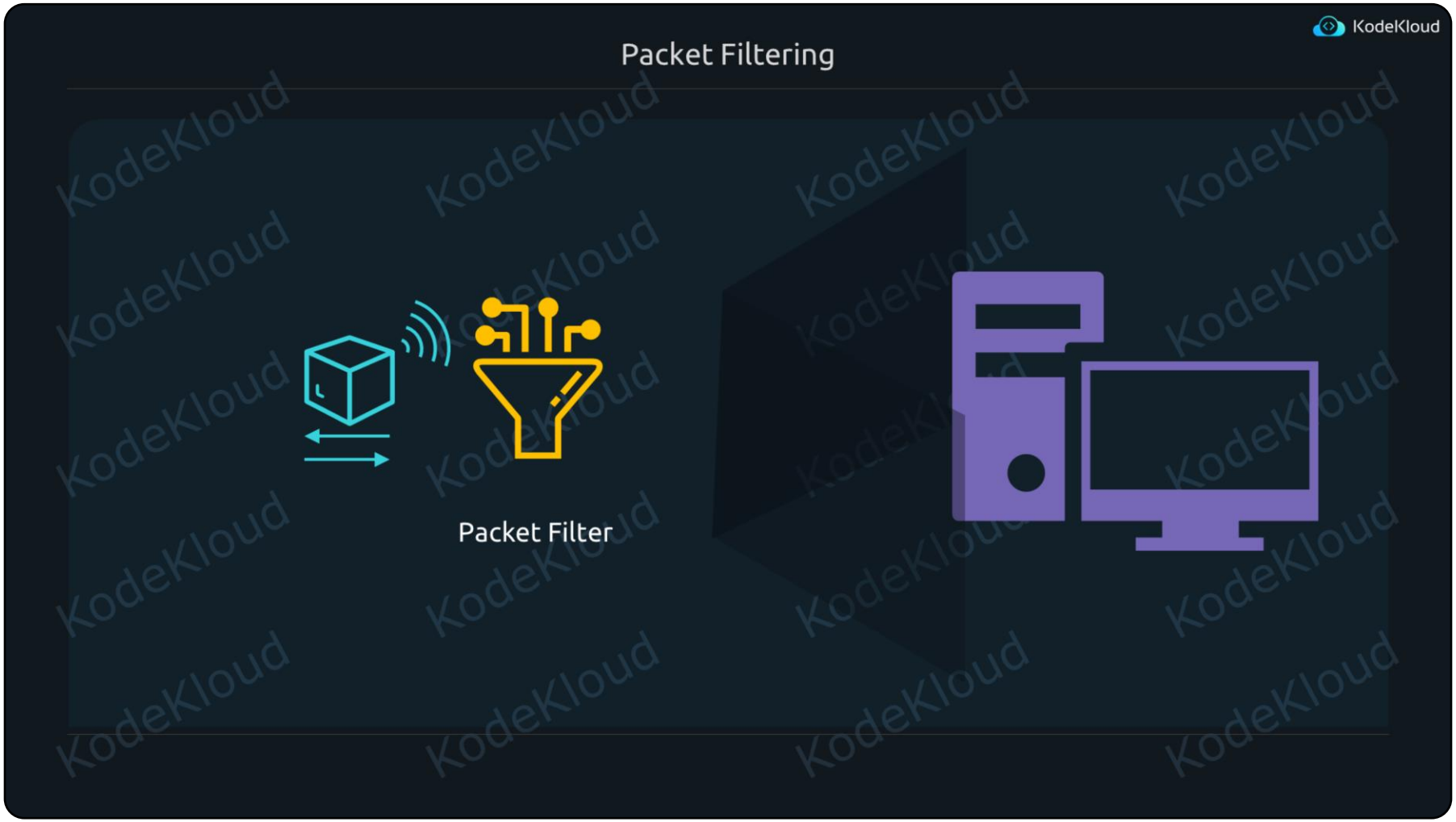


When can a firewall save the day? Here's an example. An attacker might discover that they can create a special packet of network data which can exploit the SSH daemon running on our machine. If our SSH daemon receives that packet, it might let the attacker break into our machine, log in, and do some damage.

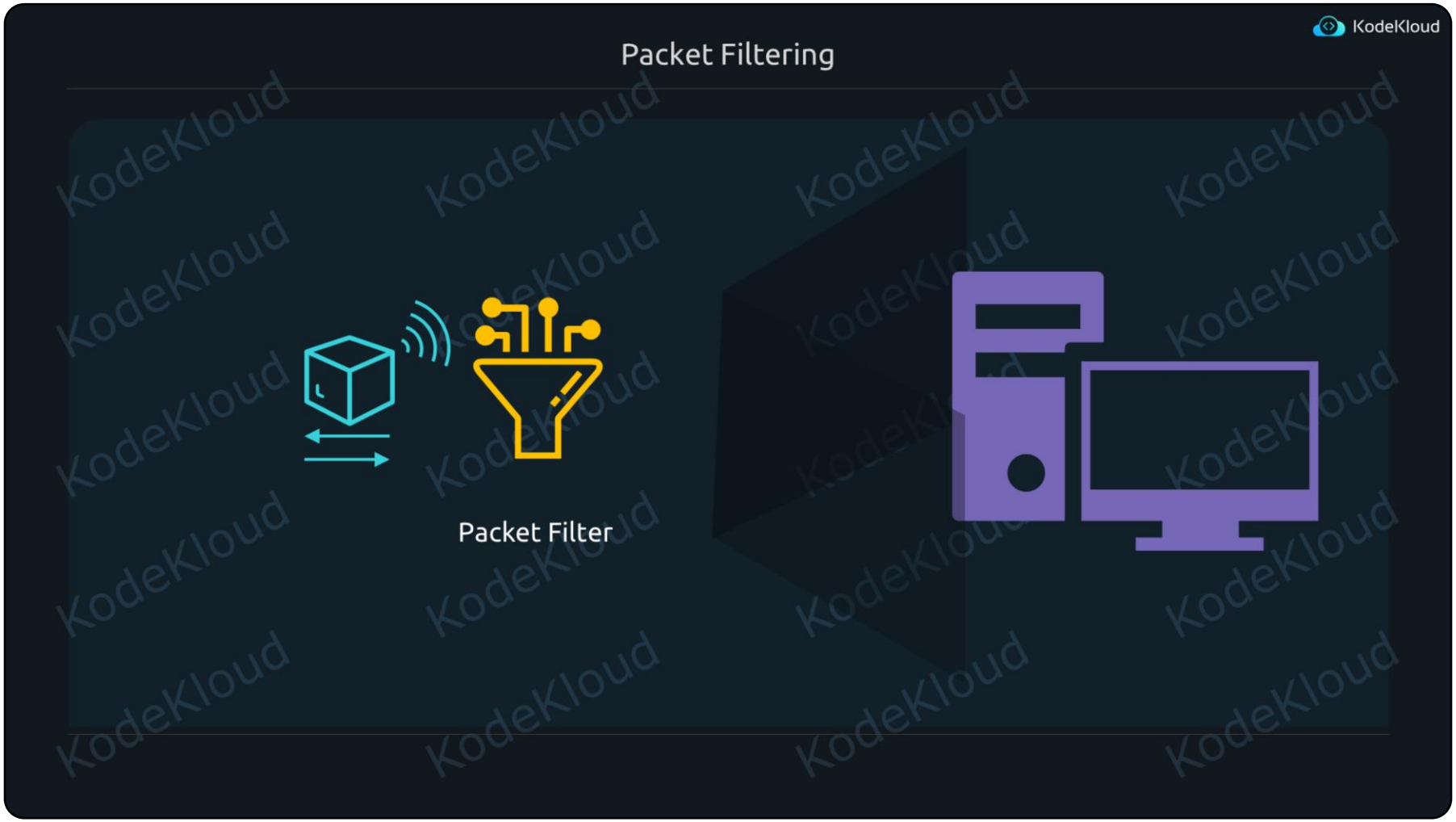
But by setting up a packet filtering firewall, we could be able to prevent this attack. Because the packet filter could look at the network data that was received, and simply reject it. This way, the malicious network packet doesn't even reach its target, the SSH daemon. So the attack is prevented.



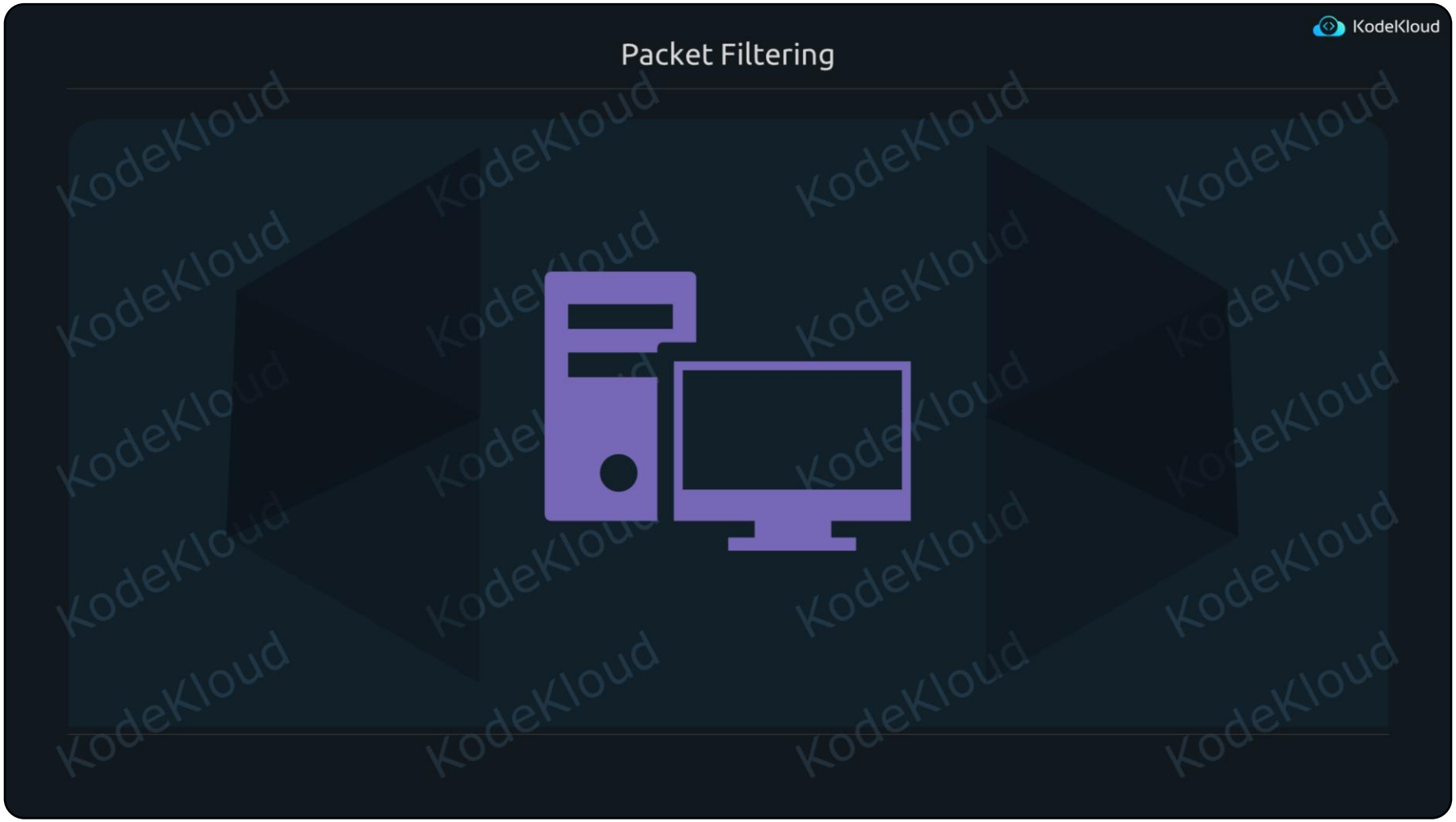
But by setting up a packet filtering firewall, we could be able to prevent this attack. Because the packet filter would look at the network data that was received, and simply reject it. This way, the malicious network packet doesn't even reach its target, the SSH daemon. So the attack is prevented.



But by setting up a packet filtering firewall, we could be able to prevent this attack. Because the packet filter would look at the network data that was received, and simply reject it. This way, the malicious network packet doesn't even reach its target, the SSH daemon. So the attack is prevented.

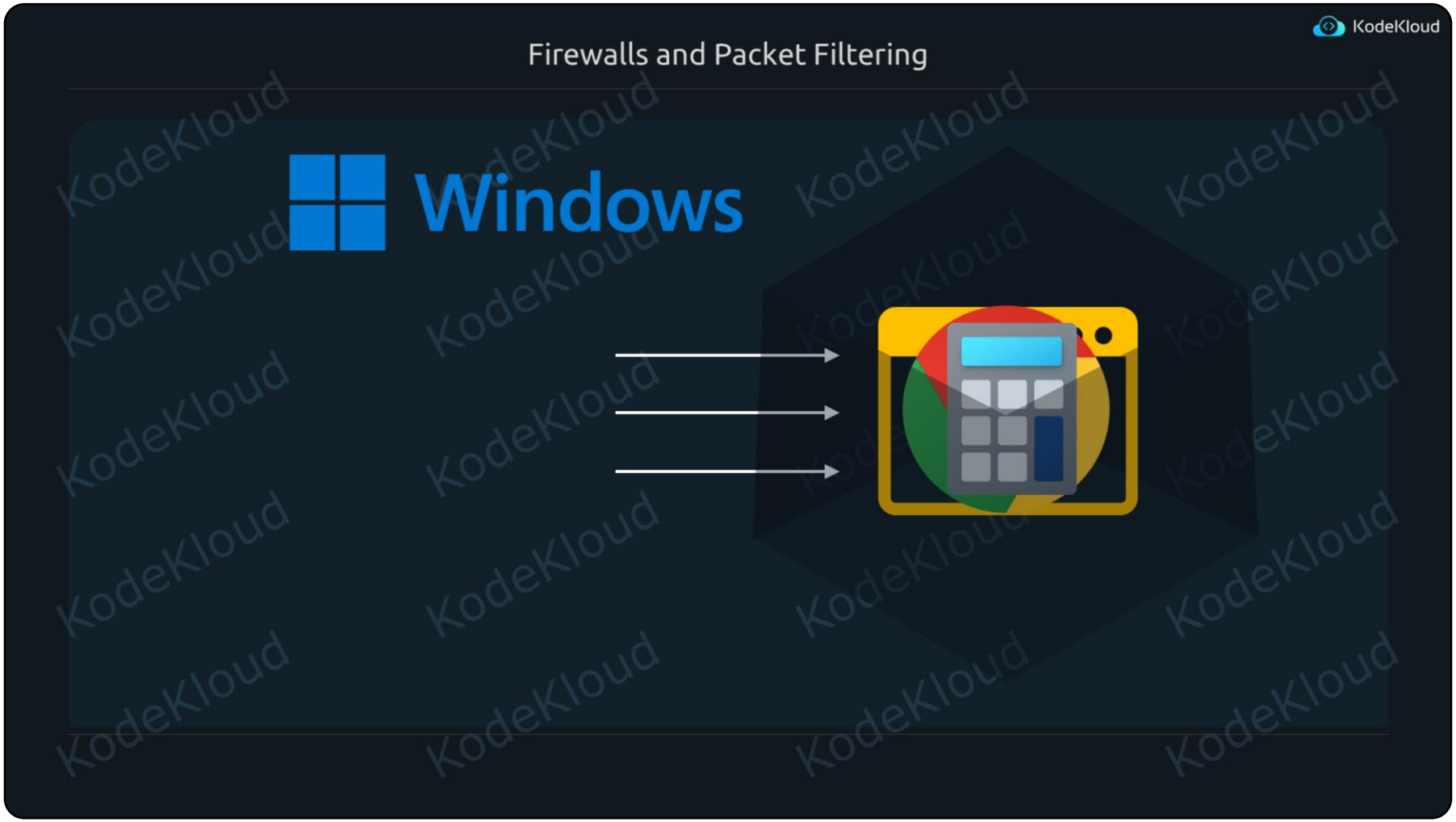


But by setting up a packet filtering firewall, we could be able to prevent this attack. Because the packet filter would look at the network data that was received, and simply reject it. This way, the malicious network packet doesn't even reach its target, the SSH daemon. So the attack is prevented.

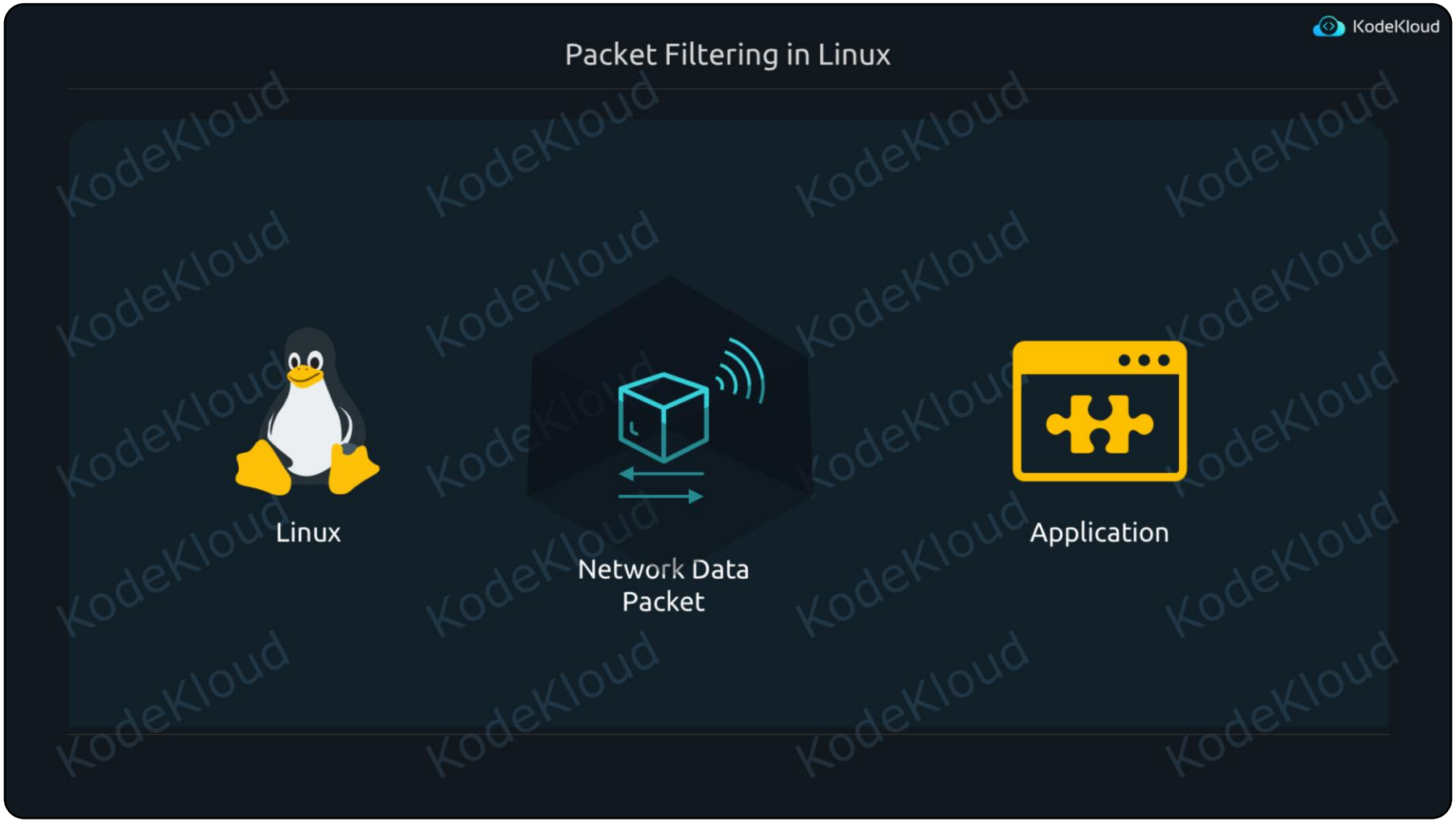


But by setting up a packet filtering firewall, we could be able to prevent this attack. Because the packet filter would look at the network data that was received, and simply reject it. This way, the malicious network packet doesn't even reach its target, the SSH daemon. So the attack is prevented.

There are many types of firewalls. For example, an application firewall can monitor individual apps. And decide if it should allow or deny network traffic based on rules set up for each application. Something like this is implemented in the Windows operating system. It makes sense to allow network data to flow through an app like Chrome. But it also makes sense to block traffic for something like the Windows Calculator.



There are many types of firewalls. For example, an application firewall can monitor individual apps. And decide if it should allow or deny network traffic based on rules set up for each application. Something like this is implemented in the Windows operating system. It makes sense to allow network data to flow through an app like Chrome. But it also makes sense to block traffic for something like the Windows Calculator.



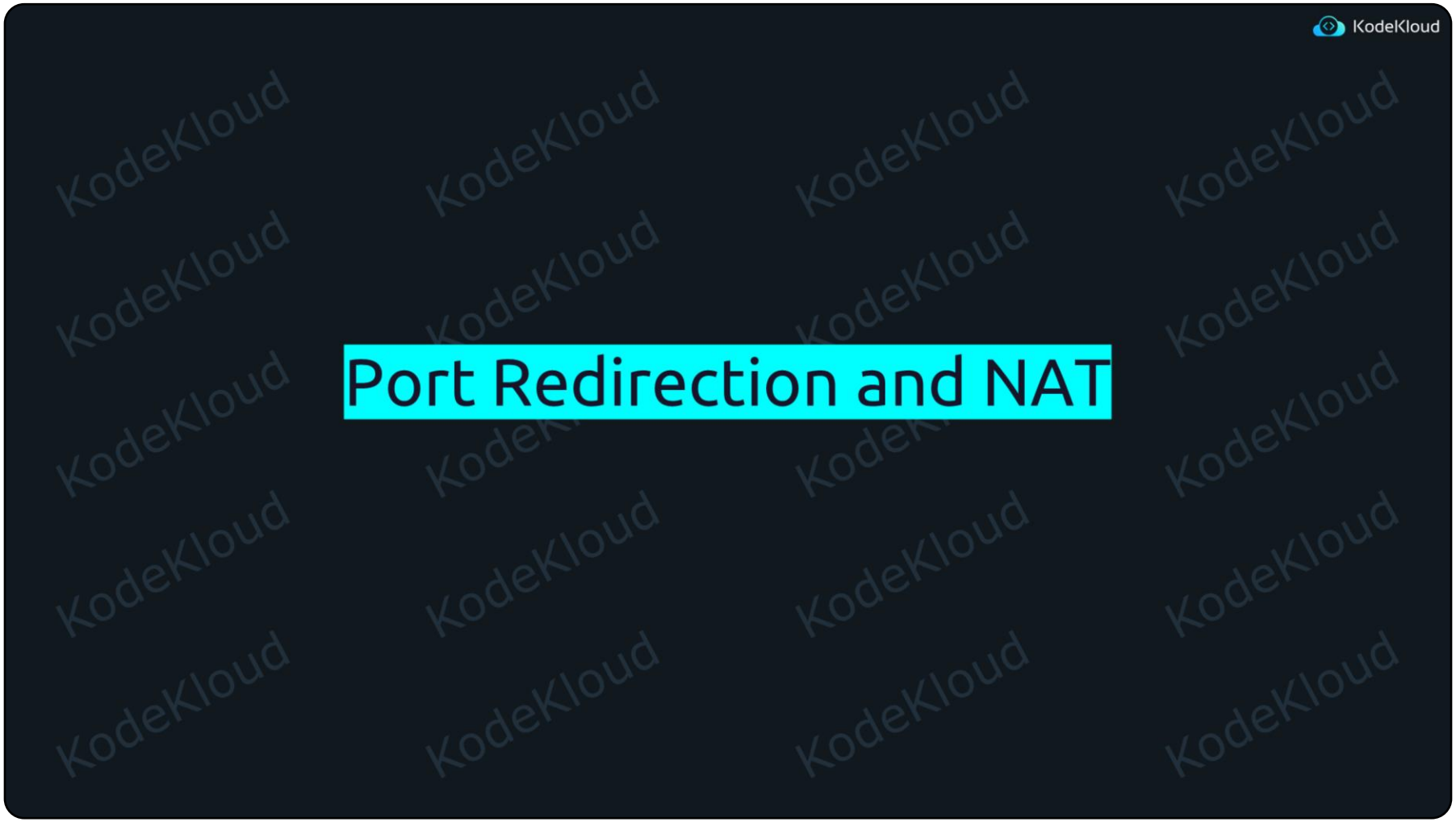
Linux, though, includes a packet filtering firewall out-of-the-box. As its name suggests it decides what to do with data based on the network packets themselves, not applications.

Now let's see how we can use a firewall in practice.

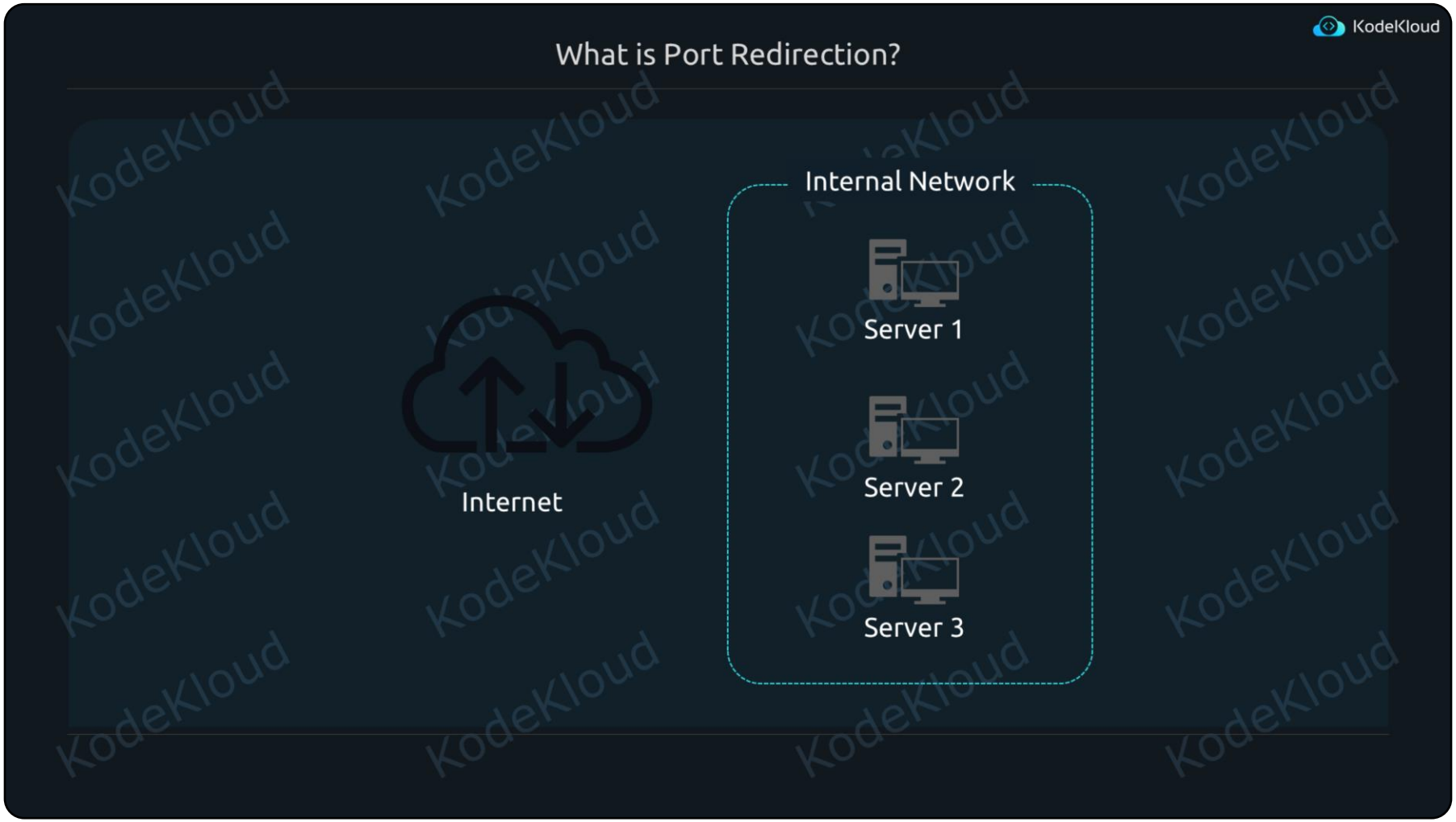
[demo starts here, see script/document]

KodeKloud

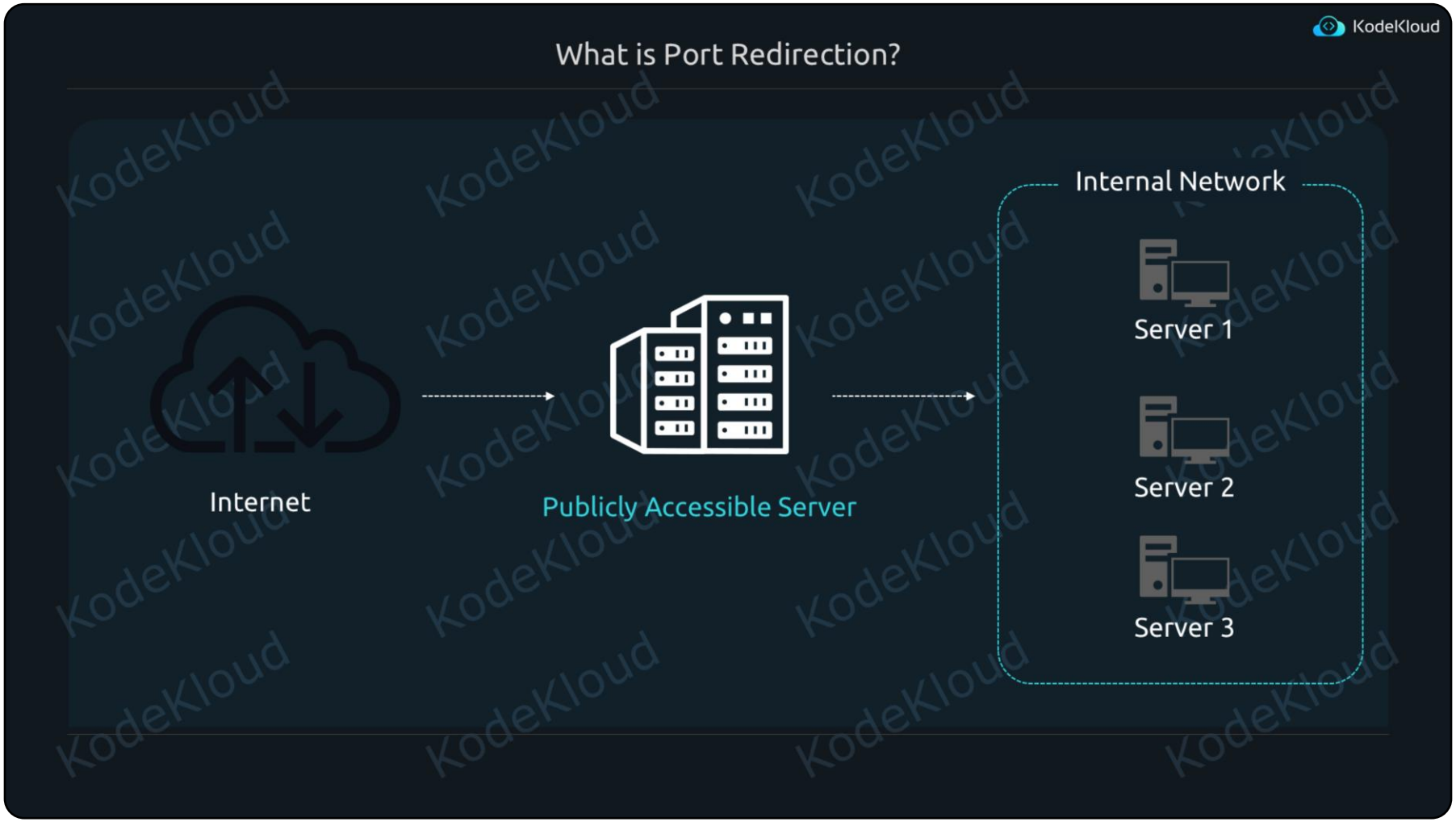
Visit www.kodekloud.com to discover more.



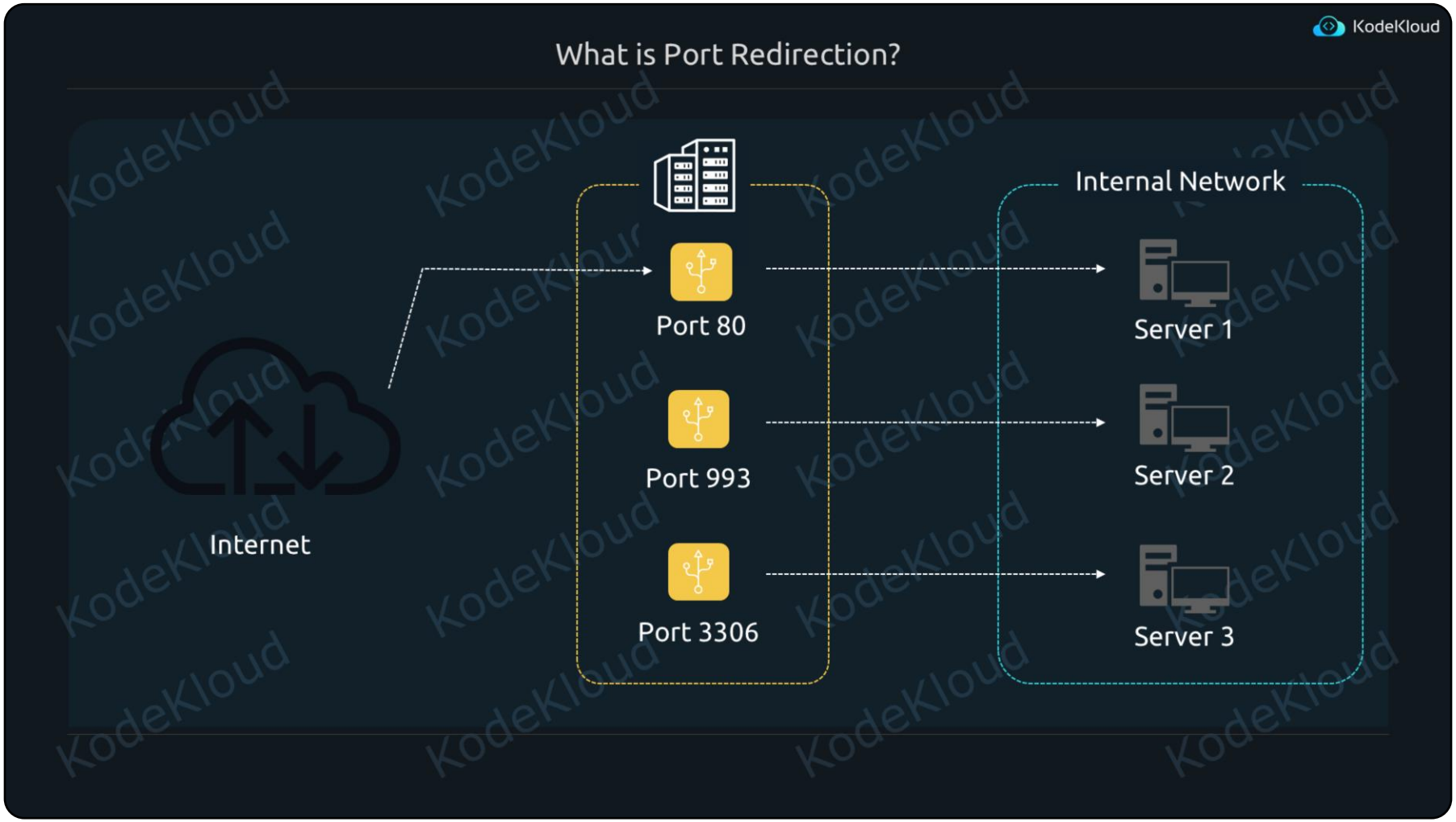
In this lesson we'll look at how we can set up port redirection and Network Address Translation, abbreviated as NAT. First, let's explore how port redirection works.



In some cases we may have some servers on a private, internal network. This network is not directly accessible from the Internet. However, we can add a publicly accessible server in the middle to get to this structure:



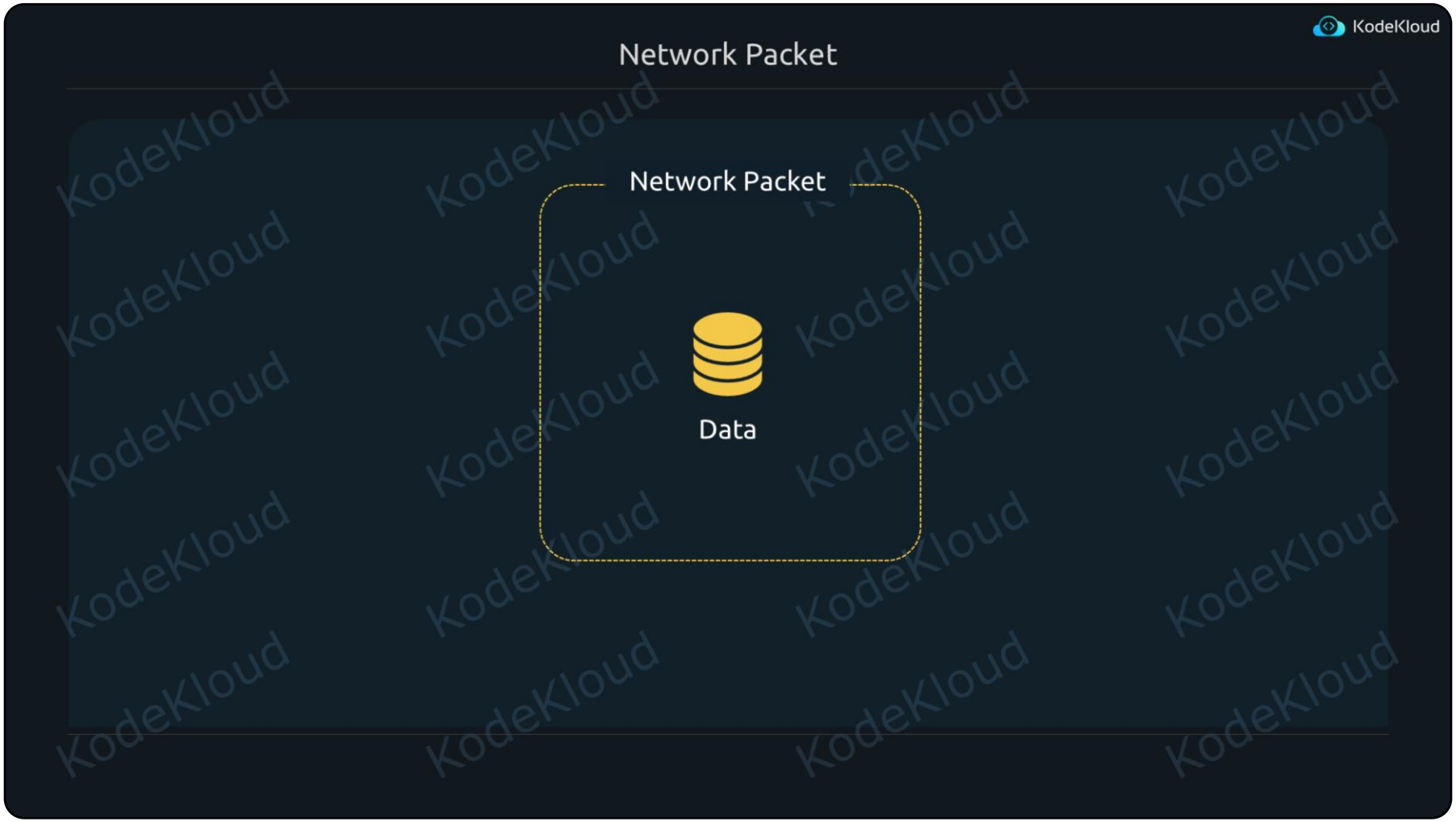
The public server is connected to both networks: The internal network at the right, and the Internet on the left. So the private servers on our internal network can now be accessed from the Internet if incoming connections go through this publicly accessible server in the middle. But not before going through additional configuration steps. Because here's the problem at this point.



Let's say that some device from the Internet contacts our public server. And it asks to connect to port 80 to access a web page. At this point our public server would be confused. Because it does not know where it should forward this connection to port 80. What private server on our internal network should deal with port 80 connections? Should it send it along to private Server 1, 2, or 3?

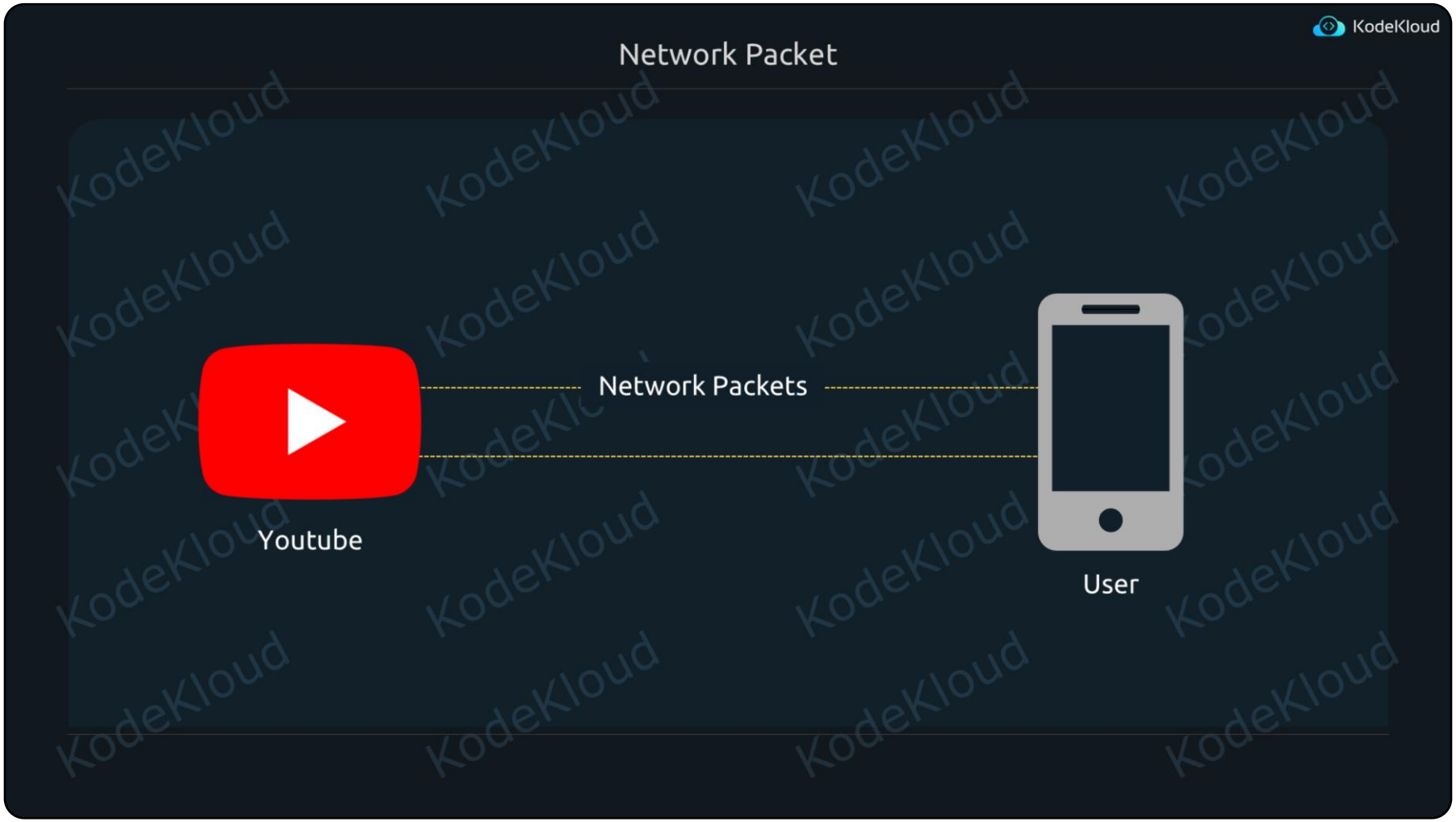
Well, with port redirection, also called port forwarding, we can solve this problem.

We can create rules that tell our public server where to redirect incoming connections to port 80, 993, and 3306. In this case, anything incoming on port 80 will be redirected, or forwarded, to Server 1. Port 993 connections would be forwarded to Server 2, and port 3306 connections to Server 3.

