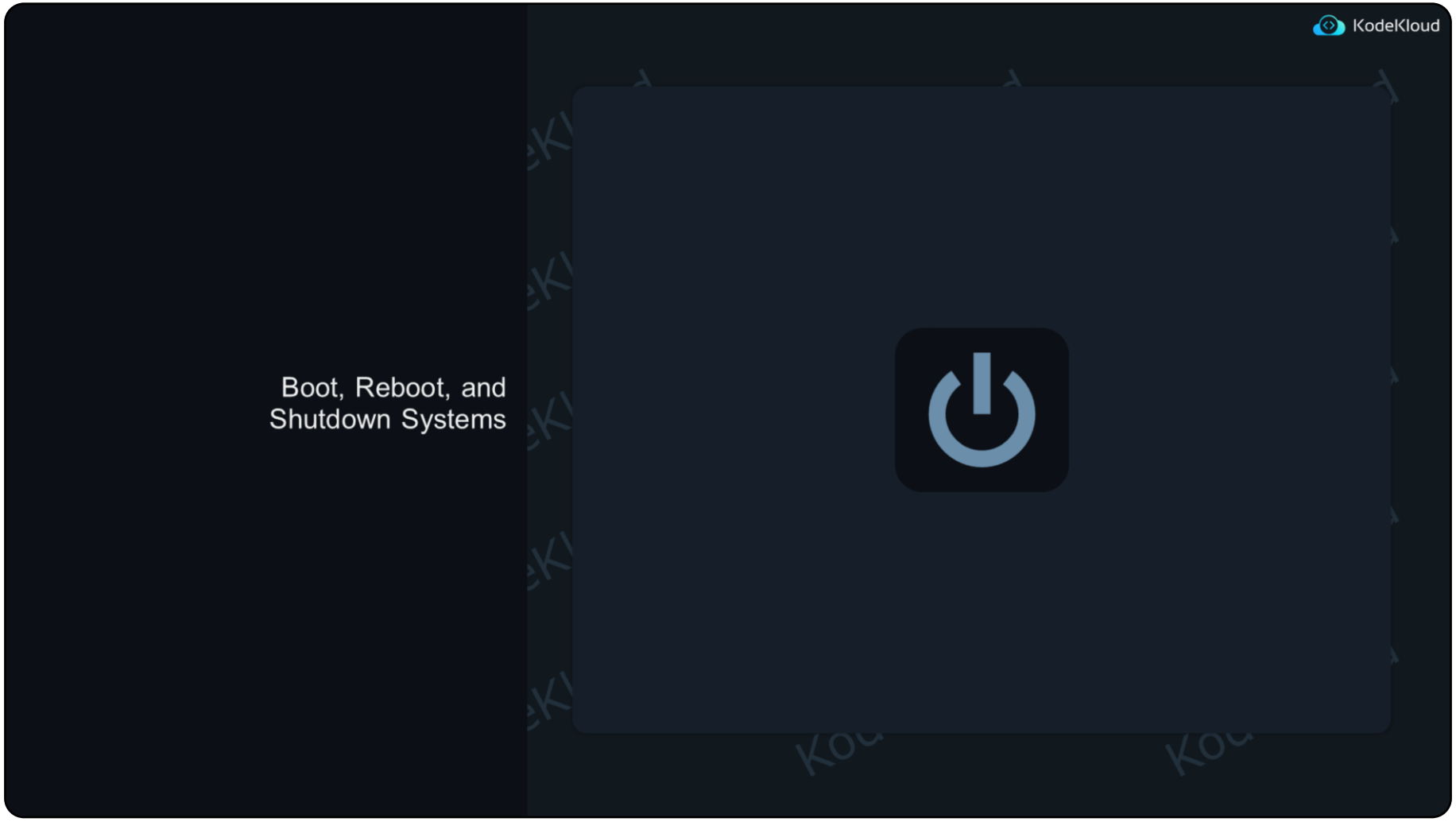
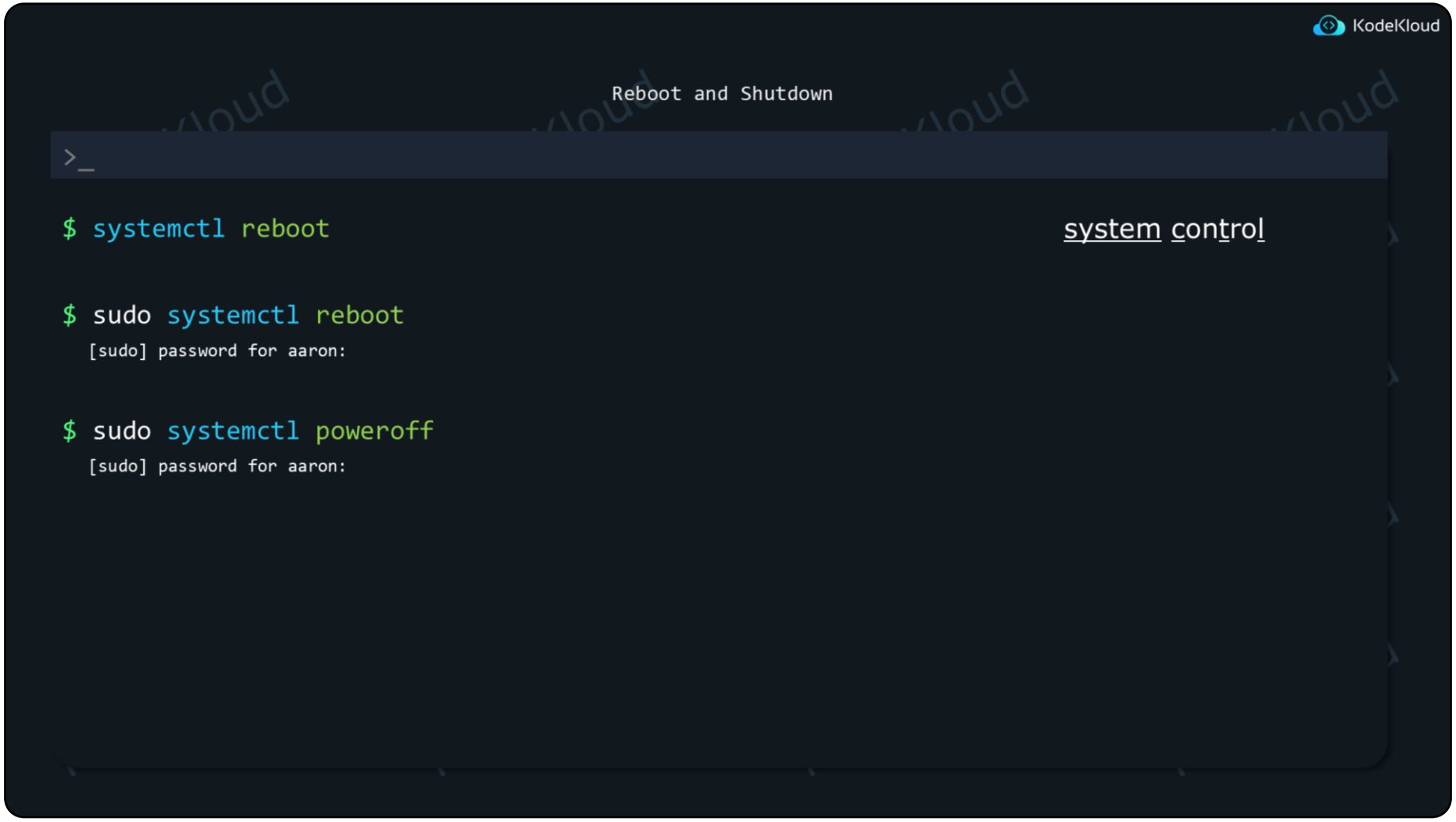
KodeKloud

Visit www.kodekloud.com to discover more.



Let’s look at how to boot, reboot, and shutdown Linux systems safely.



To reboot or shutdown a Linux machine we'll often use the systemctl (system control) command.

Some commands require system administrator privileges. The root user has such privileges. So, we can reboot a machine by simply typing systemctl reboot. Regular users cannot use commands that change the system's state. But they can temporarily get root privileges, if they add sudo in front of their commands. So, a regular user needs to type sudo systemctl reboot instead.

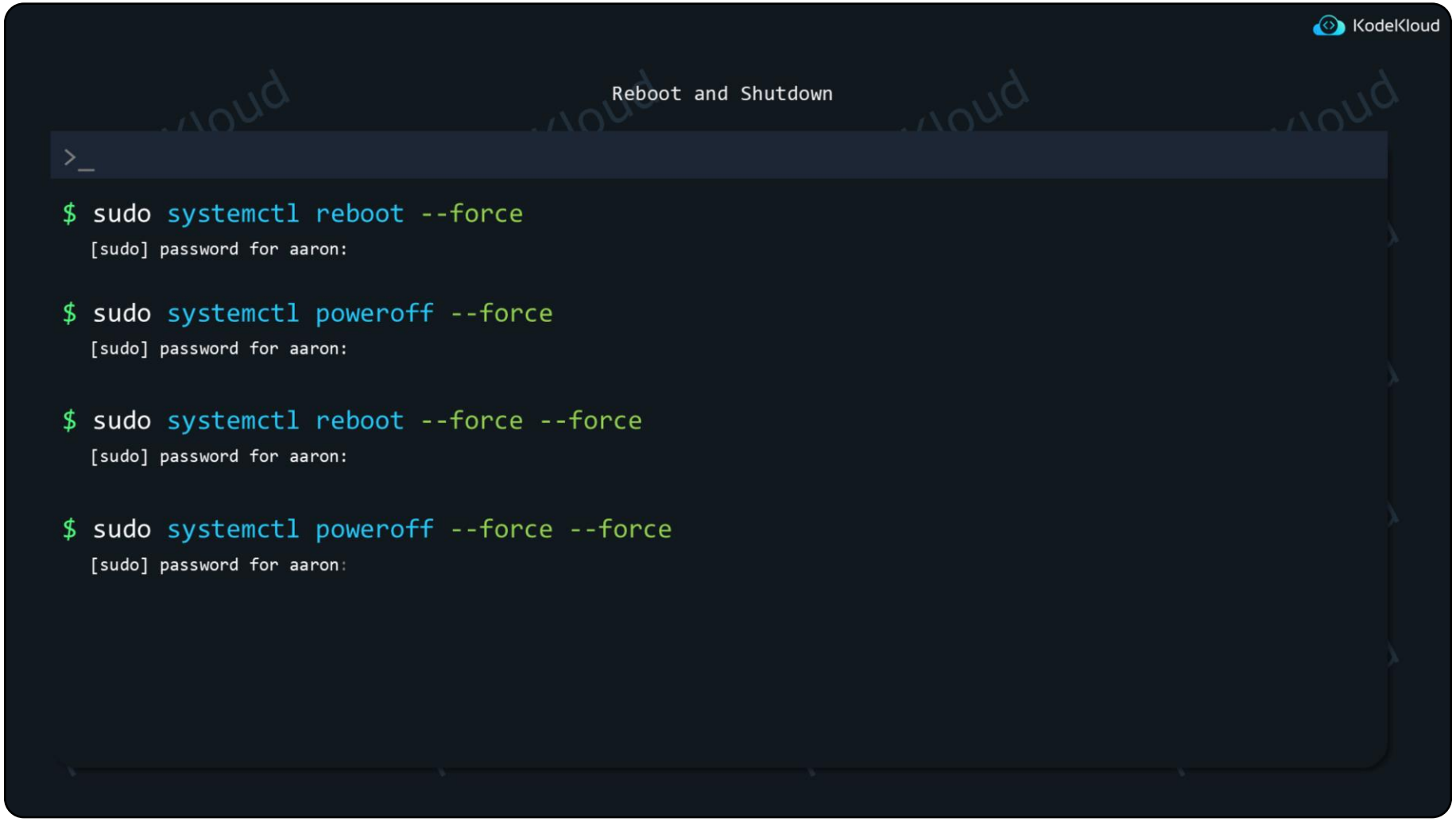
All you need to remember is this: "If I am logged in as root, I don't need sudo". So, you can skip writing the "sudo" word if you're already logged in as root. This applies to all other examples in these lessons where you see sudo.

Now to recap, to reboot we type:

sudo systemctl reboot

And to shutdown we type:

sudo systemctl poweroff



Rarely, you might find yourself in situations where the system refuses to reboot or shutdown normally. That might be because some program is misbehaving, stuck in some way and it does not want to close properly. In such abnormal situations, you can force close all such programs and reboot in a more abrupt way (not recommended to do unless absolutely necessary).

sudo systemctl reboot --force

sudo systemctl poweroff --force

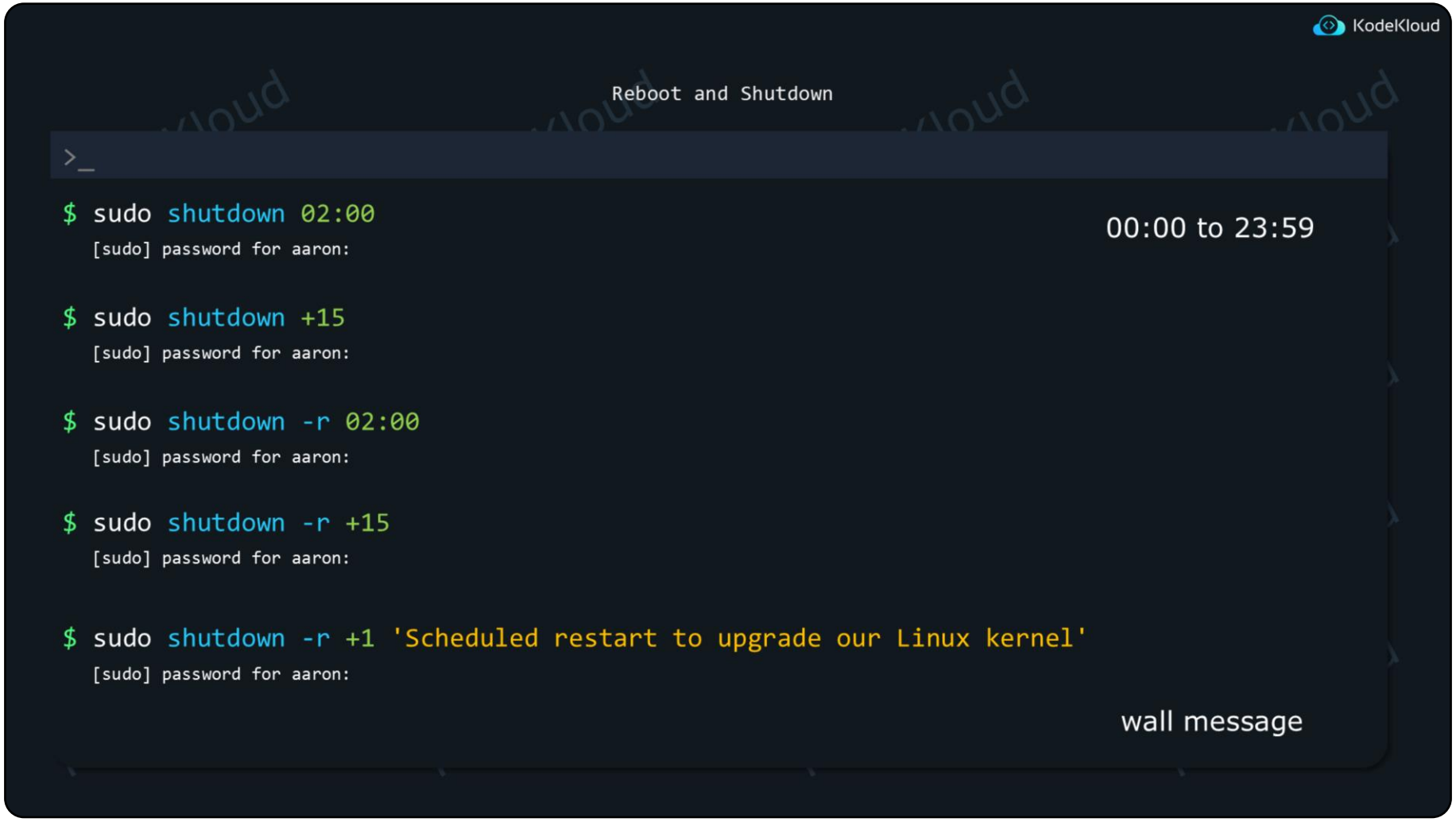
If not even this works, you can specify --force twice (only use as last resort):

sudo systemctl reboot --force --force

This is exactly like pressing the reset button. The system reboots instantly, programs have no chance to close properly or save their data.

systemctl poweroff --force --force

This is exactly like unplugging a computer from its power source.



You'll often find that you need to reboot some servers in the middle of the night, say 2 or 3AM. It's inconvenient to have to wake up just to reboot some device, so you can instead instruct Linux to do this on its own.

The shutdown command is better suited for scheduled reboots or shutdowns.

To shutdown at 02:00 AM:

sudo shutdown 02:00

The time is in 24-hour format, so you can use anything between 00:00 and 23:59.

If you want to shutdown x minutes later, use +x instead. To shutdown after 15 minutes:

sudo shutdown +15

To reboot instead, add the -r, reboot option:

sudo shutdown -r 02:00

sudo shutdown -r +15

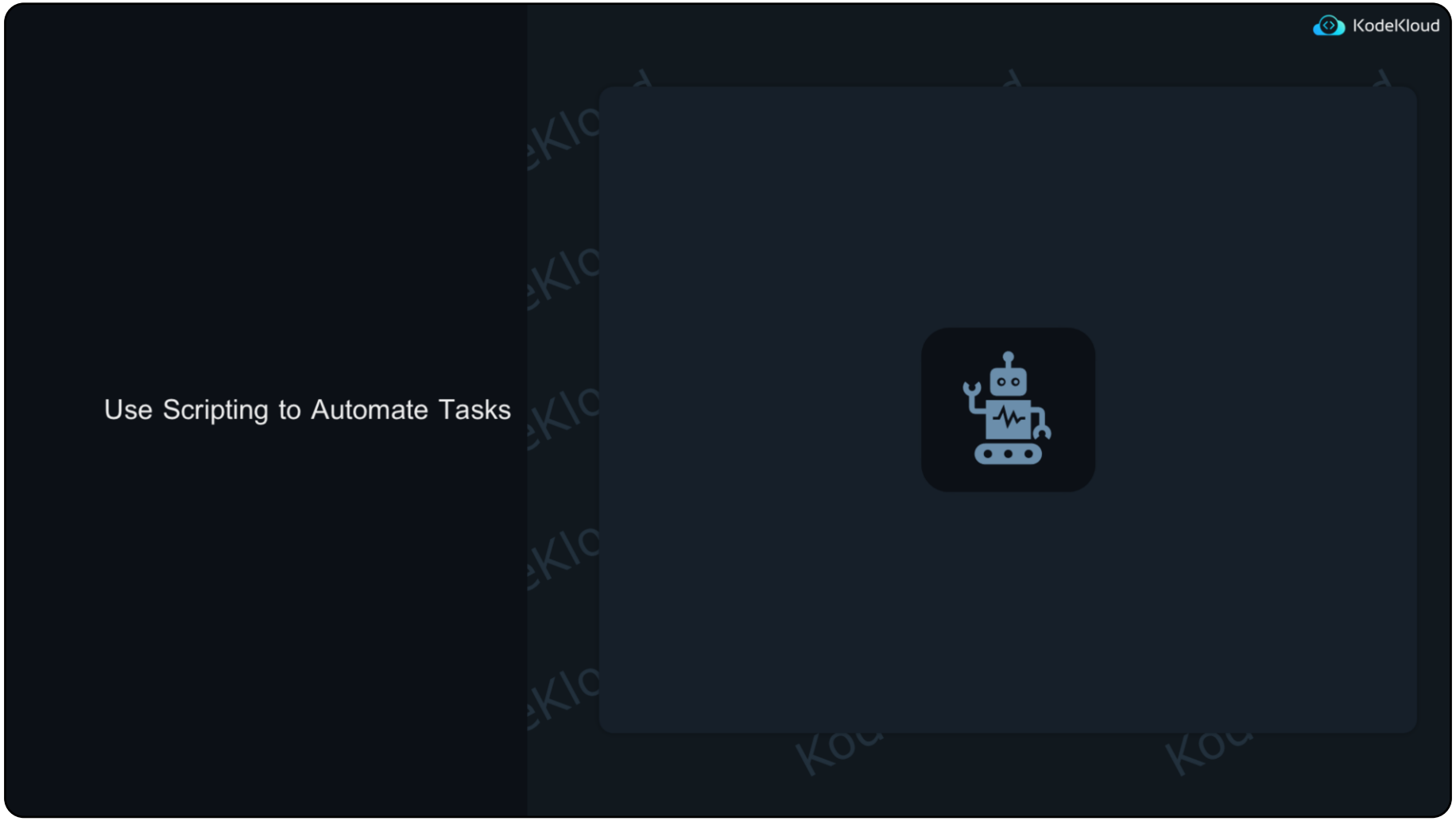
You can also set what is called a wall message. If a few users are logged in to this Linux machine, the wall message will be shown to them before the server reboots or shuts down. This gives them a chance to know in advance why and when the machine will become unavailable. It also gives them a chance to finish their work before this happens, instead of abruptly being disconnected while having no idea what happened.

You can write your wall message like this (between ' ' quotes):

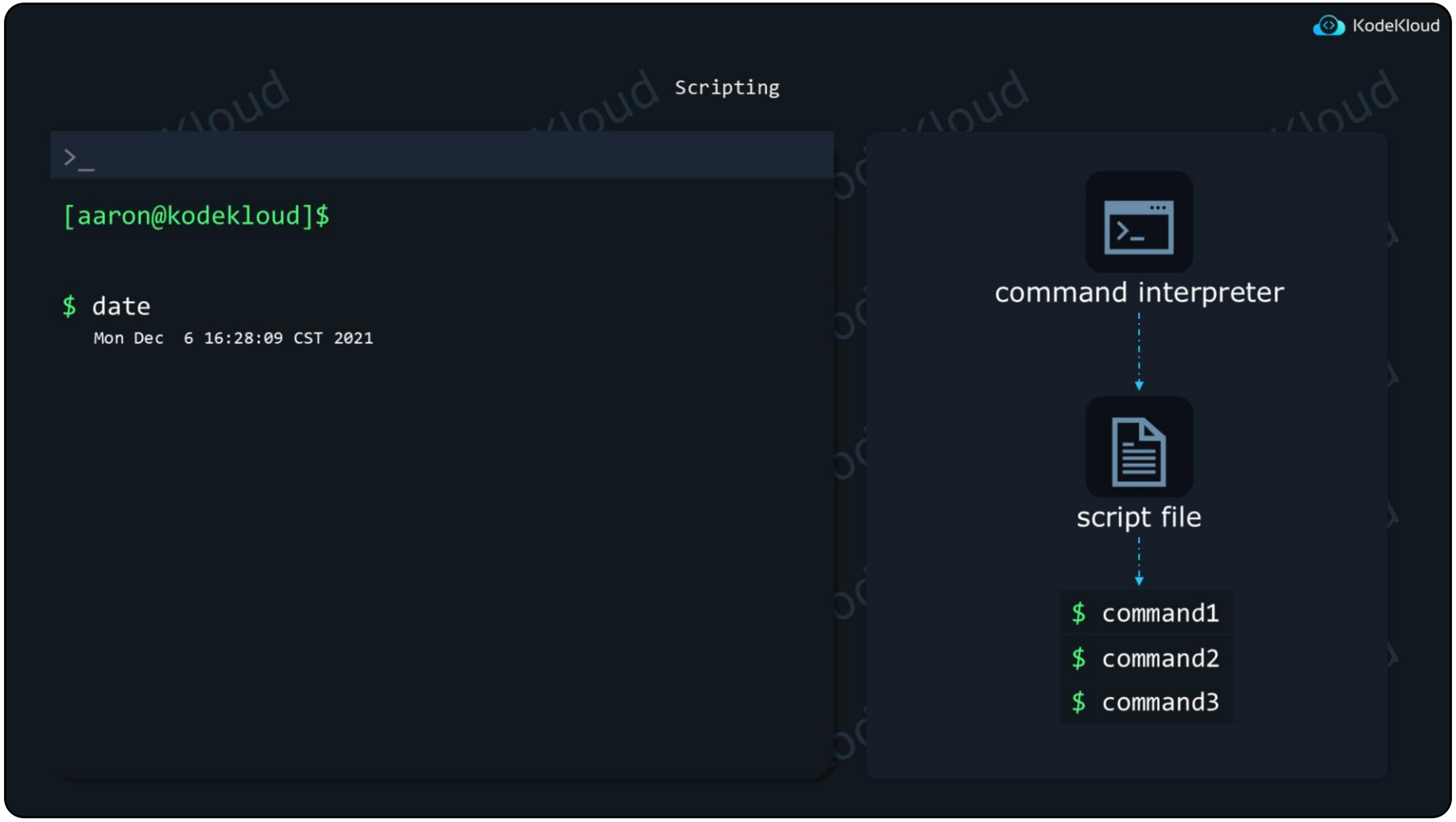
sudo shutdown -r +1 'Scheduled restart to upgrade our Linux kernel'

KodeKloud

Visit www.kodekloud.com to discover more.



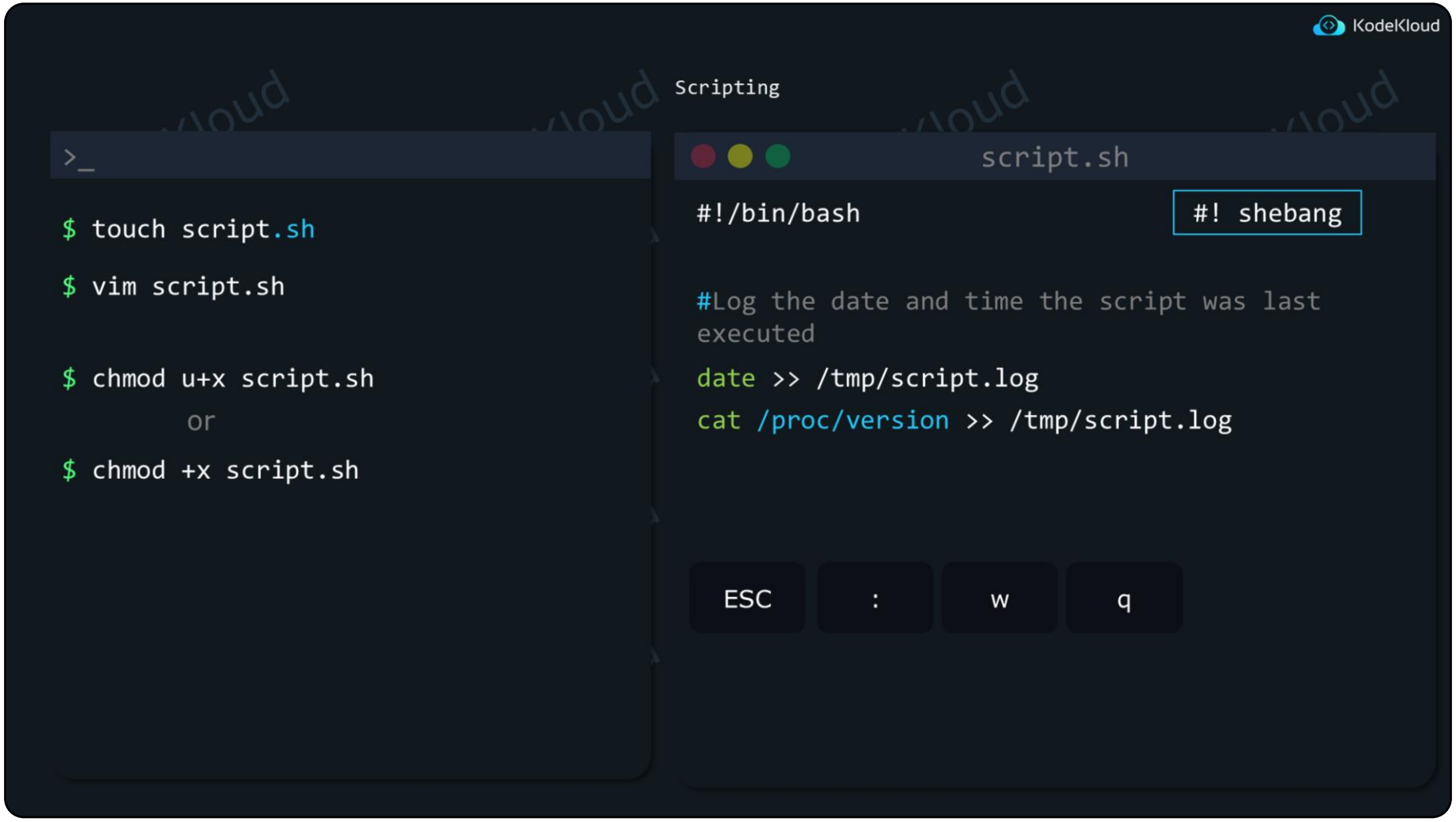
We'll now look at how to use scripting to automate system maintenance tasks.