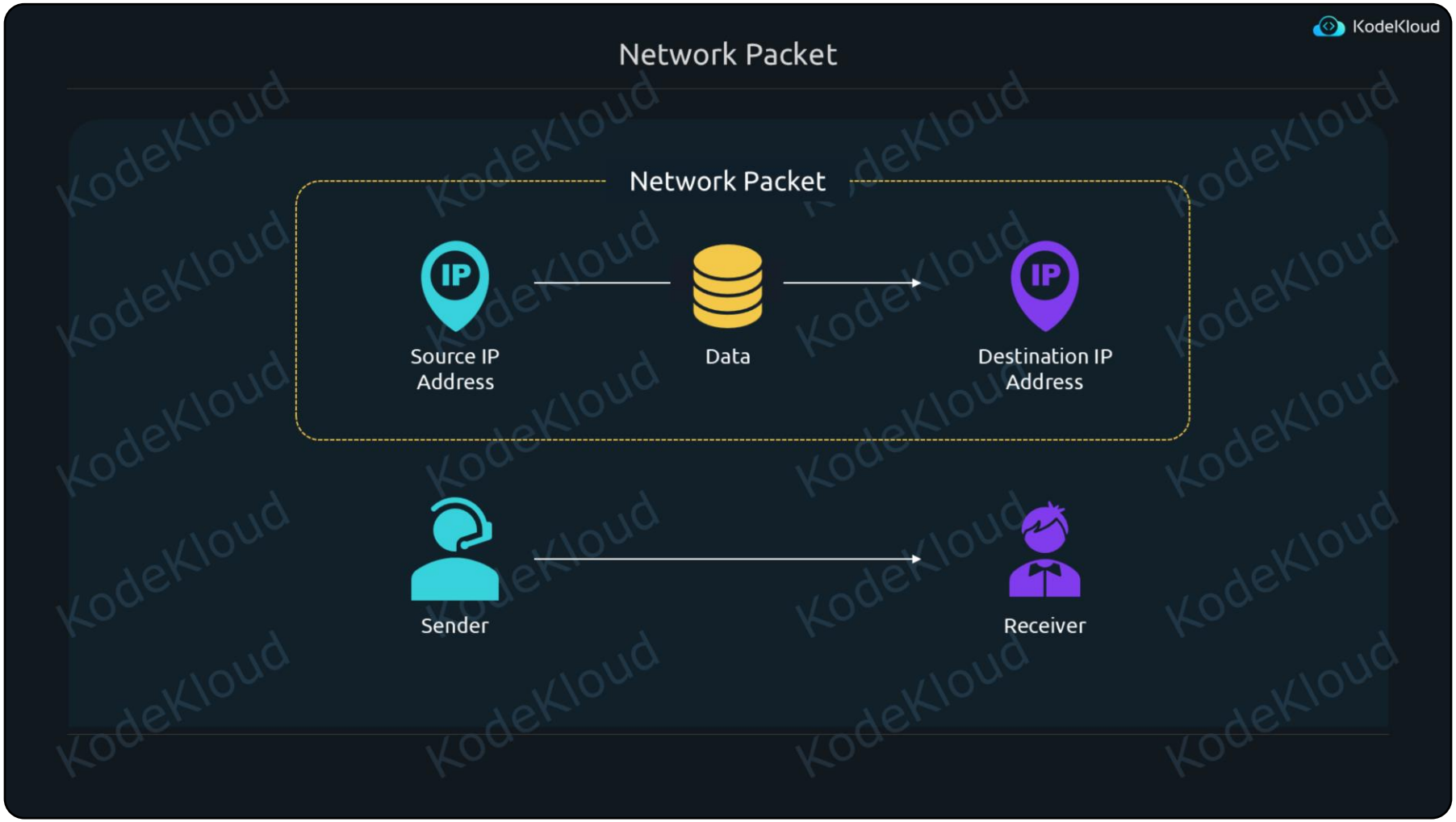


Now let's move onto Network Address Translation.

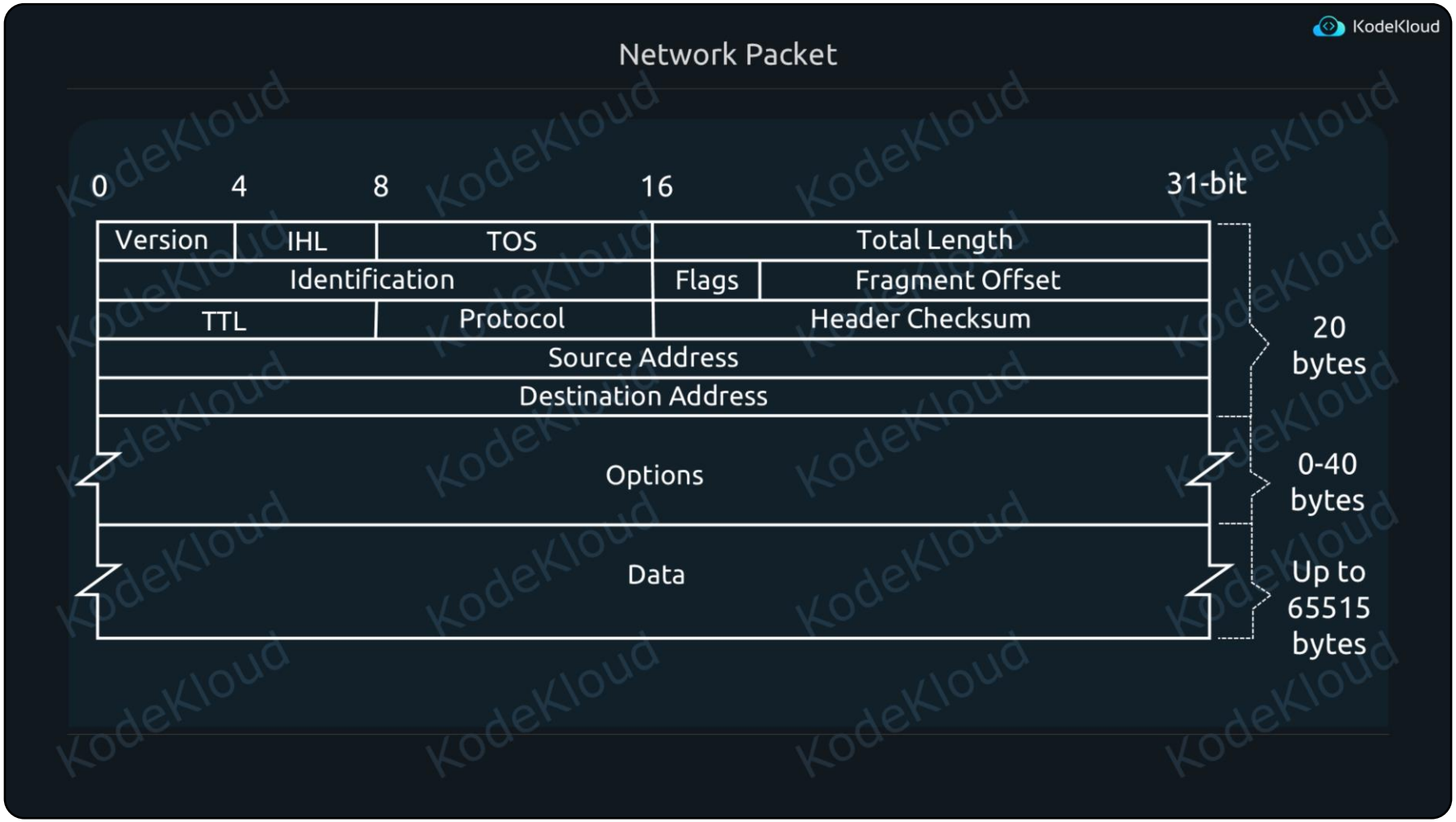
When data is sent through networks, it's first encapsulated in what are called network packets.



For example, when we watch a video on YouTube this data is sent to us in very small network packets, each containing a portion of that video.



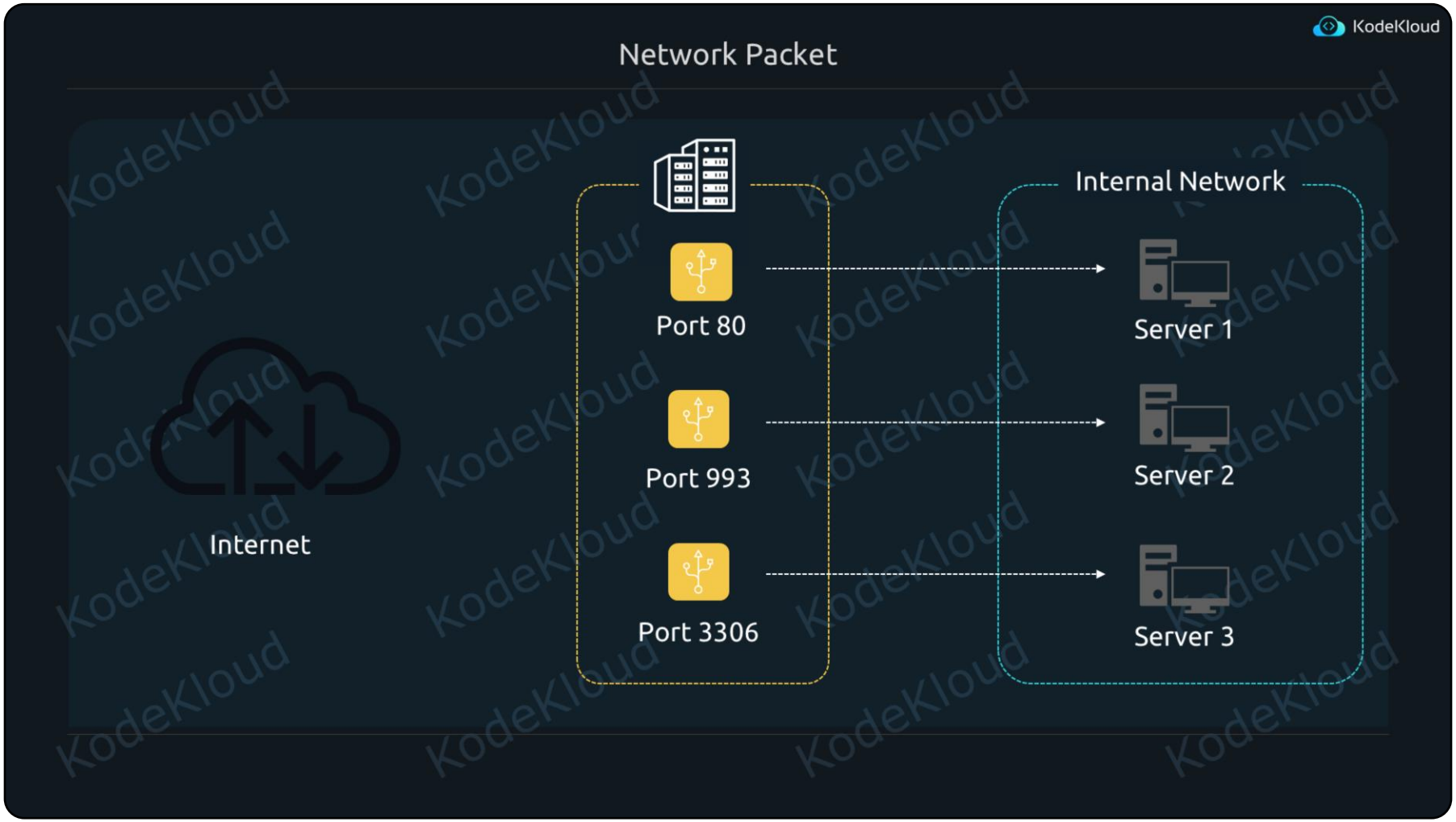
Besides the data itself, each network packet also includes the source IP address and destination IP address. This identifies who sent the packet, and who should receive it.



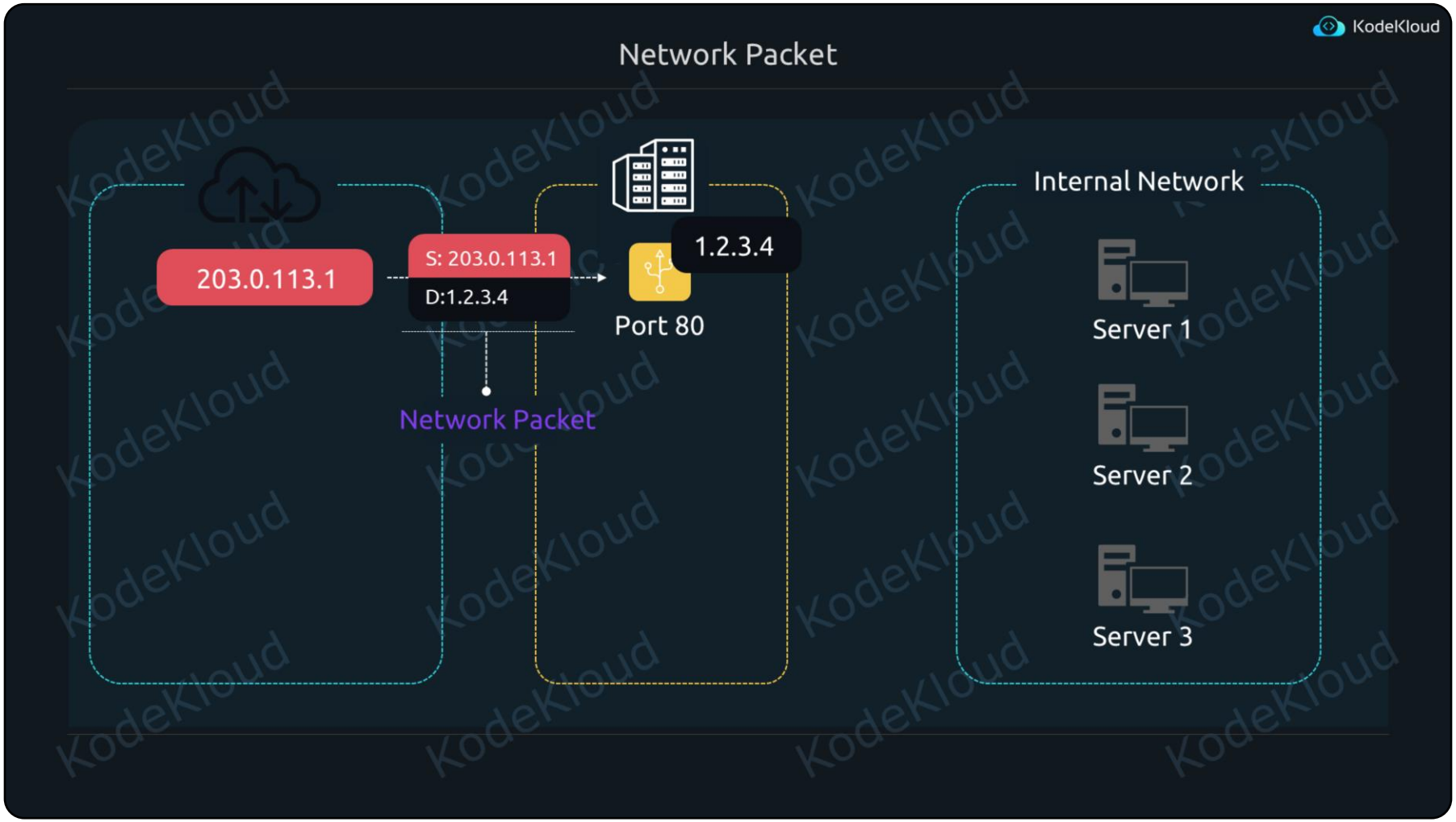
An IPv4 packet includes the stuff we see here. Mostly things that computers and programs need to make sense of what they send and receive. But the important parts that we will use in this lesson are the source address, and the destination address we see encapsulated in the middle of this diagram.

Data on the Internet is passed along from server to server, router to router, switch to switch, until it finally arrives at a destination. So having the source and destination addresses encoded in network packets helps every device in this chain

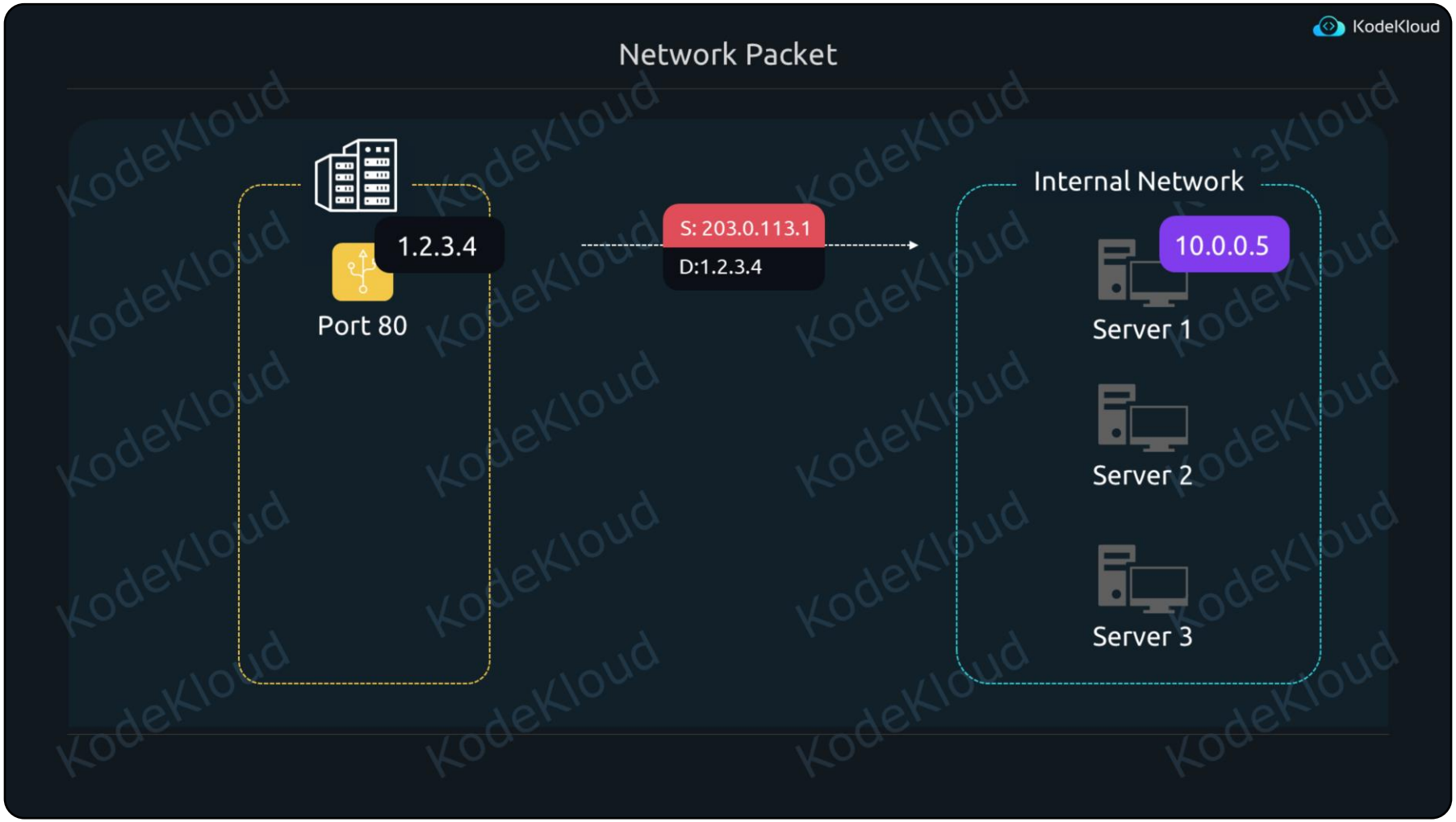
know where to forward network data, but also know where to return a response.



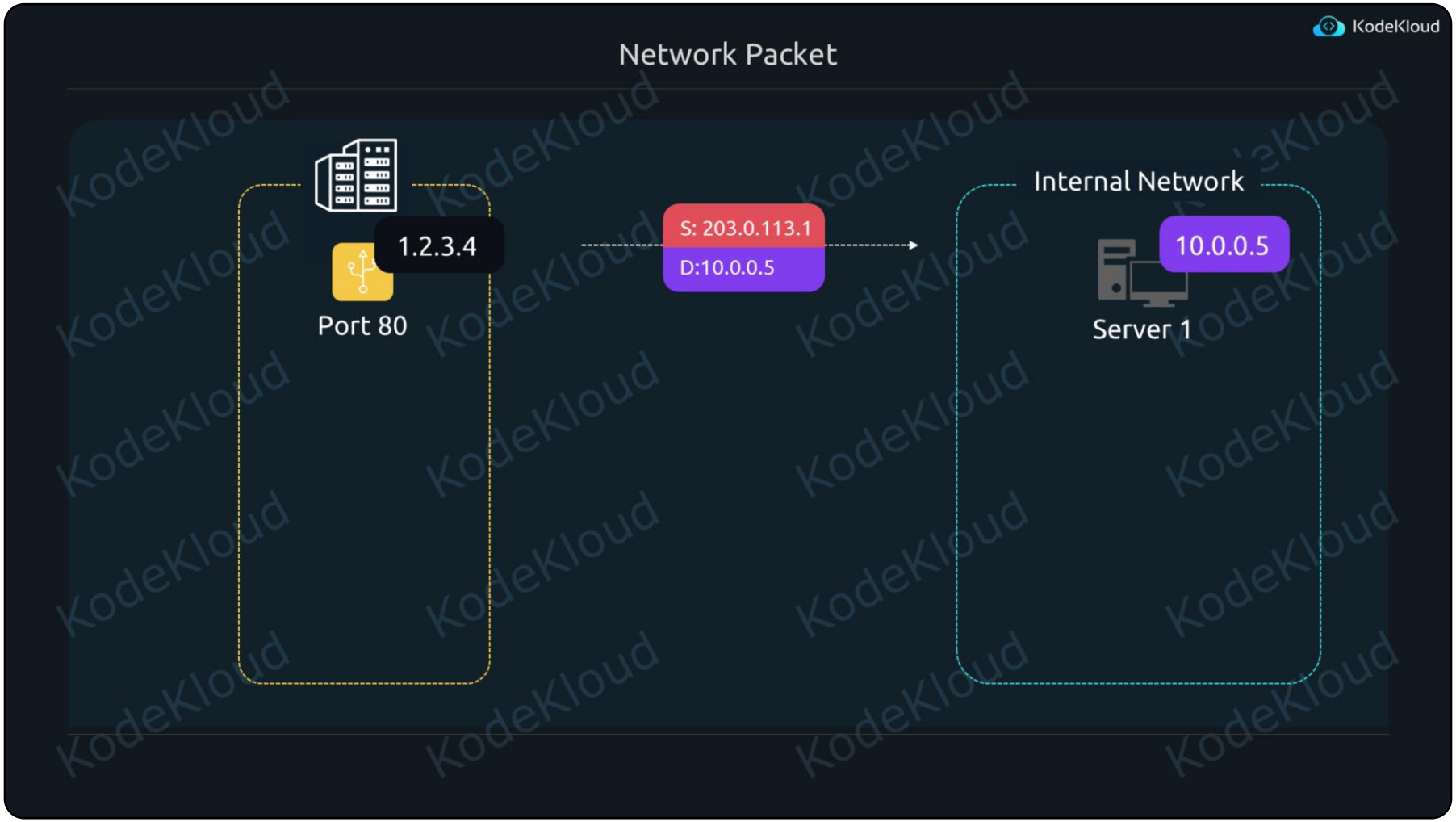
Now let's think of our previous example, with the three servers on a private network, and one server connected to both the Internet and this private network.



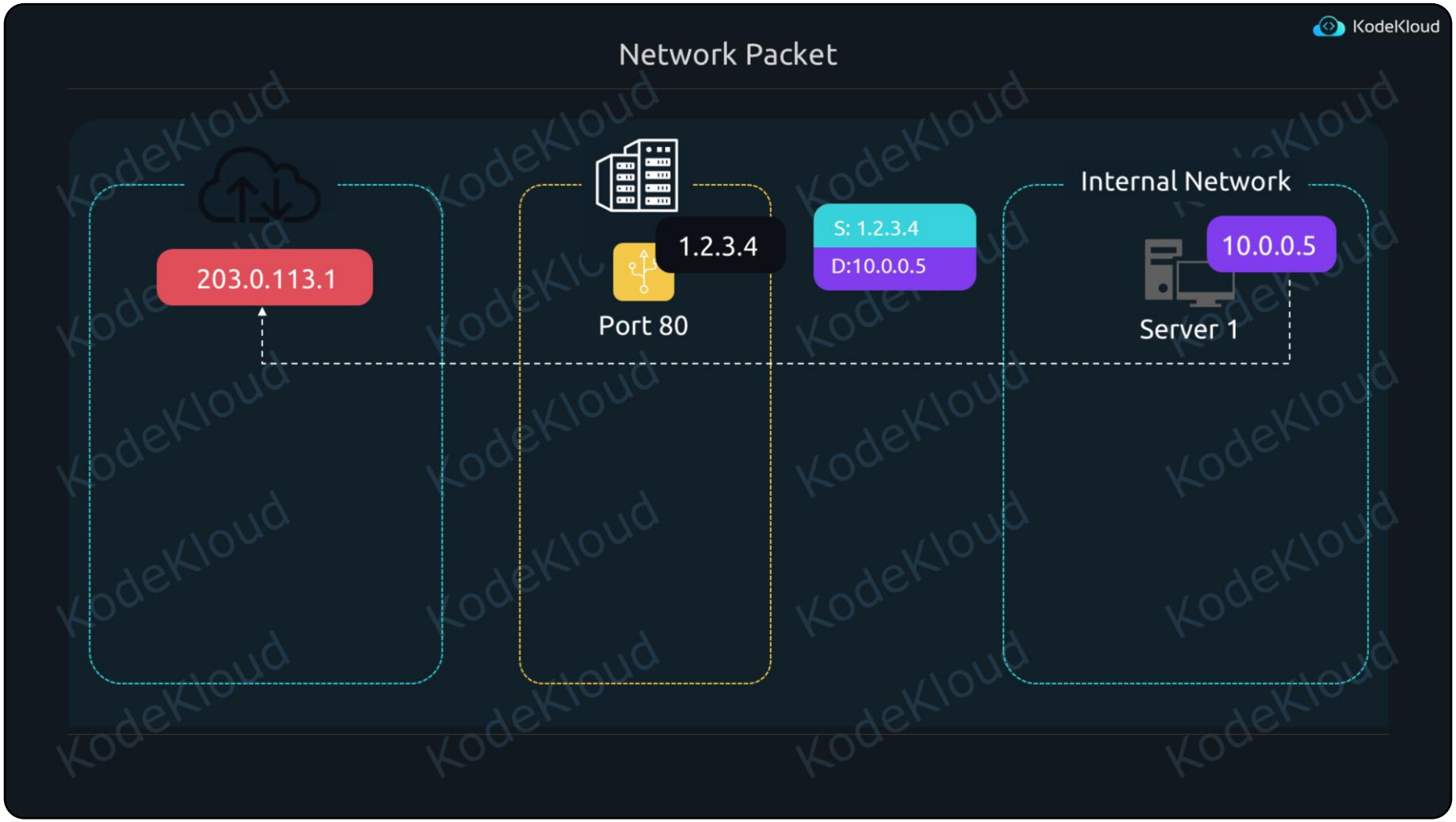
We can imagine a device on the Internet that has an IP like 203.0.113.1. It sends some data to our public server. So "203.0.113.1" will be encoded as the source address in the network packet sent out. And the destination address will be the IP of our public server, let's say 1.2.3.4. When the public server gets a network packet on port 80, it knows it has to forward it to private Server 1.



At this point, our public server actually starts using Network Address Translation. Remember, this network packet that should be forwarded has the IP of our public server (1.2.3.4) as the destination address. If it would be forwarded like this, then private Server 1, with an IP like 10.0.0.5, would think this network packet is not meant for it and simply ignore it. So what our public server does is change the destination address to the IP of private Server 1, 10.0.0.5. Now Server 1 sees this network packet matches its own IP in the destination field, so it will accept it.

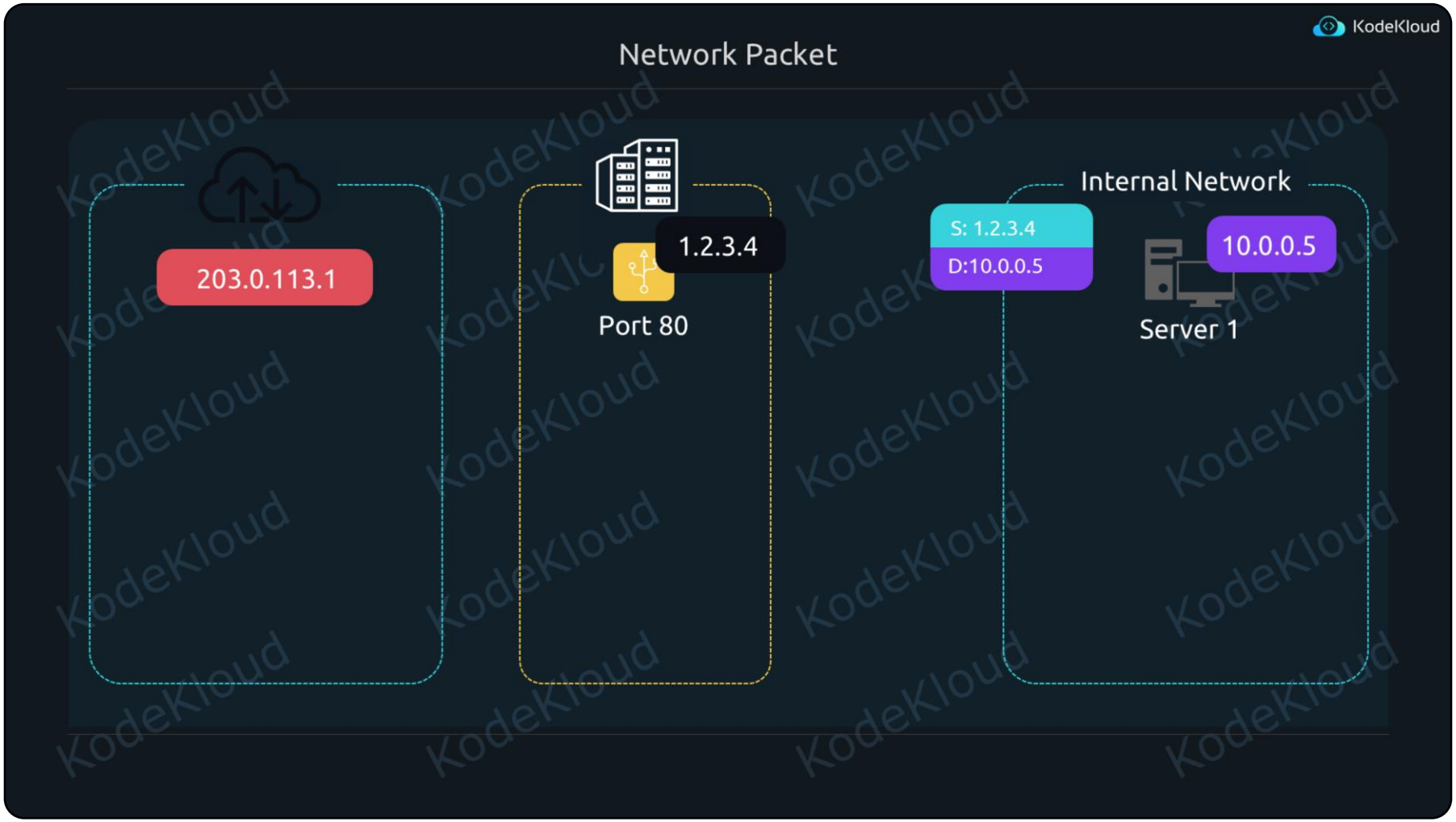


So what our public server does is change the destination address to the IP of private Server 1. The destination address 1.2.3.4 is replaced with 10.0.0.5. Now Server 1 sees this network packet matches its own IP in the destination field, so it will accept it.



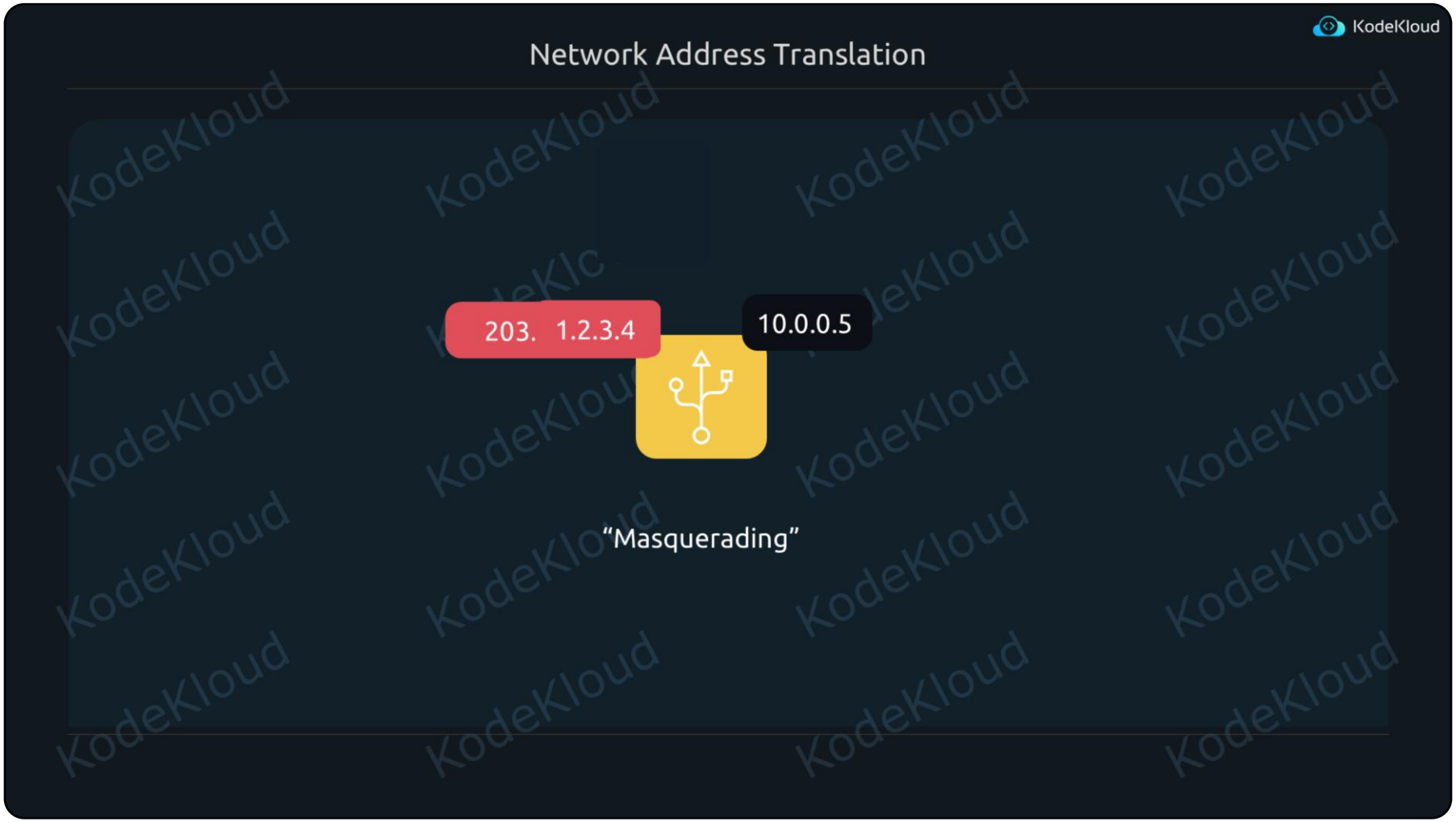
Up to this point, our public server just changed the destination address. But it might have to go one step further. Because the device from the Internet that sent this network packet had the IP 203.0.113.1. Which means 203.0.113.1 is still encoded as the source address. Private Server 1 did successfully receive this packet. But when it will try to send some data back, it will fail. Because it will attempt to send back to the source address, 203.0.113.1. And private Server 1 is not connected to the Internet, so it can't send data to that IP.

To solve this issue, our public server can replace the source address as well. Instead of keeping 203.0.113.1 as the source address, it can replace this with its own IP, 1.2.3.4. This way, when private Server 1 tries to respond to the forwarded network packet, it will see it has to send this back to our public server at 1.2.3.4. And since it's able to reach our public server, the operation will be successful this time. And with some additional NAT magic, our public server can even return this response to the original sender.

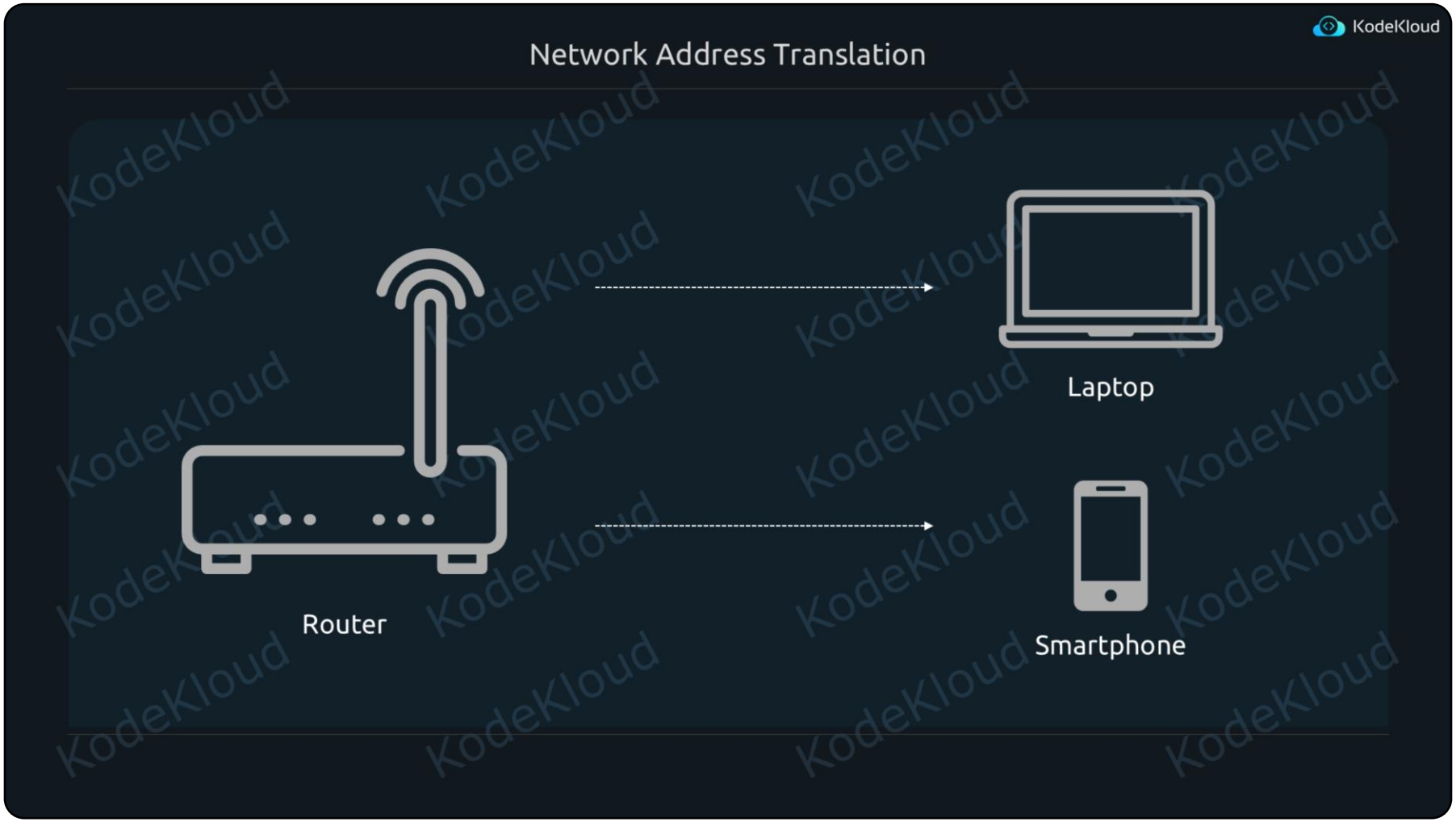


Up to this point, our public server just changed the destination address. But it might have to go one step further. Because the device from the Internet that sent this network packet had the IP 203.0.113.1. Which means 203.0.113.1 is still encoded as the source address. Private Server 1 did successfully receive this packet. But when it will try to send some data back, it will fail. Because it will attempt to send back to the source address, 203.0.113.1. And private Server 1 is not connected to the Internet, so it can't send data to that IP.

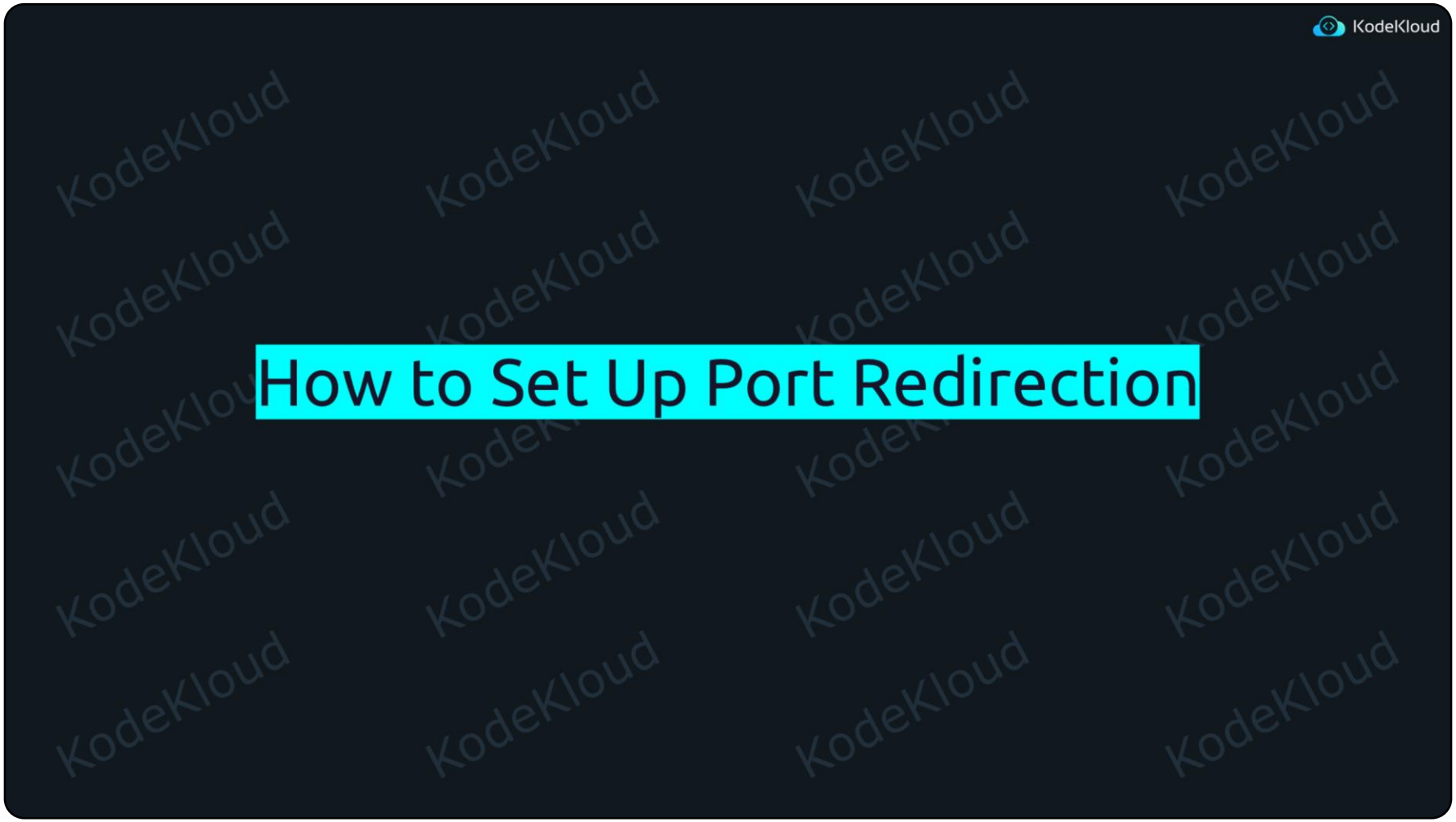
To solve this issue, our public server can replace the source address as well. Instead of keeping 203.0.113.1 as the source address, it can replace this with its own IP, 1.2.3.4. This way, when private Server 1 tries to respond to the forwarded network packet, it will see it has to send this back to our public server at 1.2.3.4. And since it's able to reach our public server, the operation will be successful this time. And with some additional NAT magic, our public server can even return this response to the original sender.