When we log into a Linux operating system, we're automatically dropped at a command line like this:

What actually happens is that after we successfully log in, a program called Bash opens up. And we are dropped within its text-based environment. All the commands we type are interpreted by Bash, which figures out what needs to happen to execute them. That's why Bash is also called a command interpreter (or shell). In this case, we are using it interactively. We write something, we press Enter, it gets executed and we get a result. But Bash can be used in a different way, with scripts.

Scripts are simply files where we can add multiple instructions for our command interpreter. The interpreter can then read this file and execute instructions in order. First, it will execute the instruction on the first line, then the one on the second line, and so on. But it's easier to understand how scripts work if we build one, so let's start experimenting.



First, we'll create a file called script.sh

touch script.sh

It's standard practice to add the .sh extension to this file. It's not mandatory, but it makes it easier when you type "ls" in a directory to spot which files are scripts. On rare occasions, we might need to skip adding the .sh extension if we want other programs to execute them. We will see an example of this when we learn about scheduling tasks with cron/anacron, in a later

lesson.

Now let's edit script.sh and add some content:

vim script.sh

The first line is very important. We'll write this here:

#!/bin/bash

To make our script work, this has to be on the first line, not the second or third. Also, make sure there is no space before #.

#! is called a shebang. What follows #! is the full path to the command interpreter that we want to run this script. In this case we choose /bin/bash.

On the third line we can add this:

Log the date and time when the script was last executed

at the beginning of a line marks this as a comment. A commented line is something that the command interpreter will ignore. It will not execute anything here, no matter what we type. A comment is useful when we want to document our script, to inform other people that may read it, about what our next line or lines do.

Now on the next line we'll add an actual command to this script:

date >>/tmp/script.log

The date command prints out a date and time. We also use redirection to save the output of the date command to the /tmp/script.log file. We're essentially building a simple log from within our script. Note how we can write these commands the same way we write them at the command line. We can redirect output, errors, pipe to other programs with | and so on.

On the next line, we'll add this command:

cat /proc/version >>/tmp/script.log

/proc/version is a special file that has detailed information about the current Linux kernel version that is running our operating system. So we're sending the content of this special file to our log file at /tmp/script.log

In a nutshell, we've built a script that can keep track of what kernel versions ran on our server in the past.

Let's press ESC, then type :wq and Enter to save our file.

To be able to run this script, we must now make it executable. If we only want the owner of this file (our current user) to be able to execute it, we can add this permission:

chmod u+x script.sh

If we want everyone to be able to execute it, we can just run

chmod +x script.sh

This will grant execute permissions for everybody: the user owner, the group owner and others.

To run this, we either must provide the full path to the script and type:

/home/aaron/script.sh

Or we can type this:

./script.sh

. represents our current directory. So, when we type ./script.sh, the . becomes /home/aaron/ since we're already inside this directory. /script.sh is added at the end so it's the same as typing /home/aaron/script.sh

Let's see what our script did. We'll read the script.log file:

cat /tmp/script.log

Pretty nice result! Further down the line, we could set this up to automatically run each day and we'd have a log with all Linux kernel versions we ever used. When something malfunctions, you could consult this log and say "Hey, this problem started when we updated to the 5.15.0-x version".

So that's the core part of creating a script. You just make a file executable and then you add in its contents a shebang pointing to the path of a command interpreter, followed by some instructions for that interpreter. However, smarter scripts can be created with the help of what are called Bash built-ins. To get a quick list of built-ins available type:

KodeKloud

Scripting

```
>_

$ help
job_spec [&]                            history [-c] [-d offset] [n] or hist>
(( expression ))                        if COMMANDS; then COMMANDS; [ elif C>
. filename [arguments]                  jobs [-lnprs] [jobspec ...] or jobs >
:                                       kill [-s sigspec | -n signum | -sigs>
[ arg... ]                              let arg [arg ...]
[[ expression ]]                        local [option] name[=value] ...
alias [-p] [name[=value] ... ]          logout [n]
bg [job_spec ...]                       mapfile [-d delim] [-n count] [-O or>
bind [-lpsvPSVX] [-m keymap] [-f file>  popd [-n] [+N | -N]
break [n]                               printf [-v var] format [arguments]
builtin [shell-builtin [arg ...]]       pushd [-n] [+N | -N | dir]
caller [expr]                           pwd [-LP]
case WORD in [PATTERN [| PATTERN]...)>  read [-ers] [-a array] [-d delim] [->
cd [-L|[-P [-e]] [-@]] [dir]            readarray [-n count] [-O origin] [-s>
command [-pVv] command [arg ...]        readonly [-aAf] [name[=value] ...] o>
compgen [-abcdefgjksuv] [-o option] [>  return [n]
complete [-abcdefgjksuv] [-pr] [-DE] >  select NAME [in WORDS ... ;] do COMM>
compopt [-o|+o option] [-DE] [name ..>  set [-abefhkmnptuvxBCHP] [-o option->
continue [n]                            shift [n]
coproc [NAME] command [redirections]    shopt [-pqsu] [-o] [optname ...]
declare [-aAfFgilnrtux] [-p] [name[=v>  source filename [arguments]
dirs [-clpv] [+N] [-N]                  suspend [-f]
disown [-h] [-ar] [jobspec ... | pid >  test [expr]
echo [-neE] [arg ...]                   time [-p] pipeline
```

help

There are many available, but we'll explore two of these to get a glimpse of what they make possible. We'll look at "if" and "test".

Let's create a different script to archive the contents of the /etc/apt/ directory. We can run the next command, even if the archive-apt.sh file does not exist. Vim will create it automatically once we save the content.

vim archive-apt.sh

Let's imagine we want this script to backup the contents of the /etc/apt/ directory. We'll use tar to create an archive that will be saved at /tmp/archive.tar.gz

```
#!/bin/bash

tar acf /tmp/archive.tar.gz /etc/apt/
```

Once the file is created, we will make the script executable.

```
chmod +x archive-apt.sh
```

And run it

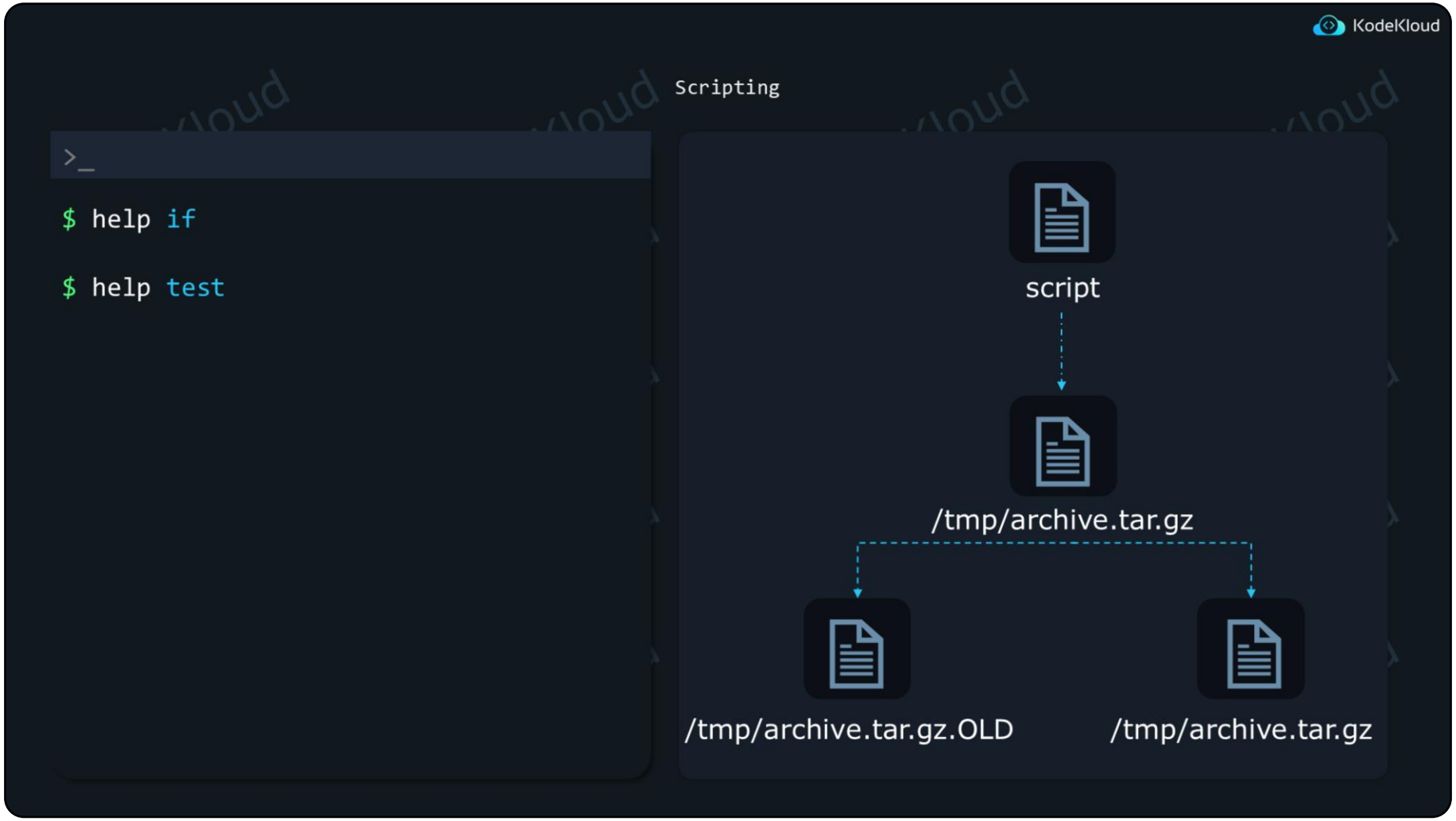
```
./archive-apt.sh
```

We see that indeed our simple script created an archive with the content of the /etc/apt/ directory

```
ls /tmp/
```

We can look at the list of files inside using:

```
tar tf /tmp/archive.tar.gz
```



But now imagine this. Someone accidentally deletes some files in /etc/apt/. Next, our backup script runs. Then /tmp/archive.tar.gz gets created, overwriting the previous backup. Now we're left with a broken /etc/apt/ directory and also a broken /tmp/archive.tar.gz since it backed up the broken /etc/apt/ directory. If only we still had the old backup around! The one that contained the old content of /etc/apt/, which was still good.

Well, we can make our script smarter. We can make it implement the following logic:

Check to see if /tmp/archive.tar.gz exists using "if" and "test". Two built-in functions of Bash. If it does, rename it to /tmp/archive.tar.gz.OLD and then create a new /tmp/archive.tar.gz. Now we'll always have two backups: the new one, and the previous, older one.
If /tmp/archive.tar.gz does not exist, simply create the first backup in this series.

As mentioned, we'll use "if" and "test" Bash built-ins to make this possible.

To get information about a built-in and its syntax, use help followed by its name. For example:

help if

and help test

Let's create a new script called archive-apt-2.sh

vim archive-apt-2.sh

And add this content that implements our desired logic:

```
#!/bin/bash

if test -f /tmp/archive.tar.gz; then
    mv /tmp/archive.tar.gz /tmp/archive.tar.gz.OLD
    tar acf /tmp/archive.tar.gz /etc/apt/
else
    tar acf /tmp/archive.tar.gz /etc/apt/
fi
```

Now let's analyze this line by line.

```
if test -f /tmp/archive.tar.gz; then
```

"if" starts a condition check. It verifies if the expression that comes after it is true or false.

Inside our if condition, we use the "test" built-in:

```
test -f /tmp/archive.tar.gz;
```

As "help test" command shows us, this can perform various types of checks. For example, for files, it can test if it's a certain type of file, if you can execute it, and so on.

In this case, the -f option makes it check if the file at /tmp/archive.tar.gz exists, and is a regular file. If it does, the "test" built-in returns true. If it doesn't, it returns false.

Our "if" built-in cannot know where our condition check ends. So, we tell it where it ends by adding ";" at the end of the condition we want to be verified. Finally, the word "then" signals what will happen if our "test" function returns true. So, when this is true, the following two commands get executed:

```
 mv /tmp/archive.tar.gz /tmp/archive.tar.gz.OLD
    tar acf /tmp/archive.tar.gz /etc/apt/
```

This simply renames the pre-existing archive.tar.gz file to archive.tar.gz.OLD and then creates a new archive named archive.tar.gz. Note how we have four spaces before each command. This is just standard convention to make the script more easily readable. Because the lines are "moved to the right" this way, we can easily spot what will happen if our test function returns true.

The next line is

```
else
```

This signals what will happen if our "test" doesn't return "true" and returns "false" instead. In this case, what will happen if /tmp/archive.tar.gz file does NOT exist. If it returns false, a simple archive will be created (no renaming of old archive this time because there isn't one available).

```
tar acf /tmp/archive.tar.gz /etc/apt/
```

Finally, on the last line we have

```
fi
```

This is just "if" in reverse. It signals that this is the end of our "if" block.

Let's save our script and then let's make our script executable.

```
chmod +x archive-apt-2.sh
```

And run it:

```
./archive-apt-2.sh
```

If we look in /tmp/

ls /tmp/

we'll see this:

archive.tar.gz
archive.tar.gz.OLD
script.log

Which means our script is working as intended. The old archive was moved to a file with the .OLD extension. And a new archive was created. So now we have both versions available.