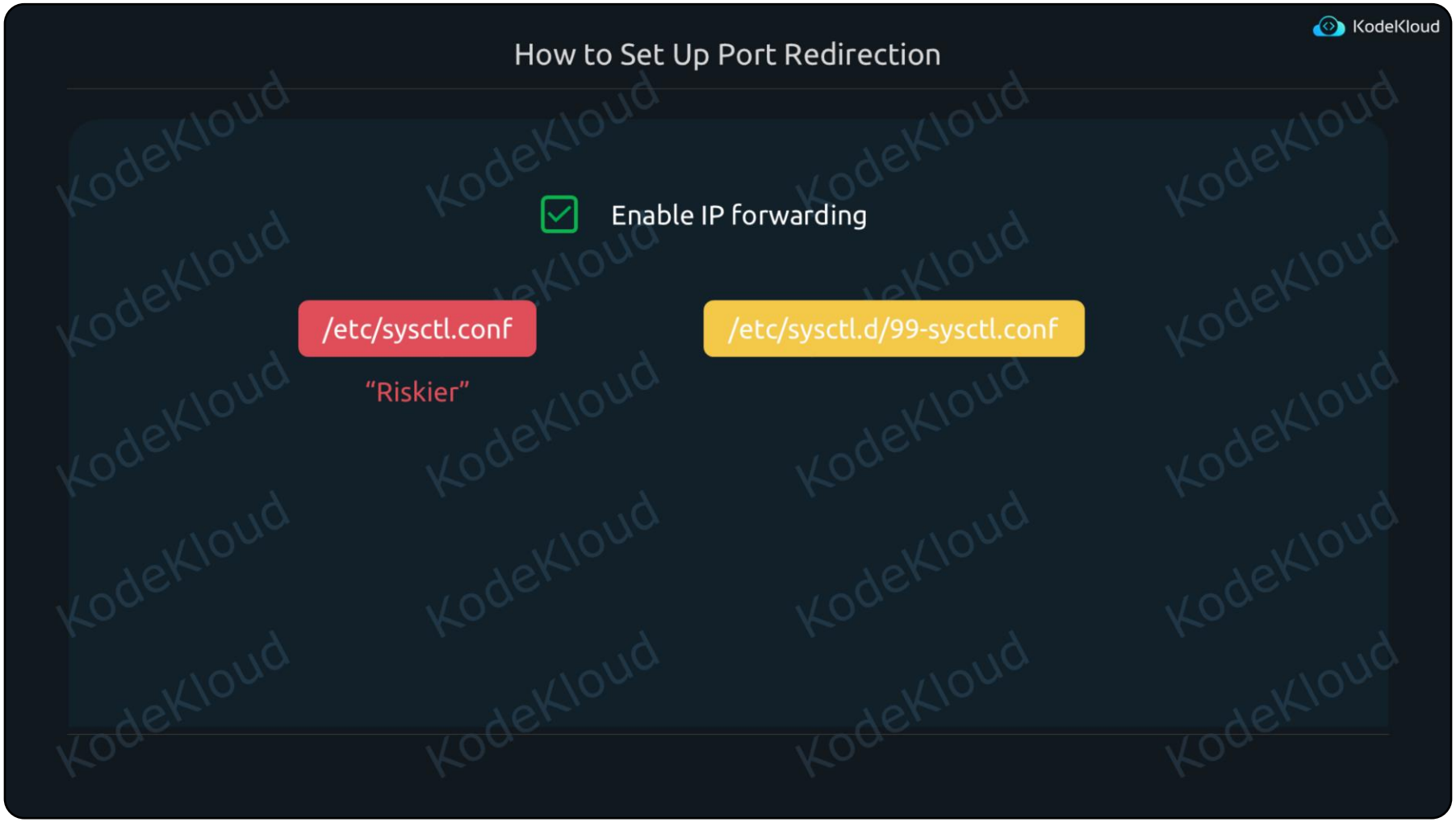
When a machine manipulates the source, or sender's IP address like this, it's effectively pretending to be someone else. That's why this technique is also called masquerading. In this case, when it's replacing 203.0.113.1 with its own IP address 1.2.3.4, it's effectively pretending to be that device from the Internet, sending a message to our private server.



As far as our private server is concerned, it will have no idea who the original sender was. But through NAT done in both directions, our public server will remember who sent what and where network packets need to go. The same thing that our routers at home do. When data is coming in on a router, it just knows what it should send to our smartphone, and what should be sent to our laptop. This is also done through Network Address Translation, and masquerading.



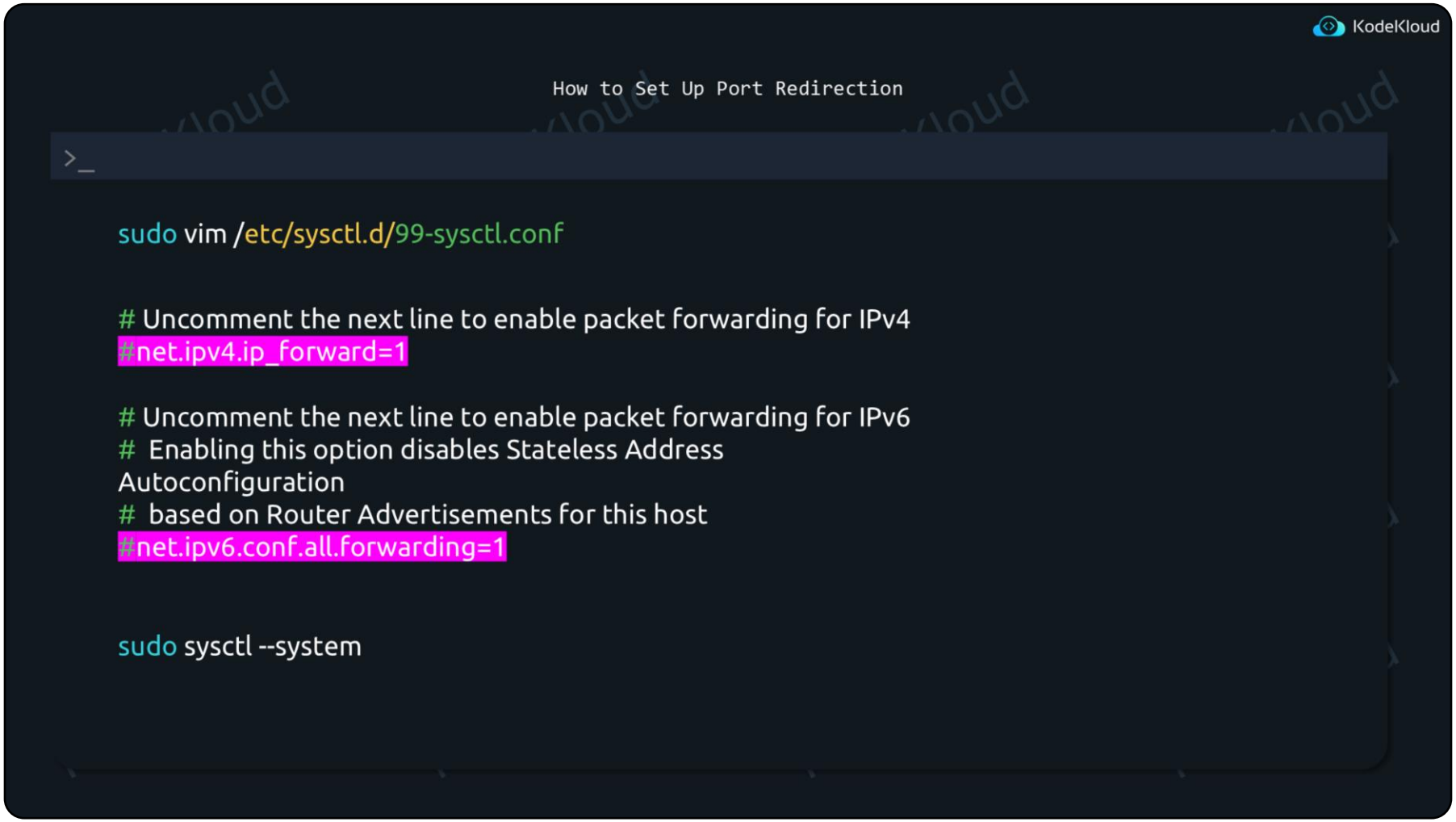
Now let's see how we can set up port redirections on Linux.



The first thing we'll need to do is enable IP forwarding on our machine. Since port redirection needs our machine to be able to forward network packets. And this ability is usually disabled by default.

On Ubuntu, we can enable this by editing one of two files -- either "/etc/sysctl.conf" or "/etc/sysctl.d/99-sysctl.conf". If possible, choose the second option. Editing "/etc/sysctl.conf" is riskier because future updates to system packages might modify this file, and we could end up losing our changes.

If we open the second file mentioned, to edit it, we can run our usual command:



sudo vim /etc/sysctl.d/99-sysctl.conf

And we'll notice these lines:

Uncomment the next line to enable packet forwarding for IPv4
#net.ipv4.ip_forward=1

```
# Uncomment the next line to enable packet forwarding for IPv6
#  Enabling this option disables Stateless Address Autoconfiguration
#  based on Router Advertisements for this host
#net.ipv6.conf.all.forwarding=1
```

If we uncomment the first one we will enable IPv4 forwarding. And uncommenting the second one enables IPv6 forwarding.

After we save the changes to this file, we must reload all sysctl config files with:

```
sudo sysctl –system
```

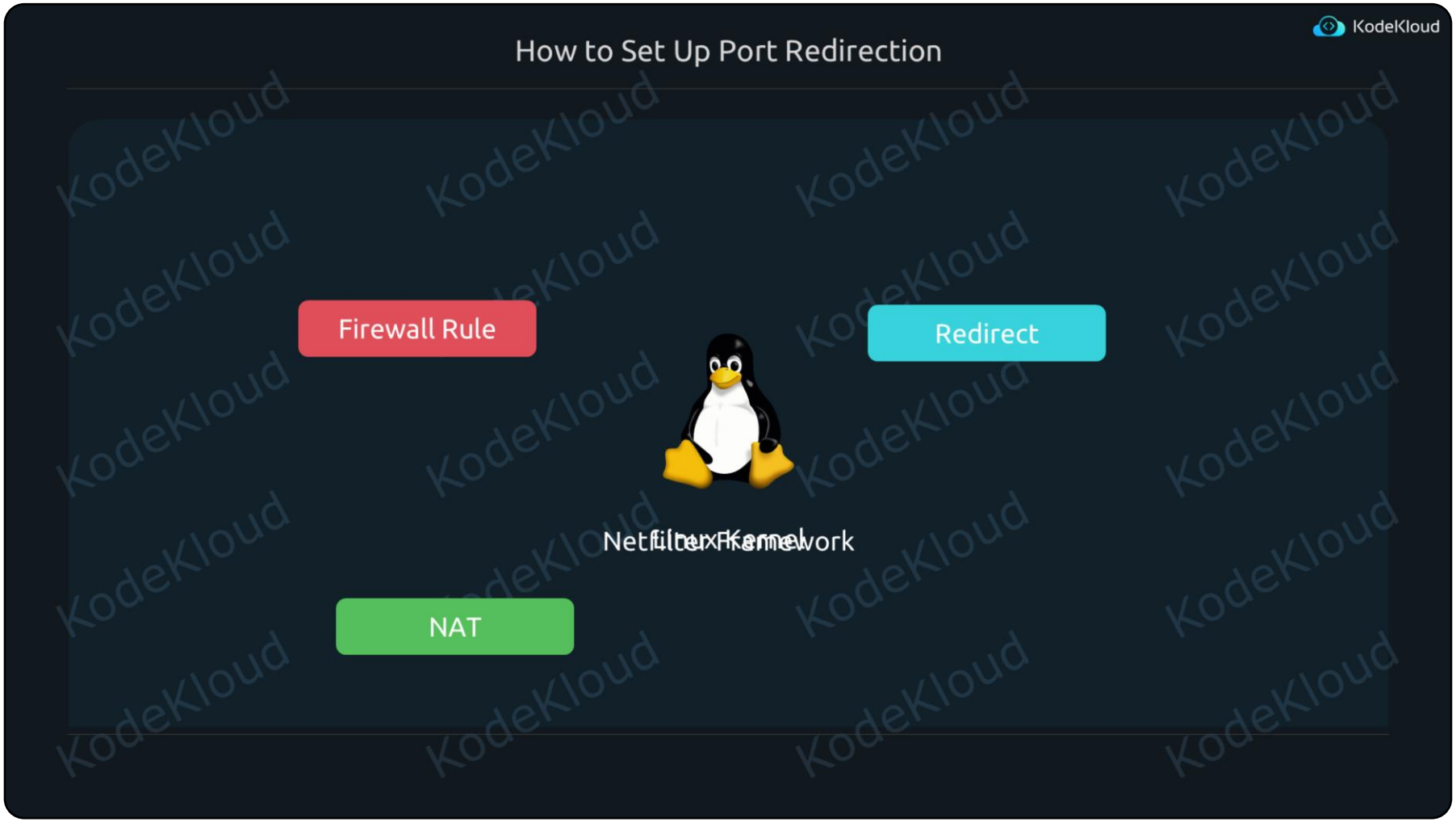
The --system parameter tells the "sysctl" command to look for configuration values in all the system default files and directories.

To check if our change took effect we can run:

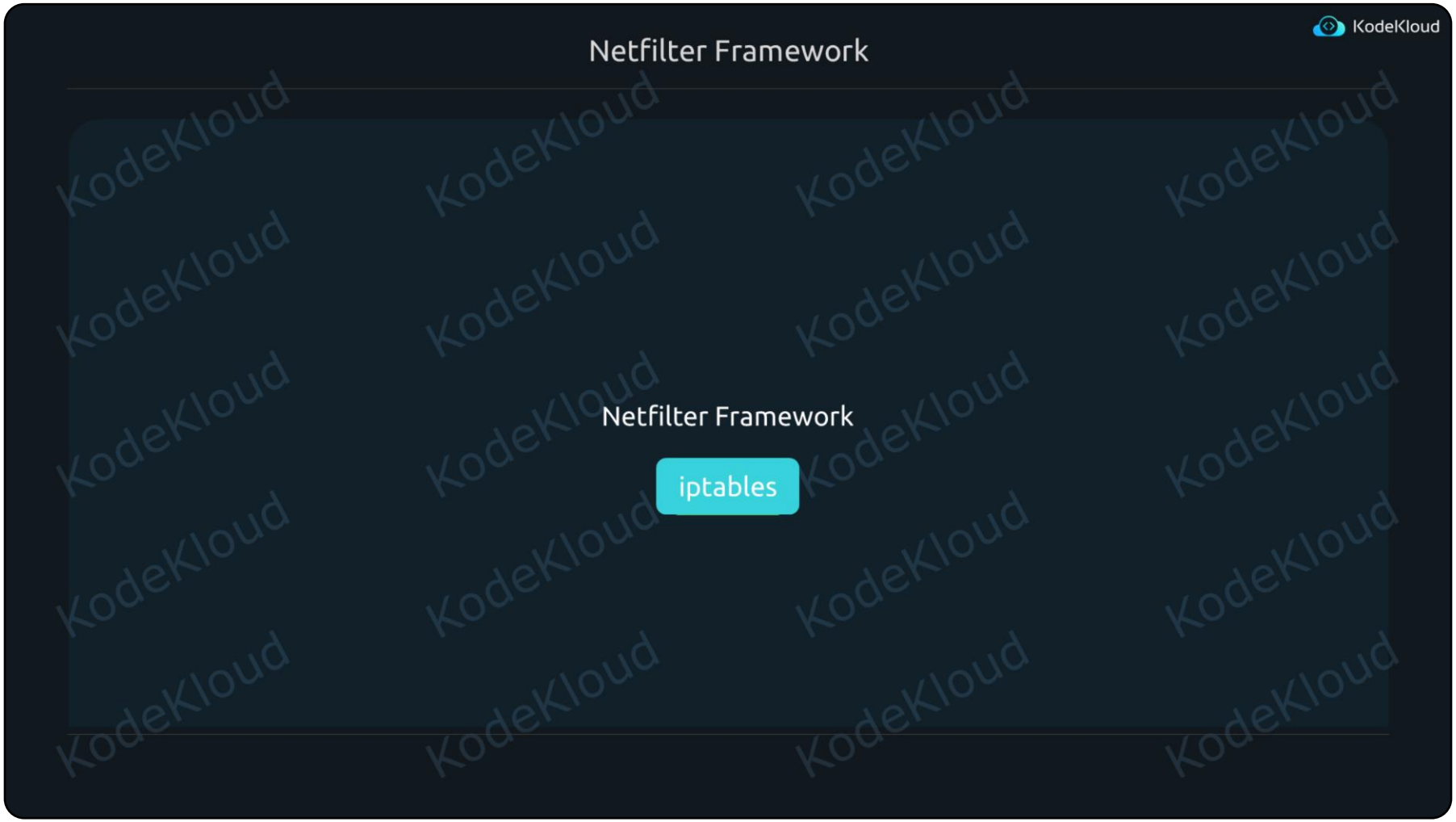
sudo sysctl -a | grep forward

And in the output we should check if both of these values are set to "1":

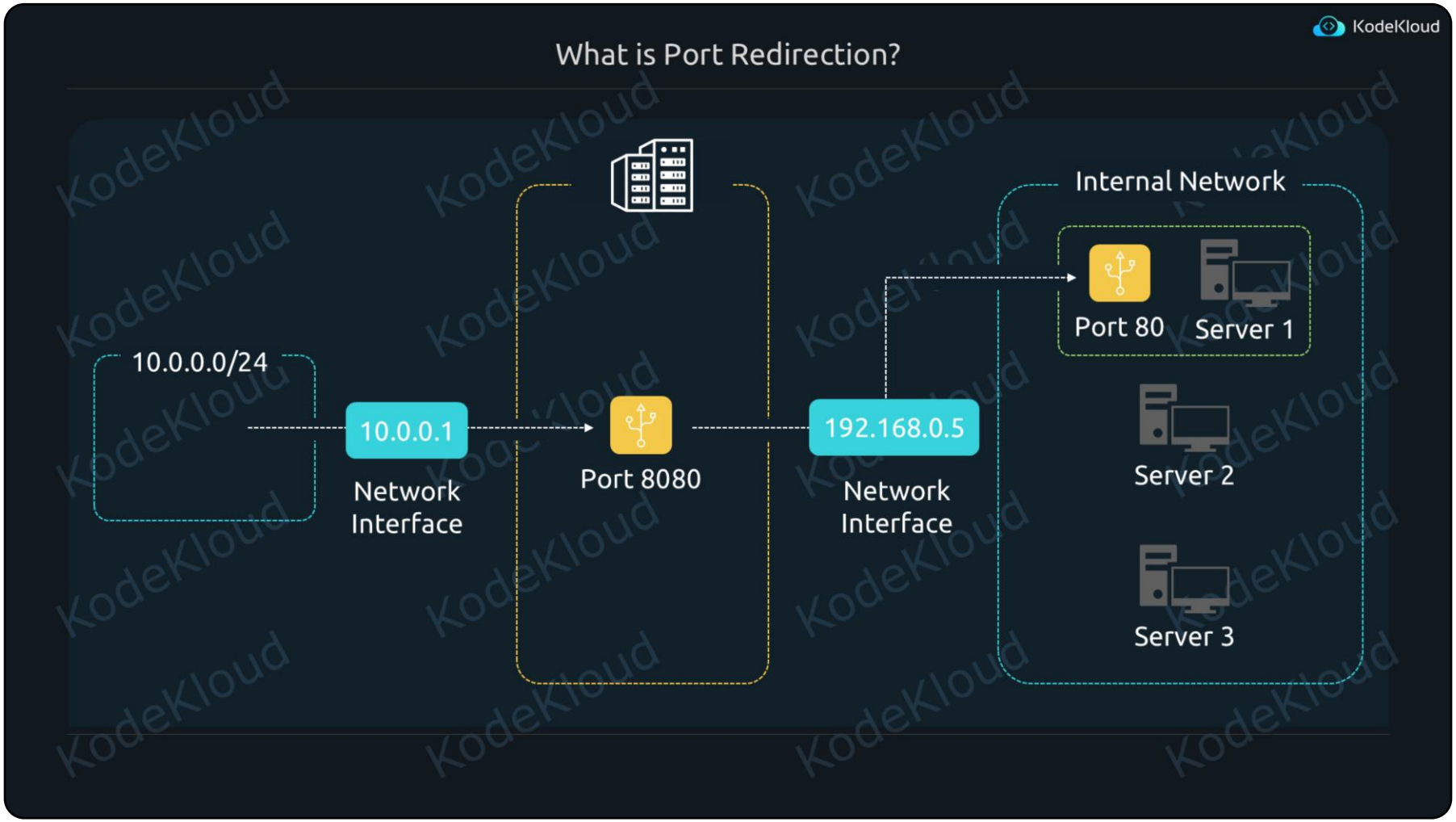
```
net.ipv4.ip_forward = 1
net.ipv6.conf.all.forwarding = 1
```



Now how do we set up port redirection? Well, first of all, all network data is initially handled by the Linux kernel. So that means that every firewall rule, every redirect, every Network Address Translation is handled by the kernel as well. Or at least, by part of it, called the Netfilter framework. And we have various commands to inform this framework about how we want it to process network packets.



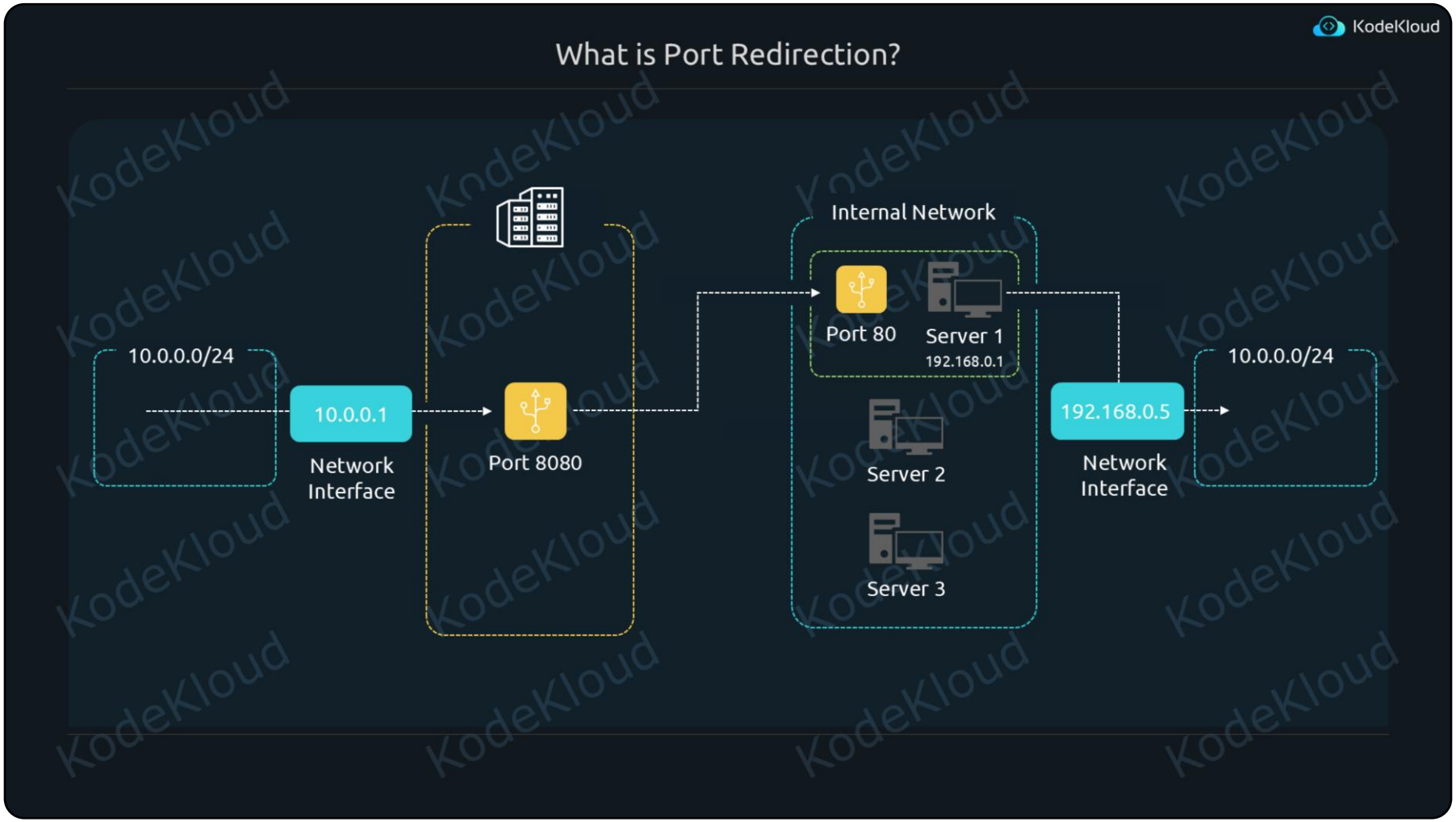
The modern command to interact with the Netfilter framework is "nft". Unfortunately, the nft commands we'd need to use to set this up are quite hard to remember without using Google. So in our examples, we'll use another command called "iptables". This is the predecessor of "nft", but it still works on modern distributions. In fact, most distributions are set up in such a way that iptables rules are automatically translated into nft rules. So even if we use the older command, things will still work as expected.



Now let's imagine this scenario: We want to redirect every request coming in from the 10.0.0.0/24 CIDR range. All data coming from that network to port 8080 on this machine should be redirected to port 80 on a remote server with IP address 192.168.0.5.

Before we can set this up, we'll want some additional data though. Namely, what network interfaces will be responsible for this redirection. Let's assume a scenario where two network cards are involved. One deals with the incoming traffic

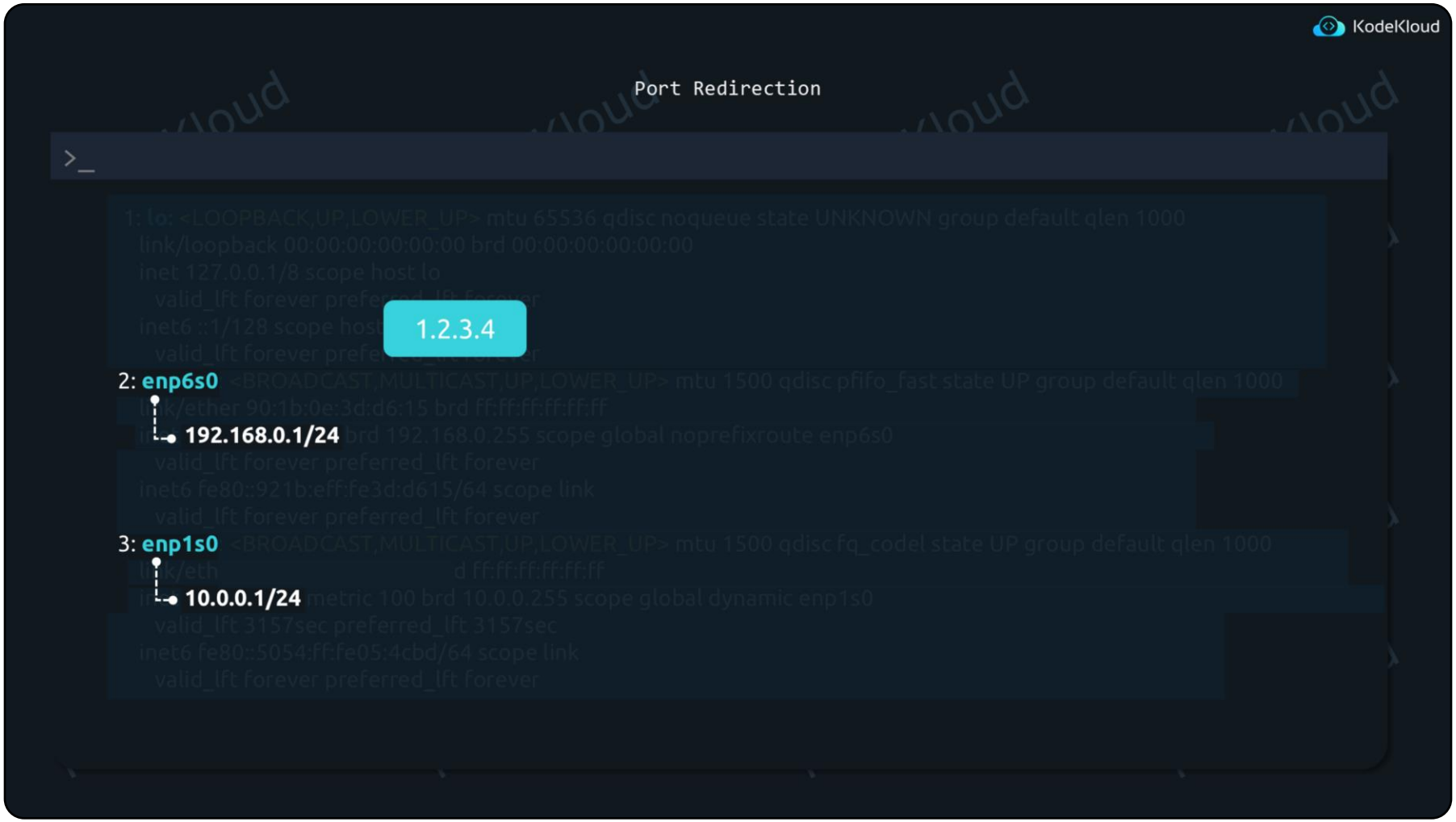
from the 10.0.0.0/24 network. And another deals with the outgoing traffic to 192.168.0.5, which sits on a different network.



Now let's imagine this scenario: We want to redirect every request coming in from the 10.0.0.0/24 CIDR range. All data coming from that network to port 8080 on this machine should be redirected to port 80 on a remote server with IP address 192.168.0.5.

Before we can set this up, we'll want some additional data though. Namely, what network interfaces will be responsible for this redirection. Let's assume a scenario where two network cards are involved. One deals with the incoming traffic

from the 10.0.0.0/24 network. And another deals with the outgoing traffic to 192.168.0.5, which sits on a different network.



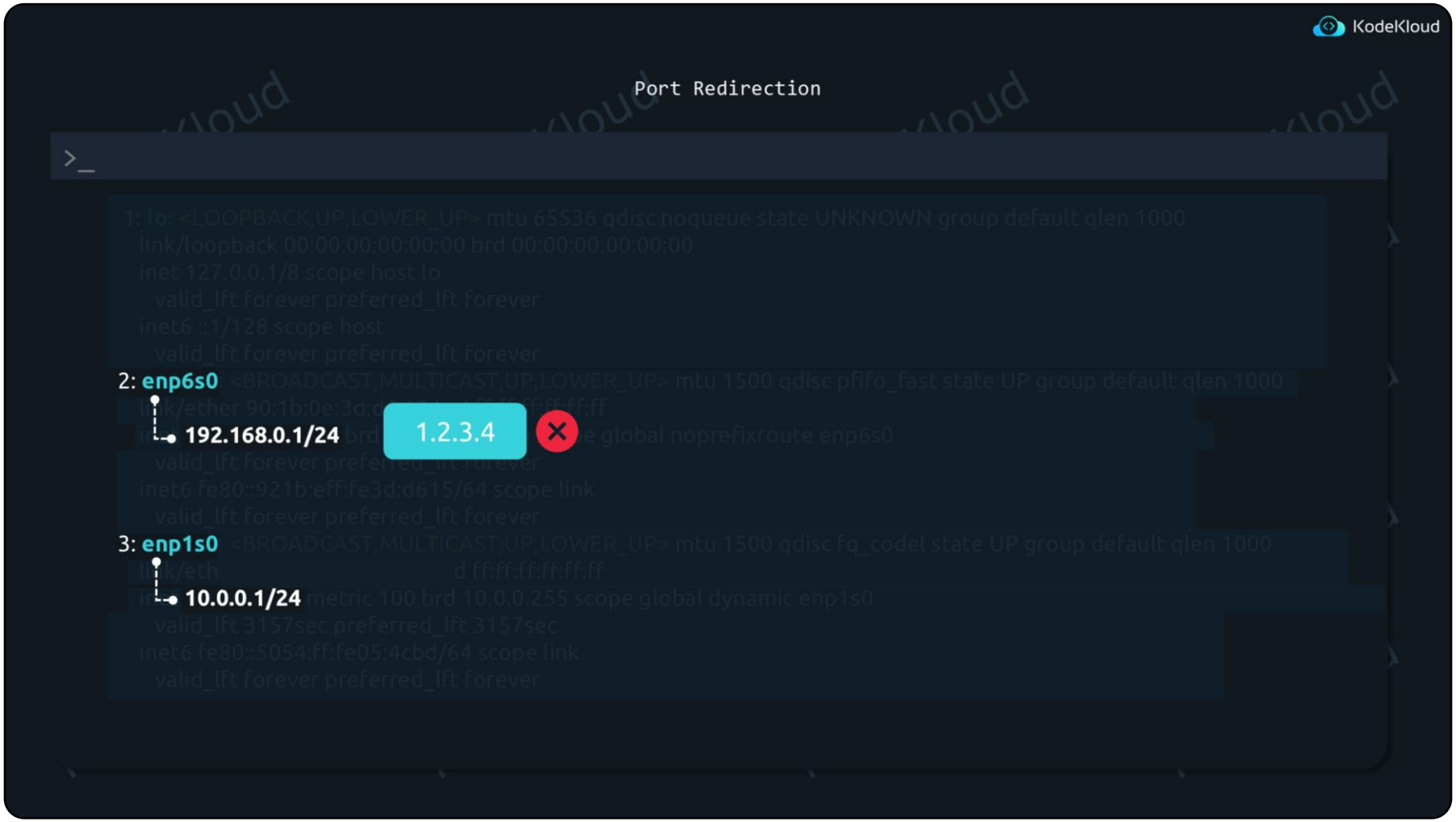
We can inspect our network addresses with an "ip a" command. And we might see something like this:

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever

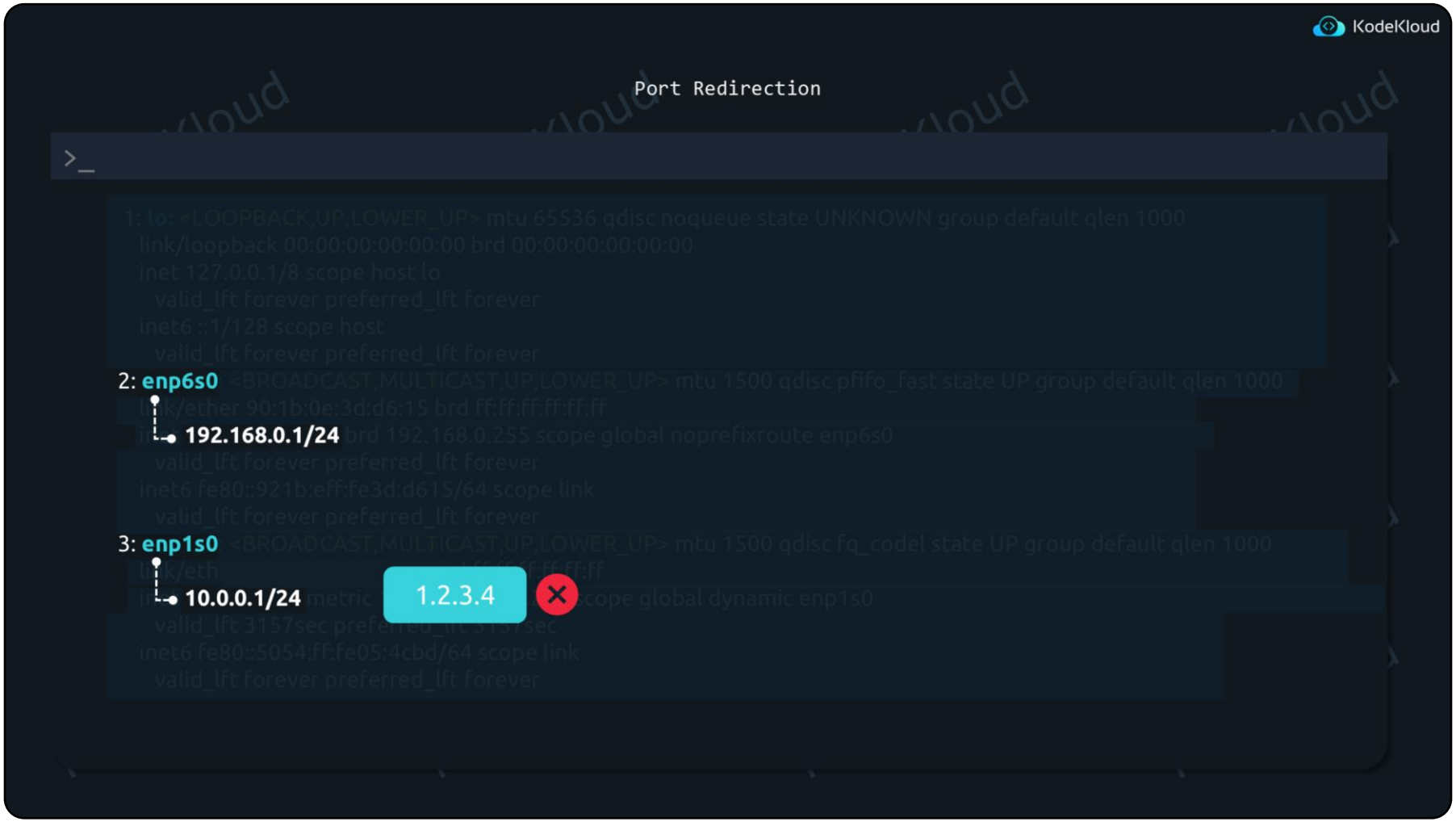
```
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp6s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen
1000
    link/ether 90:1b:0e:3d:d6:15 brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.1/24 brd 192.168.0.255 scope global noprefixroute enp6s0
       valid_lft forever preferred_lft forever
    inet6 fe80::921b:eff:fe3d:d615/64 scope link
       valid_lft forever preferred_lft forever
3: enp1s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen
1000
    link/ether 52:54:00:05:4c:bd brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.1/24 metric 100 brd 10.0.0.255 scope global dynamic enp1s0
       valid_lft 3157sec preferred_lft 3157sec
    inet6 fe80::5054:ff:fe05:4cbd/64 scope link
       valid_lft forever preferred_lft forever
```

This tells us that enp6s0 deals with the outgoing traffic that will be redirected to "192.168.0.5". And enp1s0 deals with the incoming traffic from the 10.0.0.0/24 network range.

But in some cases, we might need to do a redirect to an external network. Imagine redirecting to 1.2.3.4. This doesn't fit into any of the network ranges above.



It's not on 192.168.0.0/24,



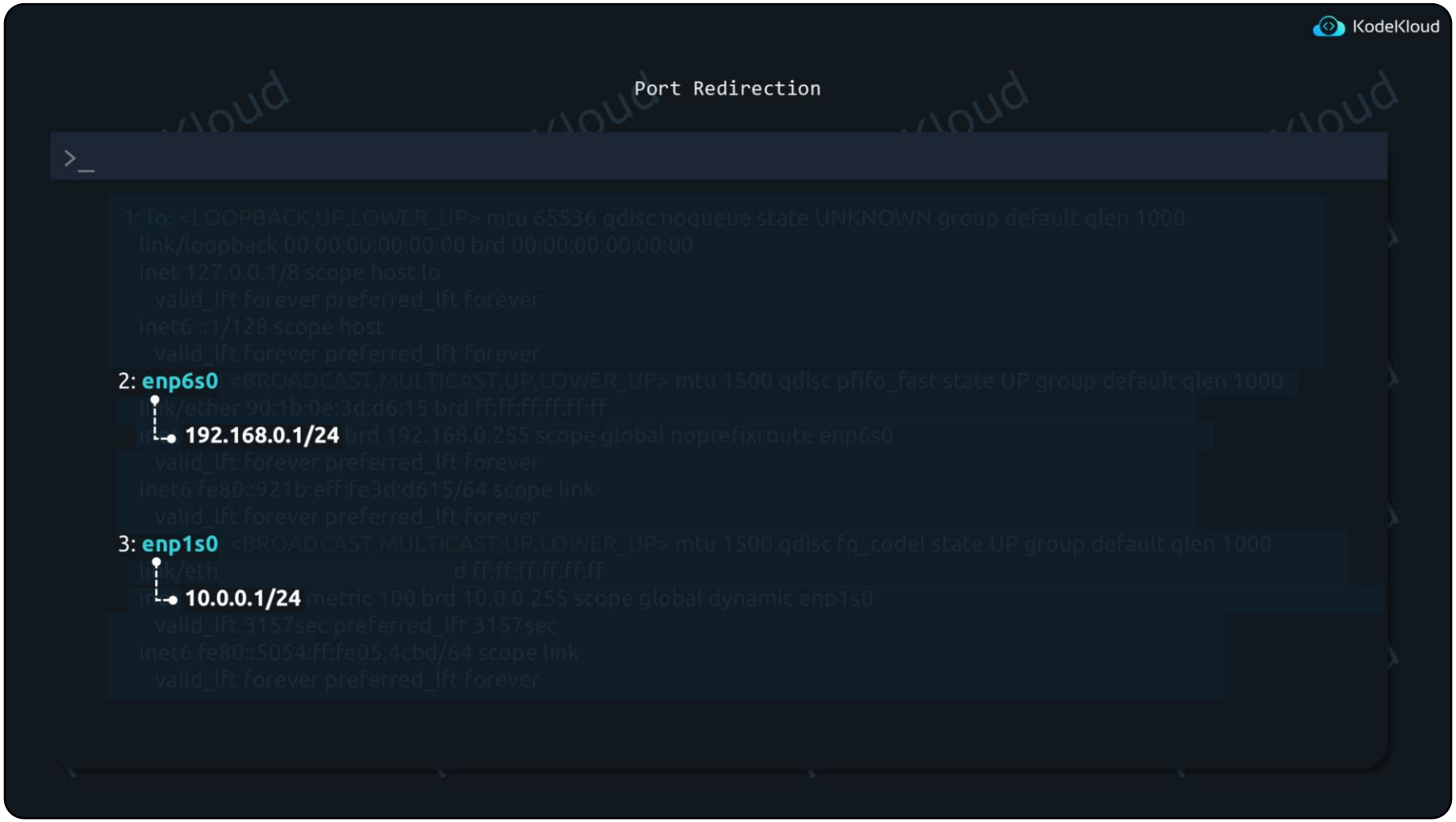
or on 10.0.0.0/24. So in that case, we can use a different trick. We can just look at the routes set up for the machine with:

ip r

And in the output, we'll search for the default route. We might see a line like this:

default via 10.11.12.1 dev enp1s0 proto dhcp src 10.11.12.14 metric 100

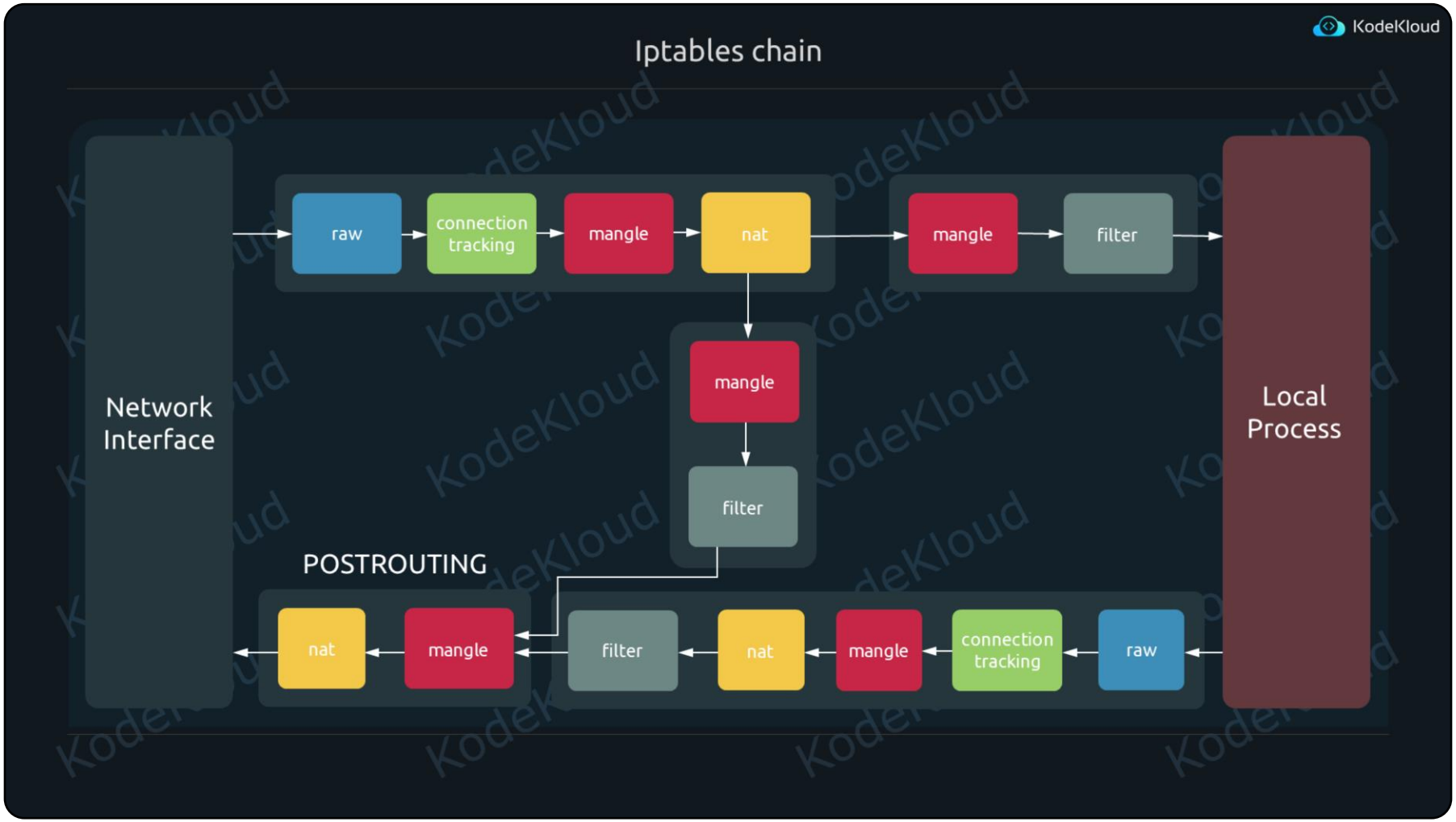
This tells us that all network packets to external networks are sent through this default gateway. And we can see that the device used is enp1s0. So we can conclude that enp1s0 will be used to send data to 1.2.3.4 when configuring our port redirection rule.