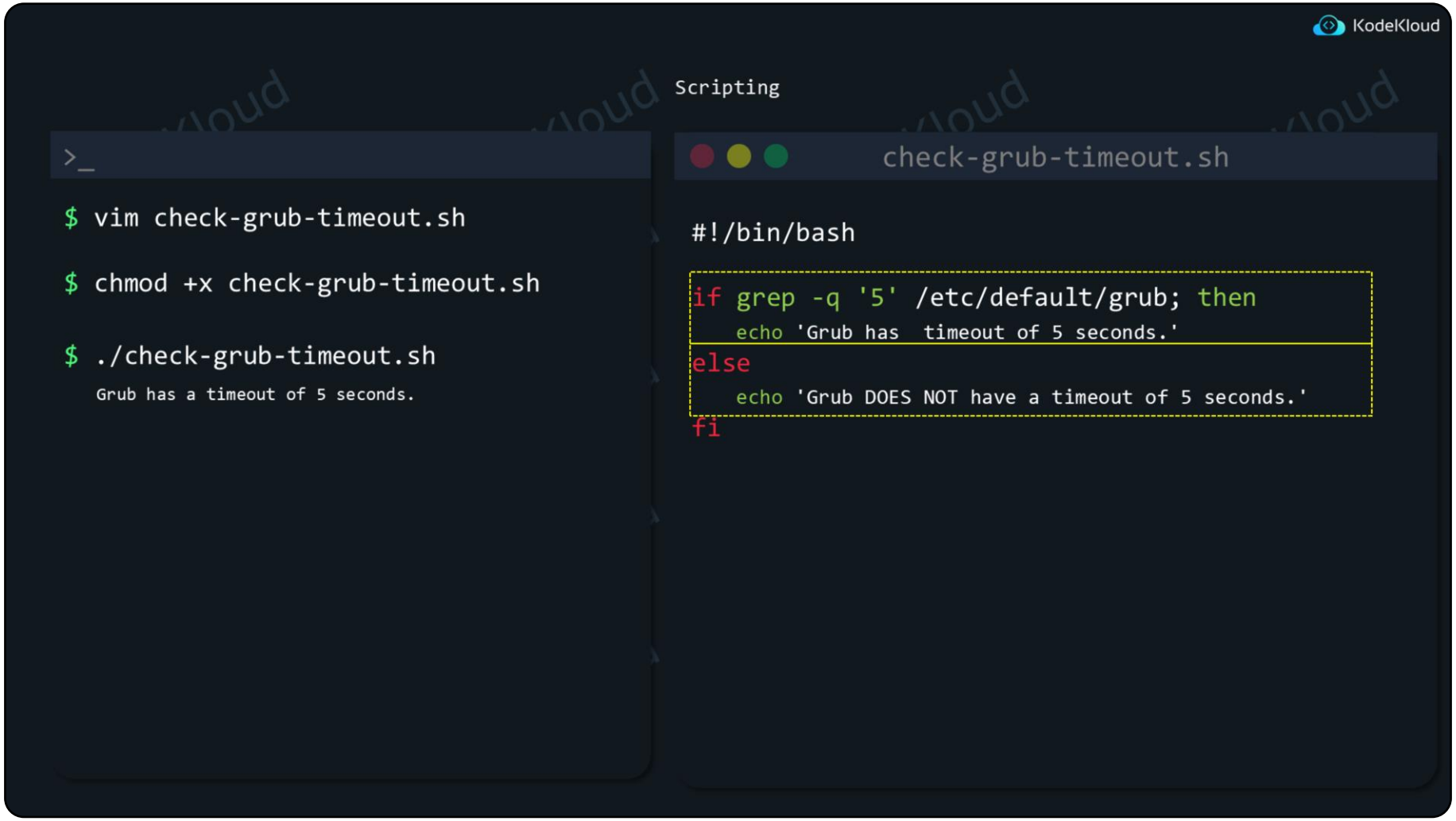
KodeKloud

Exit Status Quo (ESQ)

Most commands return exit status quo

ESQ = 0; if command is successful (True)

ESQ > 0; if command failed or encountered error (False)



It's important to note that almost any command can be used in an if block. That's because most commands return a so-called "exit status code". They return "0" if the command executed successfully. They return a value larger than 0 if the command didn't find what it was looking for, or if it encountered an error. When "if" sees that the command returned 0, it considers that this returned TRUE. When the command returns a non-zero value, "if" considers that the command returned FALSE.

For example, consider the grep command. If it finds the text you were looking for, it returns 0. If it doesn't, it returns 1. So, we could have a script like this

vim check-grub-timeout.sh

```
if grep -q '5' /etc/default/grub; then
    echo 'Grub has a timeout of 5 seconds'
else
    echo 'Grub DOES NOT have a timeout of 5 seconds'
fi
```

grep -q '5' /etc/default/grub looks for the number "5" in the /etc/default/grub file. -q stands for "quiet". This option tells grep to not display matched lines, because we don't need our script to output those lines. In this case, our script will output

Grub has a timeout of 5 seconds

If the number "5" was found.

And it will output

Grub DOES NOT have a timeout of 5 seconds

if the number "5" was not found.

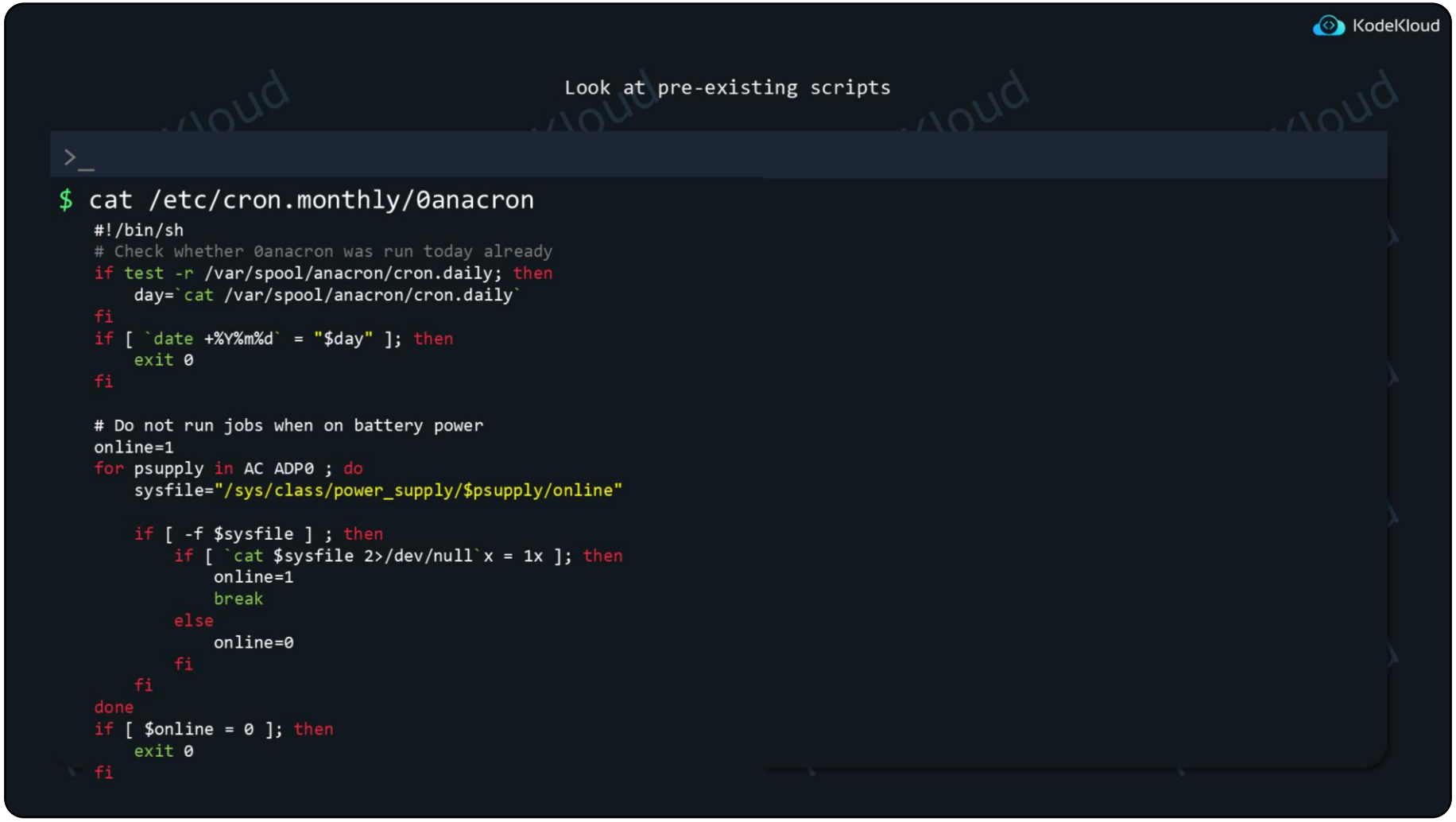
We'll now make the script executable.

chmod +x check-grub-timeout.sh

And run it

./check-grub-timeout.sh

These are the basics of scripting with Bash built-ins. If you want your future scripts to implement more advanced logic, you'll have to dive deeper into more built-ins, such as "for", and "while" loops, functions, variables, and so on. These allow you to construct scripts in a way that resembles programming logic. Thus your scripts become more "intelligent", so to speak, and they can do even more interesting things.



And if you're in need of a quick refresh of how scripting works, you could take a quick peek at files from the slash etc slash cron dot daily, weekly, and monthly directories (/etc/cron.). Usually, all files in there are scripts.

cat /etc/cron.monthly/0anacron

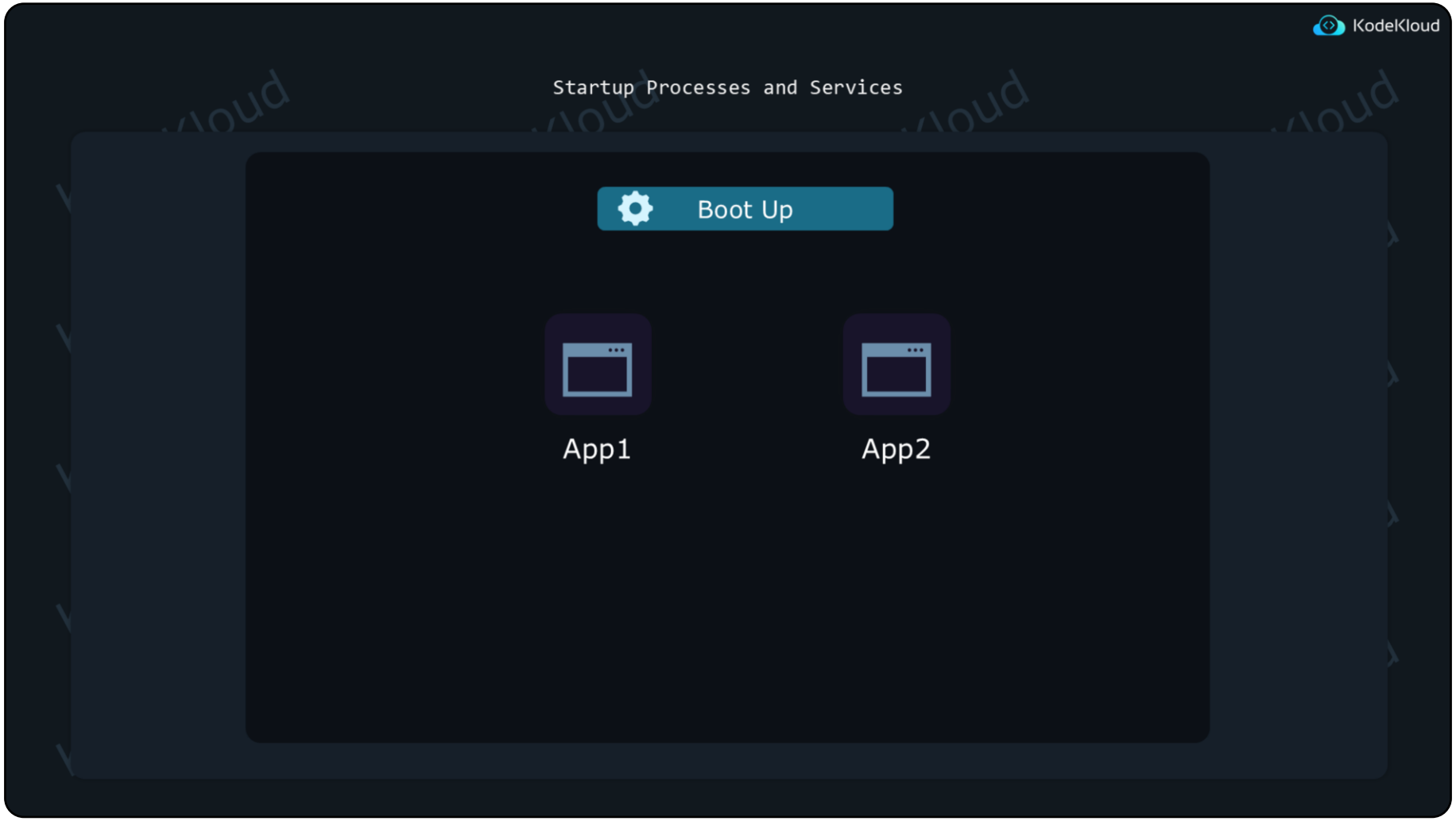
This way, you can get a sort of cheatsheet for common script syntax, like the #! shebang on the first line, the syntax of "if", and so on.

To learn more about shell scripts, check out our "Shell scripts for beginners" course on KodeKloud. It is a project driven course with many hands-on labs that will help you learn not only with easy-to-understand theory, but also practical exercises you can go through.