But let's get back to our example, where "enp1s0" is connected to the 10.0.0.0/24 network, and "enp6s0" should forward requests to 192.168.0.5.

Now that we have all the data we need, let's set up this redirect.

The first thing we should know though, iptables uses structures known as tables and chains. Here's a simplified diagram:

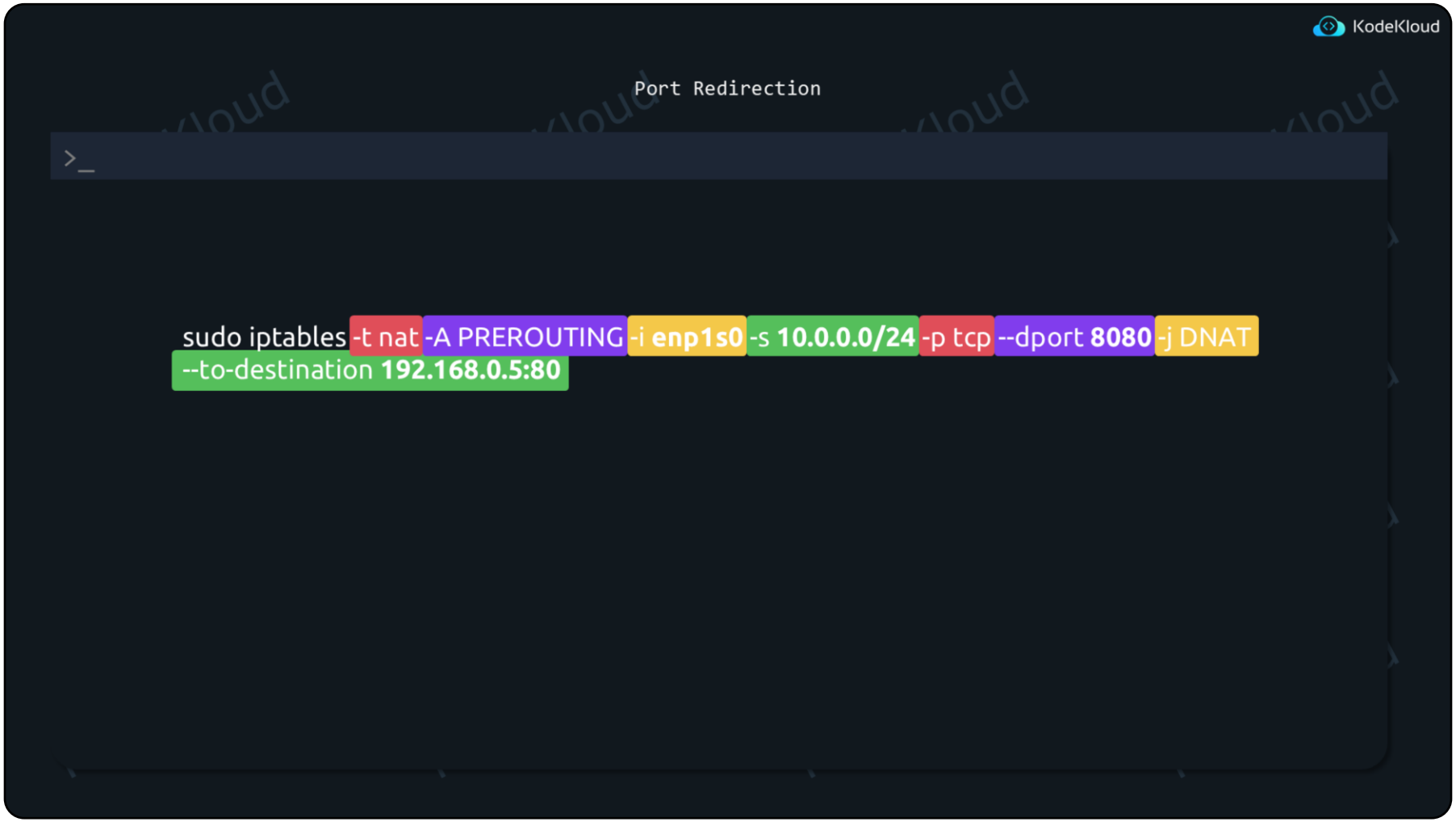


As a network packet comes in, or goes out, it travels through certain chains. And it's processed according to rules added in those chains. For example, right before a network packet goes out through a network card, it passes through the POSTROUTING chain.

You can think of an iptables chain in terms of when the network packet is processed. And you can think of tables as a way to group rules for a certain purpose. A table can reference multiple chains, but those chains will only be triggered during

a certain time of the network packet's journey.

Now let's see the command we need to set up our port redirect rule.



We'll type "sudo iptables" followed by the parameters we want:

"-t" stands for table. With -t nat we add this rule to the "nat" table. When a port is redirected, Network Address Translation is used, to change the initial destination IP and port, to a different IP and port. So whenever we need NAT, we have to add our rule to the "nat" table.

"-A" is the option for append. With -A PREROUTING we append another rule to the PREROUTING chain which deals with altering network packets as soon as they are received. Because the IP address will be modified, so this modification has to happen before the routing stage. Otherwise the packet would be routed to the wrong place, based on the old IP address instead of the new one.

"-i" stands for input. With -i enp1s0 we select enp1s0 as the input interface. This way, we only apply this rule when the packet is received on a specific interface.

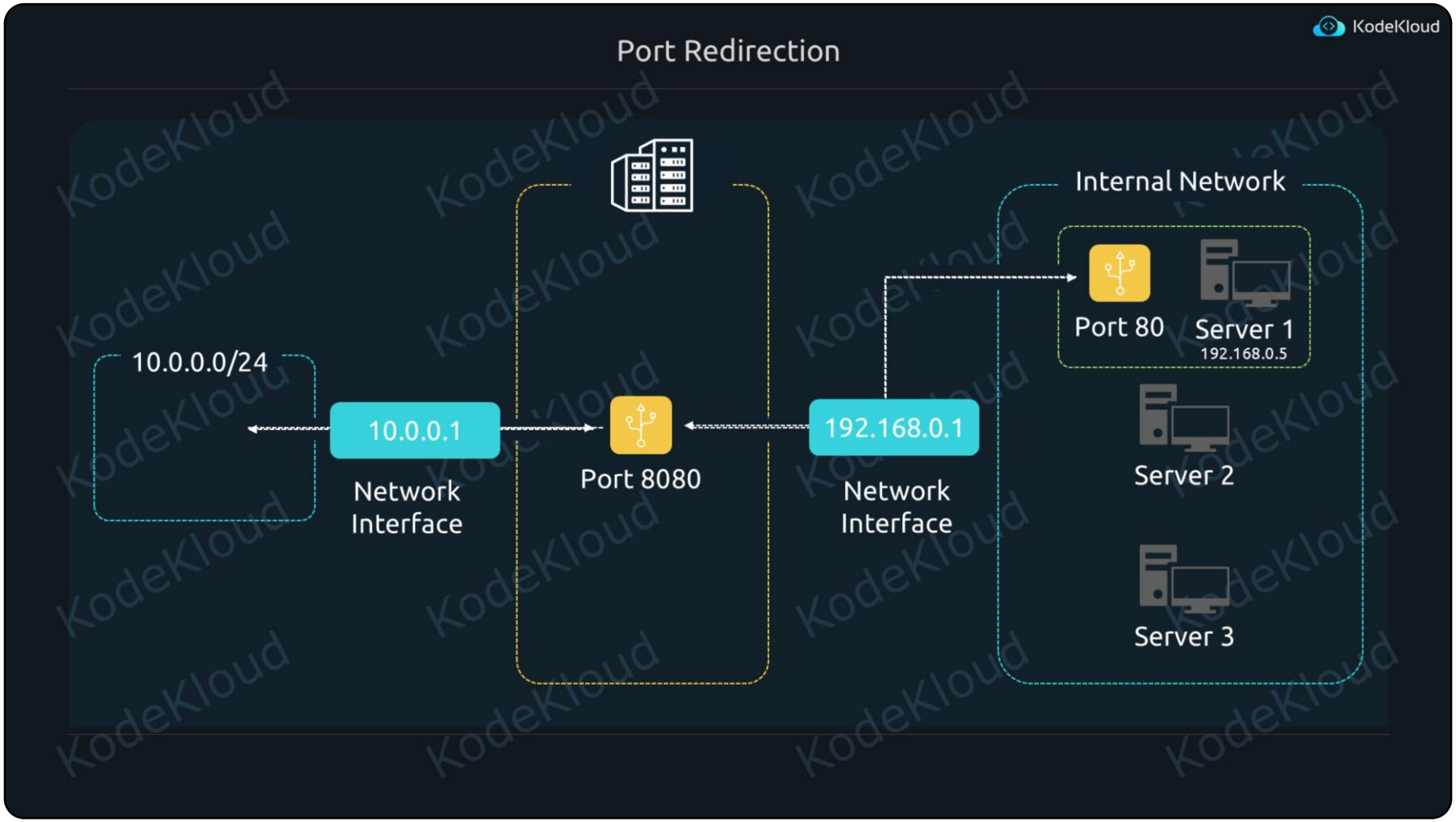
Next, with the -s option we pick the source IP. So we apply this rule only if the source IP matches anything in the 10.0.0.0/24 network range. Another way to say it is that we only redirect this packet if it comes from an IP in that range.

With -p we set the protocol to tcp. Another typical value we could use here is "udp" to add the rule for the udp protocol. It's mandatory to use this -p option for a port redirect rule. Without using "-p" we'd get an error saying that "--dport is an unknown option". Using --dport requires using "-p" as well. To set up a rule for both protocols, just copy and paste the same command. And in one use "-p tcp" while in the other change that to "-p udp" keeping all of the other options the same.

--dport sets the destination port to 8080. Meaning this redirect should happen when we receive packets on the 8080 port of this machine.

And the -j option is where the magic happens. -j tells our rule to "jump" this network packet to a so-called "target" in iptables terminology. You can think of it as sending this network packet to a mini-program, or mini-module of sorts. In this case, it's sent to an extension called DNAT, short for "destination NAT". This is the target extension that alters the packet to be redirected to a new IP and port.

Finally, --to-destination is an option we pass to the DNAT extension. So this option has to be added after the "-j DNAT" option. Here, we simply instruct the DNAT extension to redirect this packet to the 192.168.0.5 IP address, and port 80 of that machine.



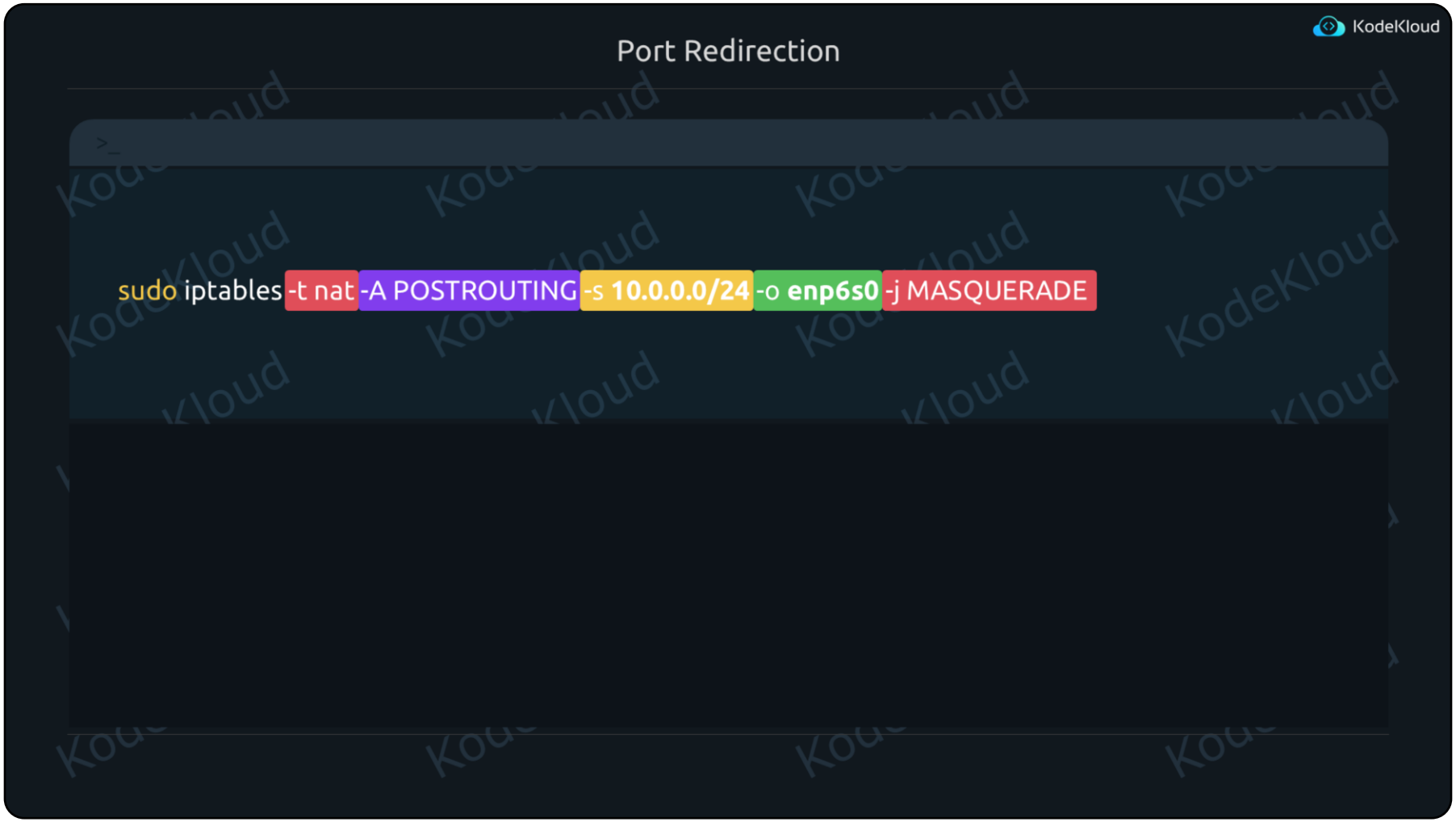
At this point, any external packet coming to port 8080 on our machine will be forwarded to this remote server, on port 80.

Please note that if you want to test this, the redirect works only when connecting from an external machine. It does not work if you connect from the same machine. The PREROUTING chain only applies to packets received from the outside world.

But even if we'd connect from an external machine, we'd notice the connection is initiated, but then it hangs. Which makes sense. Because even though network packets are forwarded, they have no way of coming back. If an IP like 10.0.0.9 sends something to our server on port 8080, it will reach 192.168.0.5, on port 80. But 192.168.0.5 would see that 10.0.0.9 sent this. When it tries to send data back, it won't be able to reach the 10.0.0.0/24 network.

So we need an additional command for our server to step in and solve this issue. We need to tell it to "pretend" that the data it's redirecting isn't coming from 10.0.0.9, but rather from its own IP address, like 192.168.0.1. Which means that when it passes the data along to 192.168.0.5, then that server will be able to return a response. Because it knows how to contact our server at 192.168.0.1, since they share a common network. Now our server receives that response and passes it back to 10.0.0.9. So communication is made possible with our server acting as a middleman, and changing the IP headers of network packets on the fly, so that they can reach the intended network devices.

iptables calls this "masquerading" which means "pretending you're someone else".



To activate masquerading, we can begin typing "sudo iptables" followed by these options:

-t nat to work with the NAT table once again.

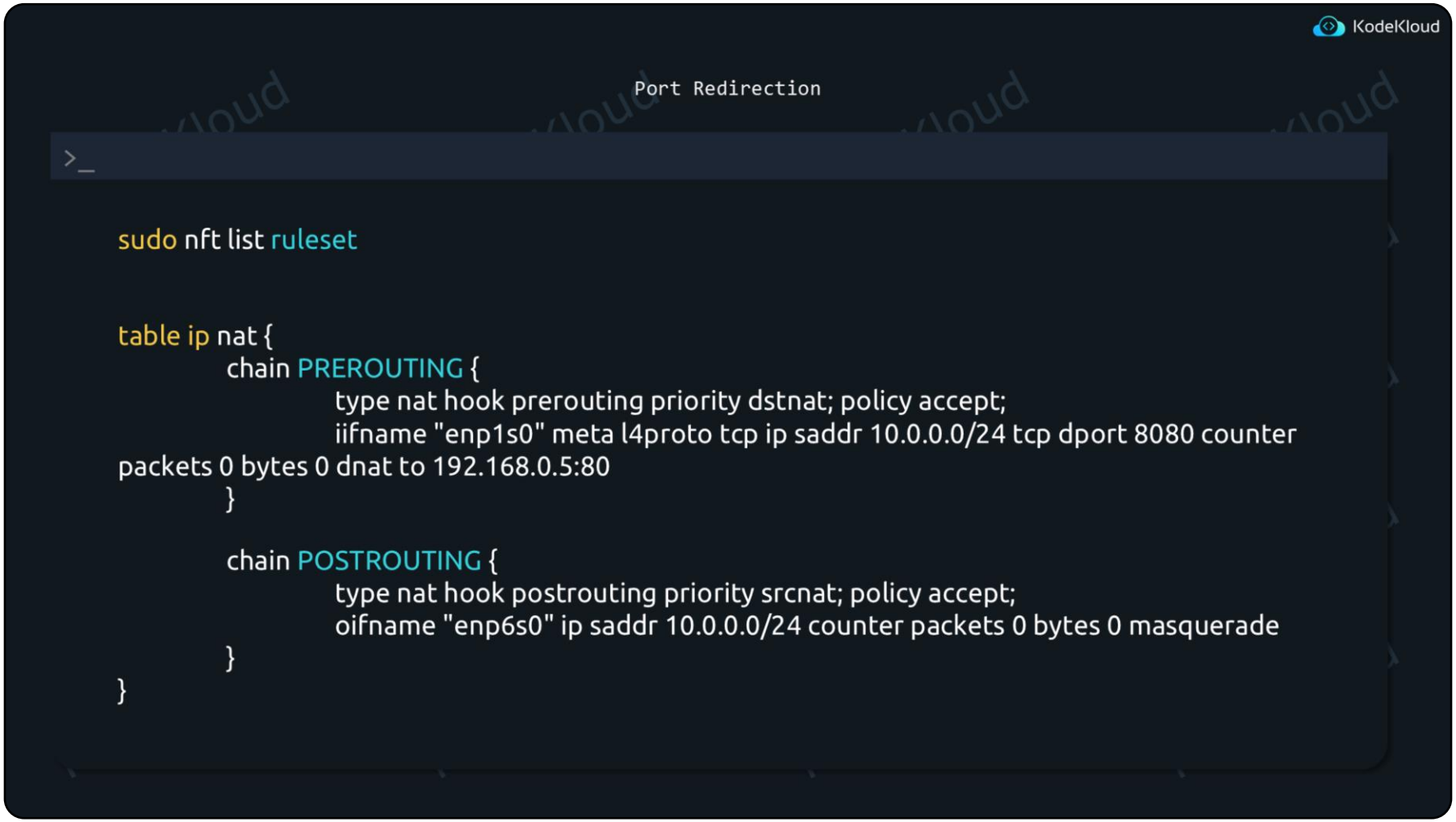
But this time, with -A we won't append to the prerouting chain, but to the POSTROUTING chain, which deals with altering network packets just before they leave the network.

With -s we do the same thing as before. And apply this rule only if the source IP that sent this packet falls in the 10.0.0.0/24 range.

And what's important to note, with -o this time we select the output interface, not an input interface as we did in the redirection rule. Namely enp6s0 which is used to send out network packets. An easy way to remember this difference is that in PREROUTING you add the network interface that receives the packet, and in POSTROUTING you add the interface that sends out the packet.

Finally, we add the -j option to jump, or send this network packet to the MASQUERADE extension this time.

Note that we don't need to specify the IP of our server. It will do its NAT magic and replace the source IP with its own IP dynamically. It will simply look at what IP it has to use to send this network packet out, and replace the initial source IP of the packet, with its own IP address.



As you can see, these iptables commands can get a bit complex. But here's a contrast with nft. We can run this to list nft rules:

sudo nft list ruleset

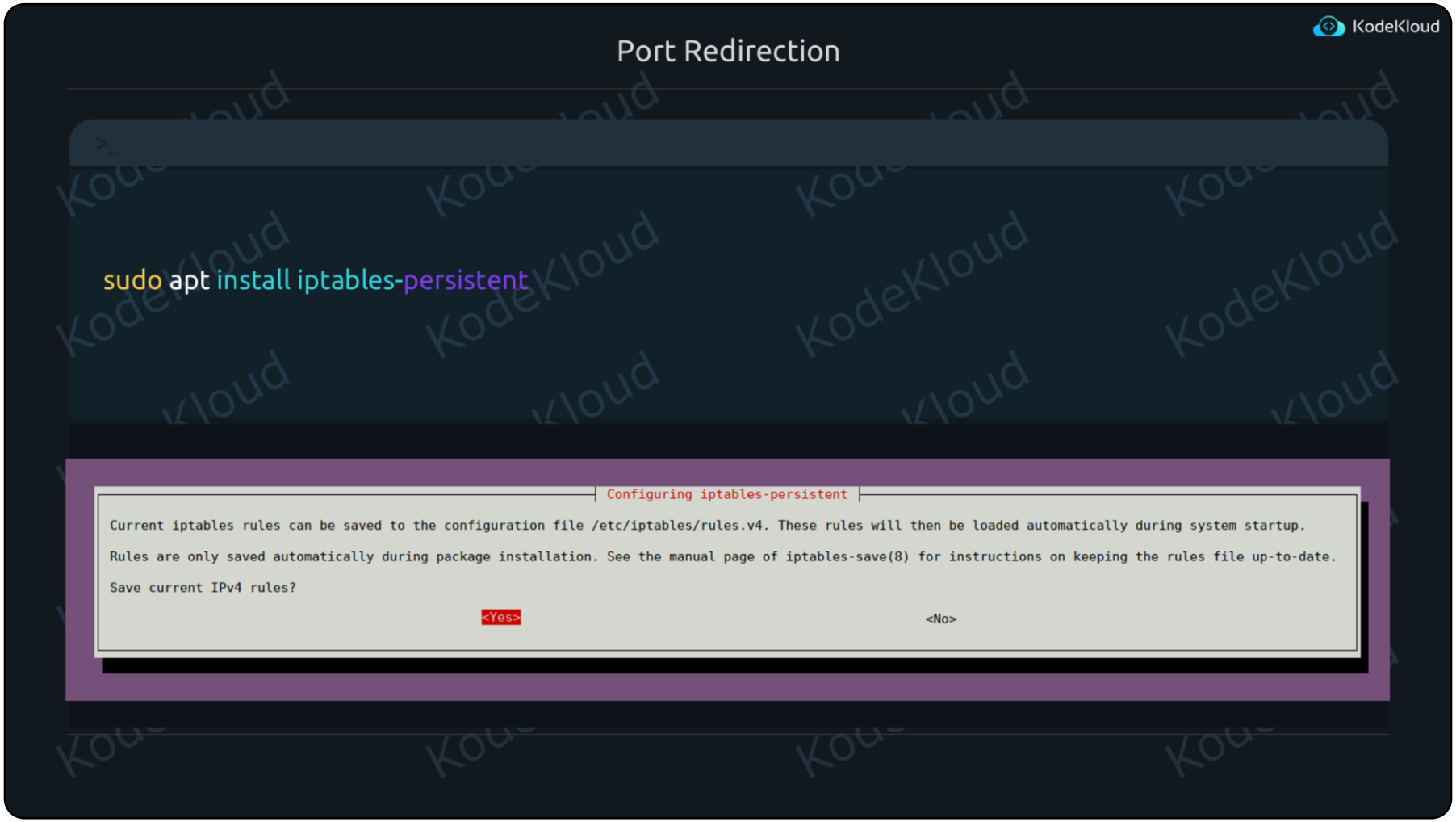
And we'll see this output:

```
table ip nat {
chain PREROUTING {
type nat hook prerouting priority dstnat; policy accept;
iifname "enp1s0" meta l4proto tcp ip saddr 10.0.0.0/24 tcp dport 8080 counter packets 0 bytes 0 dnat to
192.168.0.5:80
}

chain POSTROUTING {
type nat hook postrouting priority srcnat; policy accept;
oifname "enp6s0" ip saddr 10.0.0.0/24 counter packets 0 bytes 0 masquerade
}
}
```

So iptables is the easier solution, at least for this purpose. Creating that nft structure from zero, without any help from the Internet, would be much harder.

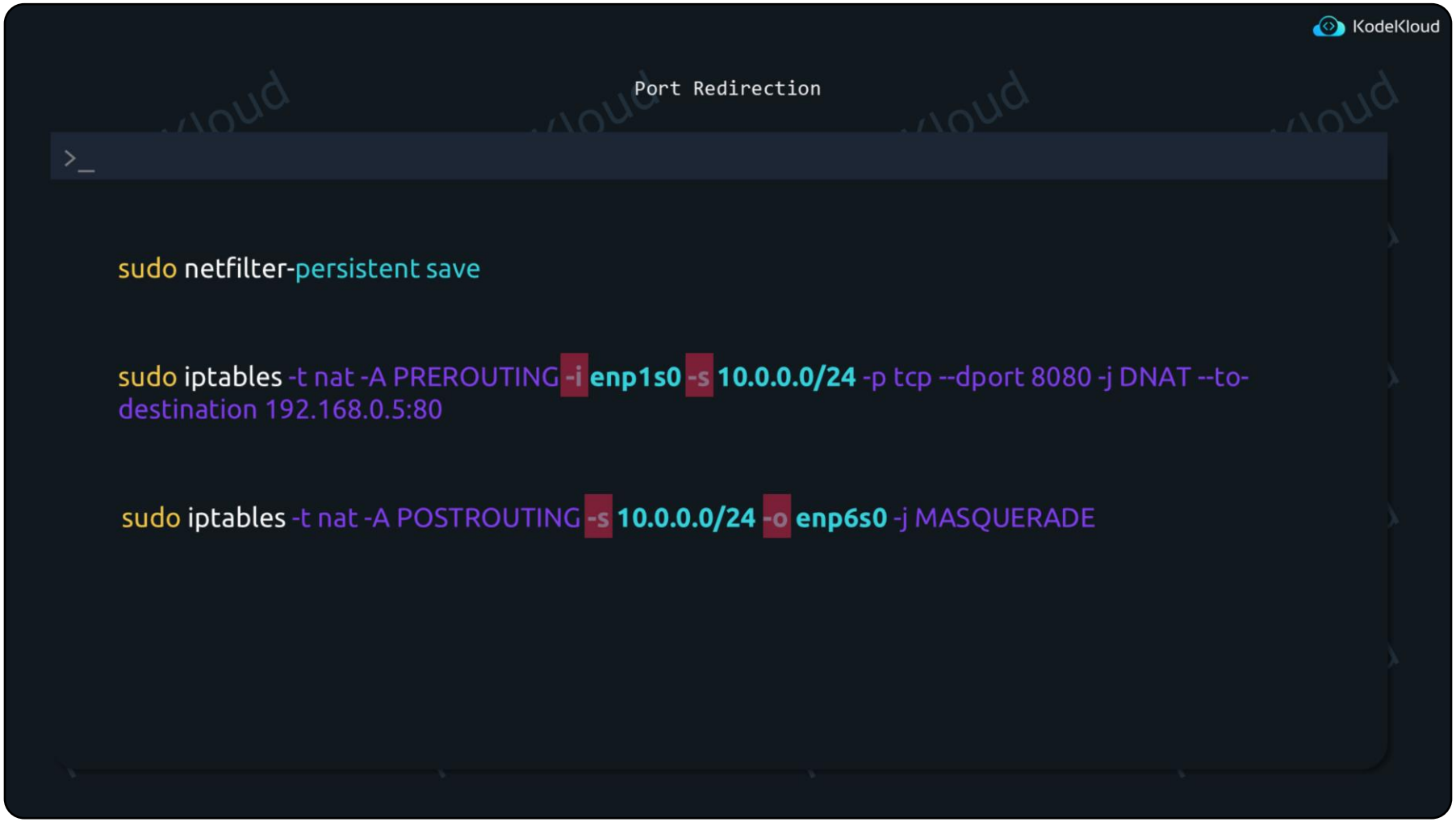
Also, notice how Ubuntu tools already converted our iptables rules to nft rules.



It's useful to know that these rules are temporary and will be lost if we reboot. To make them permanent, we can install this package:

sudo apt install iptables-persistent

We'll be asked if we want to save the rules, and we can pick the default "Yes" answer:



After this package is installed, whenever we change iptables or nft rules again, we can save them with the command:

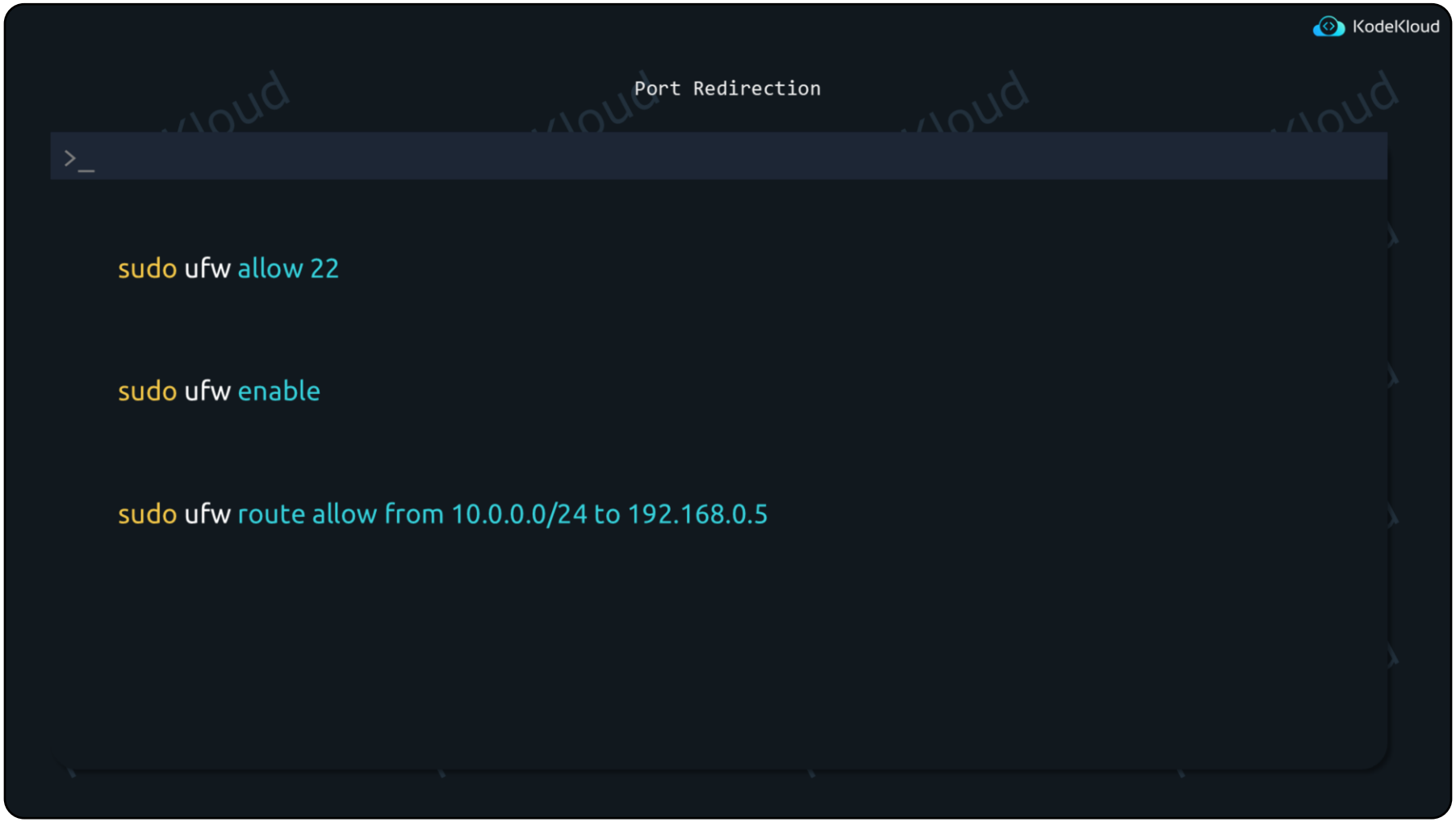
sudo netfilter-persistent save

Now let's get back to the two iptables commands we used

sudo iptables -t nat -A PREROUTING -i enp1s0 -s 10.0.0.0/24 -p tcp --dport 8080 -j DNAT --to-destination 192.168.0.5:80

sudo iptables -t nat -A POSTROUTING -s 10.0.0.0/24 -o enp6s0 -j MASQUERADE

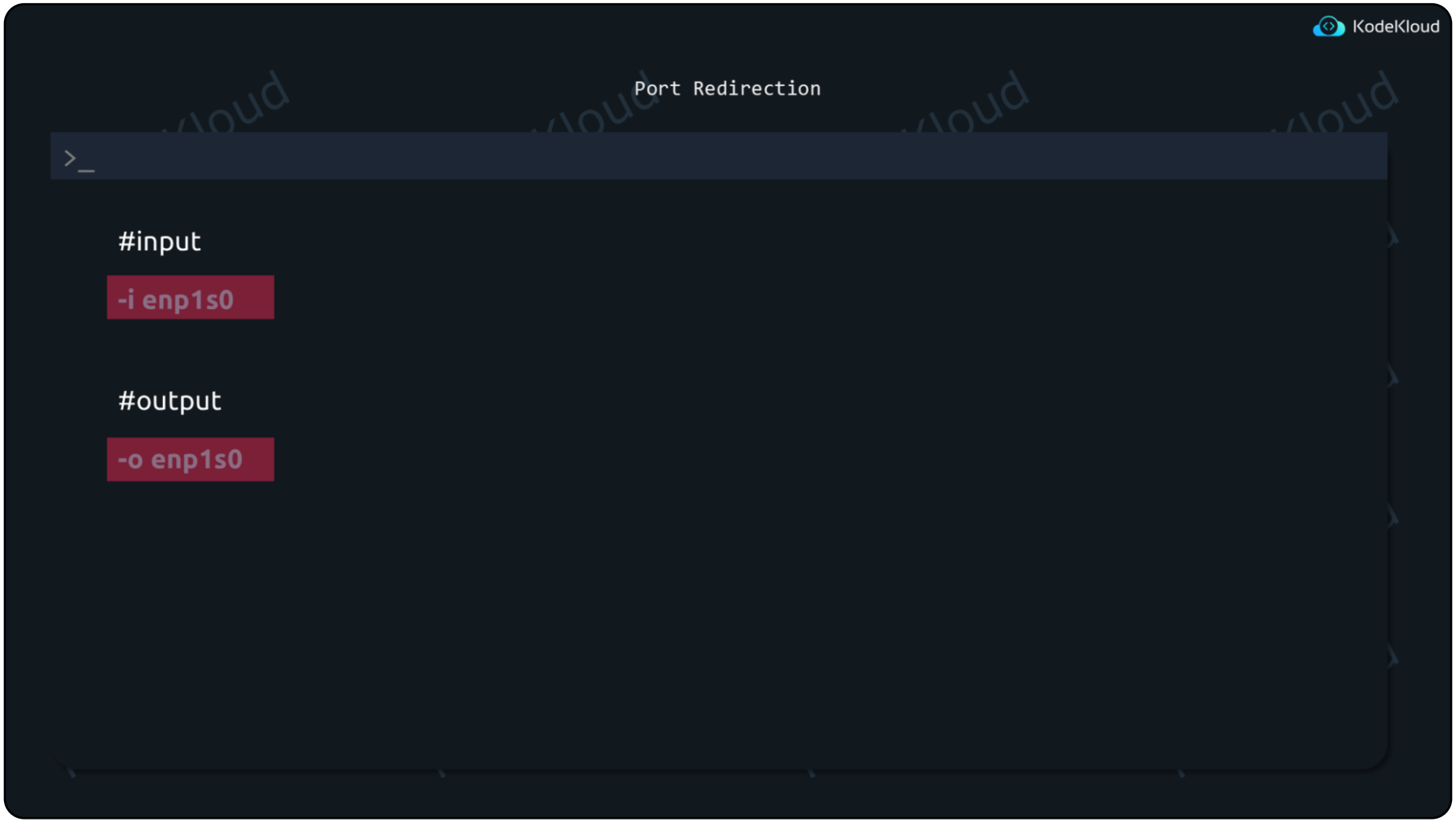
It can be useful to know that the options highlighted are optional. We can omit -i, -o, and -s. So we don't really need to specify a network input, or output interface, and a source IP, or IP range. For experimental purposes at least, they can be omitted. But in a real scenario, you would want to use them to limit who can forward what. Otherwise your machine can be abused by anyone on the Internet since it will forward network data unconditionally. In fact, you might want to apply additional firewall rules to limit network forwarding even more and prevent any misuse from unauthorized devices.



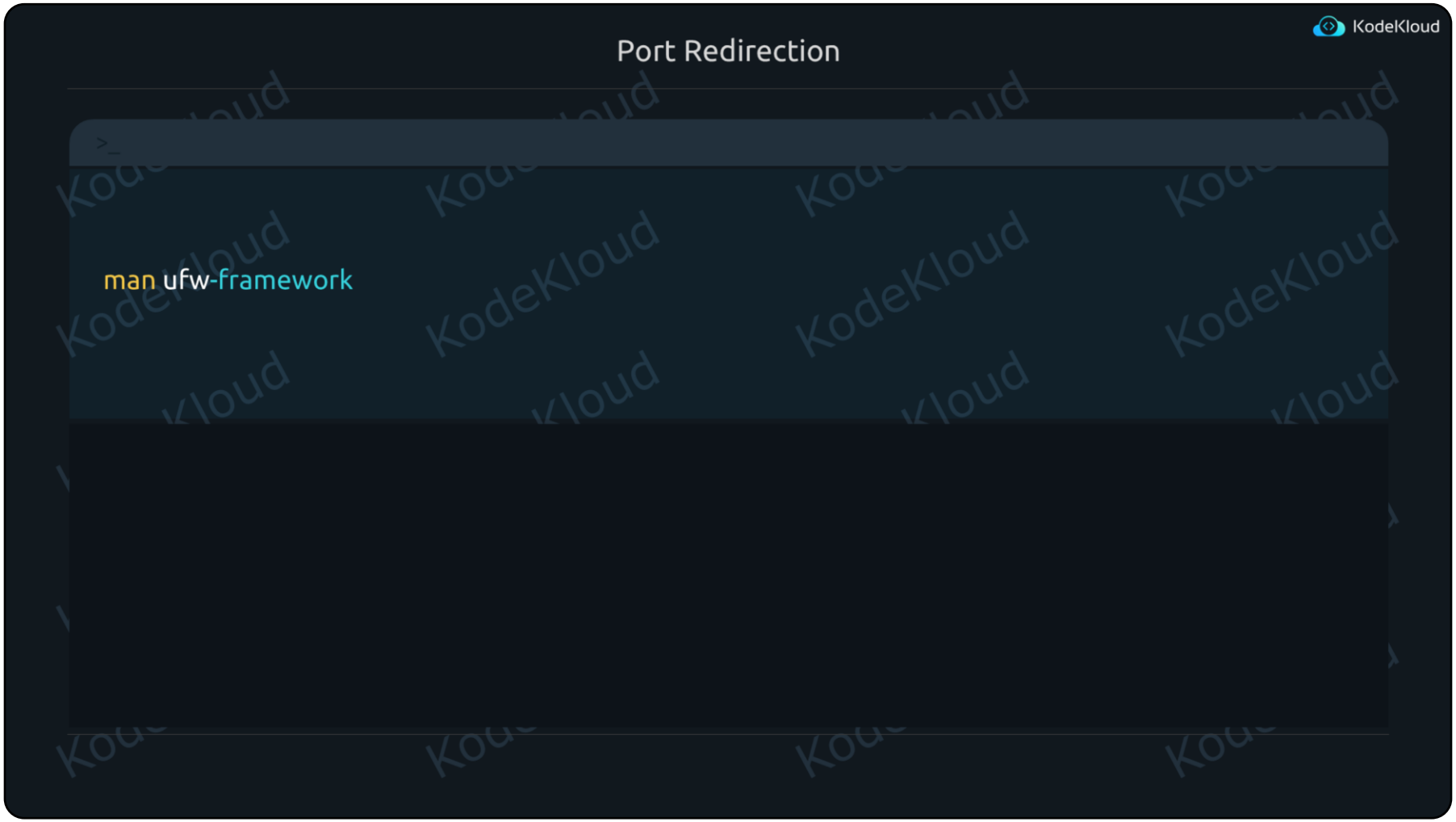
Here's a quick example of how we could further lock this down. When enabled, the default behavior of UFW is to disallow any kind of forwarding. So we can run something like this:

sudo ufw allow 22
sudo ufw enable
sudo ufw route allow from 10.0.0.0/24 to 192.168.0.5

First we would allow SSH traffic to port 22, then enable the firewall, and finally, add a route rule to allow network packets to be forwarded from the 10.0.0.0/24 network range to the 192.168.0.5 IP address.

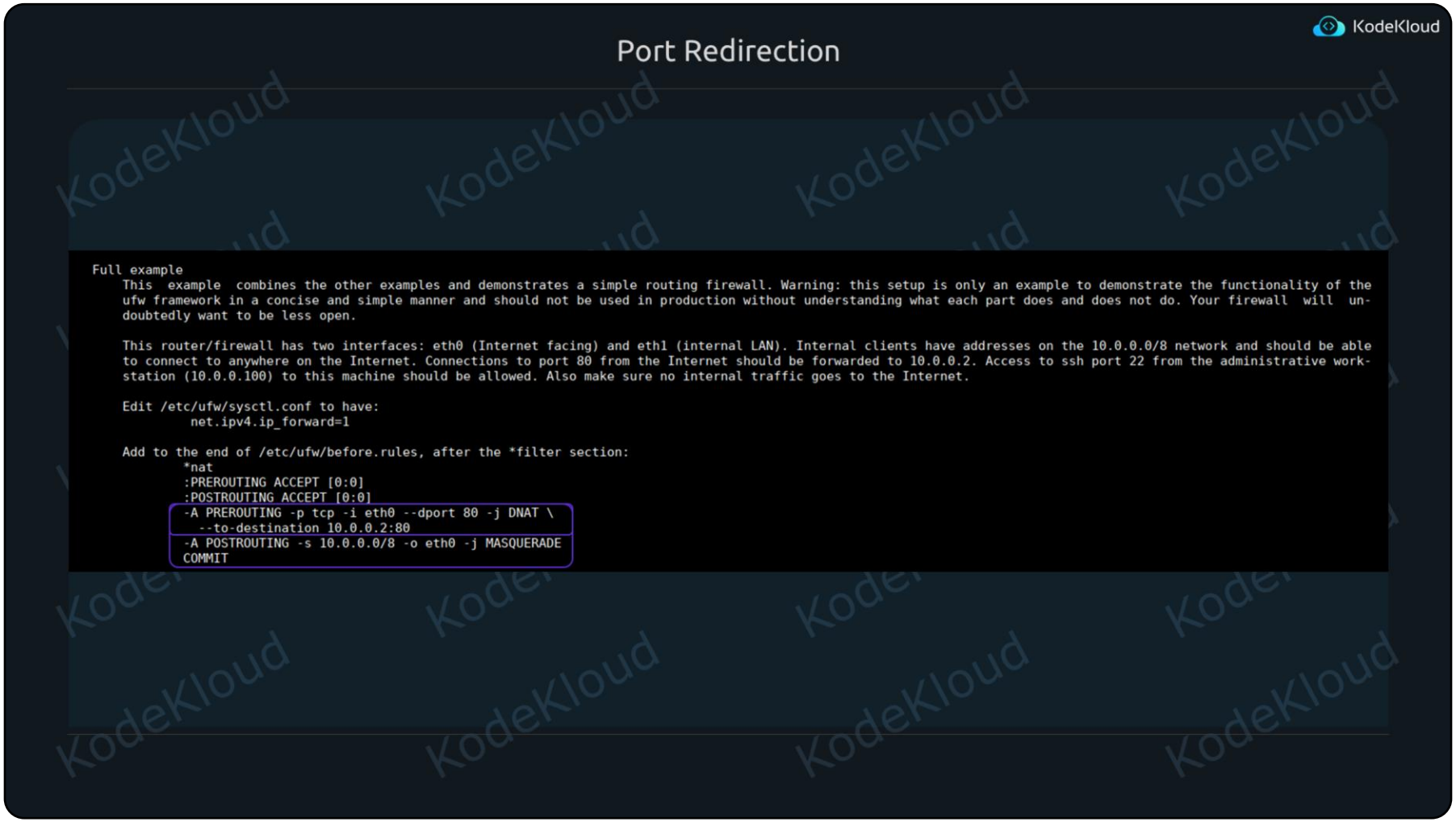


Another thing worth mentioning is that in some cases the input and output network interface can be the same one. So you could have -i enp1s0 in the first command, and -o enp1s0 in the second one. Or you can simply omit mentioning input and output interfaces. Omitting them means the redirect rule will be applied no matter what interface receives or sends out the network packet. Which is not ideal in a real scenario, but good enough for experimental purposes.



If you forget the correct iptables commands here's a way you can get a quick reminder. You can consult the manual for Ubuntu's Uncomplicated Firewall Framework:

man ufw-framework

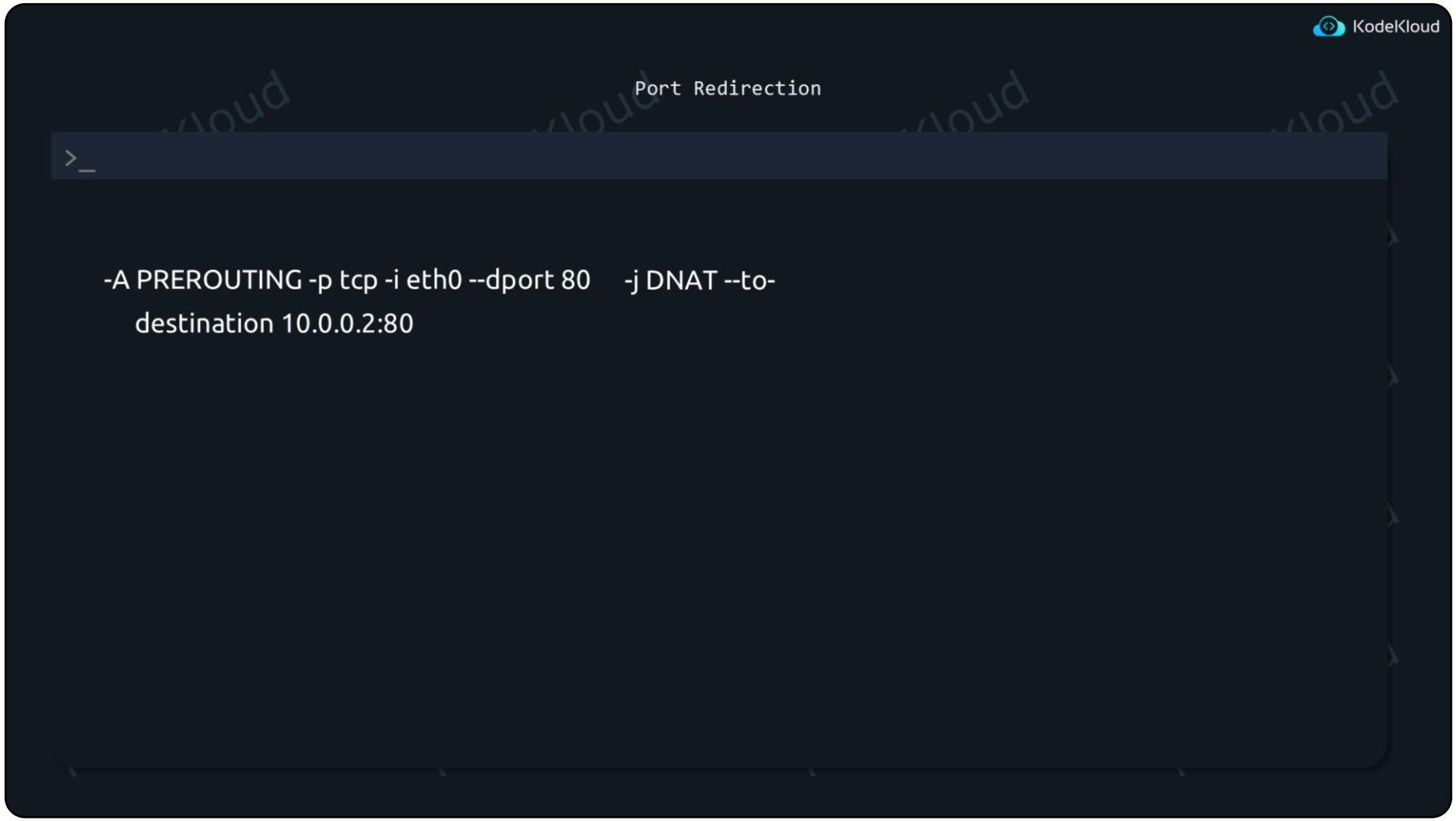


And if you scroll down to the "Full example" section, you'll notice this:

```
-A PREROUTING -p tcp -i eth0 --dport 80 -j DNAT \
  --to-destination 10.0.0.2:80
-A POSTROUTING -s 10.0.0.0/8 -o eth0 -j MASQUERADE
```

The first two lines basically reflect the first command we used. The backslash escapes a newline character. Or, in plain terms, this is actually a single list of options, but separated on two different lines by that backslash. But we can remove that character in our future command, and put all those options on a single line.

```
-A PREROUTING -p tcp -i eth0 --dport 80 -j DNAT \
  --to-destination 10.0.0.2:80
```



So just add "sudo iptables -t nat" in front of this line,

move the "--to-destination" line to the same line, and you get to:

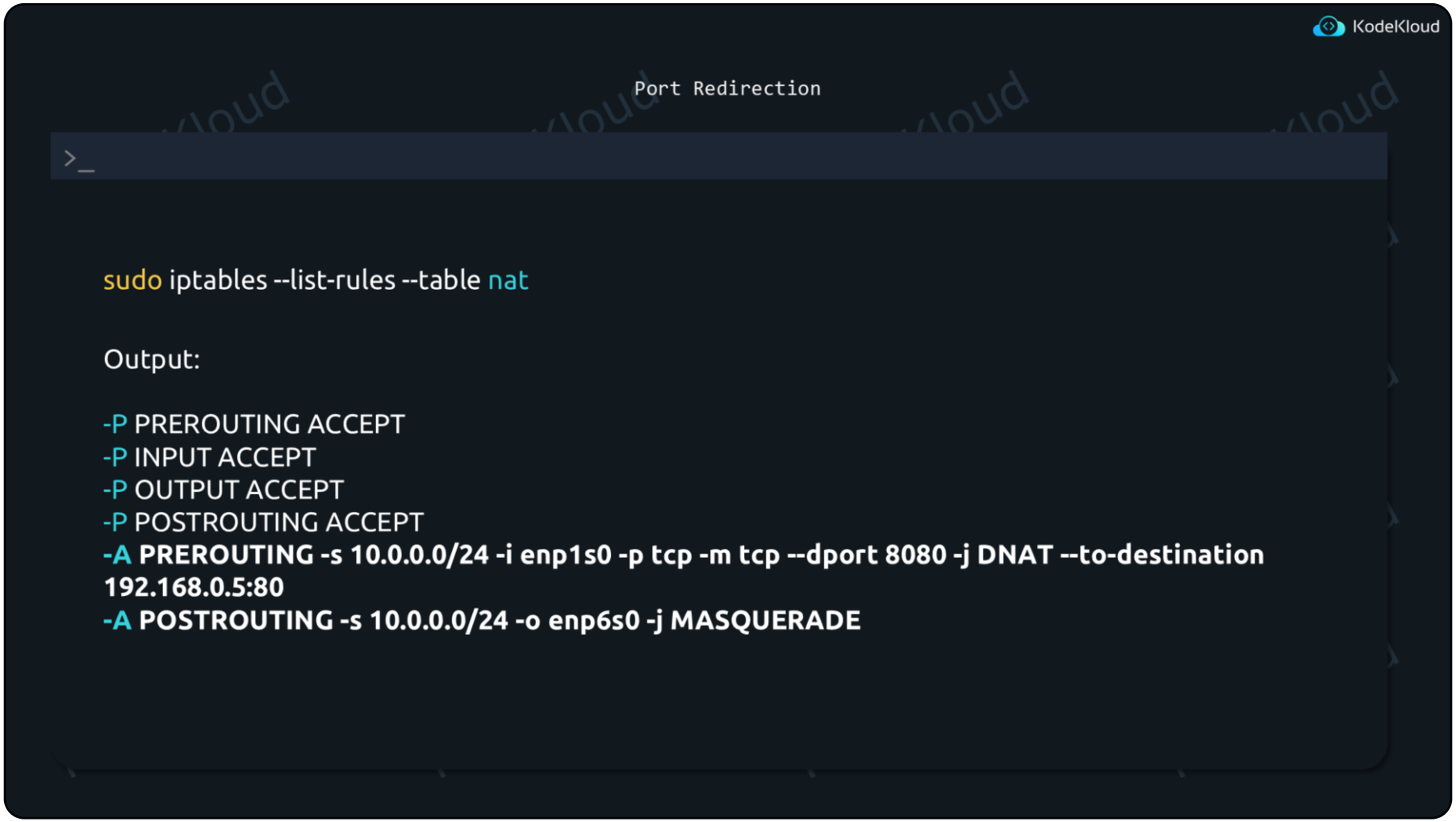
sudo iptables -t nat -A PREROUTING -p tcp -i eth0 --dport 80 –s 192.168.0.0/24 –j DNAT --to- destination 10.0.0.2:80

Then add "-s" option before the DNAT extension to select a source IP for the rule, modify the -i input interface, --dport, and --to-destination IP and port to whatever you need.
0.0/24 –j DNAT --to- destination 10.0.0.2:

Then follow a similar approach for the second example line in the manual:

-A POSTROUTING -s 10.0.0.0/8 -o eth0 -j MASQUERADE

Add "sudo iptables -t nat" in front of that example, and modify the source IP range and output interface to whatever you need.

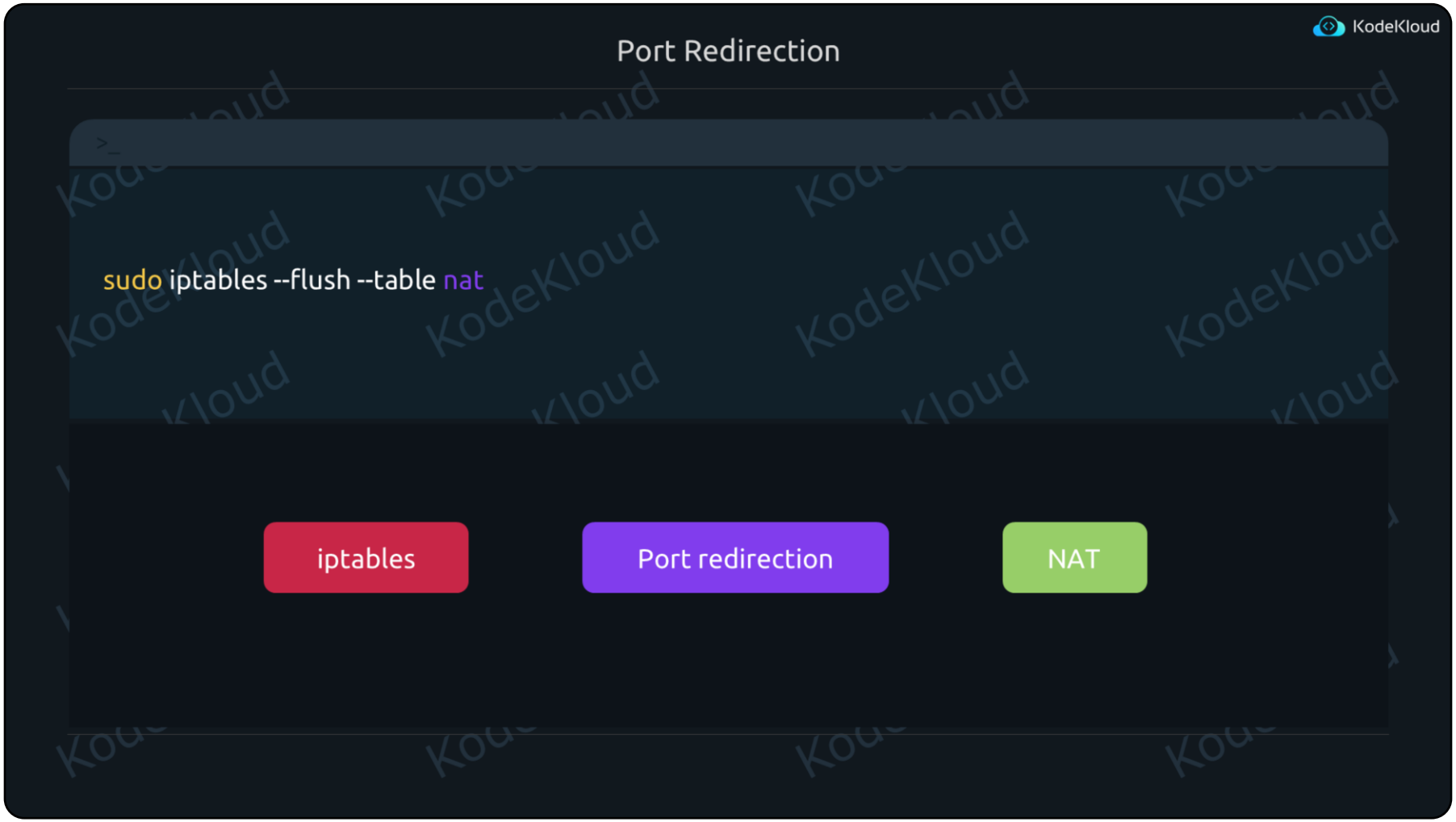


It might also be useful to know that you can list current rules in the iptables "nat" table with:

sudo iptables --list-rules --table nat

In our output we should see the prerouting and postrouting rules we just added.

```
-P PREROUTING ACCEPT
-P INPUT ACCEPT
-P OUTPUT ACCEPT
-P POSTROUTING ACCEPT
-A PREROUTING -s 10.0.0.0/24 -i enp1s0 -p tcp -m tcp --dport 8080 -j DNAT --to-destination 192.168.0.5:80
-A POSTROUTING -s 10.0.0.0/24 -o enp6s0 -j MASQUERADE
```



And if you made any mistakes, you can empty this table with:

sudo iptables --flush --table nat

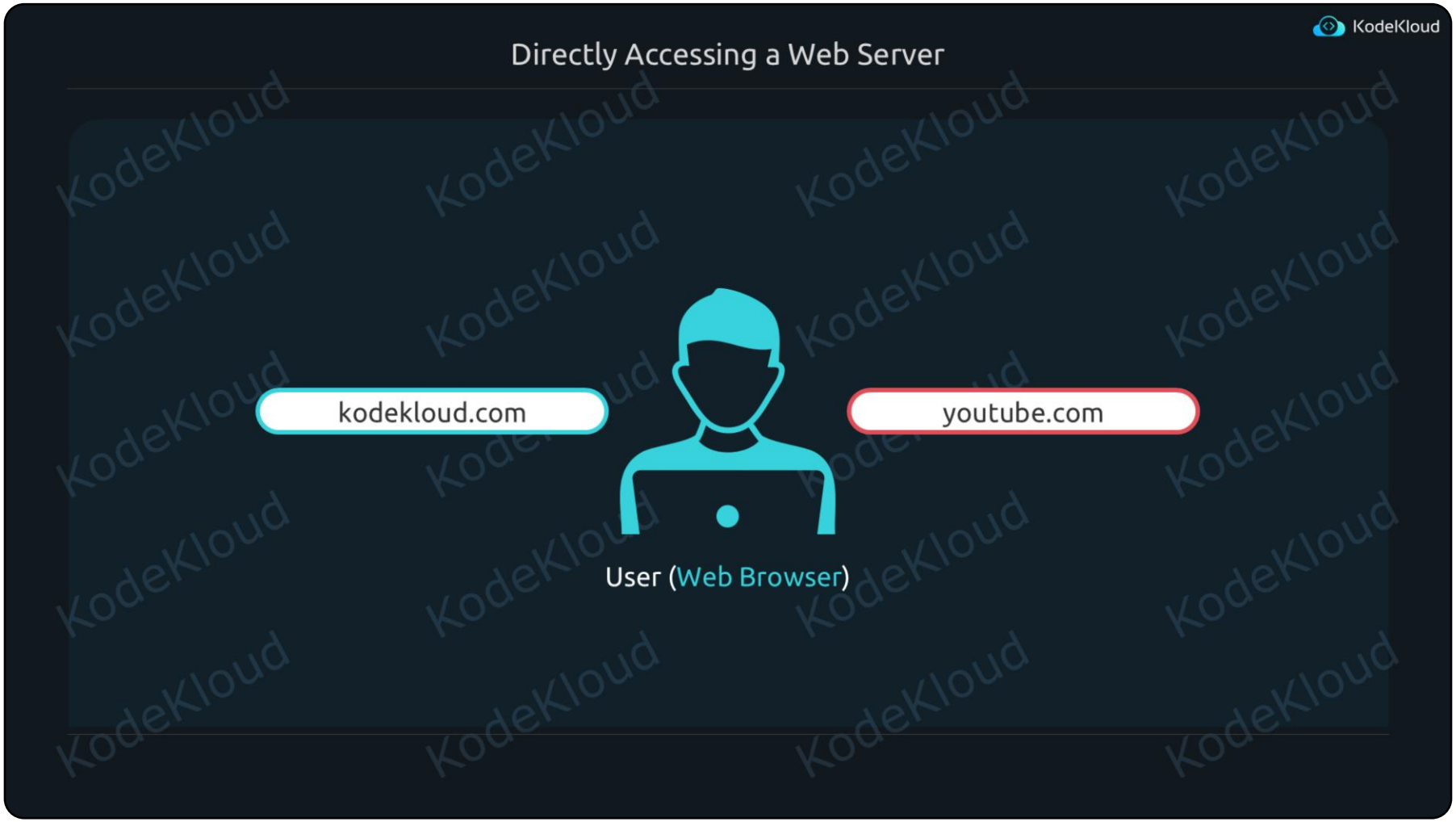
And then you can start over.

Configuring networks and firewalls is usually something most beginners struggle with. And it's understandable, as it's a very complicated and vast subject. So we hope this lesson cleared up at least some of the mysteries behind iptables, port redirection, and network address translation. Now let's jump to our next lesson.

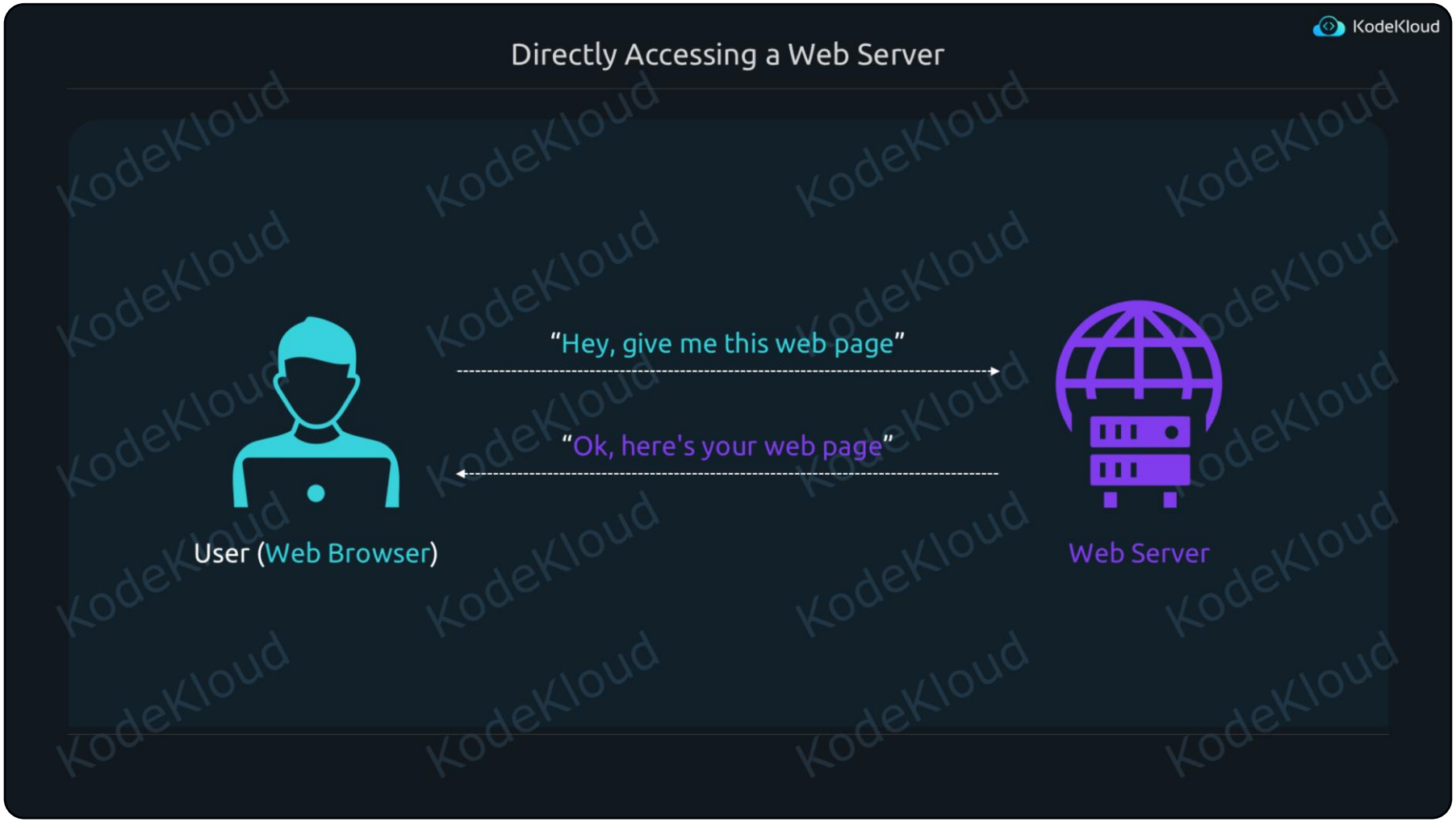
KodeKloud

Visit www.kodekloud.com to discover more.

In this lesson we'll look at how to configure reverse proxies and load balancers for web servers.



Whenever we access websites like kodekloud.com or youtube.com, the content that is displayed in our browsers comes from a web server.



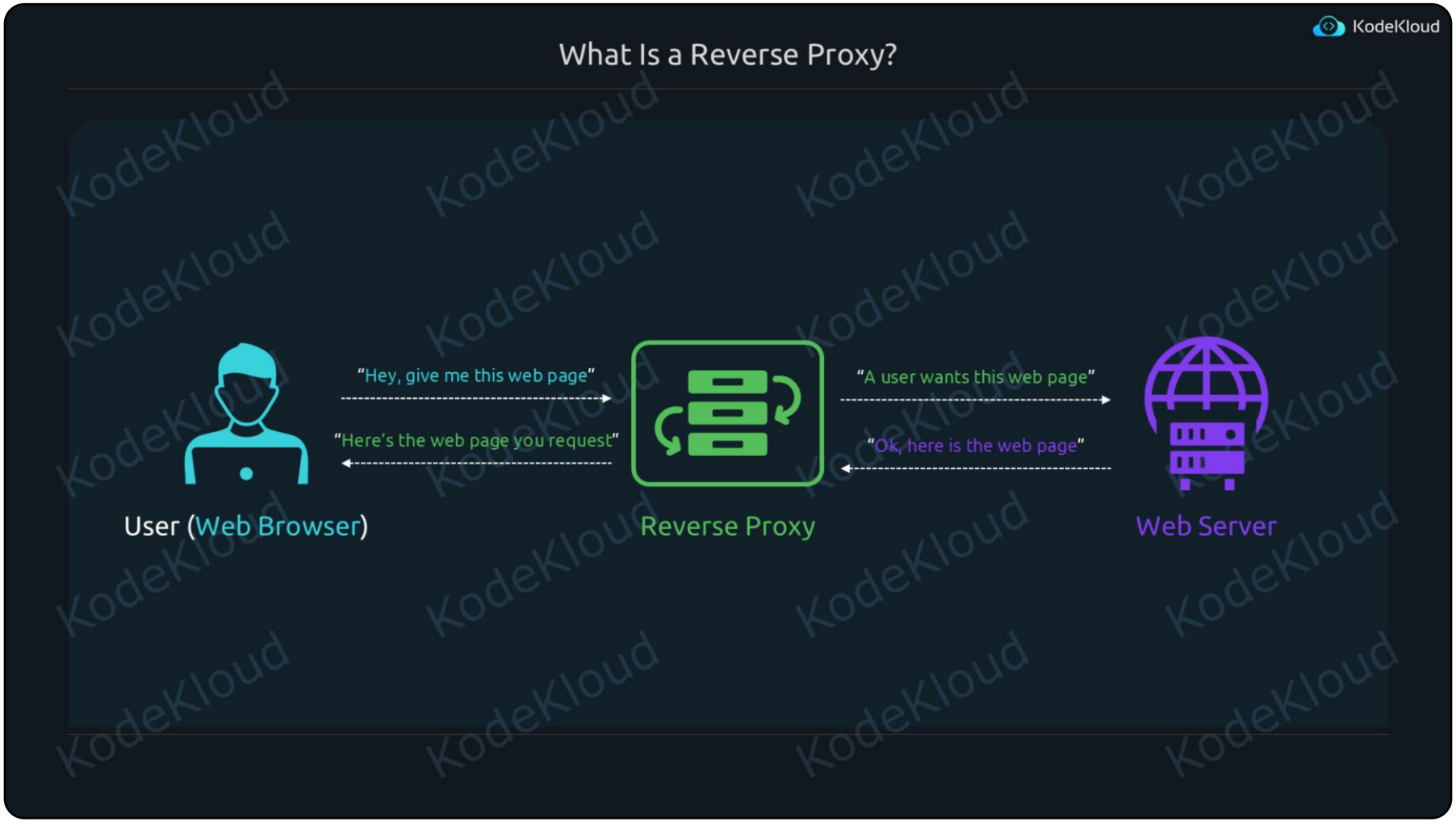
The most simplistic scenario looks something like this:

The user requests a web page. The browser sends that request to the web server, and the web server returns some content.

But nowadays, the websites that see heavy traffic are not accessed this way. Not directly. A lot of infrastructure sits in

the middle, between the user and the web server.

So let's see what changes when a reverse proxy is added into the mix.

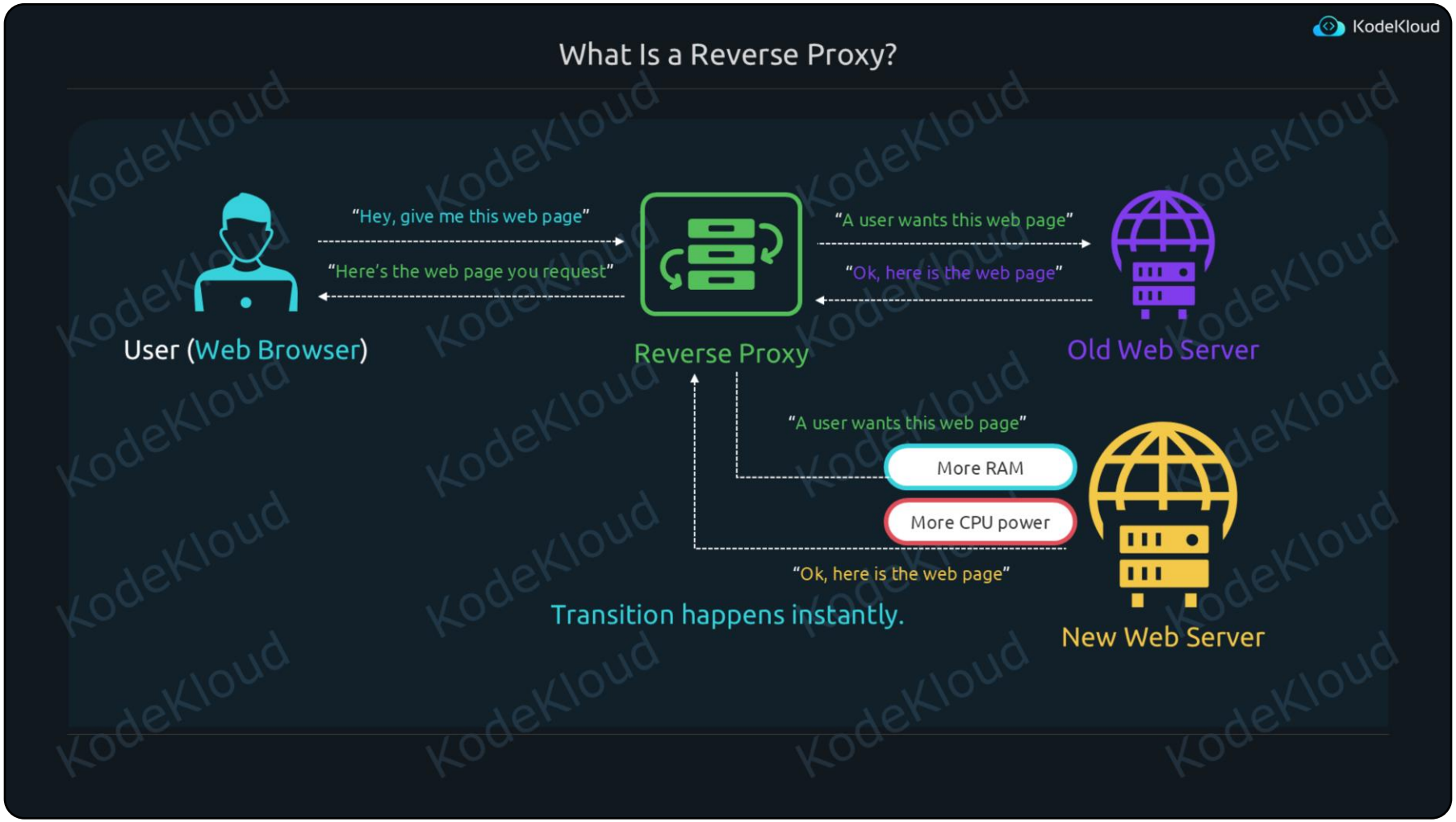


If we add a reverse proxy in front of the web server, things now look like this:

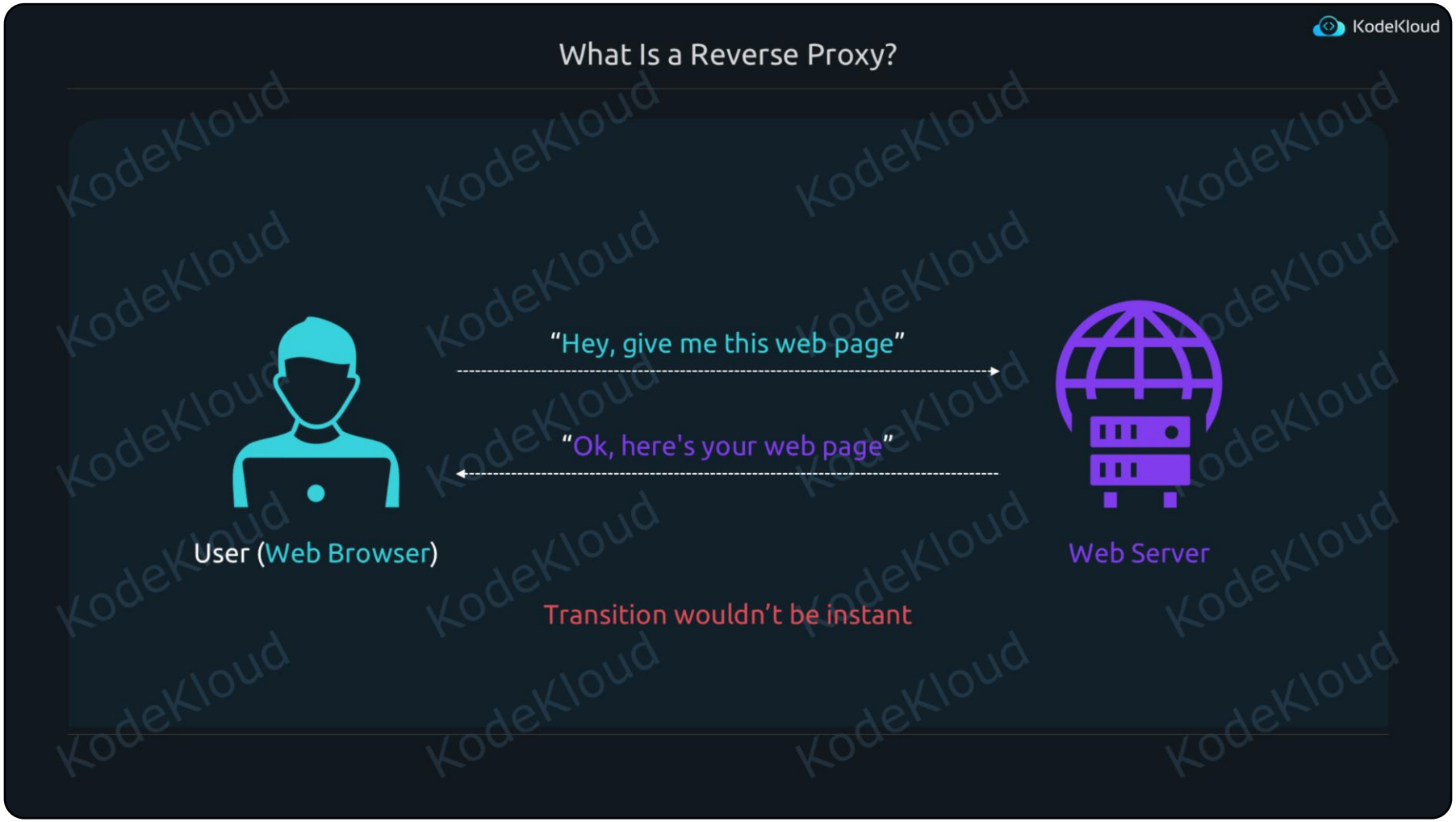
The user request now goes to the reverse proxy first. Then the proxy sends it to the web server. Next, the web server returns the page to the reverse proxy. And, finally, the reverse proxy hands it out to the user.

So why would we need to add this reverse proxy in the middle? Well, it brings a couple of advantages. Here's just one

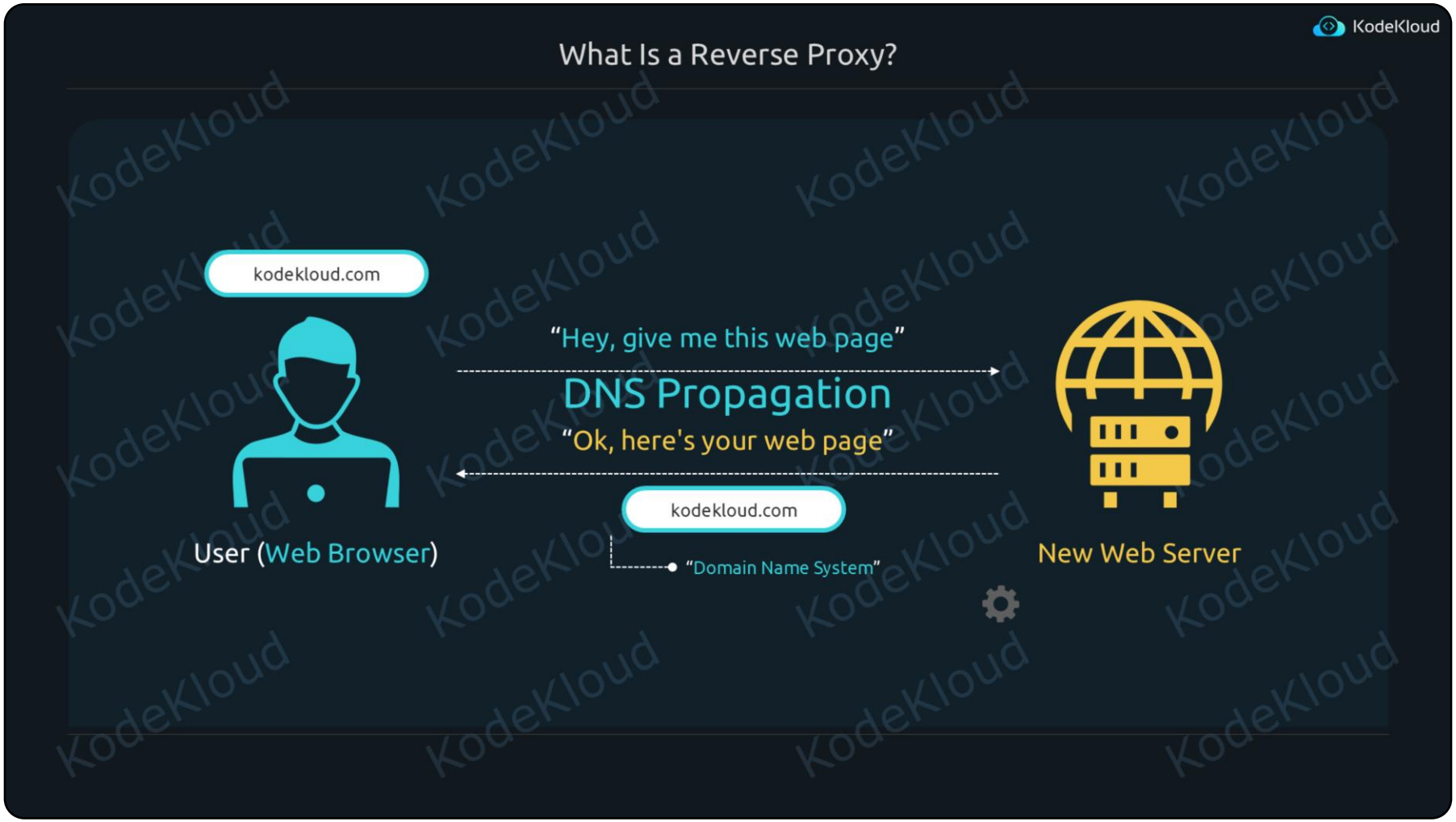
example.



Let's say we build a new and improved web server. More RAM, more CPU power; it will just work faster. And as soon as we finish building it, we can tell the reverse proxy to switch traffic to it. We basically flip a switch and traffic stops going to the old web server, and goes to the new one instead. And this transition happens instantly.



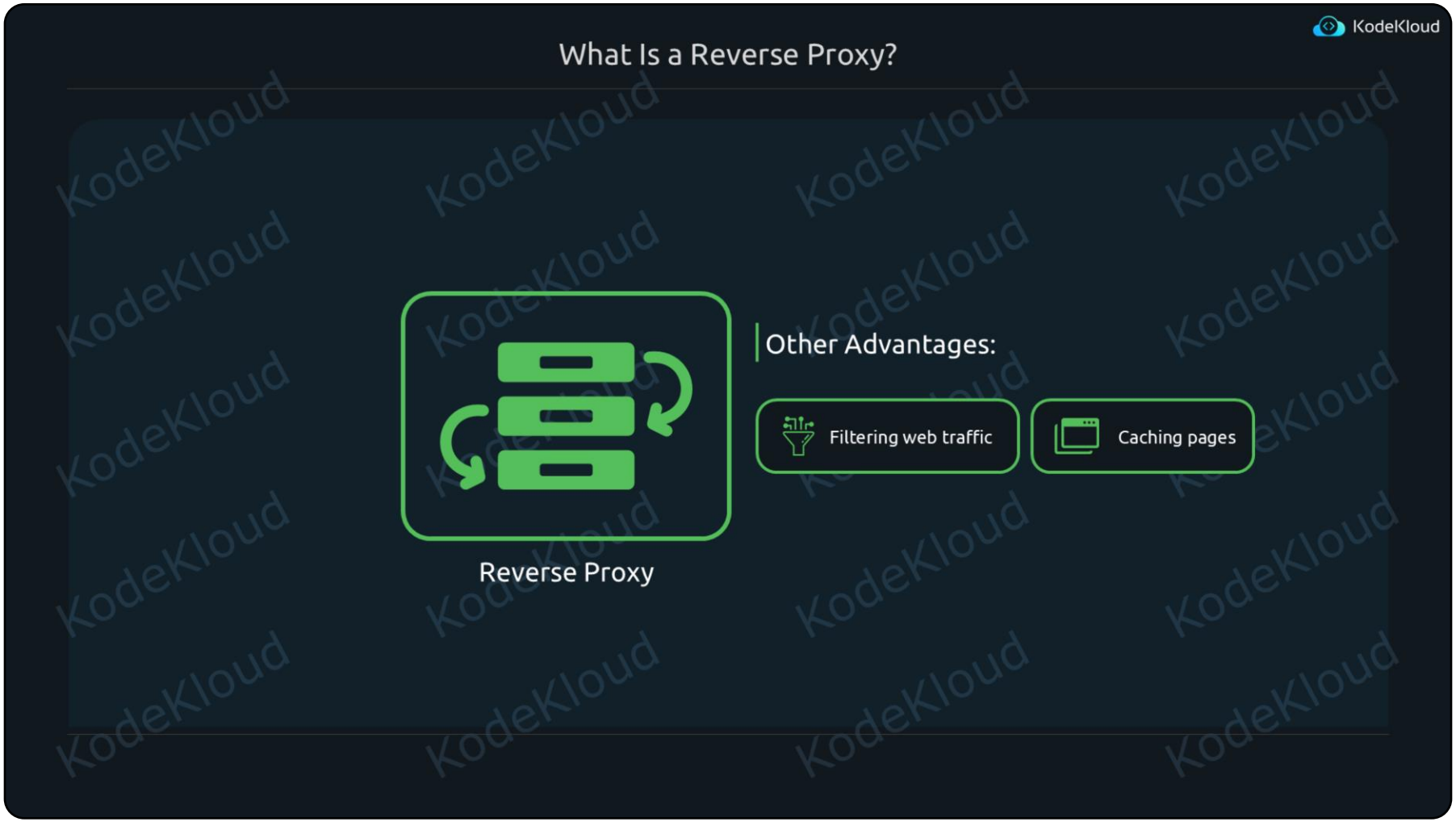
But if our setup would be the old model we looked at, with user requests going directly to the web server, then this transition wouldn't be instant.



We'd have to update our kodekloud.com domain name to point to the different IP address of the new web server. And until all users get redirected to this new IP address, it can take minutes, or hours. This is called "DNS propagation". DNS is short for Domain Name System.

A reverse proxy can bring many other advantages, such as filtering web traffic, caching pages to return results much faster to users, and so on. But it's actually the way that it can instantly redirect traffic to a different web server that

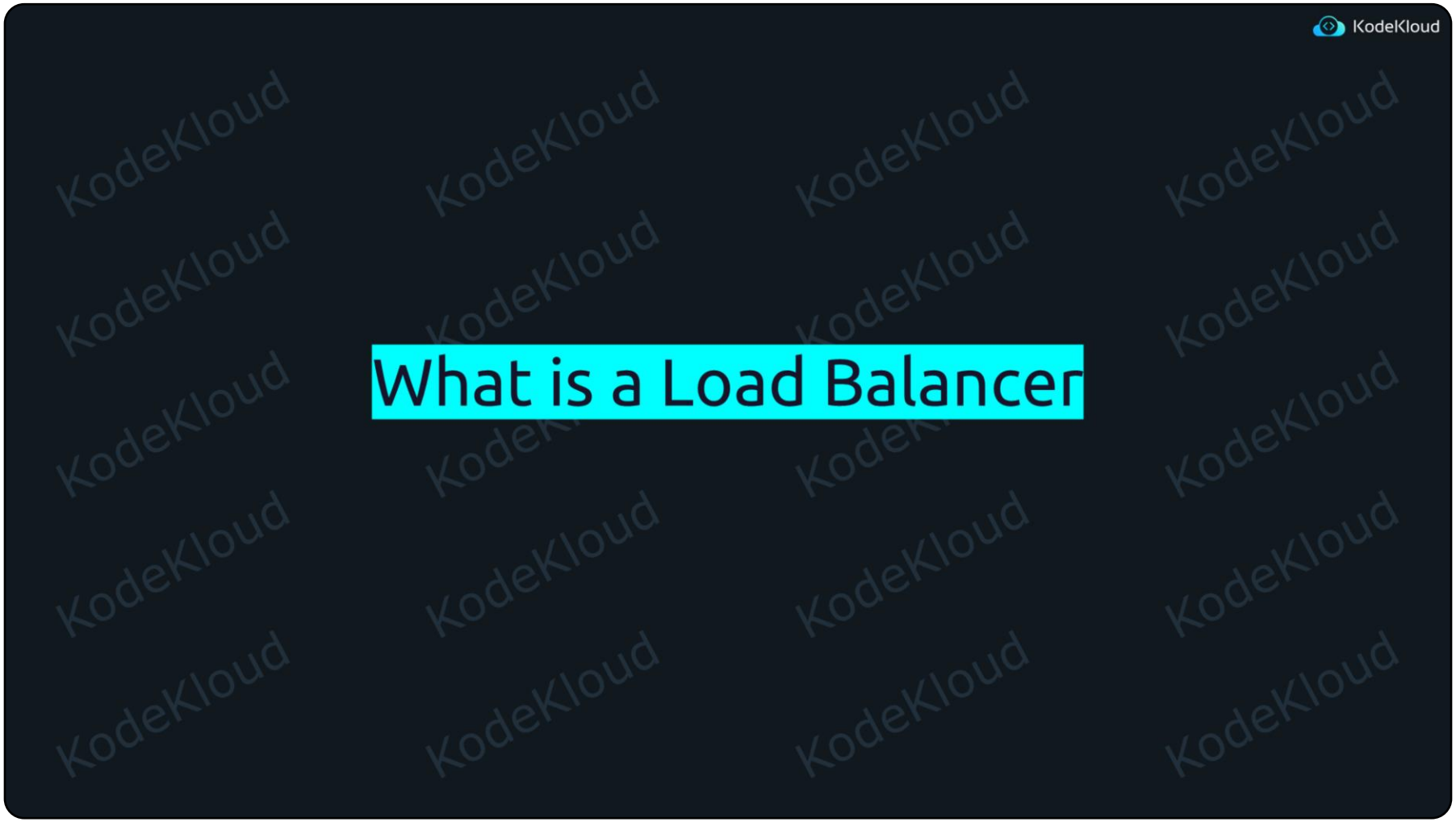
created another interesting use-case: load-balancing.