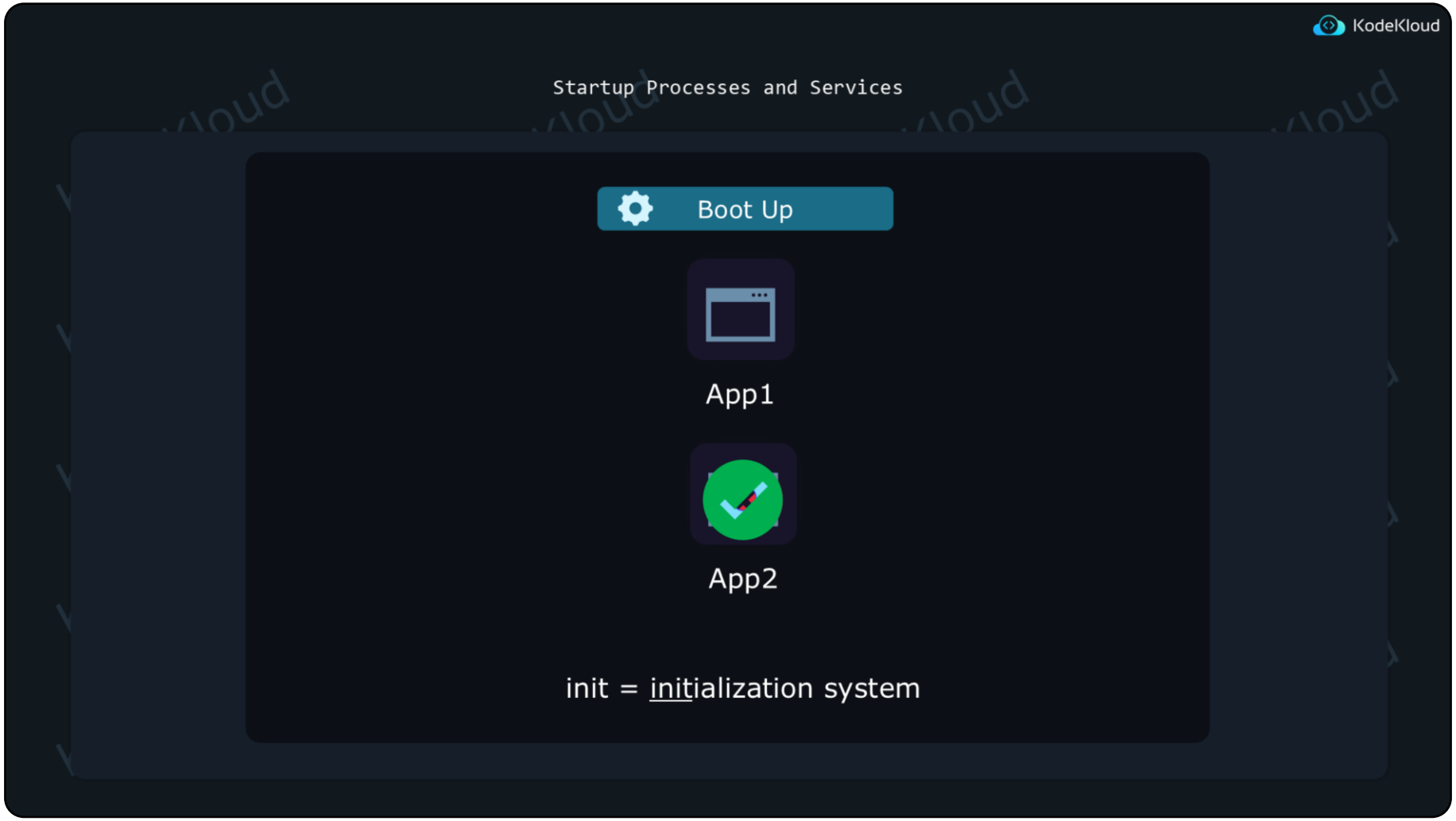
KodeKloud

Visit www.kodekloud.com to discover more.

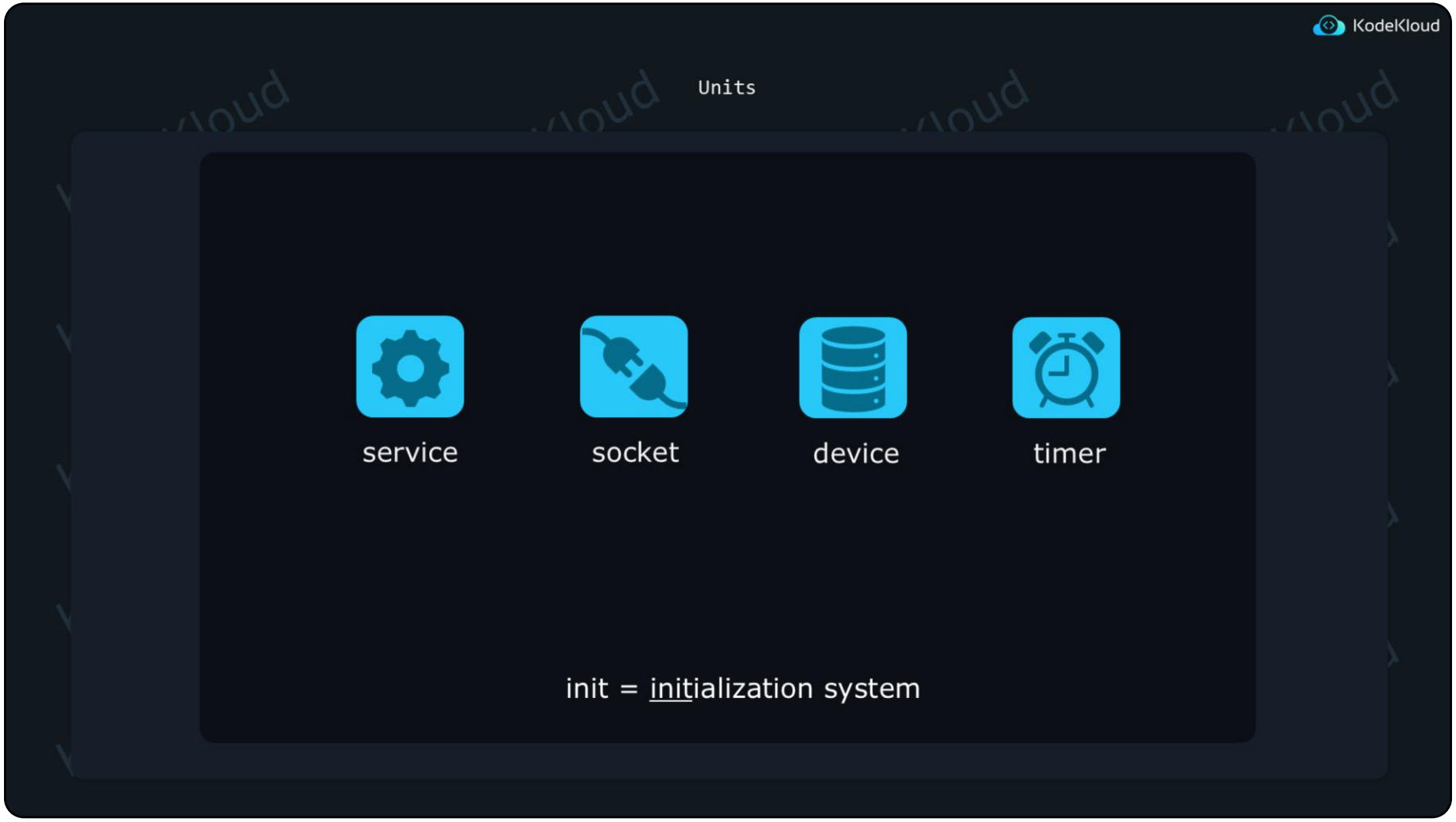
Let's examine how to manage startup processes and services in Linux.



When we boot up Linux, certain important applications start up automatically. Some of them even start up in a very specific order. For example, if App2 depends on App1, then App1 will load before App2.



All of this magic happens behind the scenes, and in a rather intelligent way. Furthermore, if important applications crash, they will be restarted automatically. This way, the system can continue to run smoothly even if there's a small hiccup like this. But how does all of this happen? With the help of what is called the init system, short for initialization system.



So how does this init system know how to start up applications, what to do when a program crashes, and so on? It needs specific instructions to know how to do its job. And sure enough, it has all the instructions it needs in what are called systemd units. These are simply text files that describe the necessary logic.

And systemd is the name of a collection of tools, components, and applications that help start, operate, and manage most of the Linux-based operating systems. systemd is also the name of the program that starts up as the init system, which can be a bit confusing. Long story short, systemd is the name of a large collection of tools and components, but also the name of the application responsible for system initalization and monitoring the system as a whole to ensure smooth operation.

Units can be of various types such as service, socket, device, timer, and others. For example, timer units let us tell the init system that it should launch a specific application once per week, maybe to clean up some files, or verify a database. But we'll be looking at service units in this lesson.

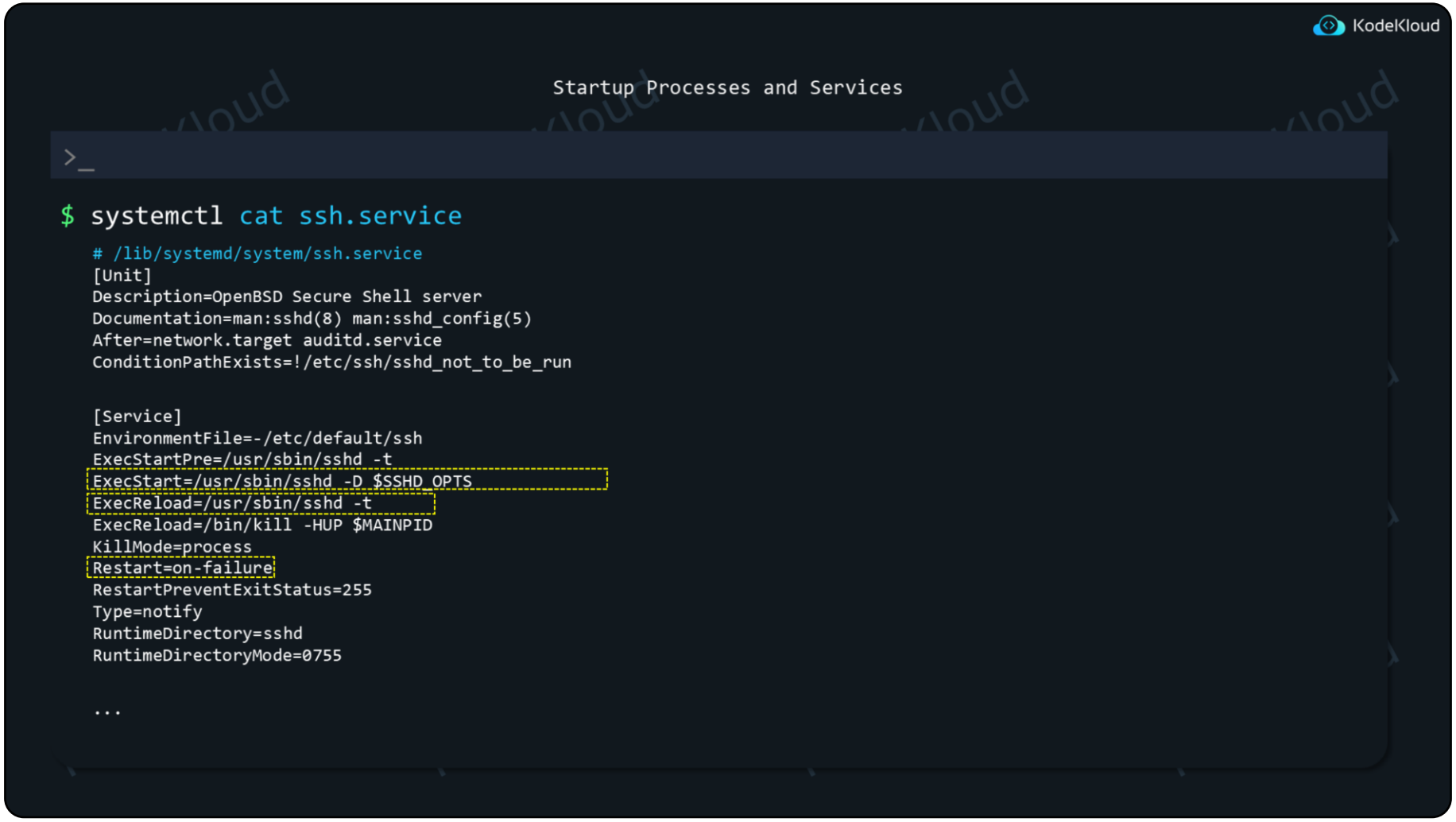
Service units have clear instructions about things such as:

What command to issue to start up a program
What to do if the program crashes
What command to issue when a program is restarted
And many more things

In a nutshell, a service unit tells the init system all it needs to know about how it should manage the entire lifecycle of a certain application.

We can see the multitude of instructions we can add in a service unit if we type

man systemd.service



Now let's look at a real example. Servers need to run the SSH daemon to let users connect to them from remote locations. And there is a service unit that instructs the init system about how to start this daemon and how to keep it running. We can look at this service file with:

systemctl cat ssh.service

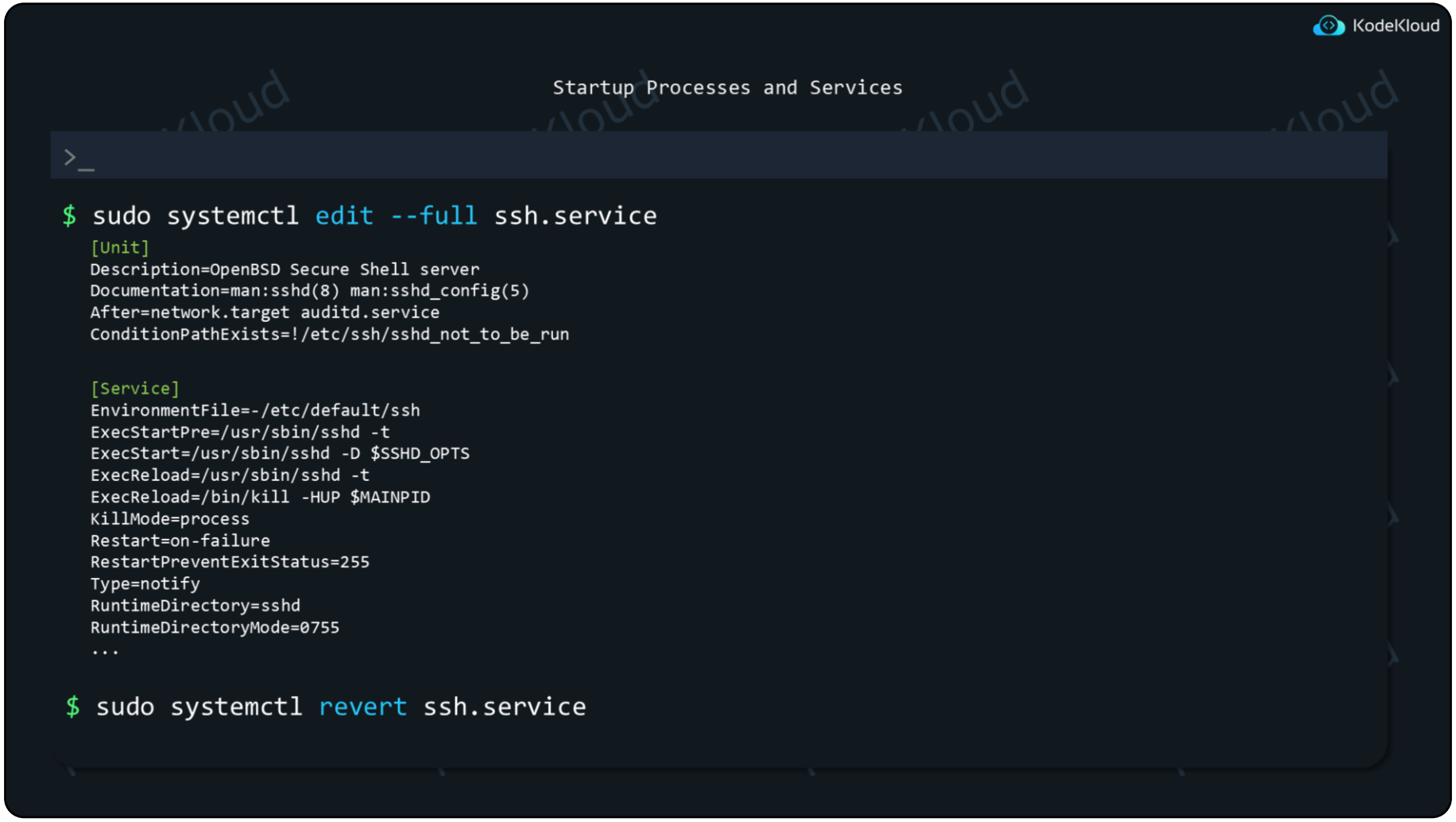
We'll see lines like this in this file:

```
ExecStart=/usr/sbin/sshd -D $SSHD_OPTS
ExecReload=/usr/sbin/sshd -t

Restart=on-failure
```

ExecStart tells the init system what command it should run when it wants to start the SSH daemon. ExecReload tells it what command it should run to reload the configuration for the SSH daemon. We'll see an example of what reloading a service means, later in this lesson.