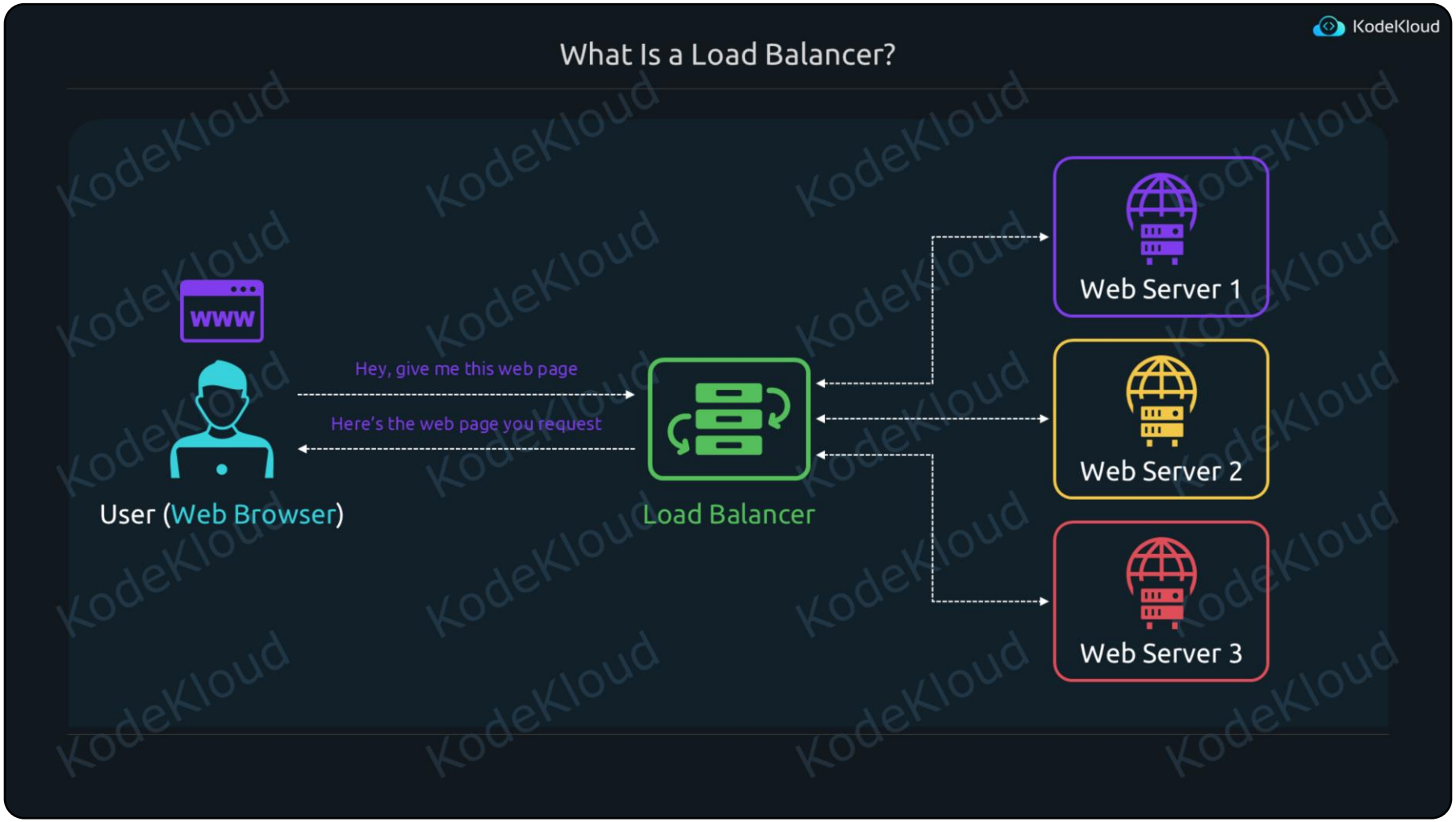
We'd have to update our kodekloud.com domain name to point to the different IP address of the new web server. And until all users get access to this new IP address, it can take minutes, or hours. This is called "DNS propagation". DNS is short for Domain Name System.

A reverse proxy can bring many other advantages, such as filtering web traffic, caching pages to return results much faster to users, and so on. But it's actually the way that it can instantly redirect traffic to a different web server that

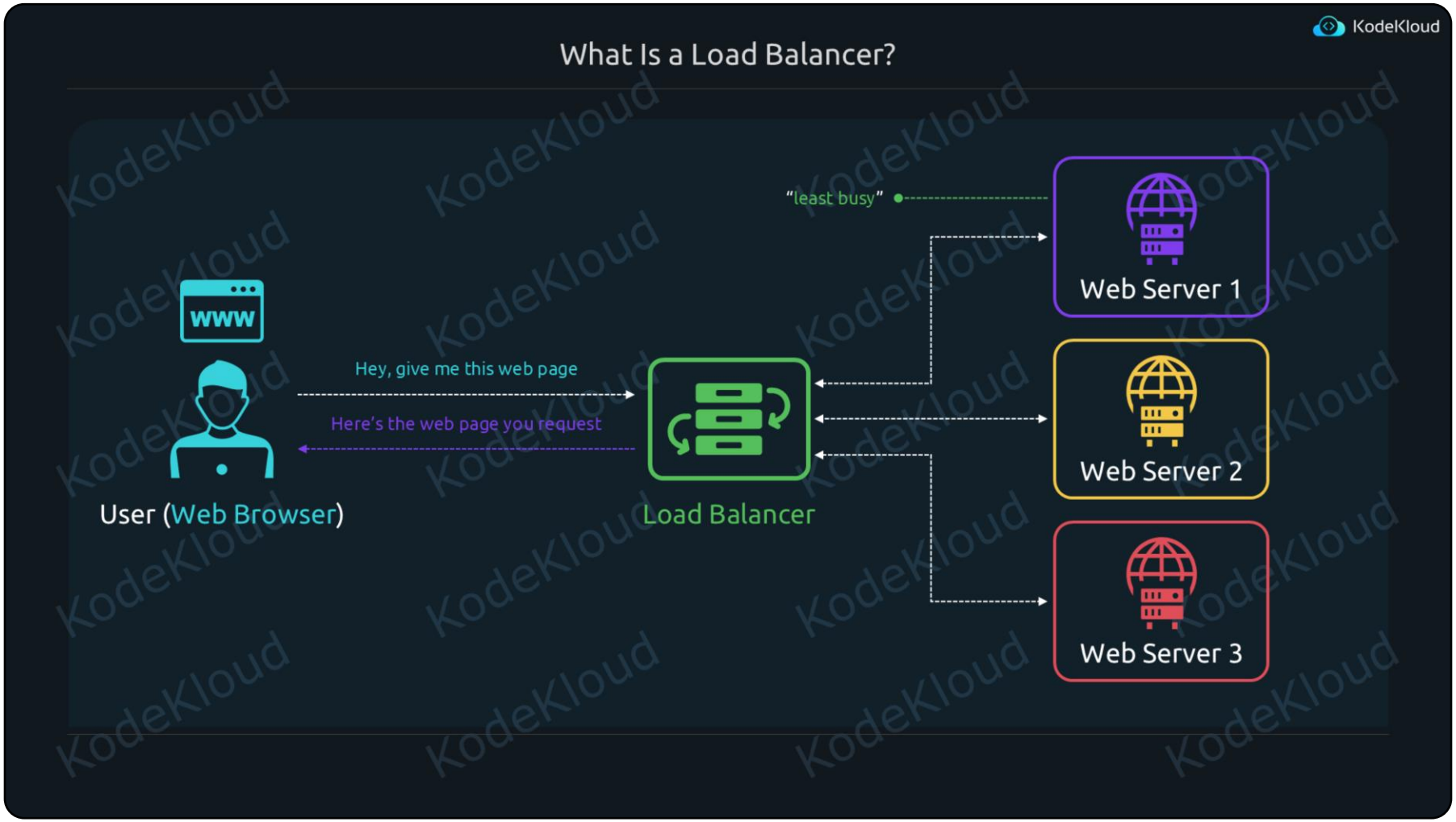
created another interesting use-case: load-balancing.



A load balancer is in a way rather similar to a reverse proxy.



But the difference is that it redirects traffic to multiple web servers, not just one. And it can dynamically pick which server is the best one to use.

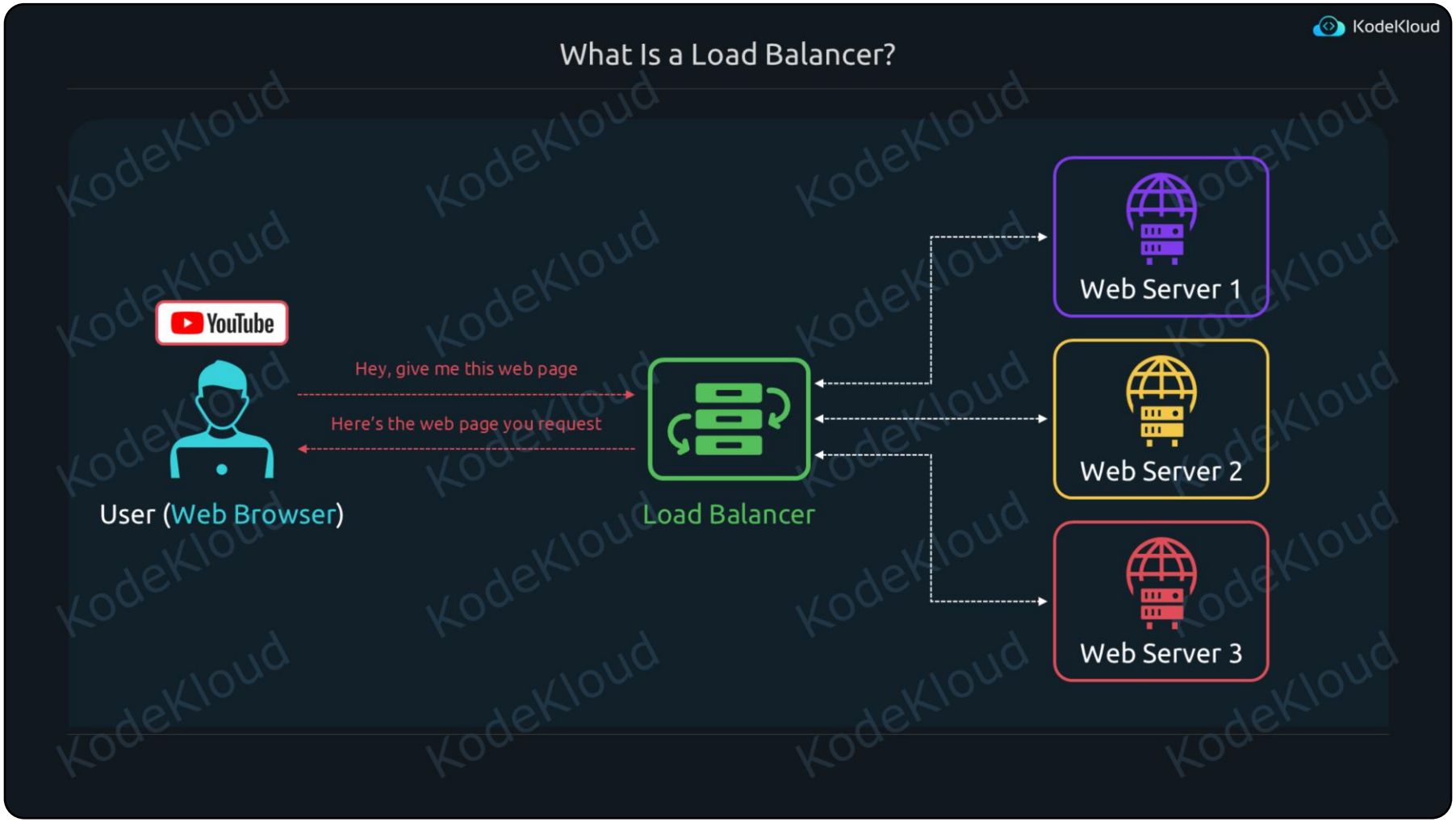


For example, it can redirect each request to the server that is least busy. This way, no server is overloaded, and they're all used evenly. This is the core idea of load balancing. To balance the load. That is, to split up the work that needs to be done so that each server gets a fair share that it can more easily deal with.

Think about sites like YouTube.com. There's no way a single web server can respond to the requests of millions of users at the same time. But load balancing distributes requests to thousands and thousands of servers. And this makes it

possible to serve all those users, without overusing any server.

Now that we know how reverse proxies and load balancers work, let's see how to create them.

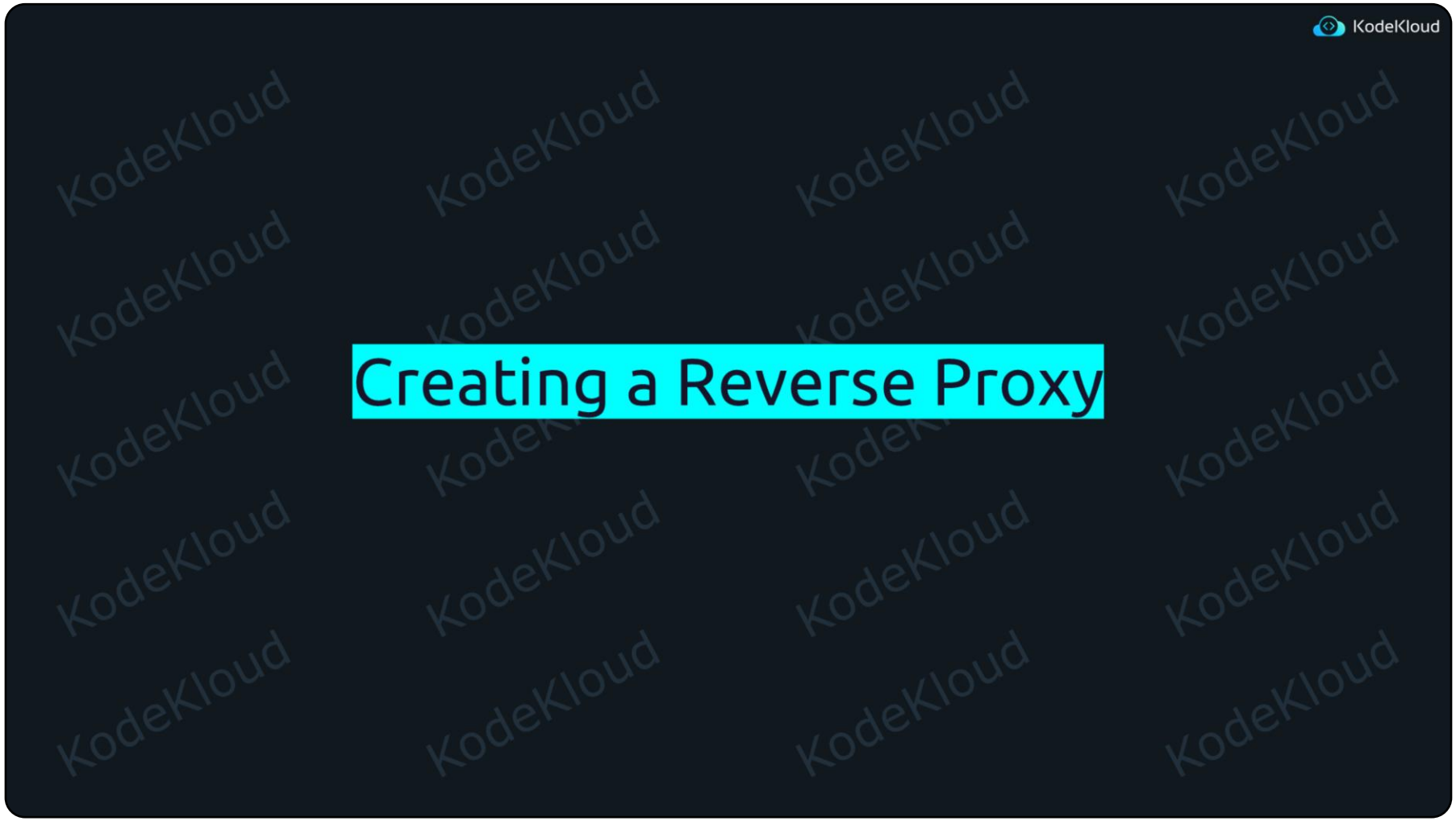


For example, it can redirect each request to the server that is least busy. This way, no server is overloaded, and they're all used evenly. This is the core idea of load balancing. To balance the load. That is, to split up the work that needs to be done so that each server gets a fair share.

Think about sites like YouTube.com. There's no way a single web server can respond to the requests of millions of users at the same time. But load balancing distributes requests to thousands and thousands of servers. And this makes it

possible to serve all those users, without overusing any server.

Now that we know how reverse proxies and load balancers work, let's see how to create them.



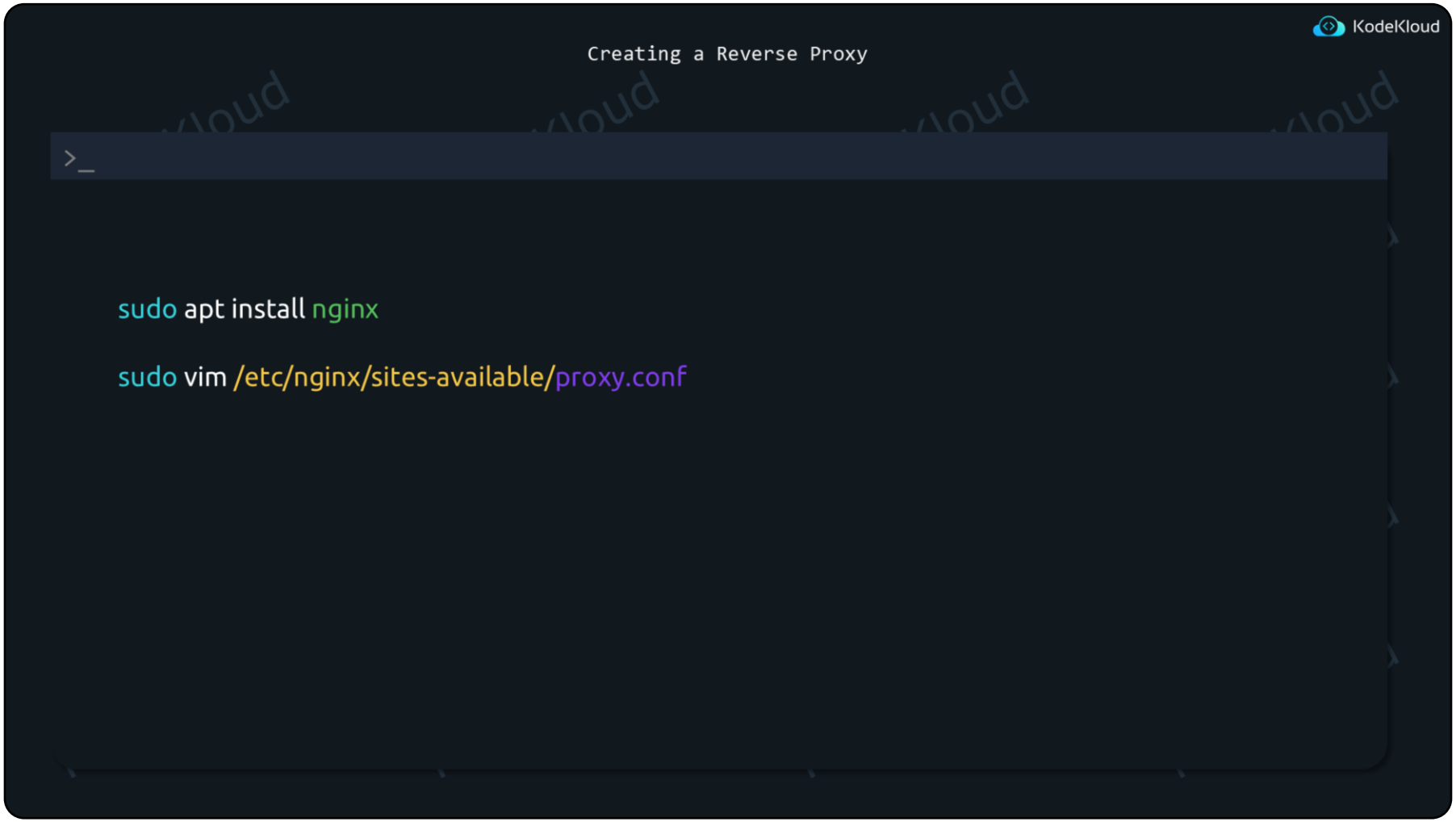
First, let's look at reverse proxies.

An application often used to create reverse proxies is Nginx.

Although alternative solutions exist, such as HAProxy, Apache, Traefik, Squid, and many others.

You might already be familiar with Nginx since it's one of the most used web servers in the world.

But even though it's a web server at heart, it can be configured to work as a reverse proxy as well. And also as a load balancer, as we'll see later on.



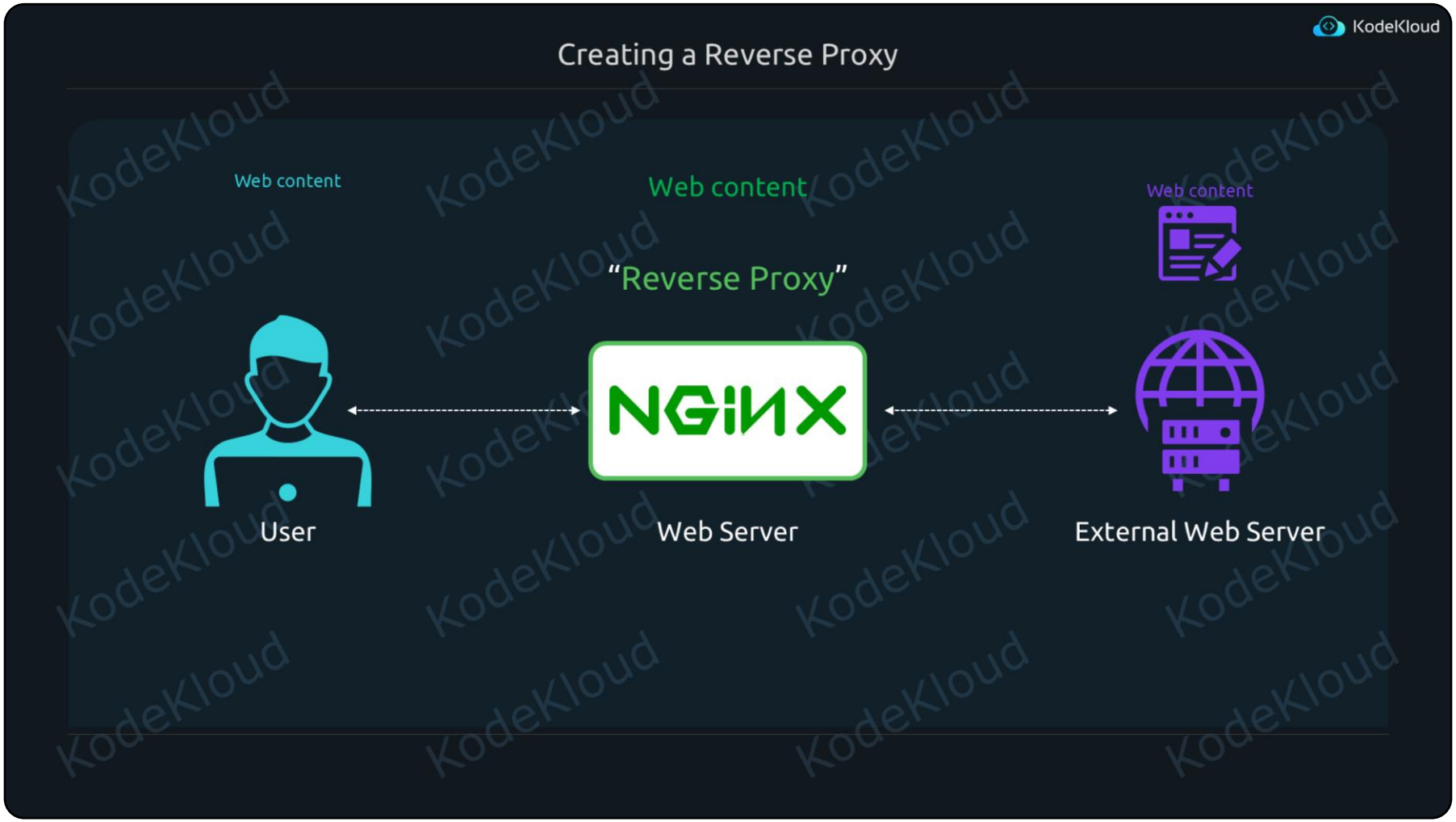
The steps to go through to create a reverse proxy are quite simple. First, we install Nginx:

sudo apt install nginx

Then, in the "/etc/nginx/sites-available/" directory we'll create a configuration file called "proxy.conf". We can pick any name we want for it, it's not mandatory to use this exact name.

sudo vim /etc/nginx/sites-available/proxy.conf

As far as Nginx is concerned, these are simply settings for a website it will host. Since technically, it still functions as a web server at its core. When users connect to it, they can get website content. But the difference is that when a user requests a web page, instead of generating it internally, we tell Nginx to fetch the content from an external web server. Which is what makes it a reverse proxy.



The steps to go through to create a reverse proxy are quite simple. First, we install Nginx:

sudo apt install nginx

Then, in the "/etc/nginx/sites-available/" directory we'll create a configuration file called "proxy.conf". We can pick any name we want for it, it's not mandatory to use this exact name.

sudo vim /etc/nginx/sites-available/proxy.conf

As far as Nginx is concerned, these are simply settings for a website it will host. Since, technically, it still functions as a web server at its core. When users connect to the Nginx reverse proxy, they can get website content. But the difference is that when a user requests a web page, instead of generating it internally, we tell Nginx to fetch the content from another web server. Which is what makes it a reverse proxy.

We'll add the following content in our configuration file:

In the server block we tell Nginx to listen for incoming connections on port 80.

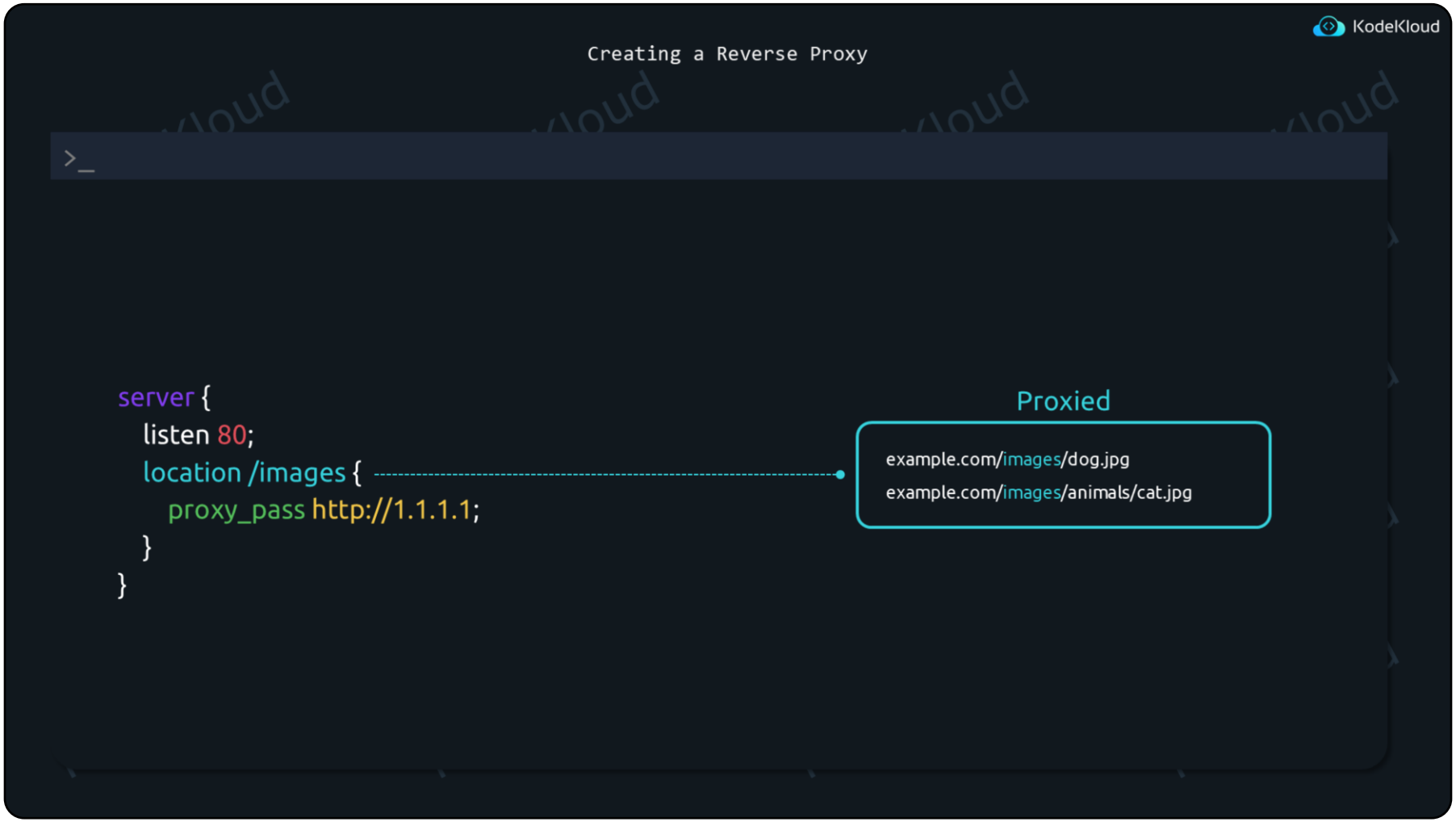
This line:

location / {

might be a bit harder to understand. But, in essence, it tells Nginx to apply the proxy_pass directive below, only for requests that match this location description. For example, let's look at these URLs:

example.com/
example.com/admin
example.com/images/dog.jpg
example.com/images/animals/cat.jpg

What do they have in common? Well, after the name of our domain, "example.com" all the URLs continue with a forward slash. Since there's no URL that doesn't add a slash after the "example.com" domain name, it means that this location directive will match all requests, for any web address the user might want to visit.



But in some cases, we might want to proxy requests only for a portion of our website. So we could change our location directive to:

location /images {

Now, only URLs like

example.com/images/dog.jpeg
example.com/images/animals/cat.jpeg

will be proxied.

But URLS like:

example.com/
example.com/admin

Won't be.

So you can look at this location directive as a filter. We tell Nginx to proxy requests only if the web address visited contains this part specified under location directly after our example.com domain name.

Finally, this line:

proxy_pass http://1.1.1.1;

is almost self-explanatory. The requests matched in the location directive will be proxied to the web server specified here. We can use an IP address, but also a hostname, or domain name, if we want to.

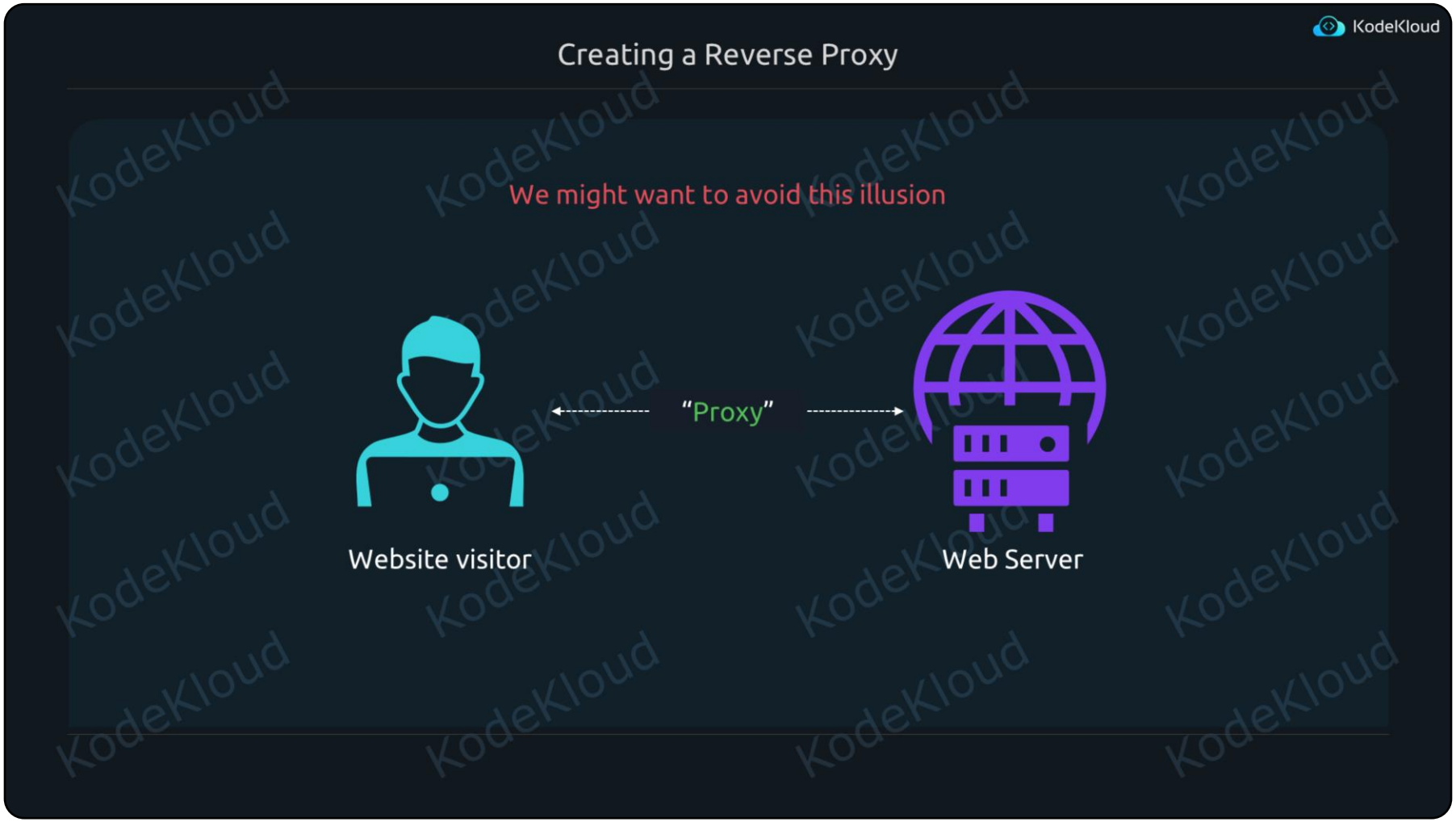
So a line like:

proxy_pass http://webserver1.example.com;

is also valid.

If the web server we are proxying requests to does not listen on port 80, we'll have to specify the port ourselves. For example, if the server listens for incoming connections on port 8081, we can modify our line this way:

proxy_pass http://1.1.1.1:8081;

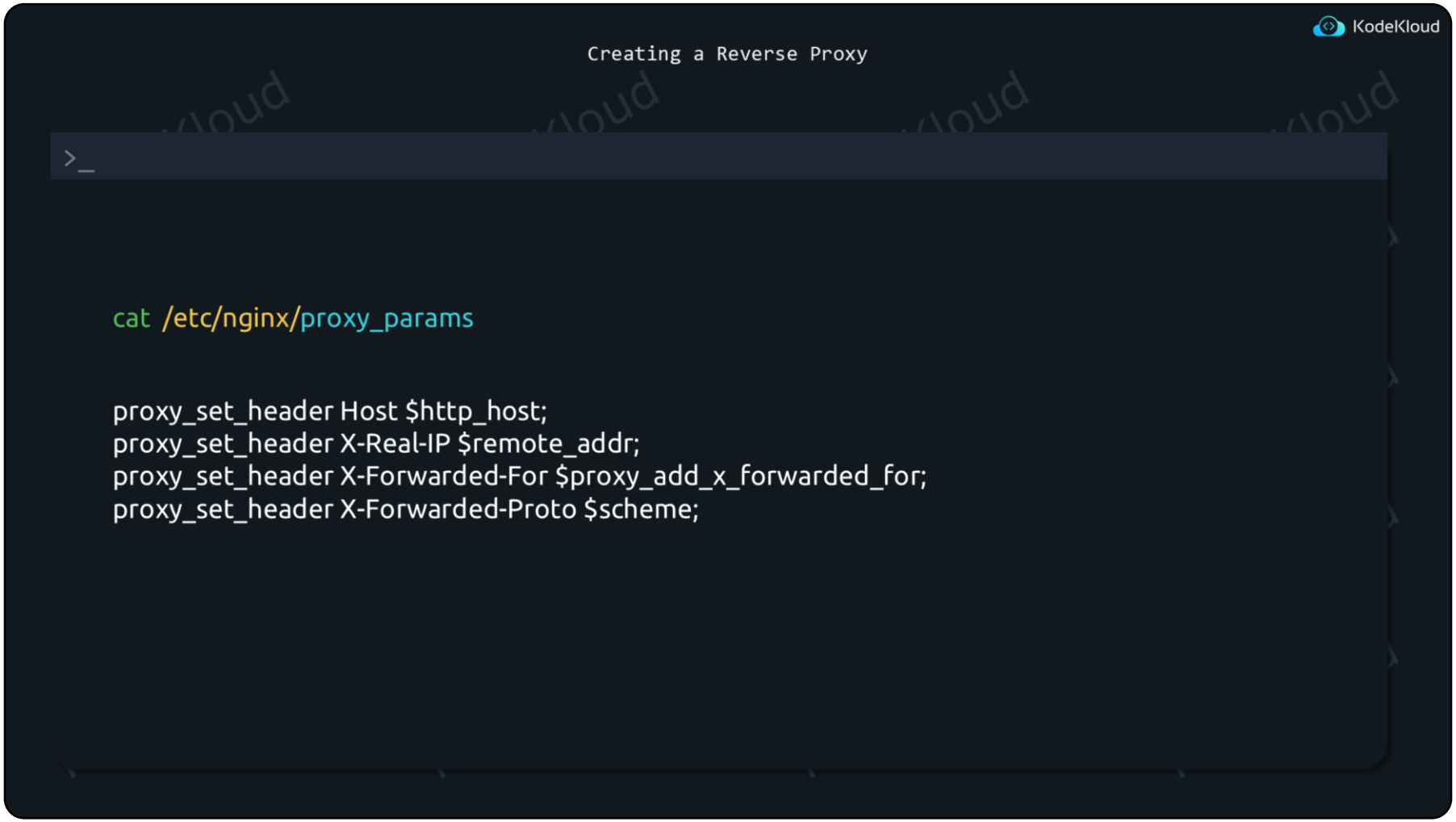


Now think of the web server. From its point of view, it looks like the website visitor is our proxy. Since it's the proxy that connects to it. Depending on our setup, we might want to avoid this illusion. For example, imagine we want to collect statistics, and see what users visited our website.

So how do we tell our server that we're actually making this request on behalf of a user? And this user has a certain IP address that should be logged. Well, we can add this line:

```
server {
   listen 80;
   location / {
```

```
        proxy_pass http://1.1.1.1;
        include proxy_params;
    }
}
```



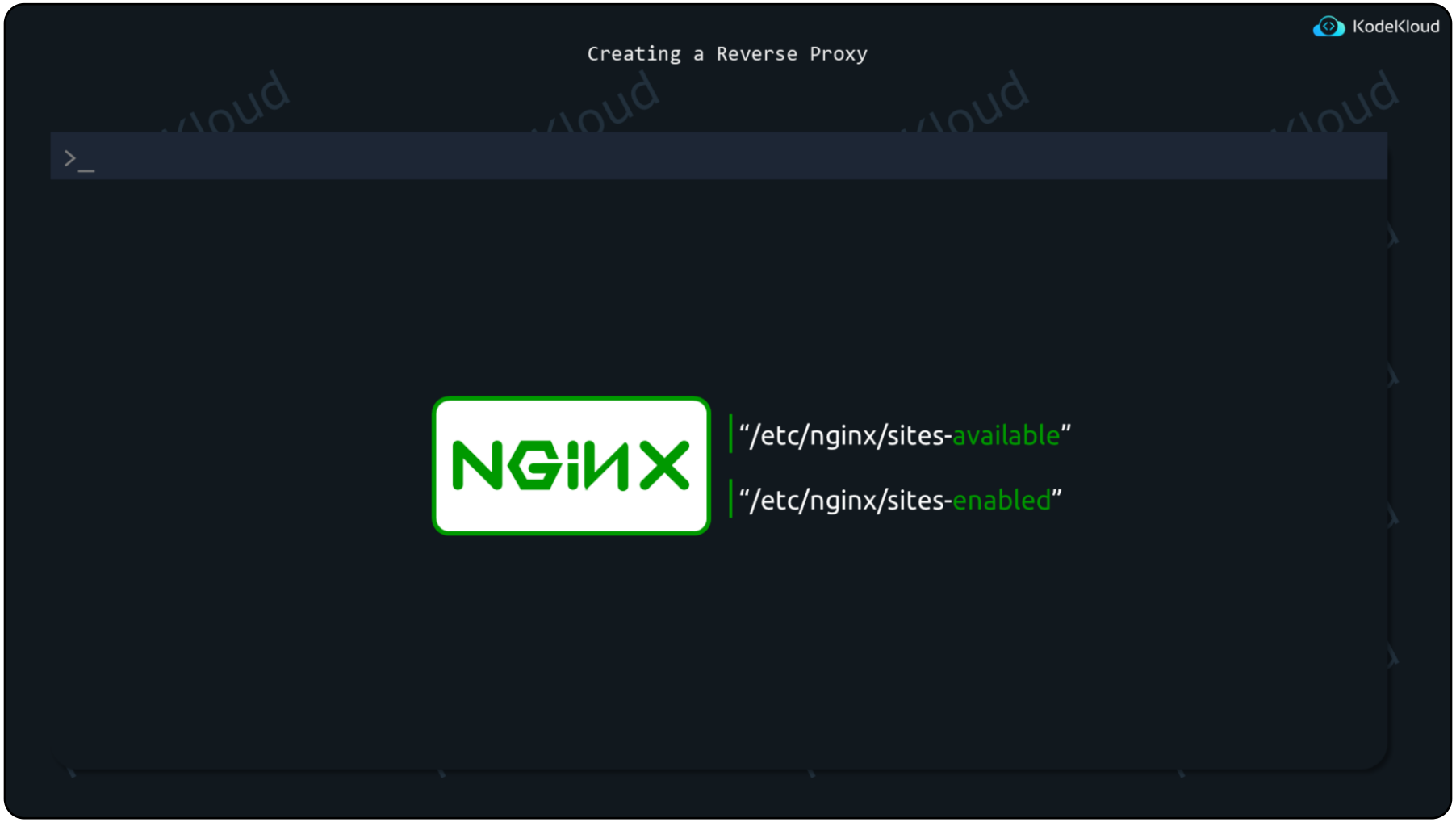
This simply includes instructions specified in another file called "proxy_params". We can find this file at "/etc/nginx/proxy_params". If we'd use the "cat" command:

cat /etc/nginx/proxy_params

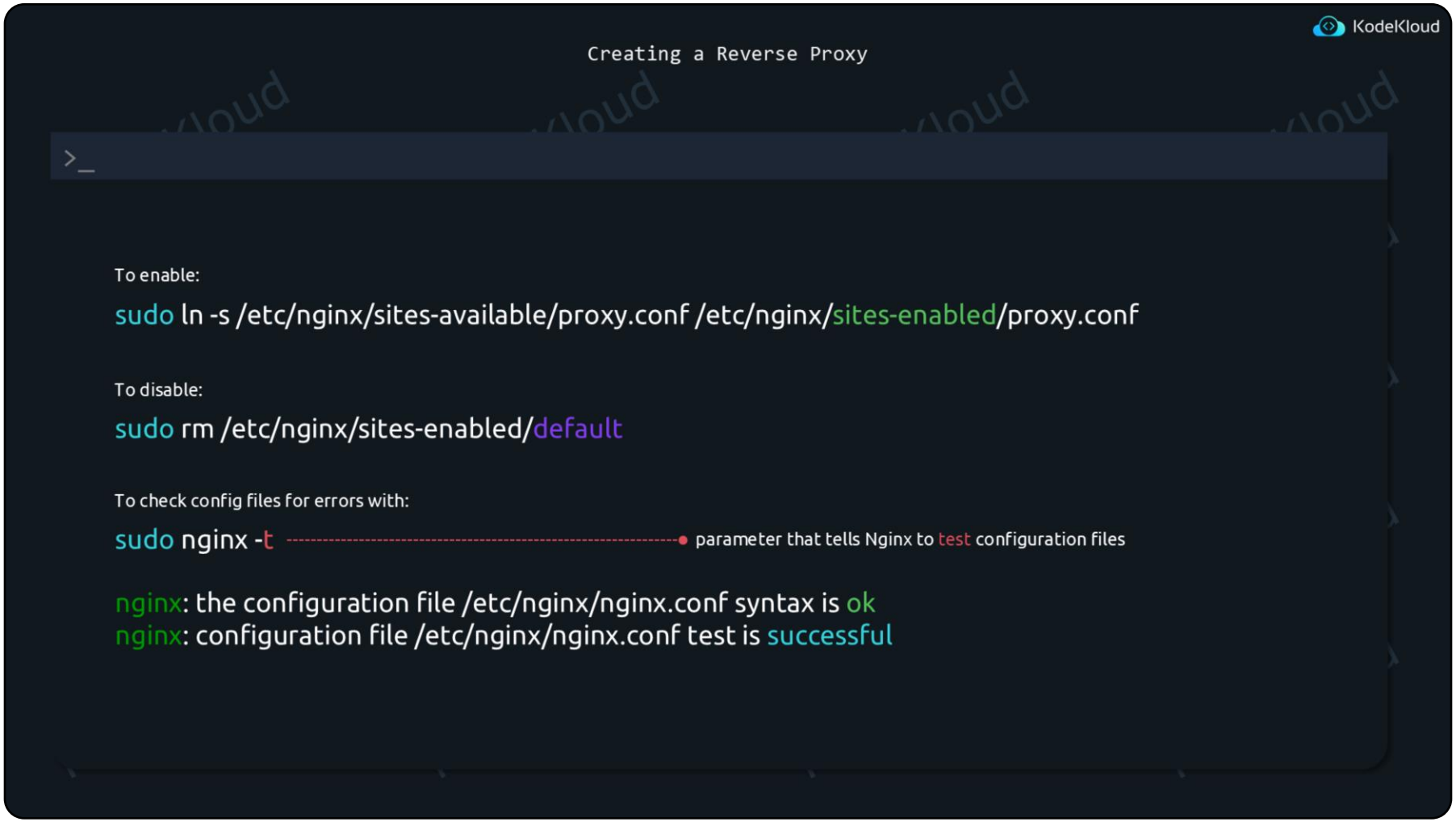
we would see this content:

```
proxy_set_header Host $http_host;
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_set_header X-Forwarded-Proto $scheme;
```

We can see that all that this does is set some headers for the requests that will be redirected to the web server. This way, the server gets some additional data in the requests it receives. Data like, the real IP address of the user that requested this web page, what protocol they used, for example http, or https, and so on.