And the Restart= line tells systemd to automatically restart this process only when something fails. For example, if the SSH daemon crashes, it's a good idea to have systemd restart it, so we can still connect to our server.

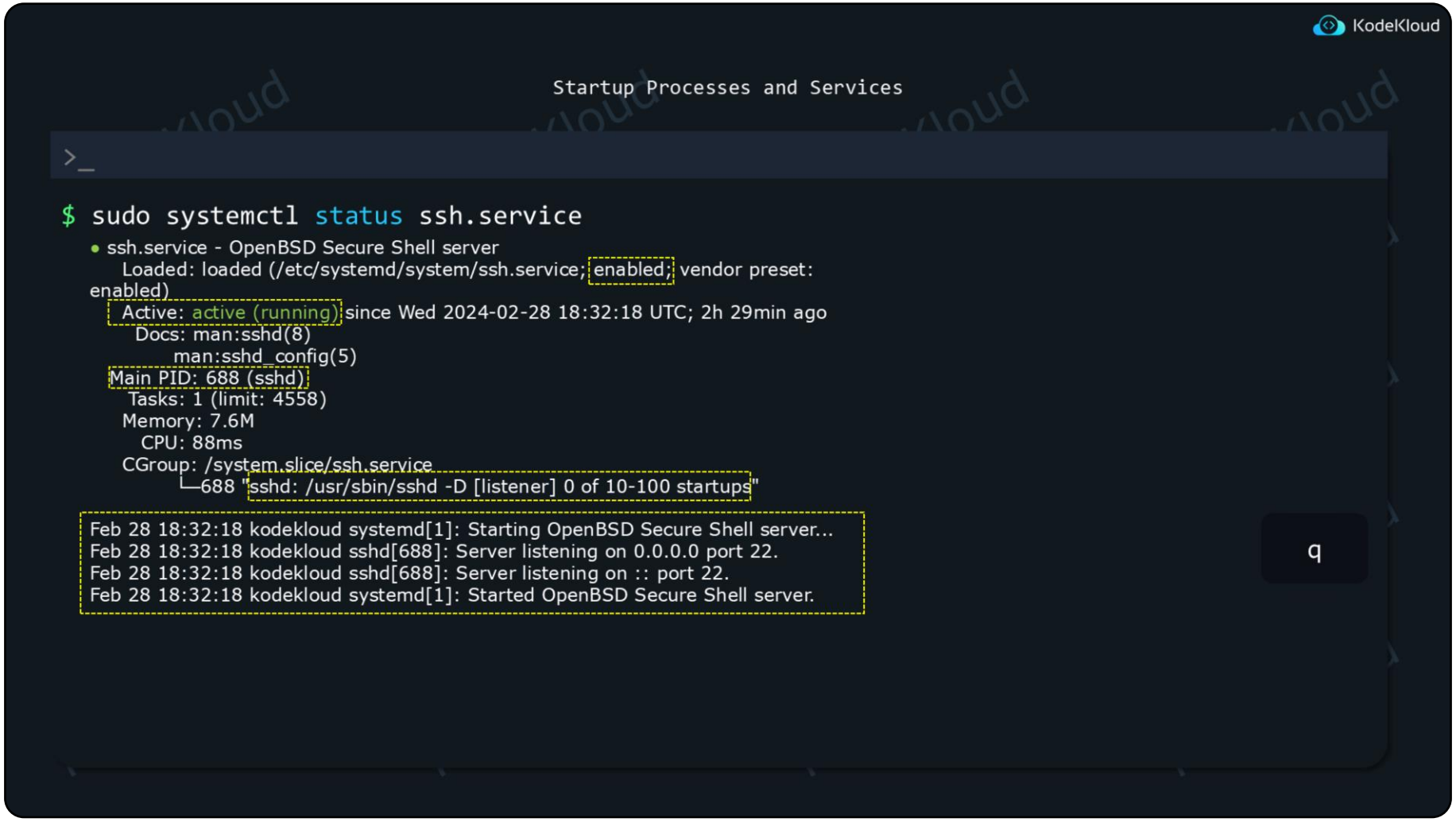


If we'd want to edit this service file and modify these instructions, we could use a command like:

sudo systemctl edit --full ssh.service

If we'd want to cancel our edits later and return this service file to its factory default settings, we could use this command.

```
sudo systemctl revert ssh.service
```



So this systemd service is responsible for the lifecycle of the SSH daemon. To start up the SSH server application, restart it when needed, and so on. To see the status of this service, we can run:

systemctl status ssh.service

This shows us a few interesting things, and here are the most important:

We can see if this service is enabled. If it's enabled, it means that systemd will automatically start up the SSH daemon when the system boots. Disabled means it won't automatically start up, but it can still be manually started by the administrator. We'll see how, later on.

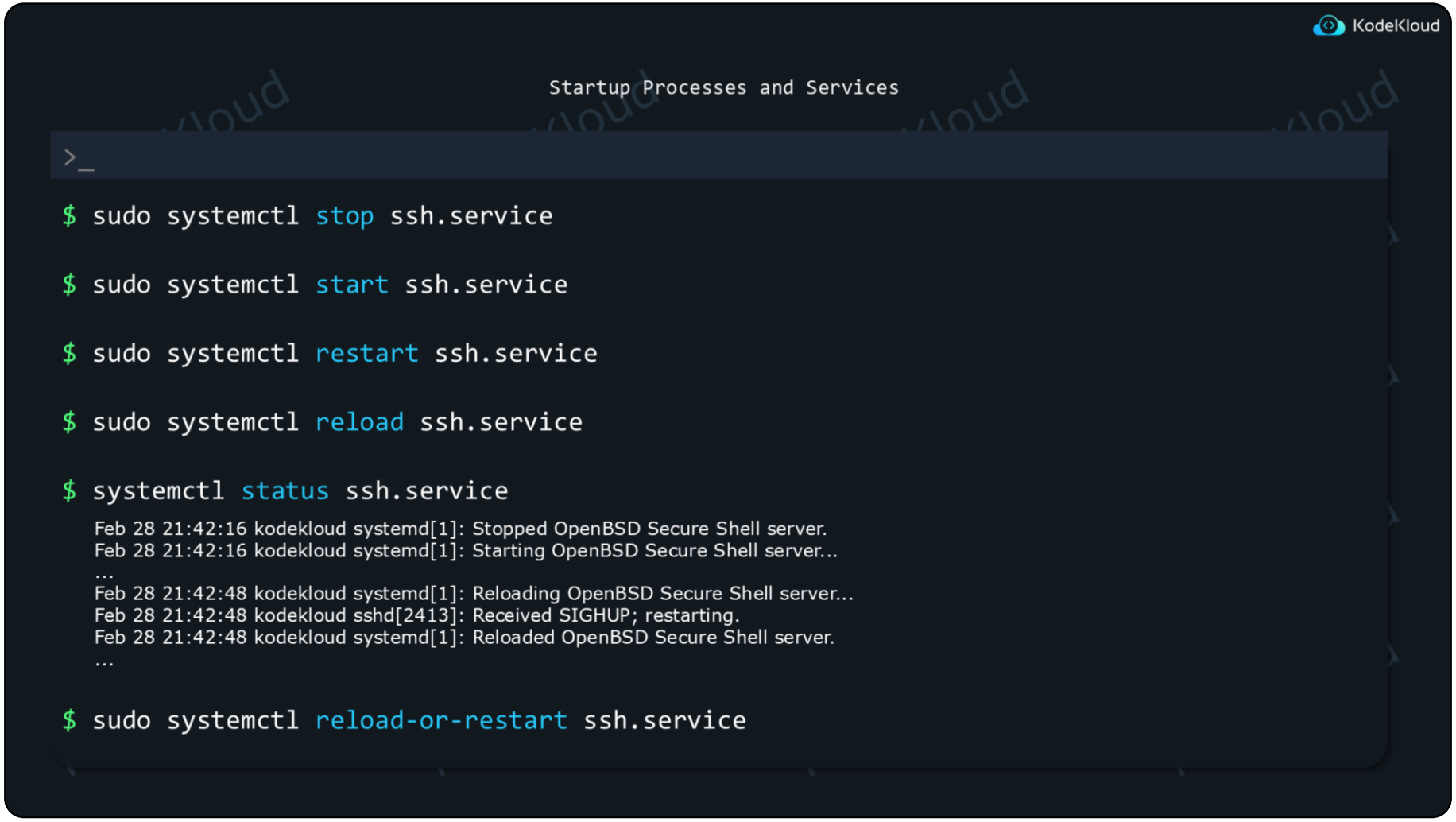
Next, we can see if the program managed by this service is running. If it's currently launched, loaded into memory, and running, we'll see a status of active (running). We could potentially encounter situations where the service is active, but the program managed by the service is not running. Because it was launched, did its job successfully, then exited. In such a case we'd see a status of active, but exited between parentheses.

If the SSH daemon is currently running, we'll also see its PID (Process IDentifier). Every time we launch a program on Linux, a process will start up. The process encapsulates the computer code and resources loaded into memory and using the CPU when it needs to execute instructions. And every process has this unique number identifying it. The PID can be used to interact with this process, as we'll see in one of the next lessons in this series.

We can also see the exact command that has been used to start up this process.

And, finally, systemctl status also shows us a few log lines: status and error messages generated by this application. These can be useful to see what happened after the program started up, if it encountered any errors, what settings are active for it, and so on.

We can press q to exit from this status display.



To stop a service, we can enter:

sudo systemctl stop ssh.service

This will close the program that this service unit controls.

We mentioned how some services do not automatically start up at boot time, but they can still be manually started up later. The command to manually start a service is:

```
sudo systemctl start ssh.service
```

Sometimes, we will change the settings of a program. For example, to modify how the SSH daemon works, we could change its settings in the file at /etc/ssh/sshd_config. After we change a configuration file like this, the process currently running will still use the old settings, until restarted. One way to force the process to restart and pick up on the new settings is with this command:

```
sudo systemctl restart ssh.service
```

However, this restart command can sometimes be a bit disruptive. For example, if a few users are actively using that program, the restart may interrupt their work, temporarily. So there's a more gentle way to reload program settings, without completely closing and reopening the application.

```
sudo systemctl reload ssh.service
```

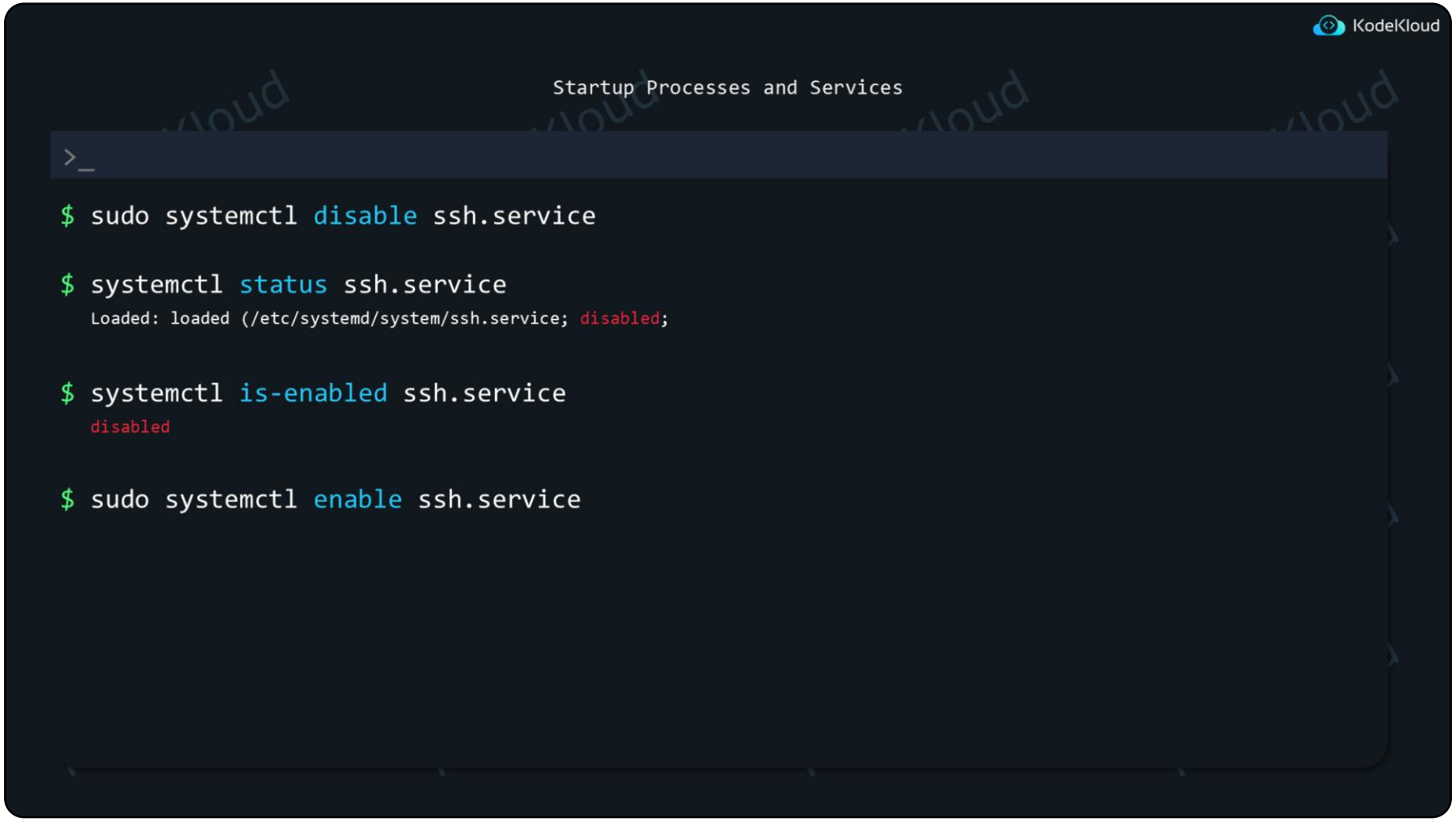
If we would check out the status for the ssh.service again

```
systemctl status ssh.service
```

we would see that the first command restarted it and the second just gracefully reloaded its settings.