After we save our file, we need to tell Nginx to use settings for this "website" we defined. Or, more accurately, the website definition which will tell it to work as a reverse proxy. Remember, we added this config file to the "/etc/nginx/sites-available/" directory. But, as the name suggests, this just lists what website configurations are available. Nginx has an additional directory called "/etc/nginx/sites-enabled/". And it only applies definitions for files it can find in that directory.



The standard practice is to simply soft link files. When we want to enable a website config from the "sites-available" directory, we soft link it to the "sites-enabled" directory. In our case, we would use this command:

[They learned about soft links in initial lessons, so no need to explain]

sudo ln -s /etc/nginx/sites-available/proxy.conf /etc/nginx/sites-enabled/proxy.conf

Also, we should disable the default website configuration that is currently active. To disable it, we simply remove the file from the directory. The file is actually called "default".

sudo rm /etc/nginx/sites-enabled/default

This file is also a soft link, so we won't lose any content since the real file can be found in the "/etc/nginx/sites-available/" directory.

At this point, it's a good idea to check our config files for errors with:

sudo nginx -t

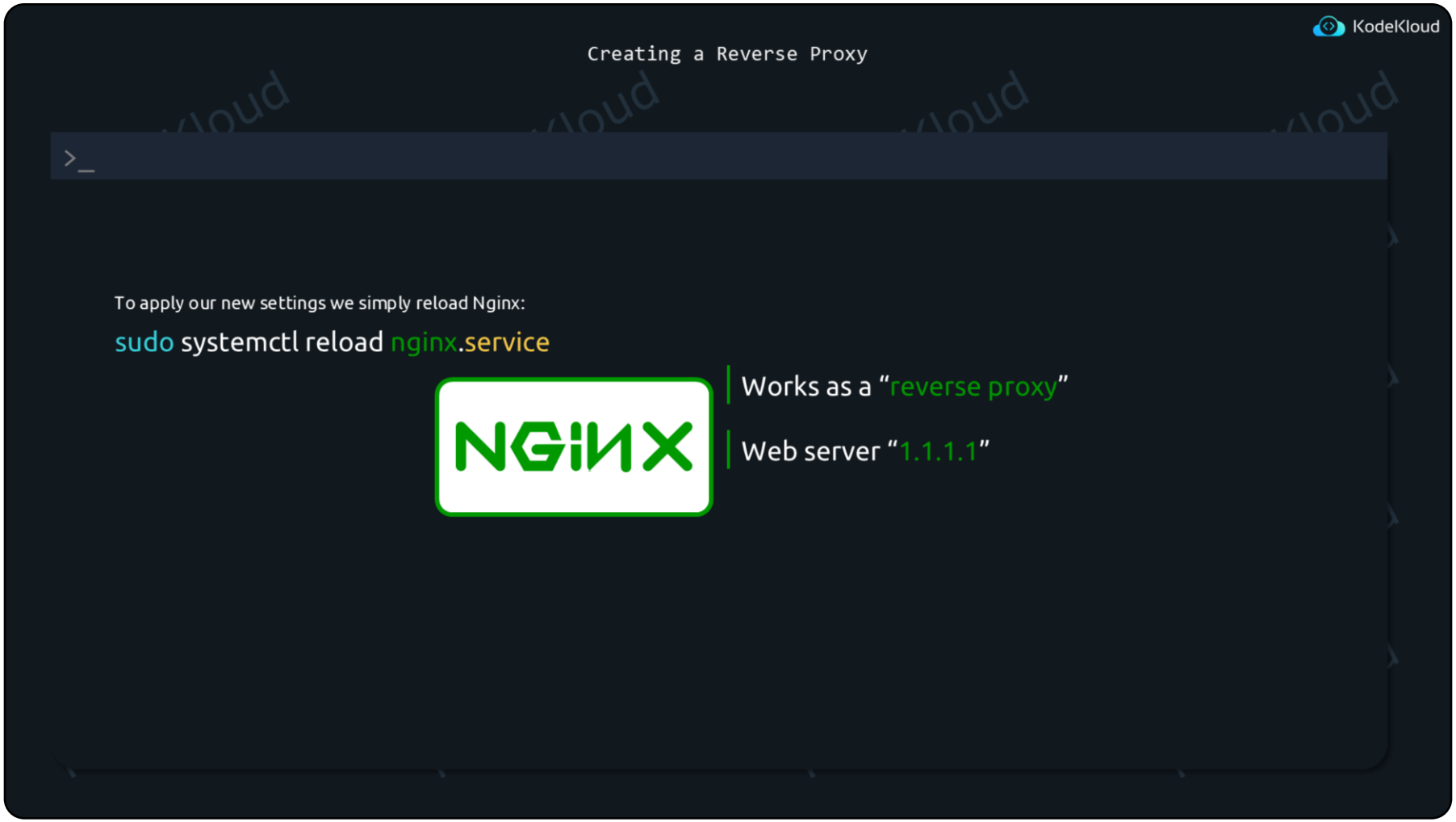
-t is the parameter that tells Nginx to test configuration files

We should see this output:

nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
nginx: configuration file /etc/nginx/nginx.conf test is successful

Otherwise, errors will inform us about the lines we have to correct.

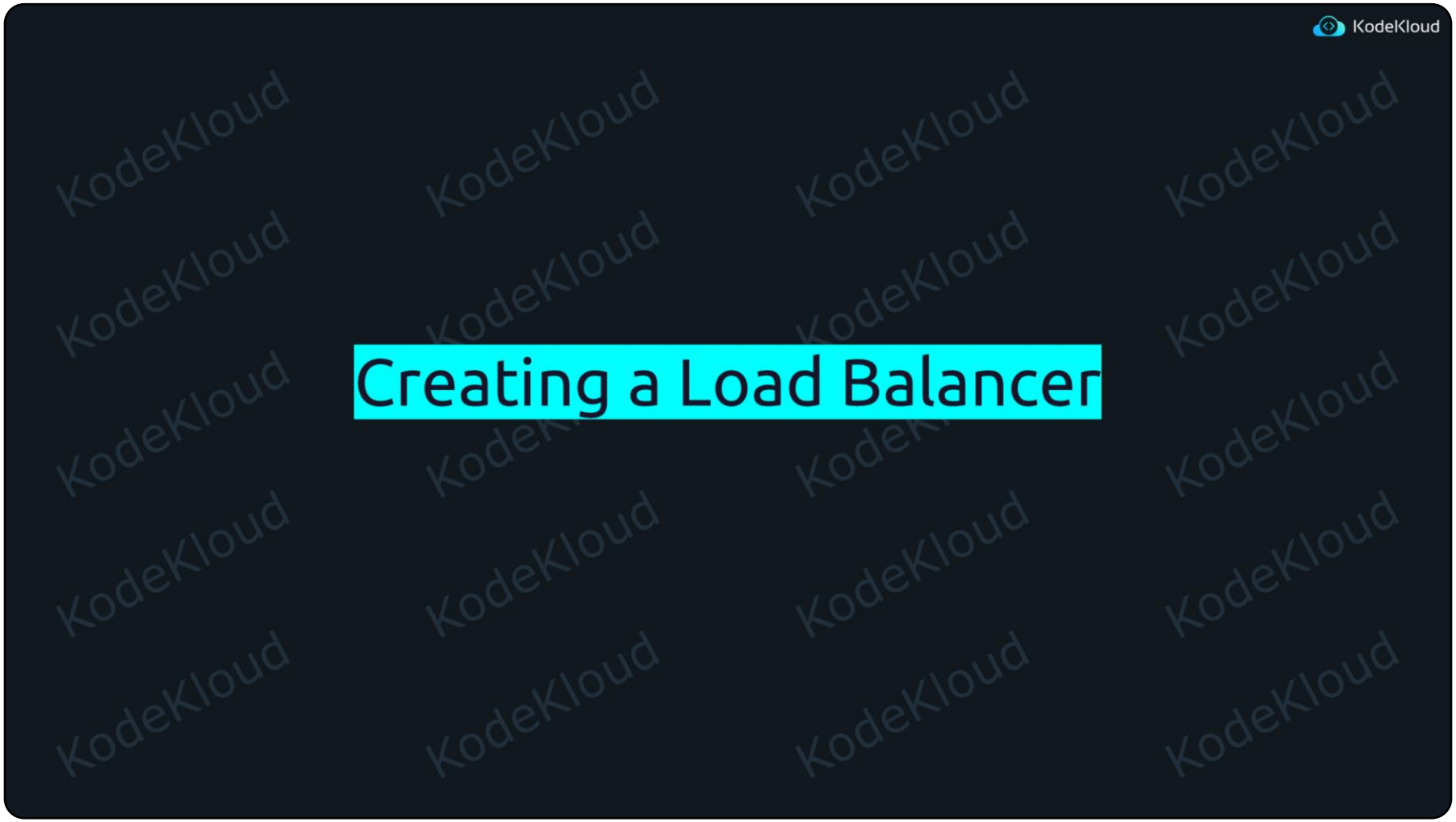
Even though this mentions "/etc/nginx/nginx.conf", this configuration file will load other files, including the "proxy.conf" file we created. So that will be tested as well even if it doesn't show up in this output.



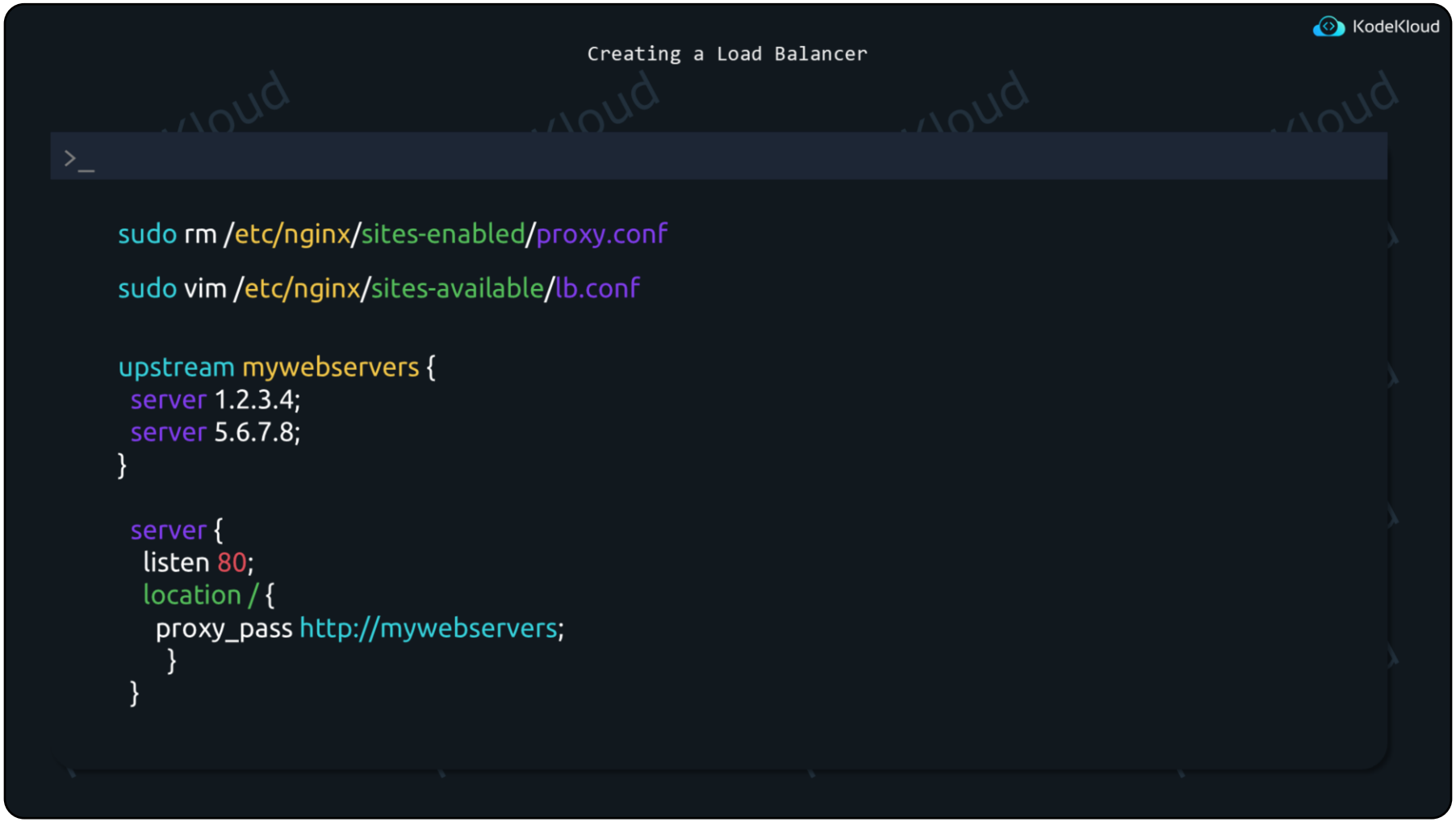
To apply our new settings we simply reload Nginx:

sudo systemctl reload nginx.service

And that's it. Nginx now works as a reverse proxy, redirecting all requests to the web server at "1.1.1.1".



Now let's see how to reconfigure this as a load balancer.



First, we should remove the link to our old configuration:

sudo rm /etc/nginx/sites-enabled/proxy.conf

Next, a new configuration file should be created in the "/etc/nginx/sites-available/" directory.

sudo vim /etc/nginx/sites-available/lb.conf

Then the content we add could look something like this:

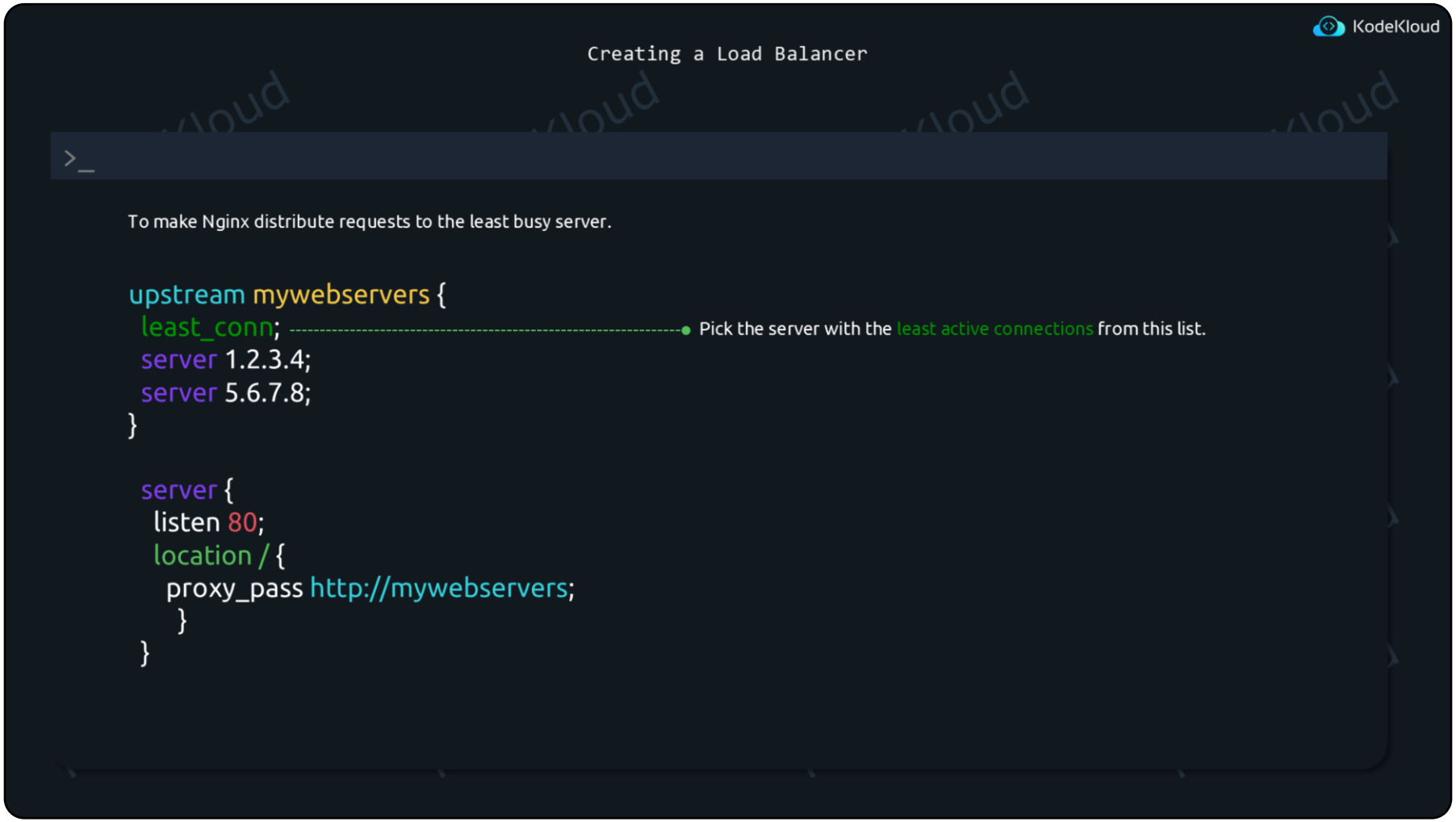
```
upstream mywebservers {
  server 1.2.3.4;
  server 5.6.7.8;
}

  server {
    listen 80;
    location / {
      proxy_pass http://mywebservers;
       }
  }
```

To understand it, let's analyze from the last lines, upward. Because we're already familiar with how this block at the bottom works. It's almost the same as what we saw in our reverse proxy section of this lesson. The only difference is that we tell Nginx to proxy requests to http://mywebservers instead of the IP address or hostname of a web server. Basically, "http://mywebservers" is something that doesn't exist out there, in the real world. But we defined what it means, internally, in the first block above.

Now the text at the top makes sense. With the upstream directive we defined a collection of servers. And we called this collection mywebservers, but we can choose any name we want here. Then we simply listed the IP addresses of various web servers.

With the current contents in this file, the load balancing method would be what is called round-robin. This means that any time a user requests something from our load balancer, this request would be redirected to a random server. Sometimes it will be 1.2.3.4, some other times 5.6.7.8.



This can work fine for a website that is not seeing very heavy traffic. But with heavy traffic, the random redirection might not be ideal. It could happen that the server at 1.2.3.4 gets more than 50% of the requests and it might get overloaded with more than it can handle. To make Nginx distribute requests to the least busy server, we could add this line:

```
upstream mywebservers {
  least_conn;
```

```
  server 1.2.3.4;
  server 5.6.7.8;
}

  server {
    listen 80;
    location / {
      proxy_pass http://mywebservers;
        }
  }
```

The least_conn keyword makes it pick the server with the least active connections from this list.

Another scenario we can encounter is that we might have web servers that are more powerful than others. So they could process more requests than our weaker servers. In this case, we can assign what are called weights.

```
upstream mywebservers {
 least_conn;
 server 1.2.3.4 weight=3;
```

```
  server 5.6.7.8;
}

  server {
    listen 80;
    location / {
      proxy_pass http://mywebservers;
       }
  }
```

If we add this keyword here, then Nginx will send many more requests to the server with the IP address "1.2.3.4". And even if we don't specify it, the default weight is equal to 1. So server 5.6.7.8 has a default weight of 1 assigned to it. With these settings, for every 4 requests, roughly 3 will be sent to 1.2.3.4, and one will be sent to 5.6.7.8.

If we need to do some maintenance work on some server, we can mark it as being unavailable, by adding the down keyword:

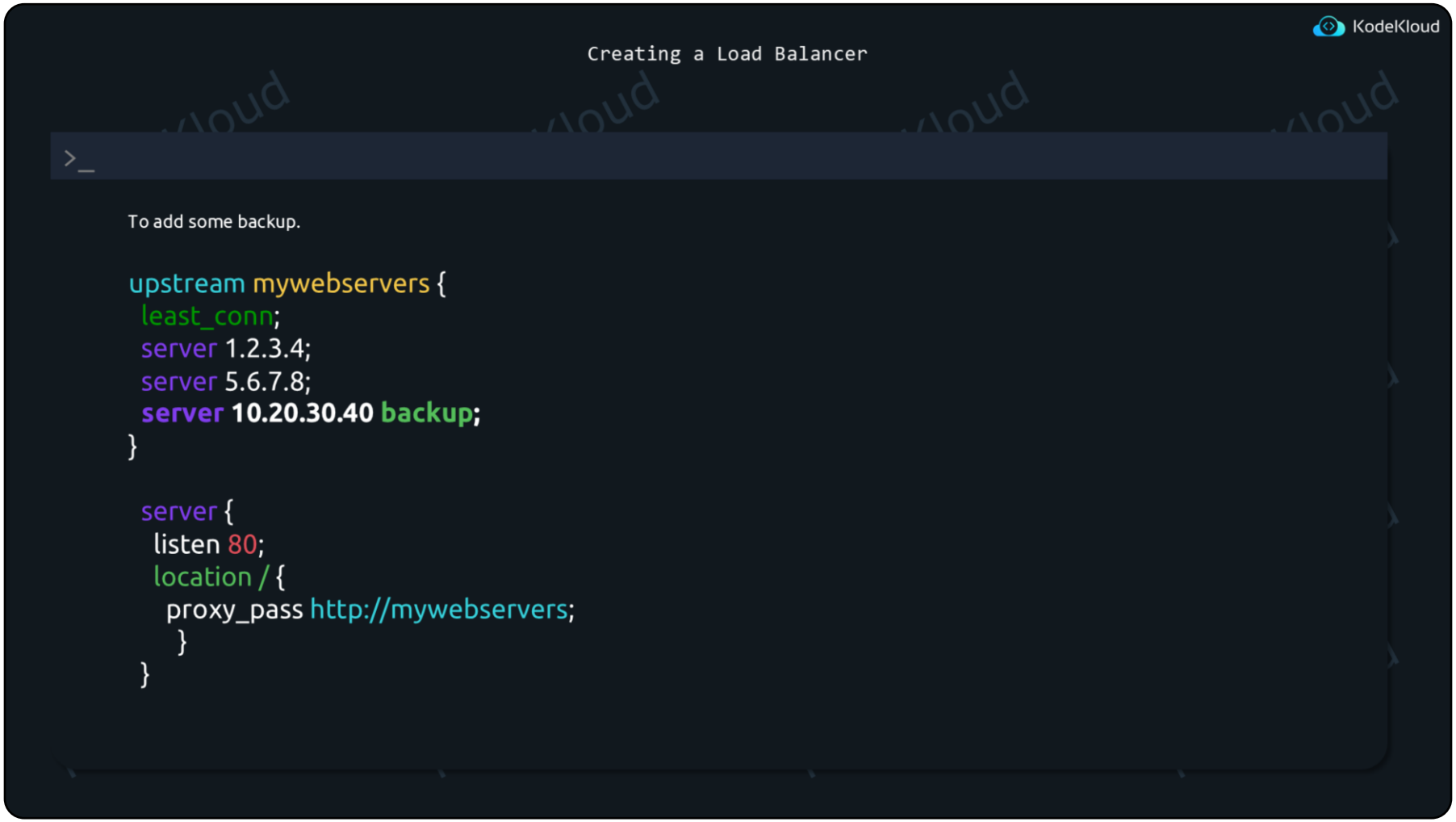
```
upstream mywebservers {
 least_conn;
 server 1.2.3.4 weight=3 down;
```

```
  server 5.6.7.8;
}

  server {
    listen 80;
    location / {
      proxy_pass http://mywebservers;
       }
  }
```

Note how we can add multiple settings to a single server specified here. We assigned it a weight equal to 3, but also marked it as "down".

Now Nginx will not redirect requests to this server anymore. And we can do maintenance work, restart it, and finally remove that keyword when the server is ready to accept requests again.



Here's a different scenario we can encounter. Maybe 2 servers are enough to handle web traffic. But if one server breaks, things will be slow with just one server left around. So we want to prevent this. We can add the following line:

```
upstream mywebservers {
 least_conn;
 server 1.2.3.4;
```

```
  server 5.6.7.8;
  server 10.20.30.40 backup;
}

  server {
    listen 80;
    location / {
      proxy_pass http://mywebservers;
       }
  }
```

What this does is add 10.20.30.40 to our collection of web servers, but it marks it as a backup. Which means traffic will not be load balanced to it yet. Instead, if 1.2.3.4, or 5.6.7.8 breaks down, then 10.20.30.40 becomes active and traffic begins to be redirected to it as well. So it sits on the sidelines and it's only used if necessary.

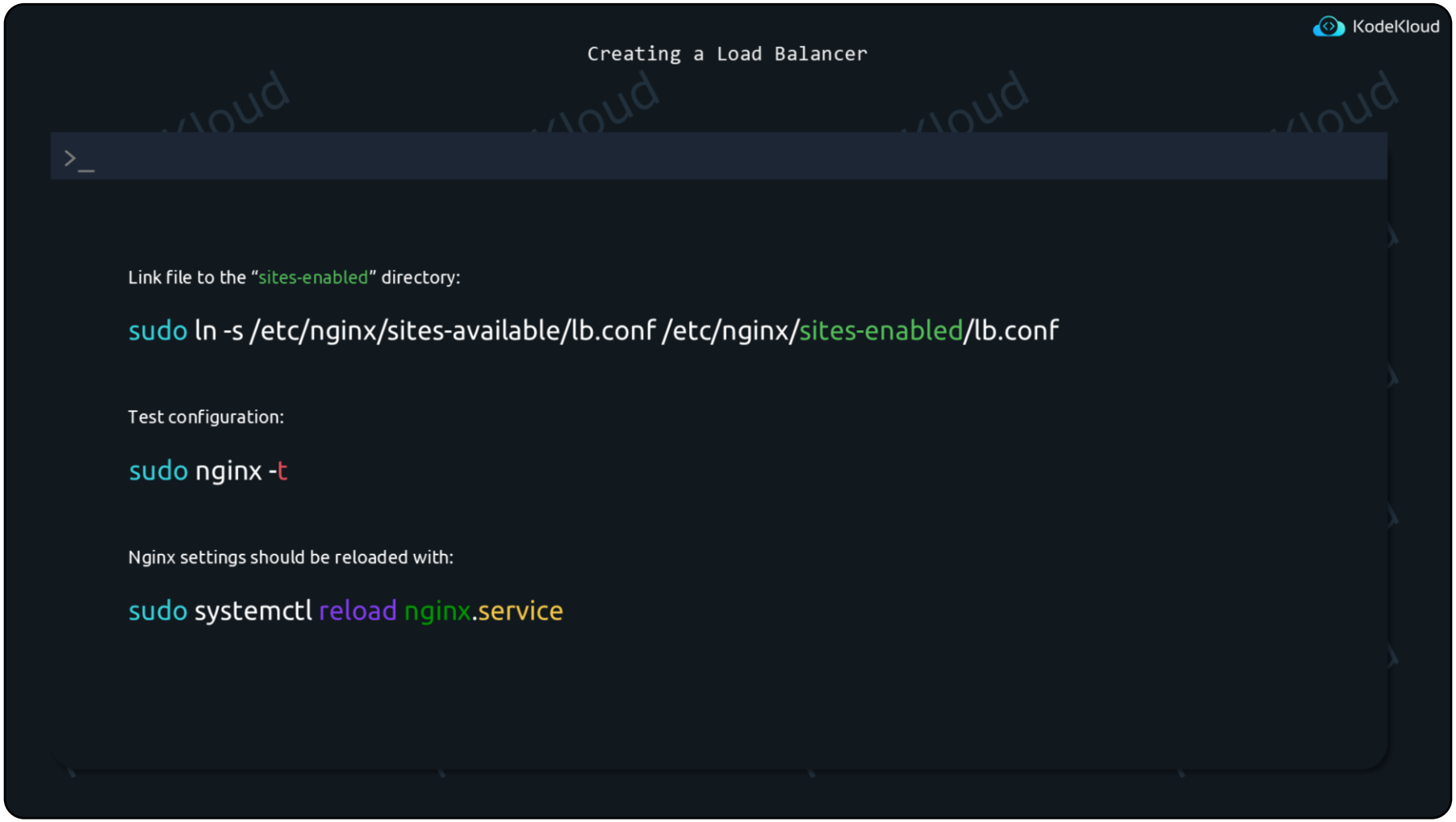
It's also worth mentioning that if these web servers are not listening for incoming connections on the standard port 80, we can specify a custom port like this:

```
upstream mywebservers {
 least_conn;
 server 1.2.3.4:8081;
```

```
  server 5.6.7.8;
  server 10.20.30.40 backup;
}

  server {
    listen 80;
    location / {
      proxy_pass http://mywebservers;
        }
  }
```

In this case, we specified that server 1.2.3.4 is listening for incoming connections on port 8081.



After a config file for the load balancer has been created, the rest of the steps are identical to what we did for the reverse proxy.

Meaning, first we should link this file to the "sites-enabled" directory:

sudo ln -s /etc/nginx/sites-available/lb.conf /etc/nginx/sites-enabled/lb.conf

Then the configuration should be tested:

sudo nginx -t

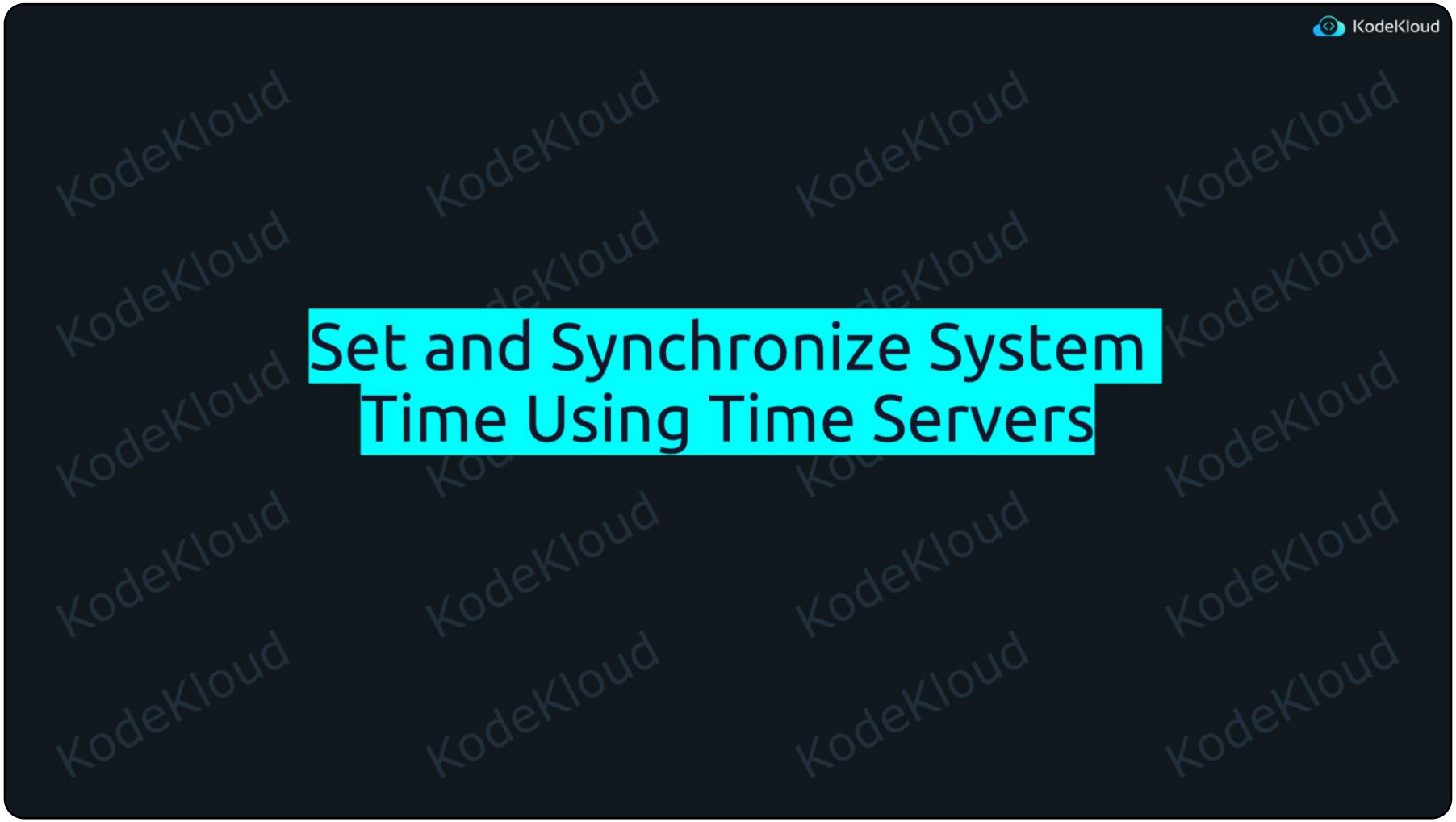
And finally, Nginx settings should be reloaded with:

sudo systemctl reload nginx.service

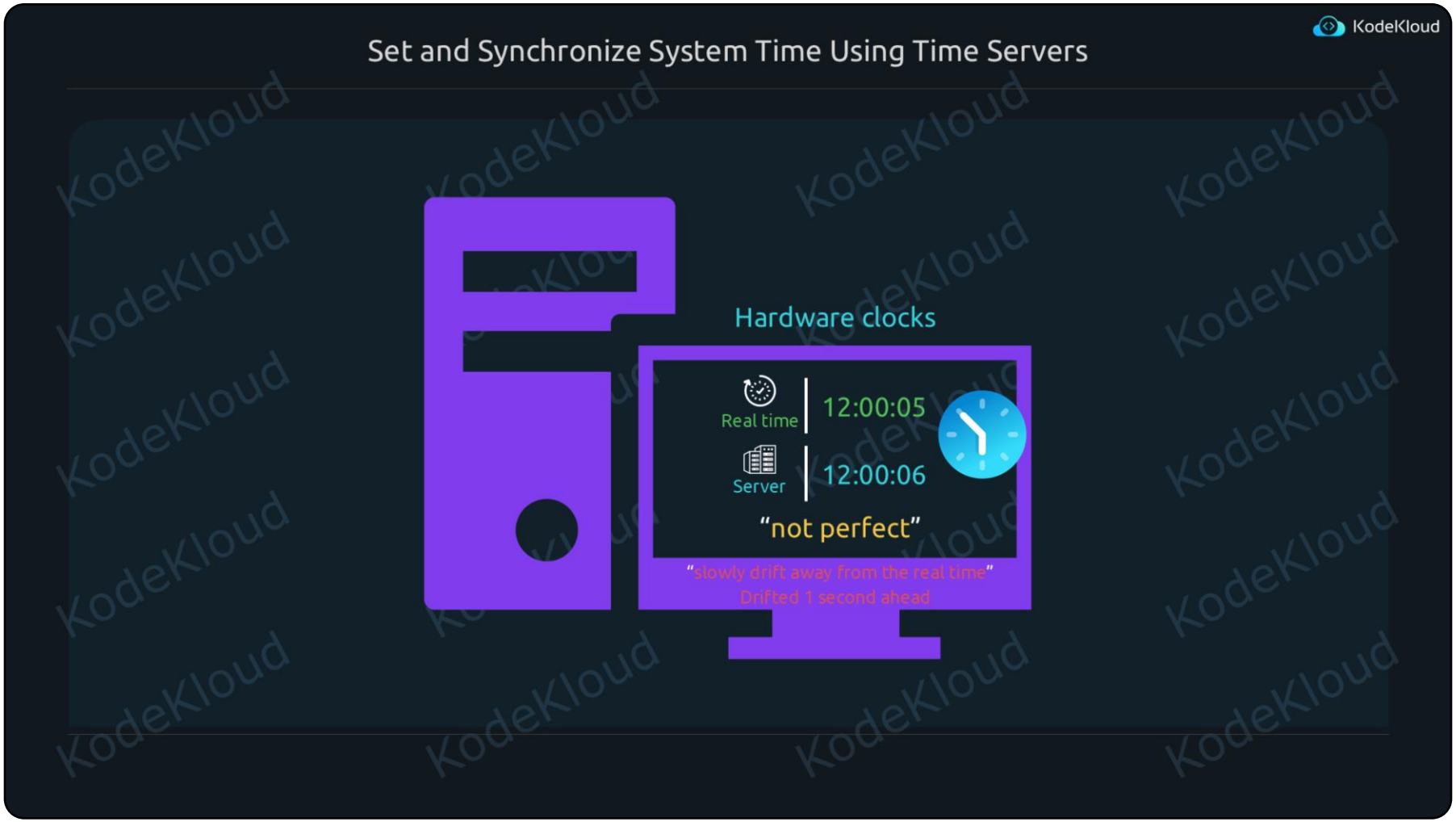
That's all for this lesson. See you in the next one.

KodeKloud

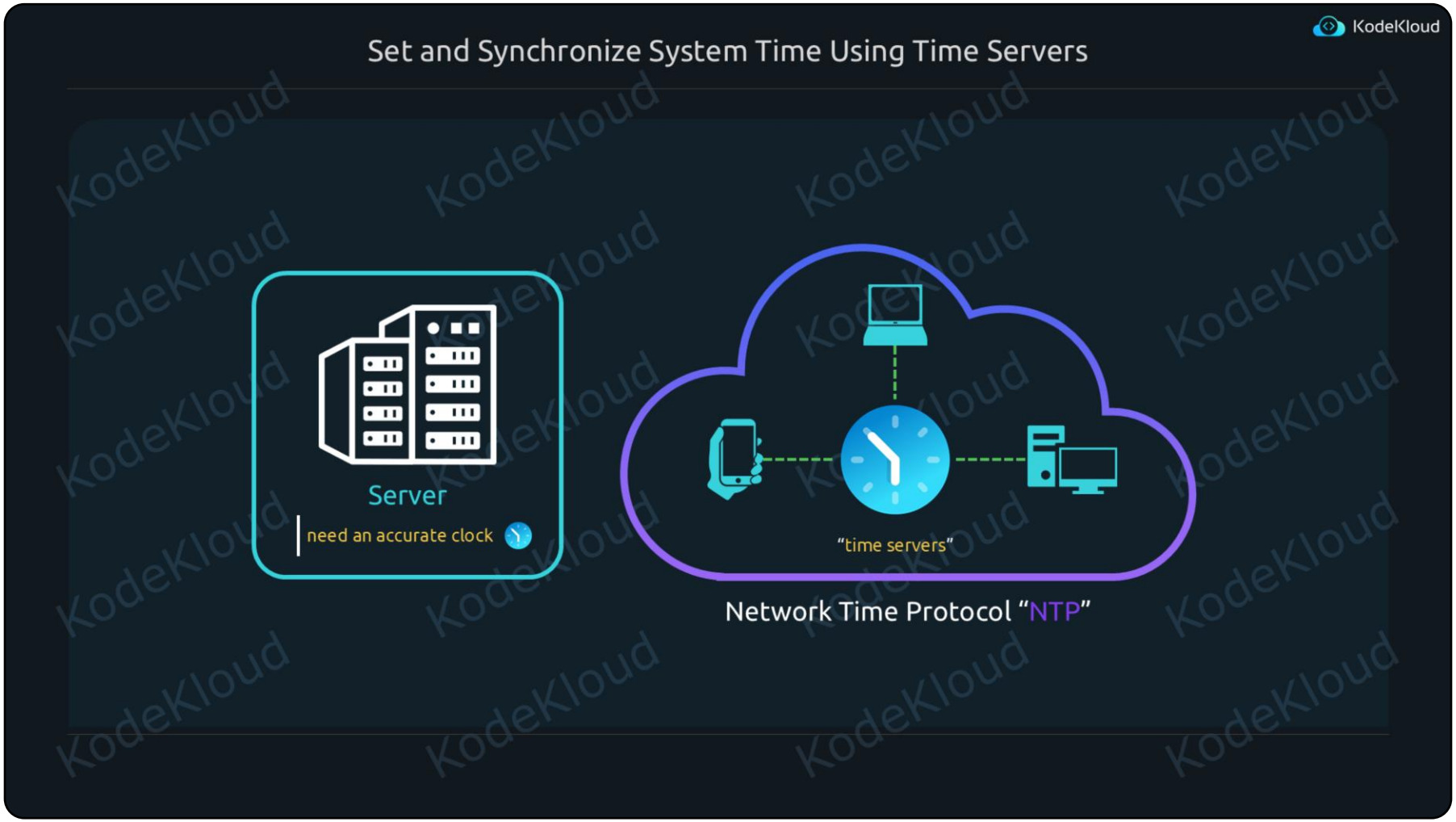
Visit www.kodekloud.com to discover more.



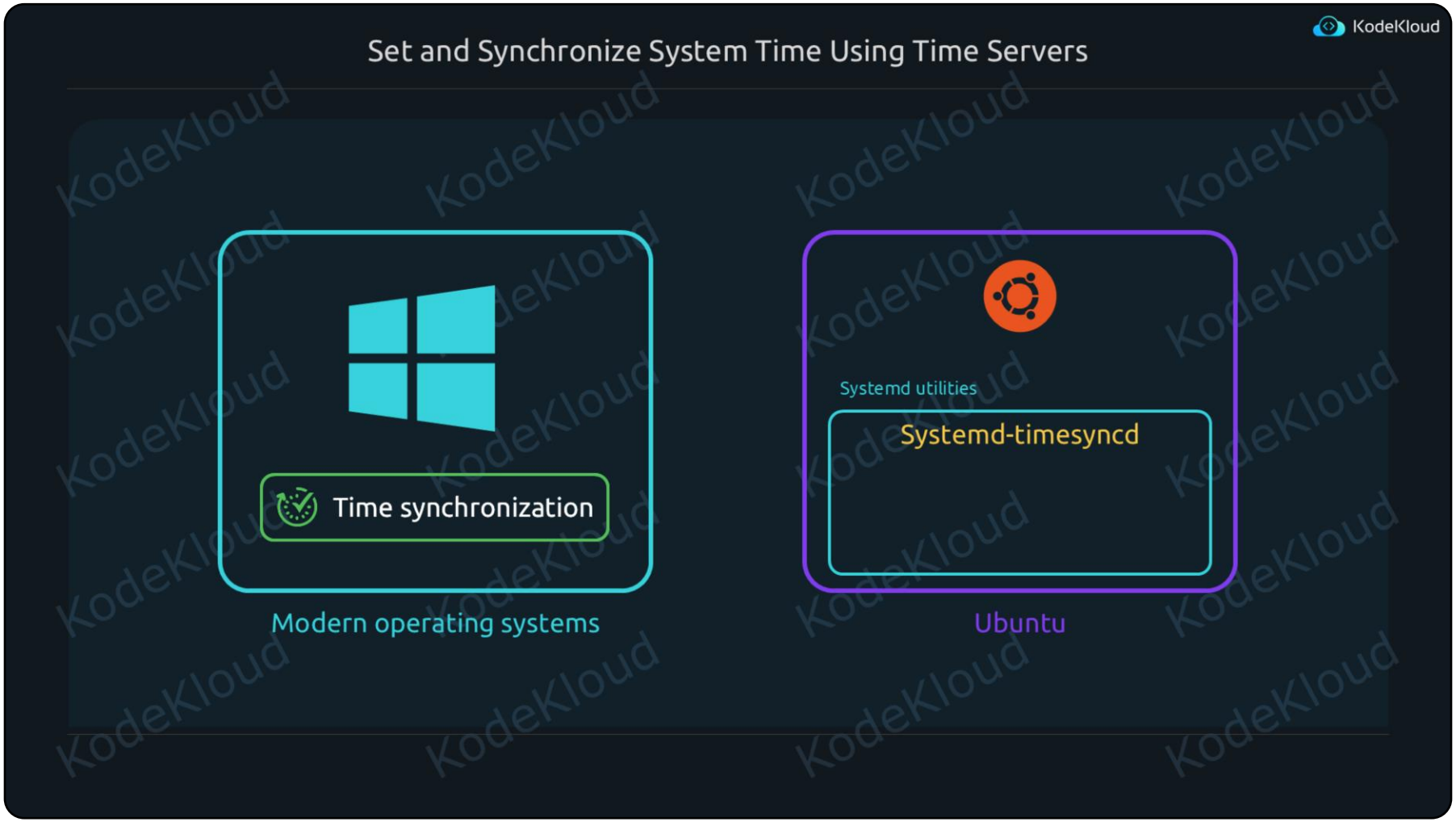
In this lesson we'll learn how to configure our servers to keep the clock accurate.



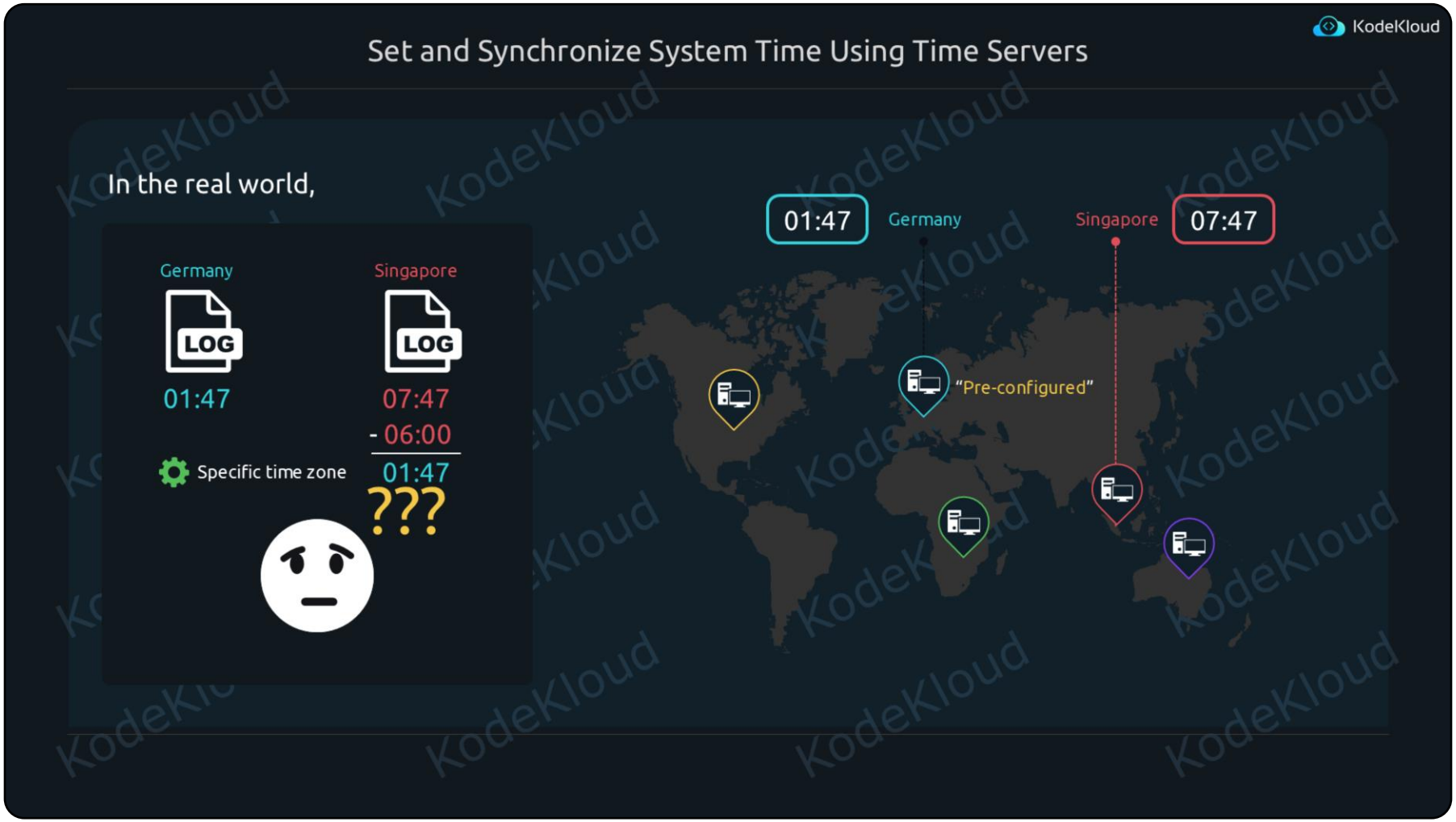
Hardware clocks in computers are not perfect. They slowly drift away from the real time. Drifting means that after a few days the real time might be 12:00:05 but our server might show 12:00:06. In this case, it drifted 1 second ahead.



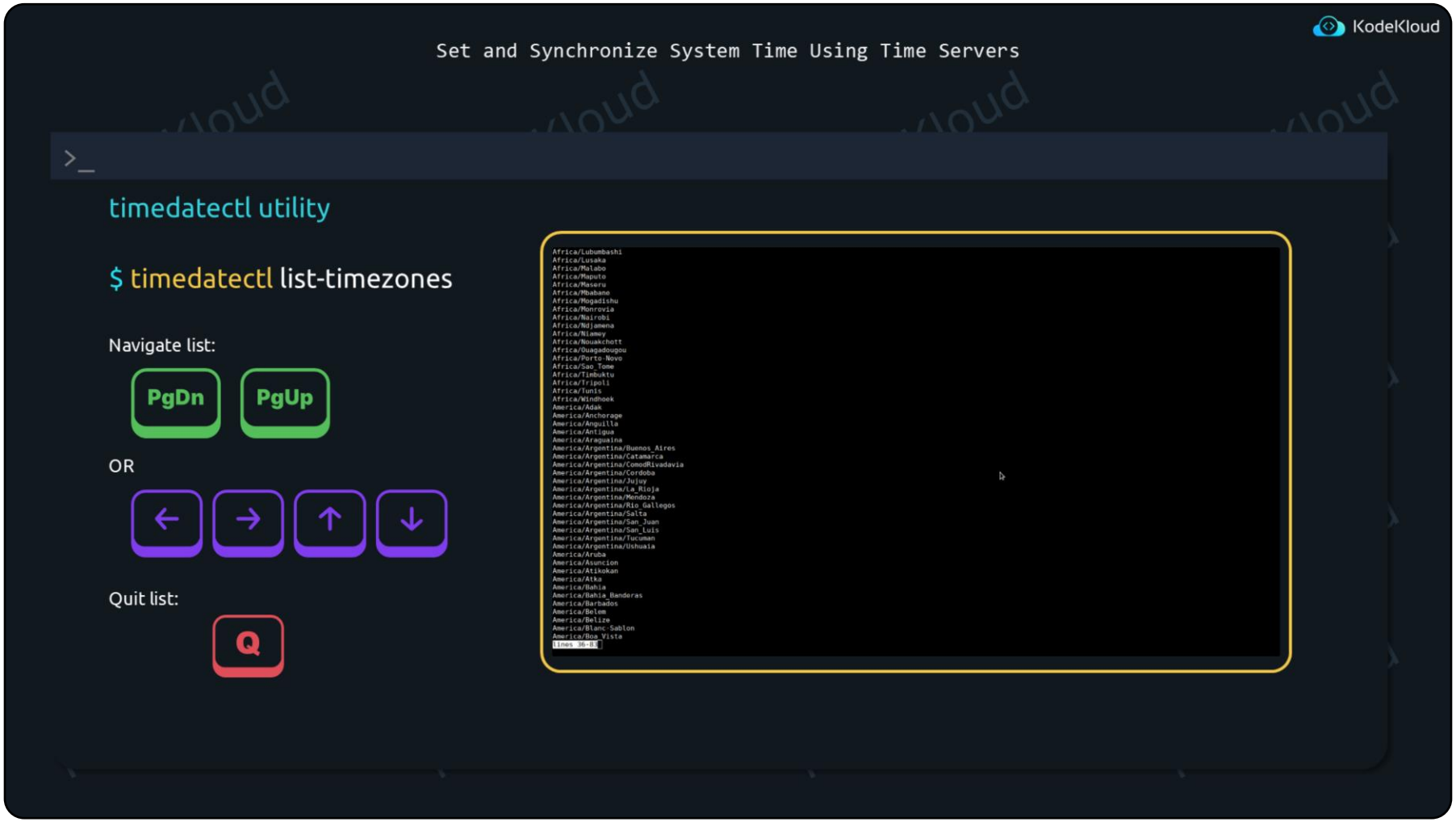
Servers need an accurate clock. But you might have noticed that, nowadays, we almost never have to manually set our clocks. Our phones, laptops, and servers somehow magically know what time it is. That's because they periodically get the exact time from the Internet. And they get it from what are called "time servers". More specifically, NTP servers. NTP is an abbreviation for Network Time Protocol.



Most modern operating systems include time synchronization software by default. In Ubuntu, the default one is included in the suite of systemd utilities, and it's called systemd-timesyncd. But before learning how to work with that utility, we need to cover another important thing.



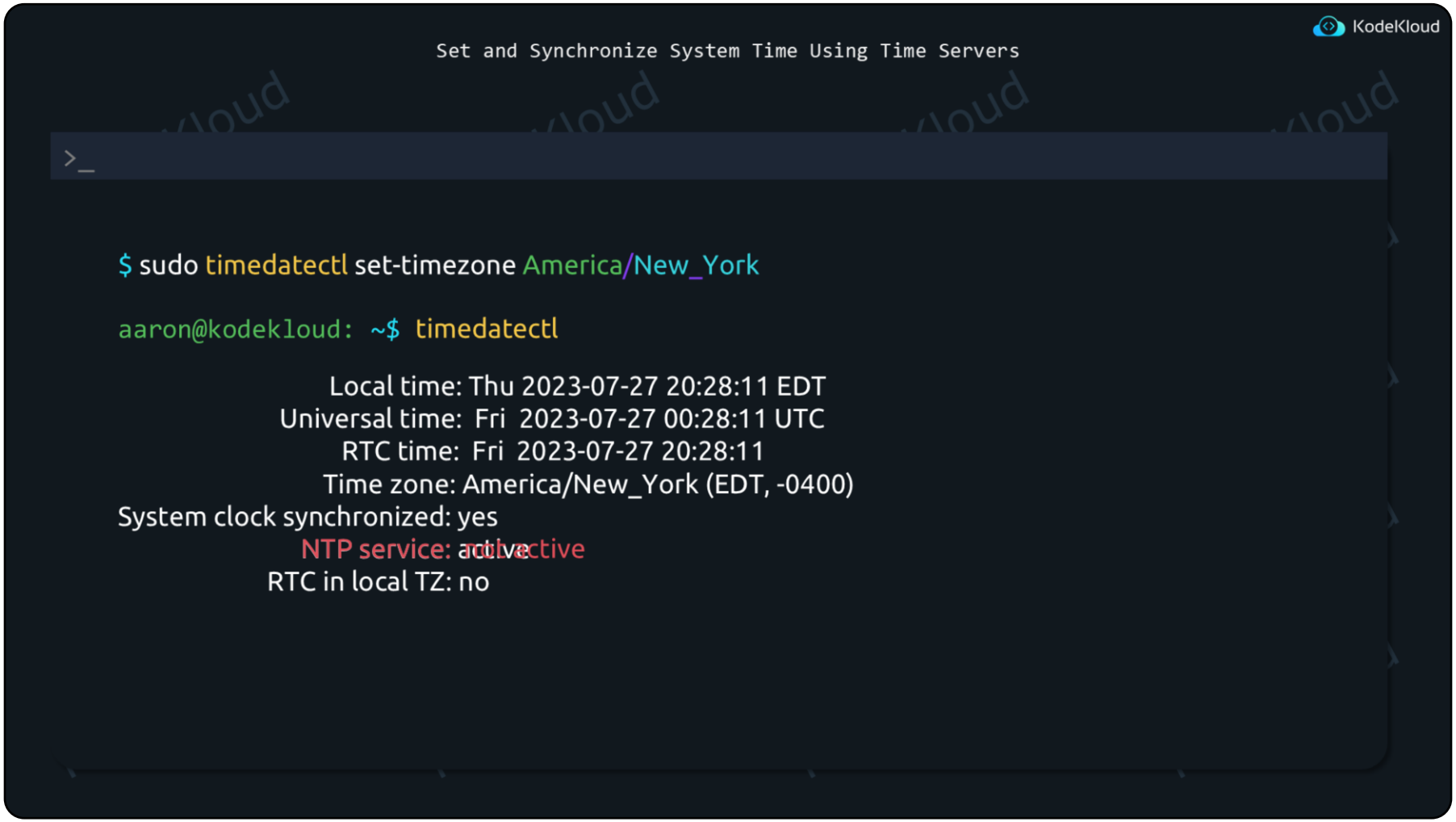
In the real world, we have a lot of time zones. When it's 01:47 in Germany, it's 07:47 in Singapore. If we launch a server in Germany, it might be pre-configured to use that zone. Now imagine we have servers in 5 different countries. Looking at logs would get confusing. We'd constantly have to add or subtract a few hours to figure out when a certain logged event took place, relative to our own country's time. So the first thing we might want to do after launching a server is to set a specific time zone for it. Preferably, the time zone where we live in, or where our company's main office is.



Time-related operations can be performed with the timedatectl utility. To see the available time zones we can type:

timedatectl list-timezones

We can navigate this list with the PAGE UP and PAGE DOWN keys. Or with the arrow keys. And we can quit by pressing "q".



We can see that these zones are specified by typing the name of the continent first, followed by a "/" slash character, then the name of a city; usually the capital city of a country or state. To set our timezone to New York, we would type:

sudo timedatectl set-timezone America/New_York

Note that when the name of the city contains two words, we separate them with the "_" underscore character, and not a

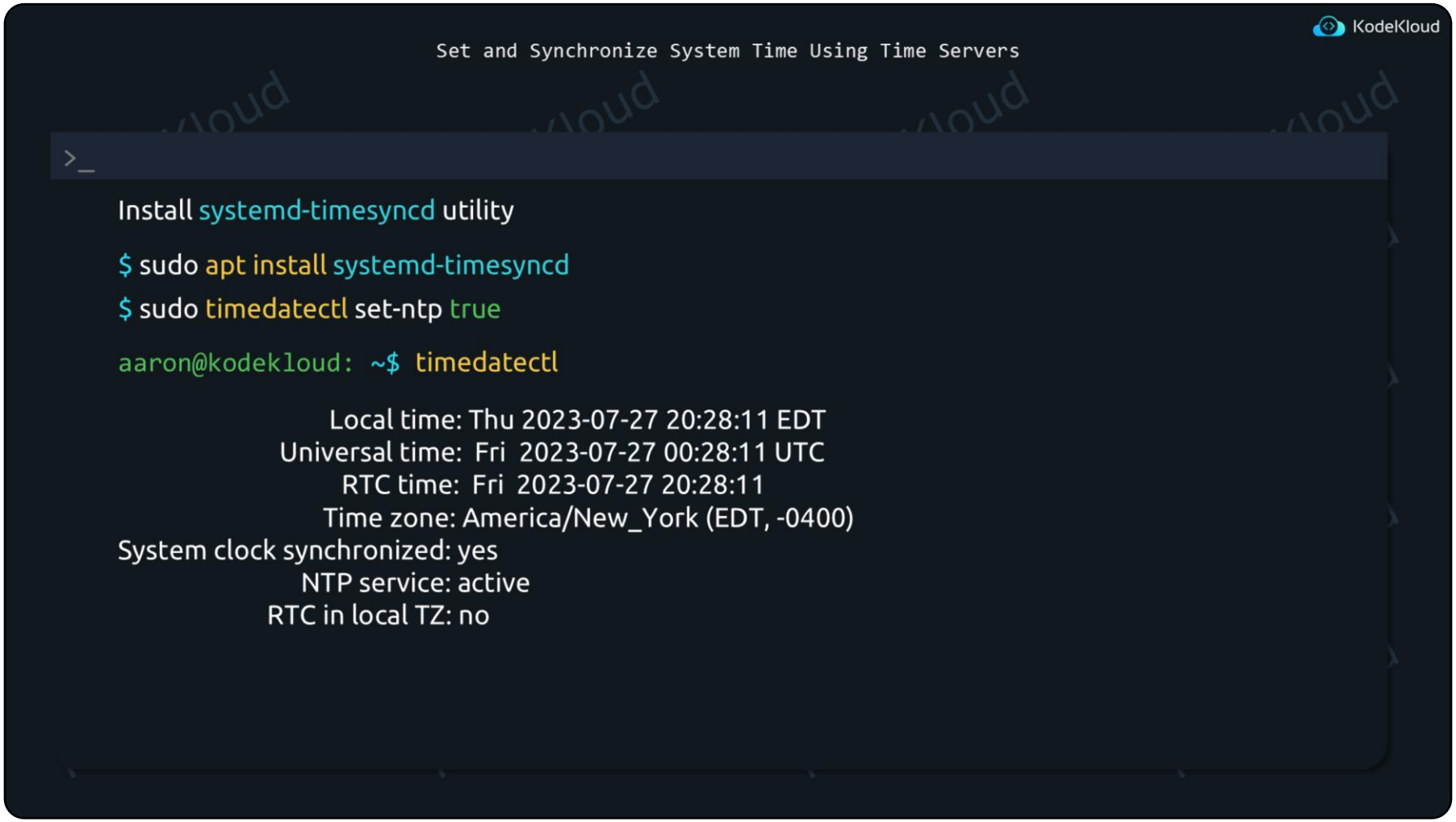
space character.

We can check if our change took effect by simply typing:

timedatectl

We'll see that the time zone has changed, and "Local time" will show us the current time in New York.

Note that this output also tells us if any NTP service is active. If it's active, time is automatically synchronized with accurate time servers from the Internet.



If we encounter a server where timedatectl says the NTP service is not active, here's what we can do.

First, we can install the systemd-timesyncd utility by typing:

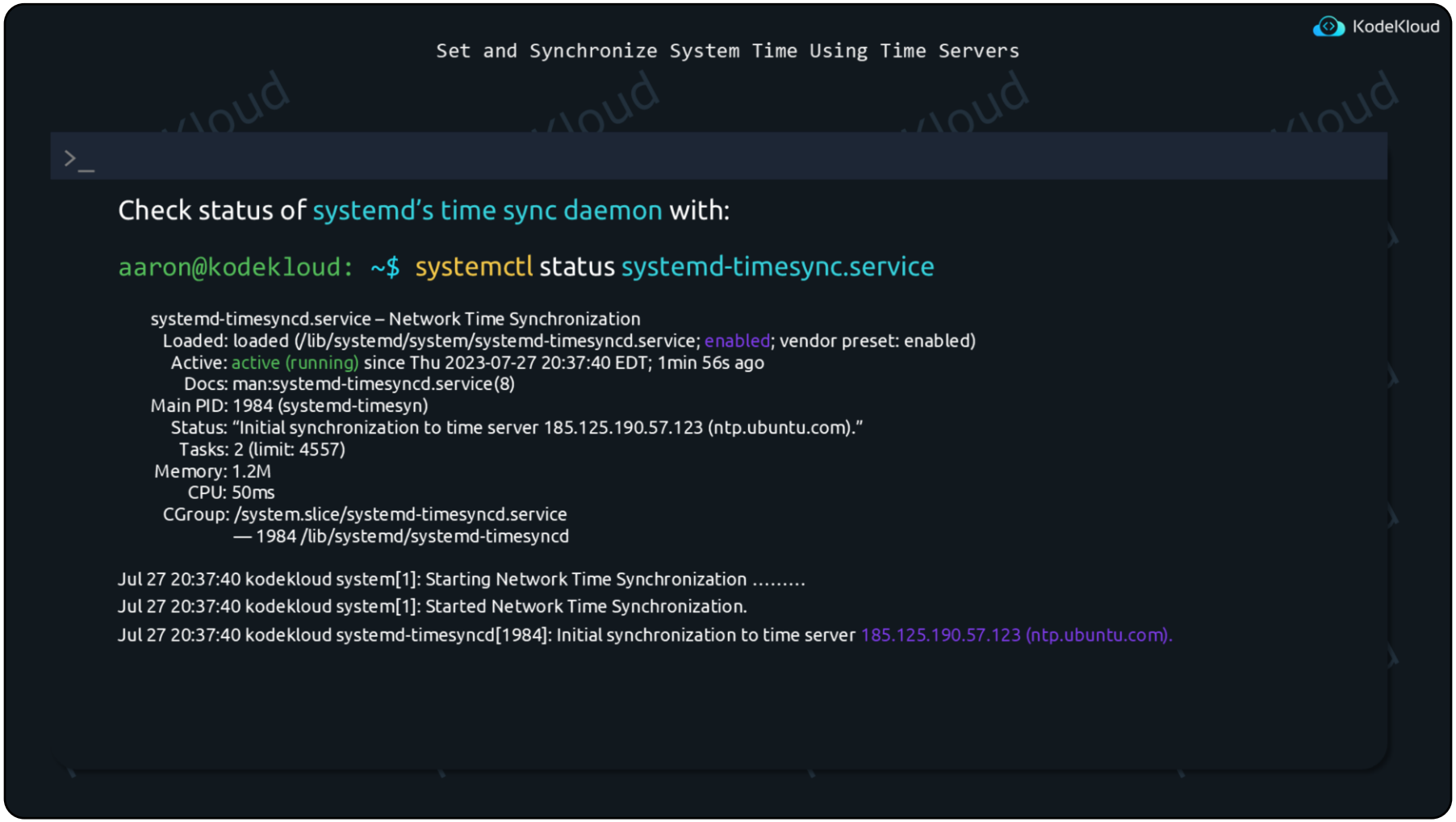
sudo apt install systemd-timesyncd

Then we can tell timedatectl to turn on synchronization with NTP servers:

```
sudo timedatectl set-ntp true
```

Finally, we can use the timedatectl command, with no arguments, to see if the NTP service is now active.

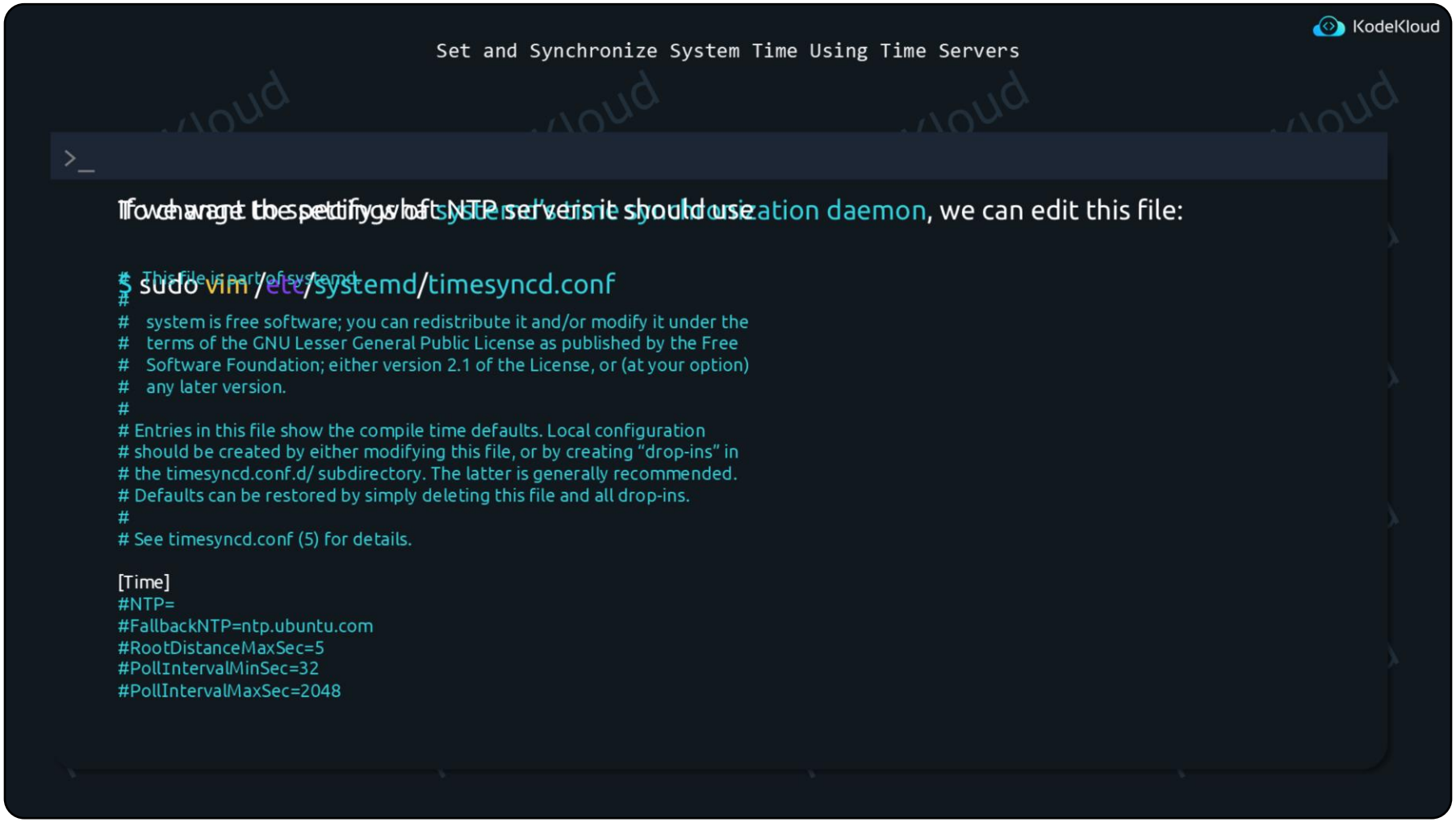
```
timedatectl
```



We can also check the status of systemd's time sync daemon with:

systemctl status systemd-timesyncd.service

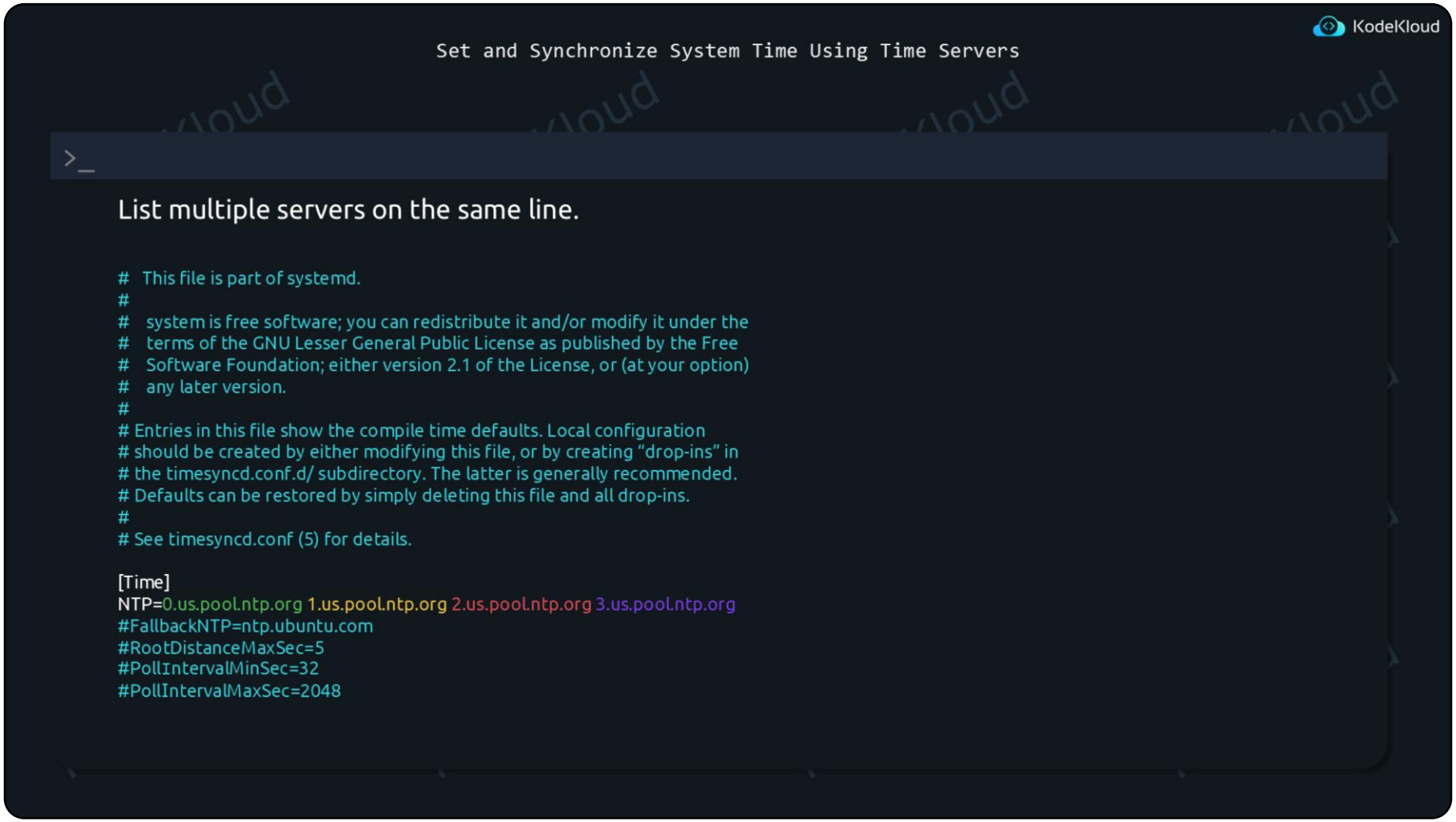
It should show us that the service is active, and enabled to autostart at boot. We'll also see a few log lines showing us the NTP server it's using.



To change the settings of systemd's time synchronization daemon, we can edit this file:

sudo vim /etc/systemd/timesyncd.conf

If we want to specify what NTP servers it should use,



we can uncomment this line by removing the preceding "#" hash sign:

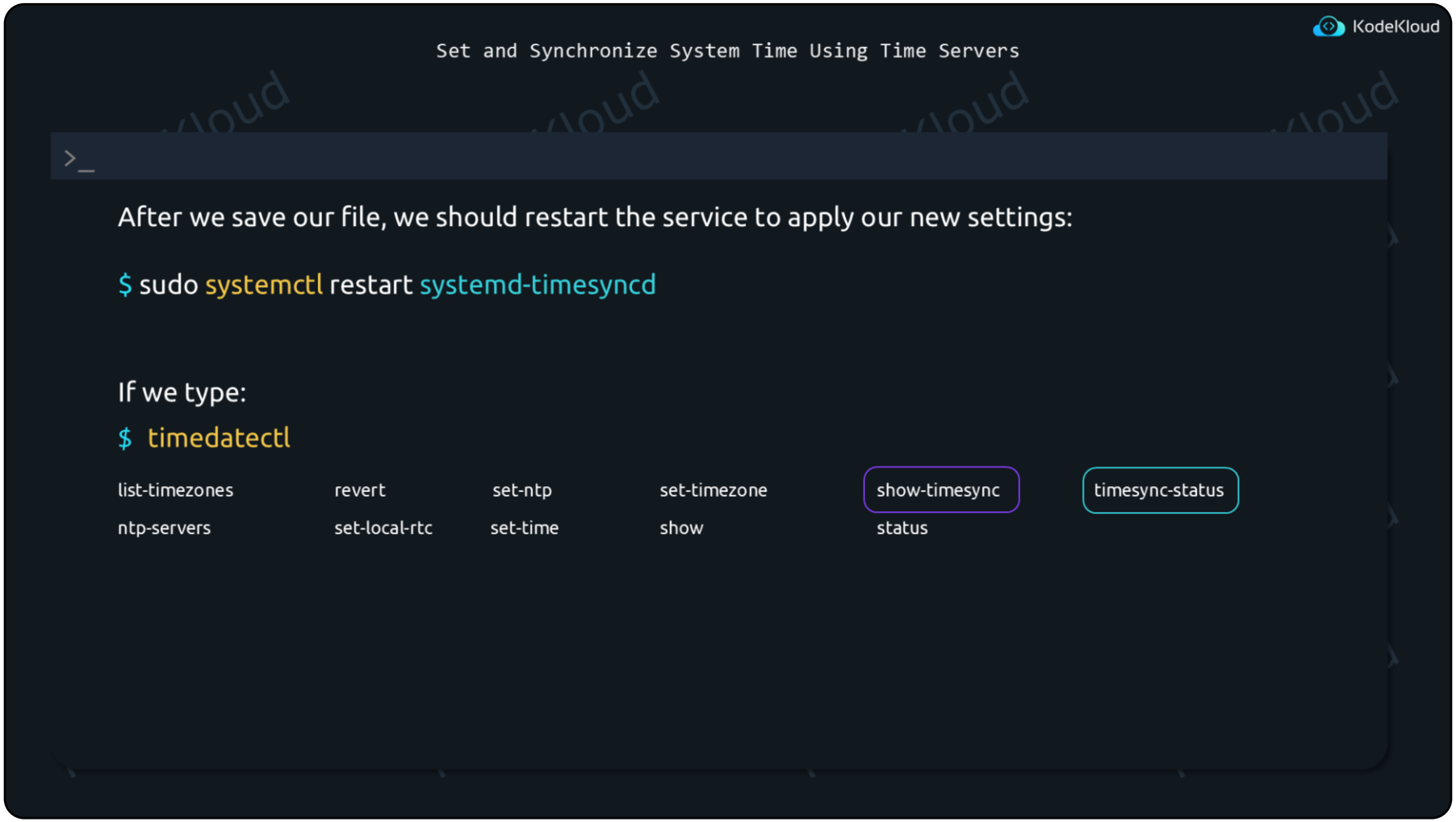
We can list multiple servers on the same line. For example, we can type:

0.us.pool.ntp.org 1.us.pool.ntp.org 2.us.pool.ntp.org 3.us.pool.ntp.org

to use these four.

Note the format for the hostnames of these NTP servers. They're often laid out in a similar way. If the main NTP website is hosted at ntp.org, then the NTP servers might be referenced by a subdomain such as us.pool.ntp.org. In this case, "us" is the country code for the United States. And each collection of NTP servers is assigned to a further subdomain, represented by a number, from 0 to n. So the first collection will be assigned to the 0.us.pool.ntp.org subdomain. The second collection to 1.us.pool.ntp.org, and so on.

The rest of the commented lines show us timesyncd's default options. The FallbackNTP will be used if nothing is specified on the NTP line. Hence, that was used before we made our modifications.
The RootDistanceMaxSec entry specifies the maximum time, in seconds, that it can take for the NTP server to send network data to our machine. If the NTP server exceeds this time, then the timesyncd utility will move on to a different NTP server. The PollIntervals specifiy how often the timesyncd utility should contact NTP servers. For example, we can see a minimum poll interval of 32 seconds here, and a maximum of 2048. This means that it should wait at least 32 seconds to ask for new data. But it shouldn't wait more than 2048 seconds. After making its first request, timesyncd will adjust the poll interval between these two minimum and maximum values. So it can be something random, like 248 seconds. It will choose the poll interval based on the clock drift it observes. If the server's clock doesn't drift too bad, it can poll less often.



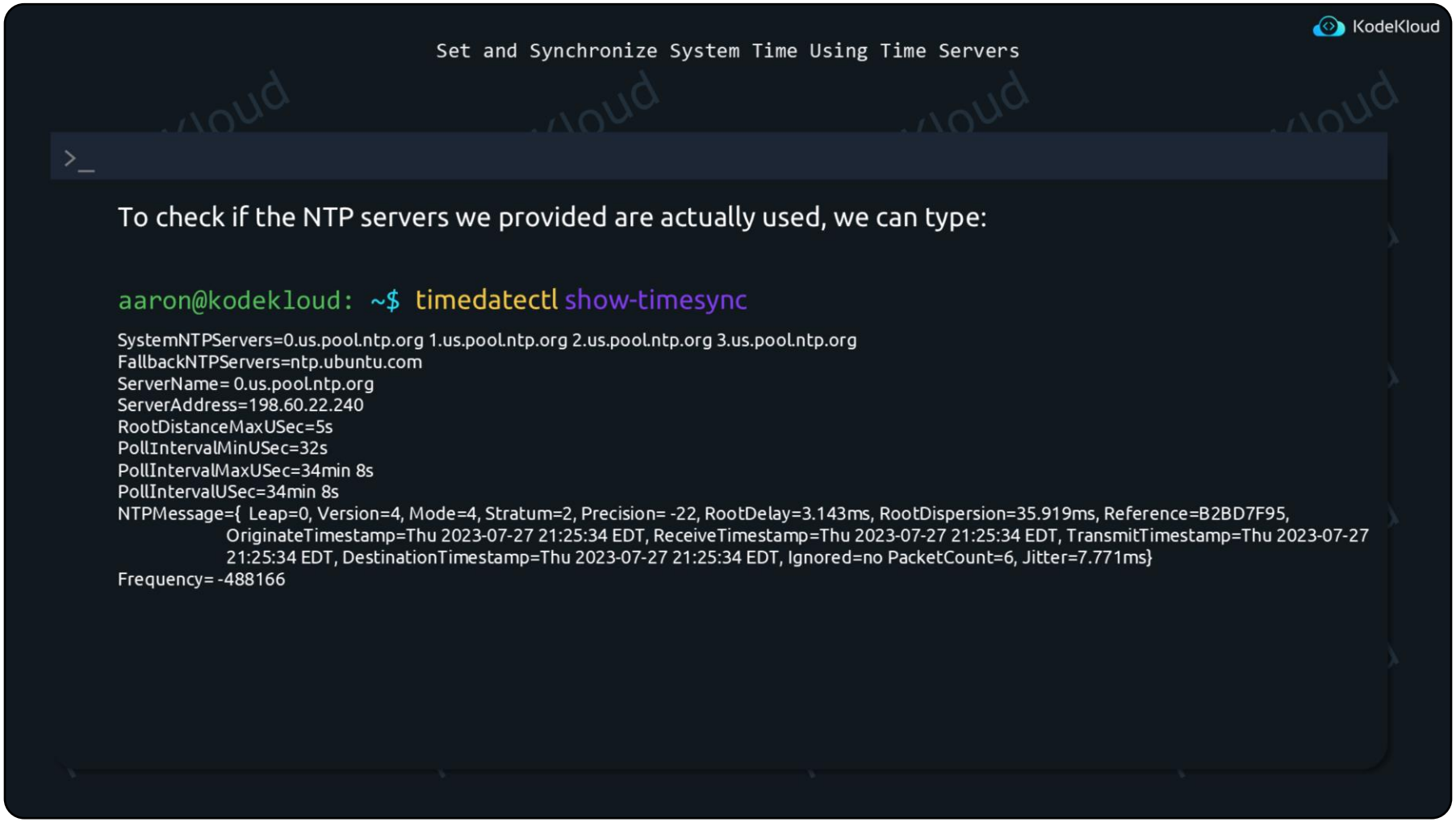
After we save our file, we should restart the service to apply our new settings:

sudo systemctl restart systemd-timesyncd

If we type:

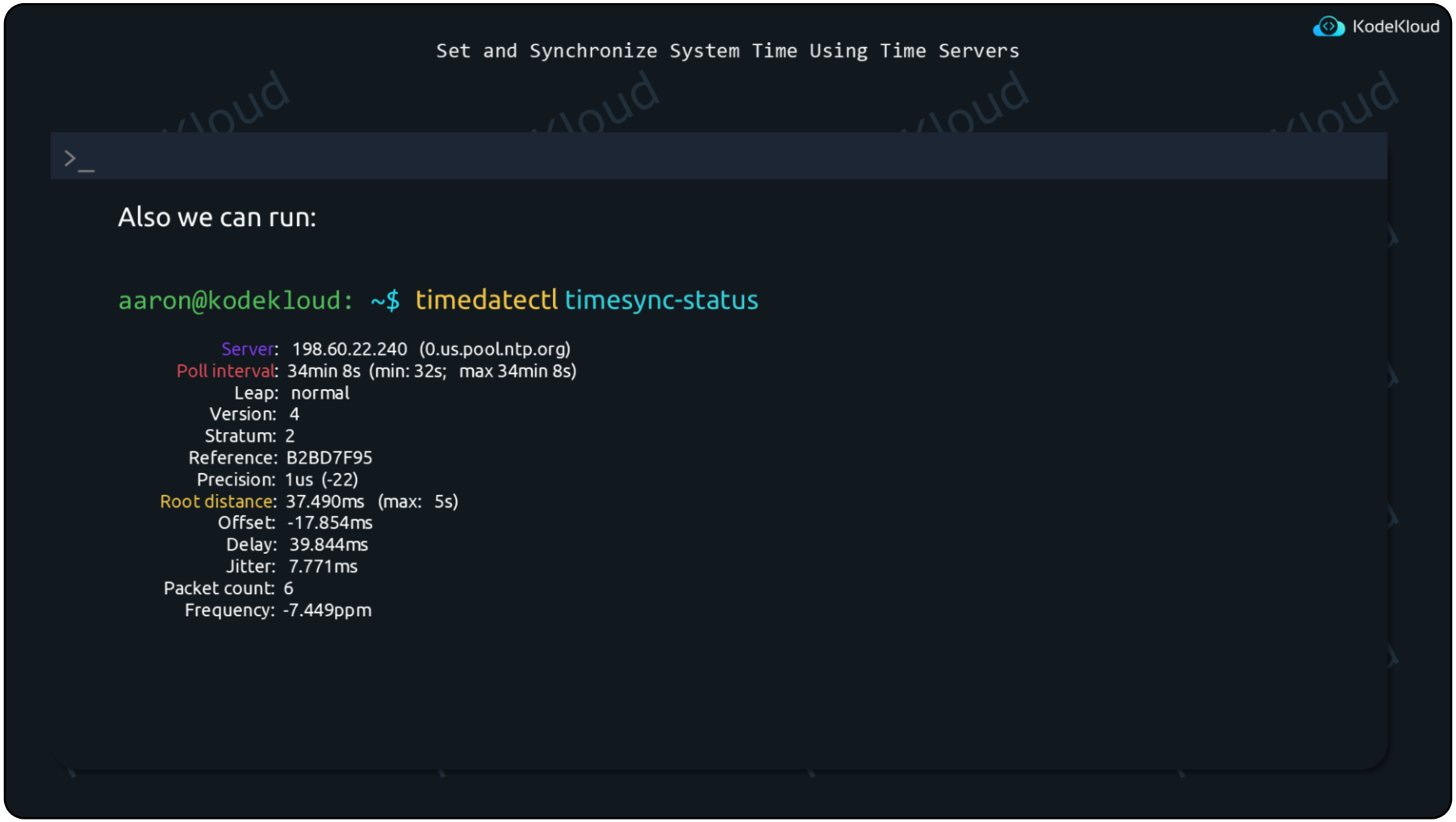
timedatectl

then add a space character, and press the TAB key two times, we can see a few other commands that can help us here: show-timesync and timesync-status.



To check if the NTP servers we provided are actually used, we can type:

timedatectl show-timesync



Also, we can run:

timedatectl timesync-status

to see both the current poll interval, and the root distance for the NTP server used.

This wraps up our lesson about time synchronization. Let's progress to the next one.

KodeKloud

Visit www.kodekloud.com to discover more.