It's worth noting that not all applications support being reloaded like this. But we have a command that can automatically try a graceful reload first and then a restart, if a reload is not supported by that app.

```
sudo systemctl reload-or-restart ssh.service
```



If we'd want to disable SSH logins completely, we could prevent the SSH daemon from automatically starting up at boot time:

sudo systemctl disable ssh.service

Now

systemctl status ssh.service

would show us that this is now disabled

Loaded: loaded (/etc/systemd/system/ssh.service; disabled;

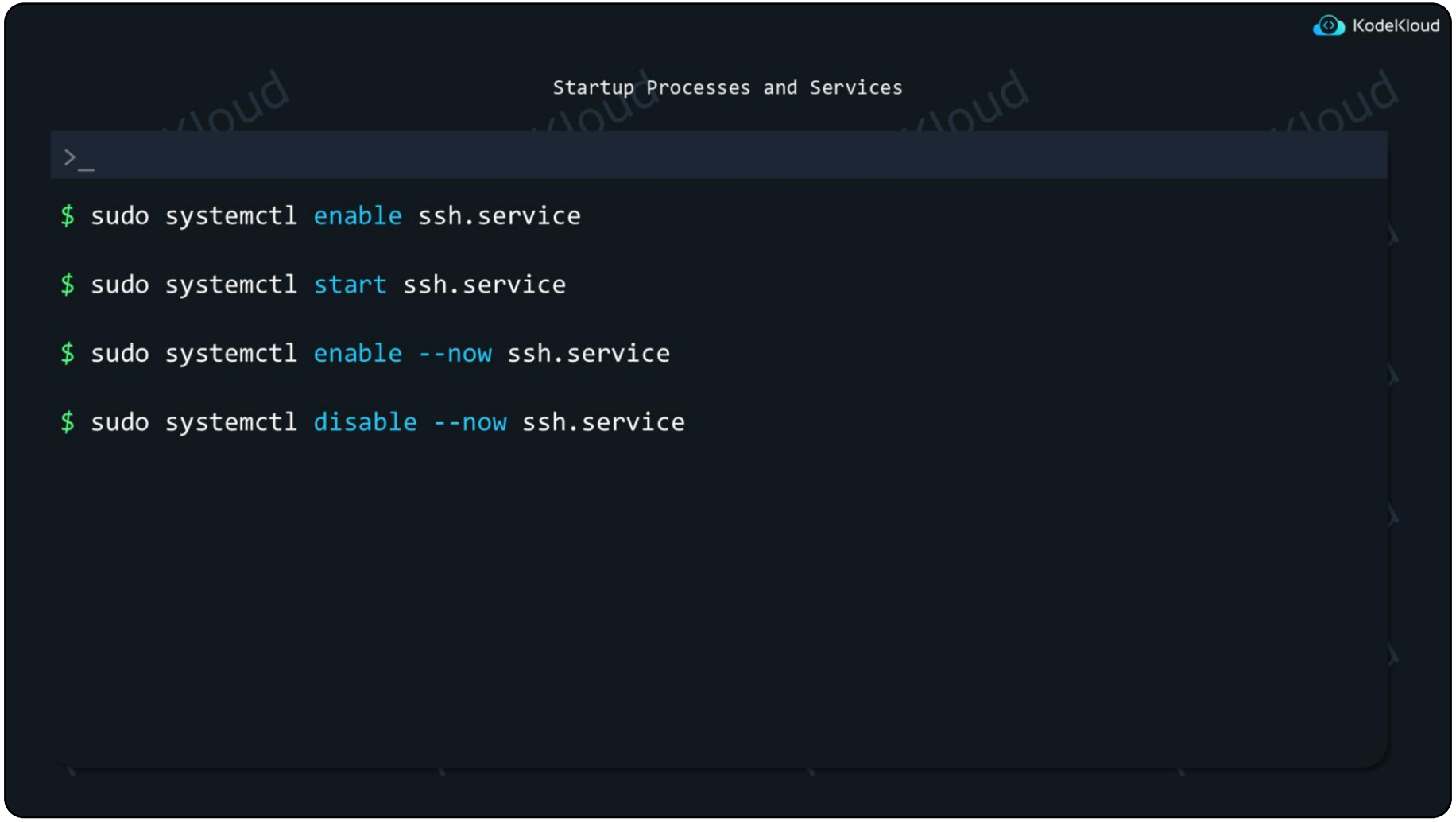
But we can also check with:

systemctl is-enabled ssh.service

To re-enable the SSH daemon and make it automatically start up at boot:

sudo systemctl enable ssh.service

Quick note if you went through these steps in your own virtual machine. If you disabled your ssh.service, please re-enable it now to ensure you can still log in the next time you boot your Linux OS.



Usually, after we install a new application, especially if it's a daemon or a server of some kind, like a database server application, or an HTTP (web) server application, we'll want to do two things:

1. Enable it to autostart every time the operating system boots up
2. Manually start it so we can begin to use it immediately.

Operating systems from the Debian and Ubuntu family usually take these steps automatically after you install certain programs. But other operating systems, for example those belonging to the Red Hat family don't. So it's useful to know how to perform these actions manually.

One way to do this is with the two commands we explored earlier. First we tell our init system to automatically start the service every time the operating system boots up:

```
sudo systemctl enable ssh.service
```

Then we tell our init system to also start the service immediately:

```
sudo systemctl start ssh.service
```

Otherwise this service will only start the next time we boot up the operating system, but it would not run for our current session.

But we also have an alternate command that does both things in one shot:

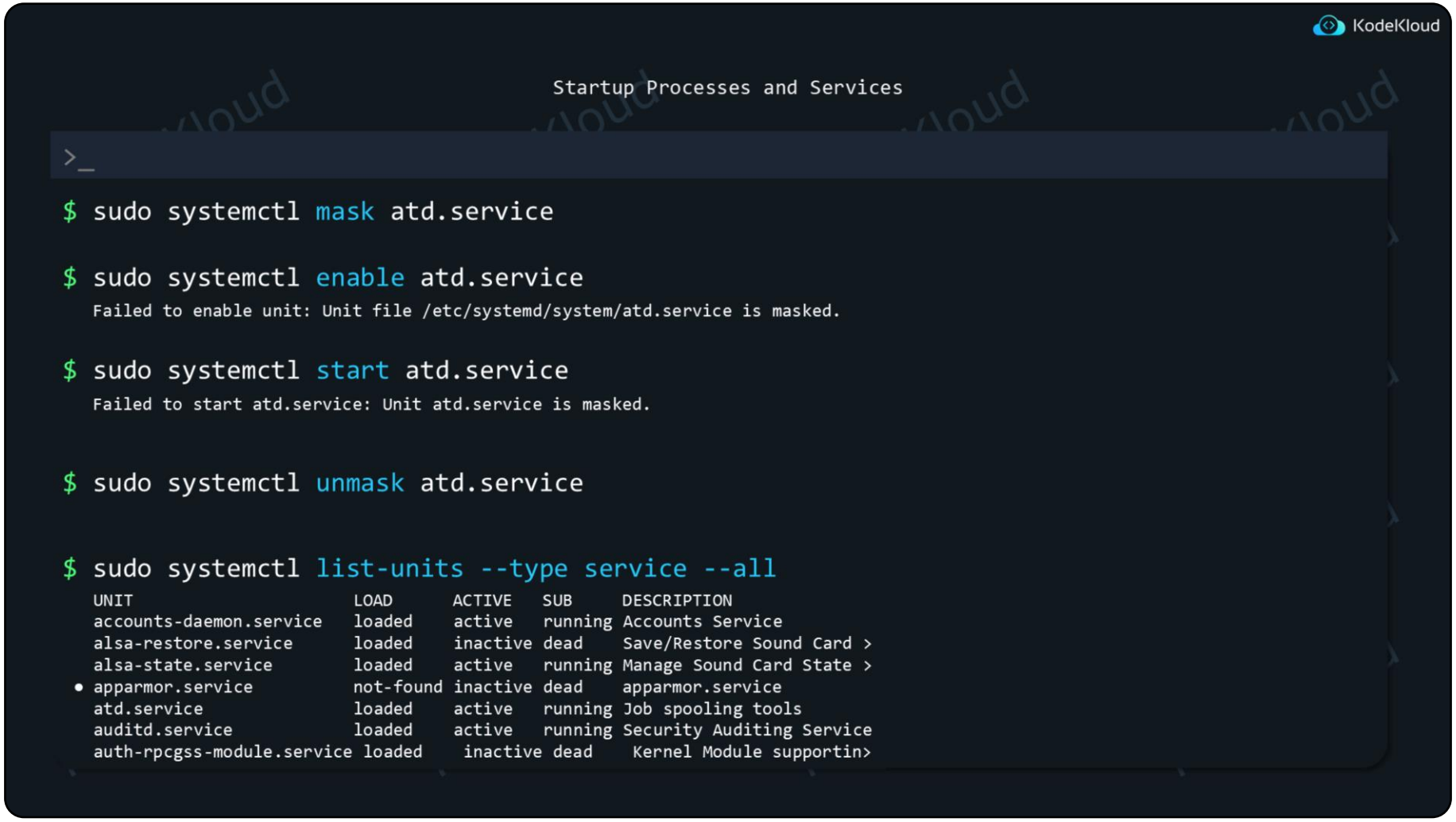
```
sudo systemctl enable --now ssh.service
```

The --now option tells systemctl to both enable the service to auto-start every time the system boots up, but also start the service now.

The disable command also supports the --now option. If you're practicing in a virtual machine, don't enter this next command, as you won't be able to log in through SSH anymore.

```
sudo systemctl disable --now ssh.service
```

This would disable the SSH daemon from automatically starting up when the system boots up, and also stop the service and the program it manages. It's basically like running "sudo systemctl disable ssh.service" followed by "sudo systemctl stop ssh.service".



With some services, we might notice that they're "stubborn". That is, even if we disable them from autostarting at boot, and we stop them, we might notice later on that they're still running. Somehow, they mysteriously started up on their own. We'd be left wondering, "How did this happen?" Well, some services can automatically start up other services, if they want to do so. So there can be a Domino effect. service1 can start up service2, even if you disabled and stopped service2. But there's a brute-force way to prevent this from happening and it's called masking.

Let's say we want to mask the service responsible for managing the "at" daemon. The "at" daemon is a program that lets us schedule tasks we want to run in the future. We'll learn about it in one of our next lessons. It's worth noting that on some operating systems (like Ubuntu Server variants) the "at" daemon might not be pre-installed, so it needs to be manually installed.

To mask the service responsible for managing this application we'd run:

sudo systemctl mask atd.service

A masked service cannot be enabled or started. These commands:

sudo systemctl enable atd.service
sudo systemctl start atd.service

would fail with the following messages:

Failed to enable unit: Unit file /etc/systemd/system/atd.service is masked.

Failed to start atd.service: Unit atd.service is masked.

So, this basically ensures that the service cannot ever be started, even if other programs, users, or services want to do so.

Finally, if we masked a service and want to cancel that, we can unmask it and return to normal with:

sudo systemctl unmask atd.service

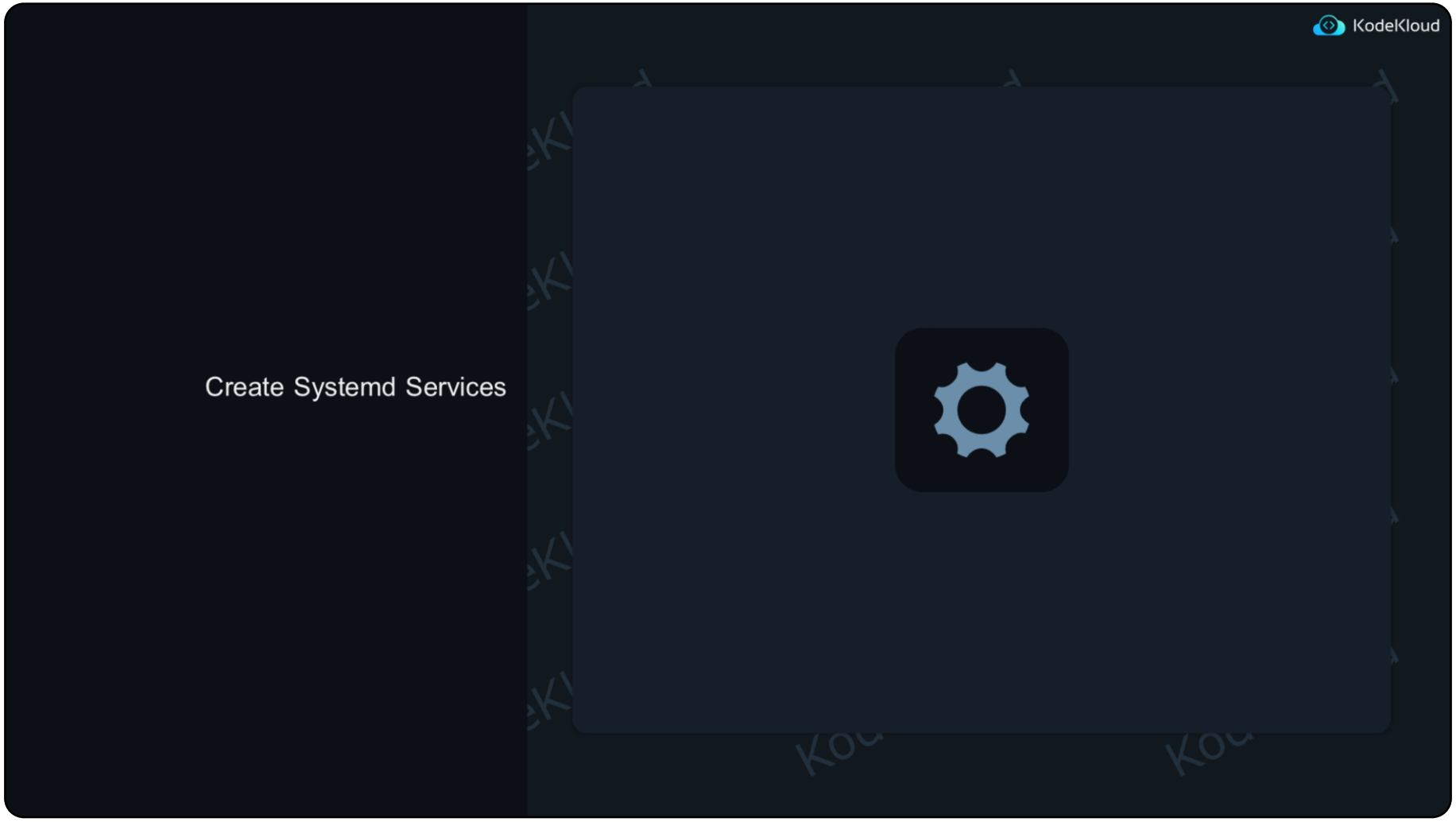
We can see that these systemctl commands are rather intuitive. What might not always be intuitive is how a service for a certain application might be named like. For example, we could install a web server like Apache and expect the service for this application to be called apache.service. But we may find out that it's called httpd.service on some operating systems like Red Hat Enterprise Linux. So, it can be useful to get a list of all service units available on the system, no matter if they're currently enabled, disabled, started or stopped. We can do that with this command:

systemctl list-units --type service –all

"systemctl list-units" would show all types of systemd units. And, remember, besides service units, other types exist, like socket units, timer units. With "--type service" we tell our command to only show us service units. Combined with "--all" we tell it to show us all service units, no matter their status.

KodeKloud

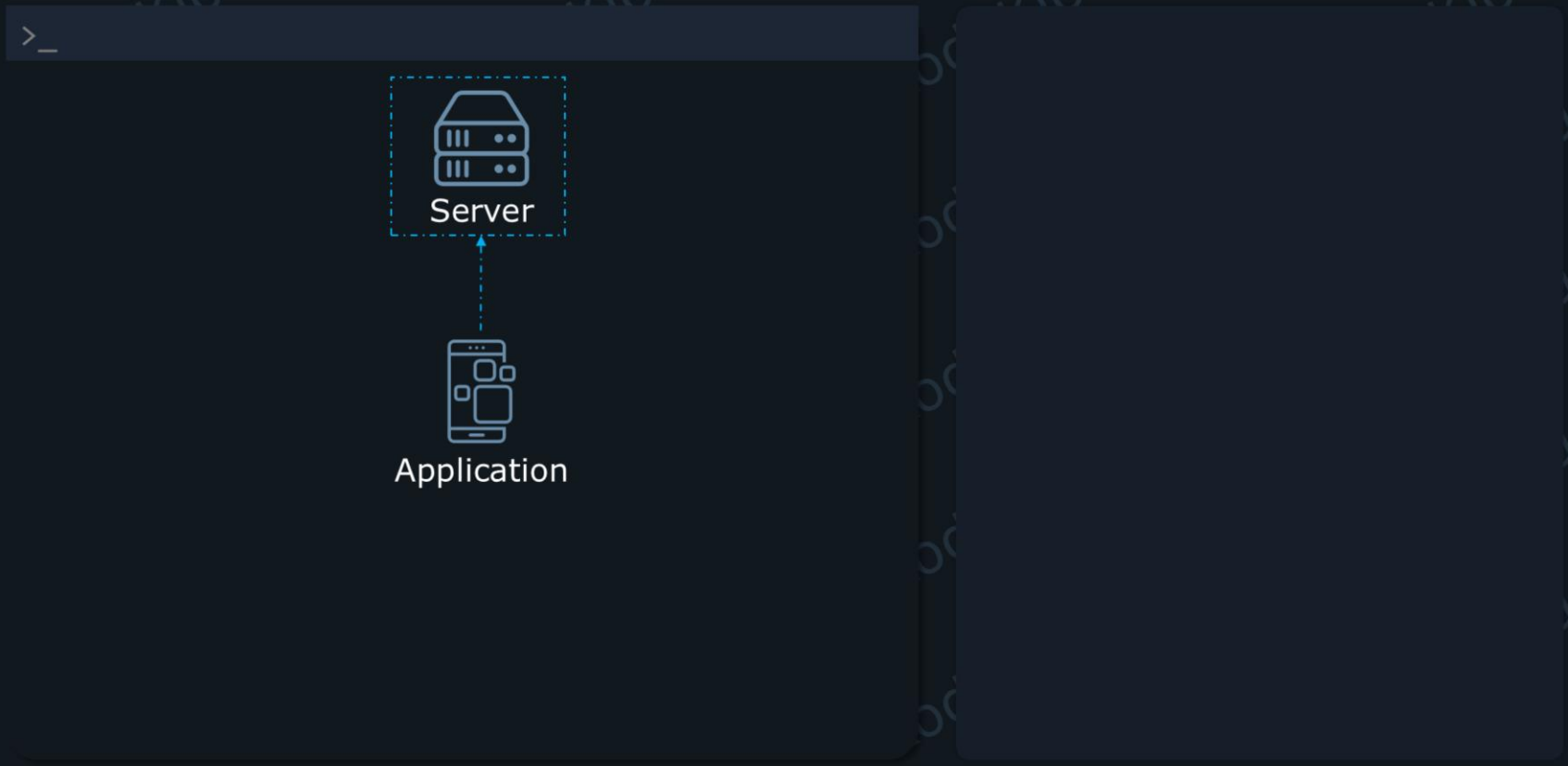
Visit www.kodekloud.com to discover more.



Let's examine how to manage startup processes and services in Linux.

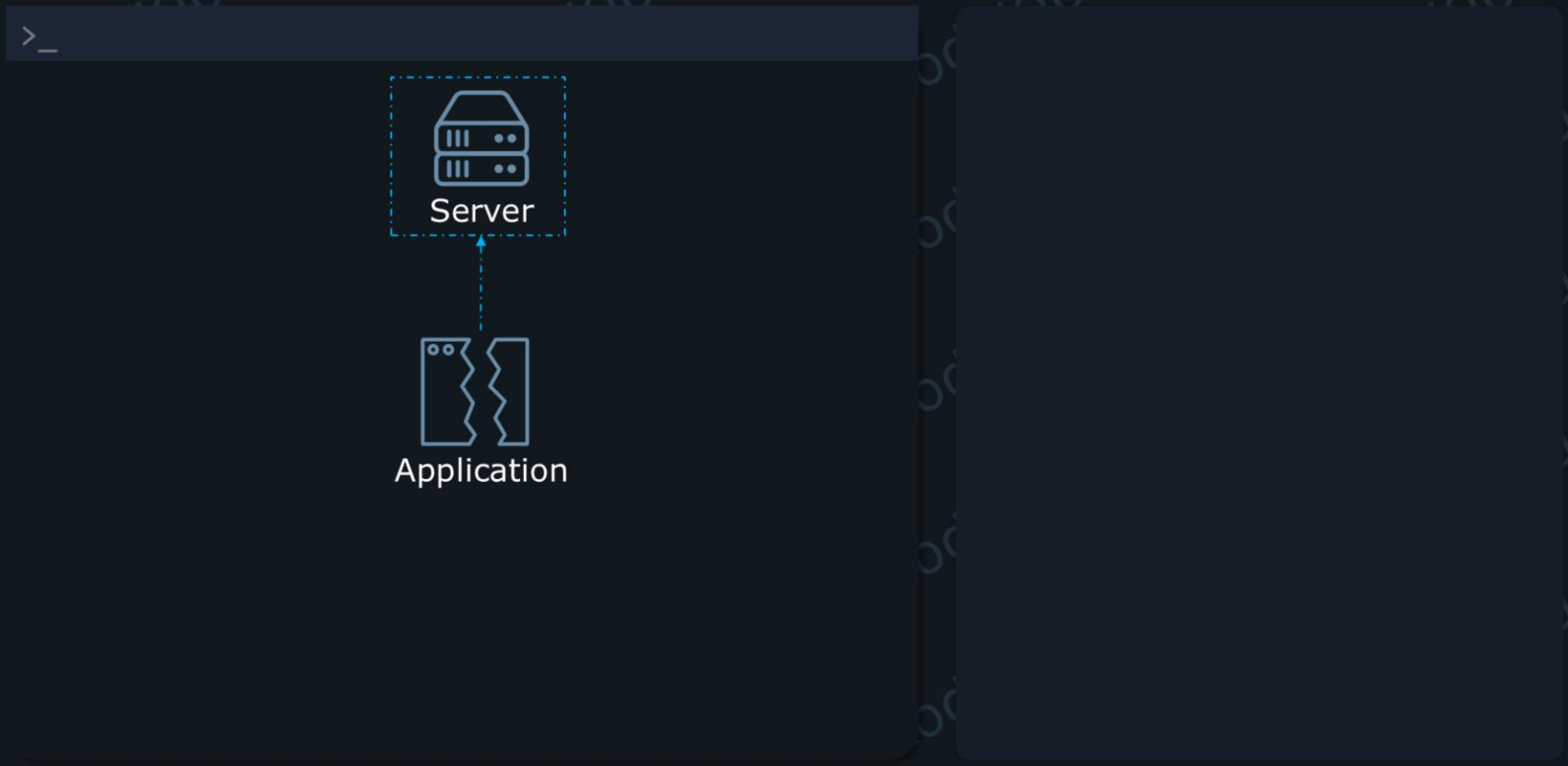
KodeKloud

Building Application



KodeKloud

Building Application



KodeKloud

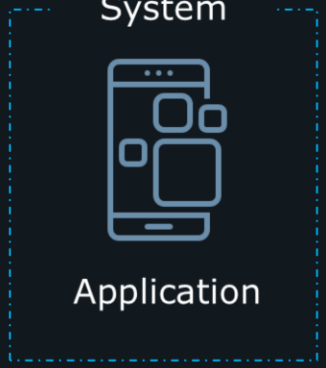
Building Application

>_

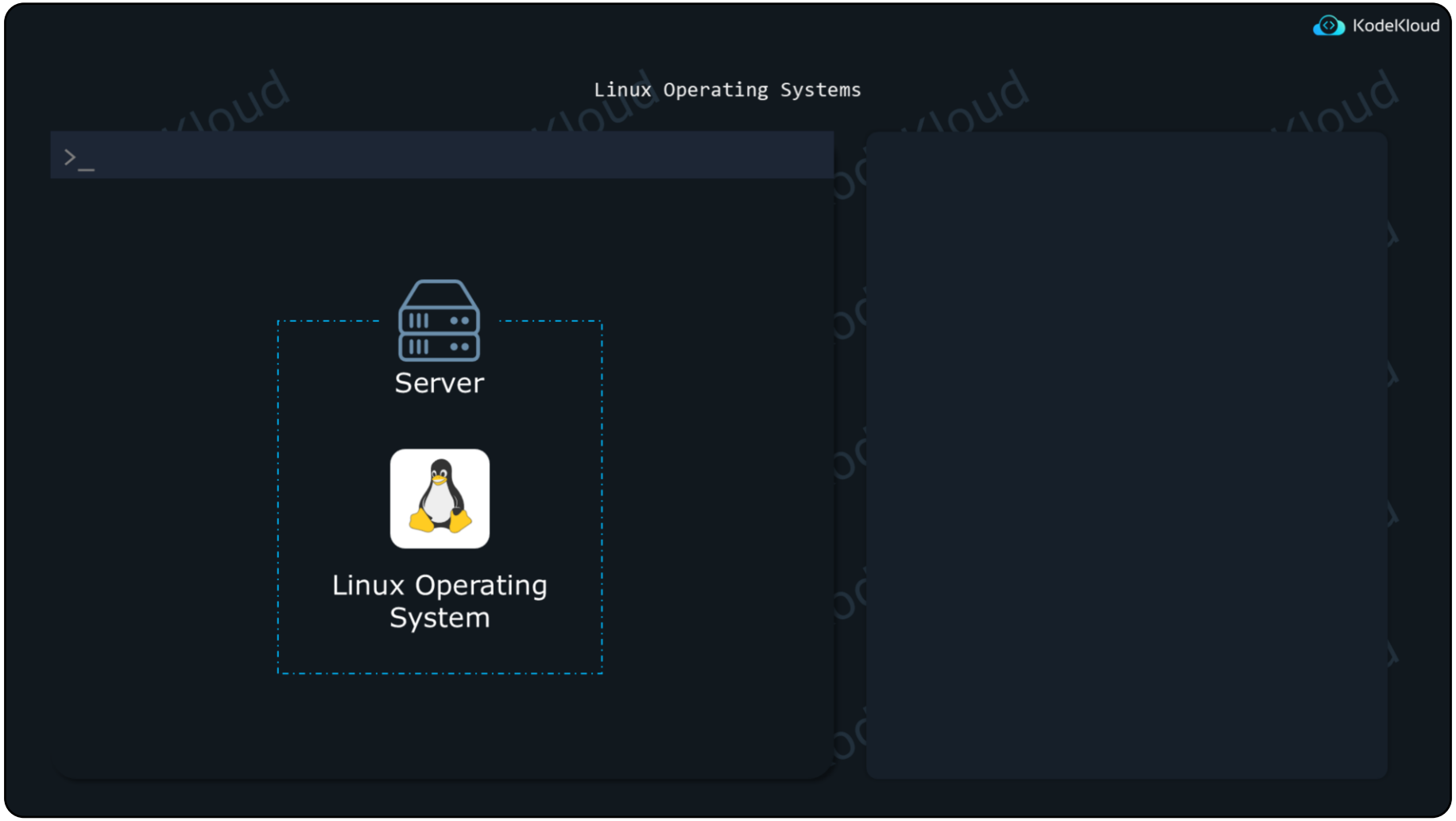
KodeKloud

Building Application

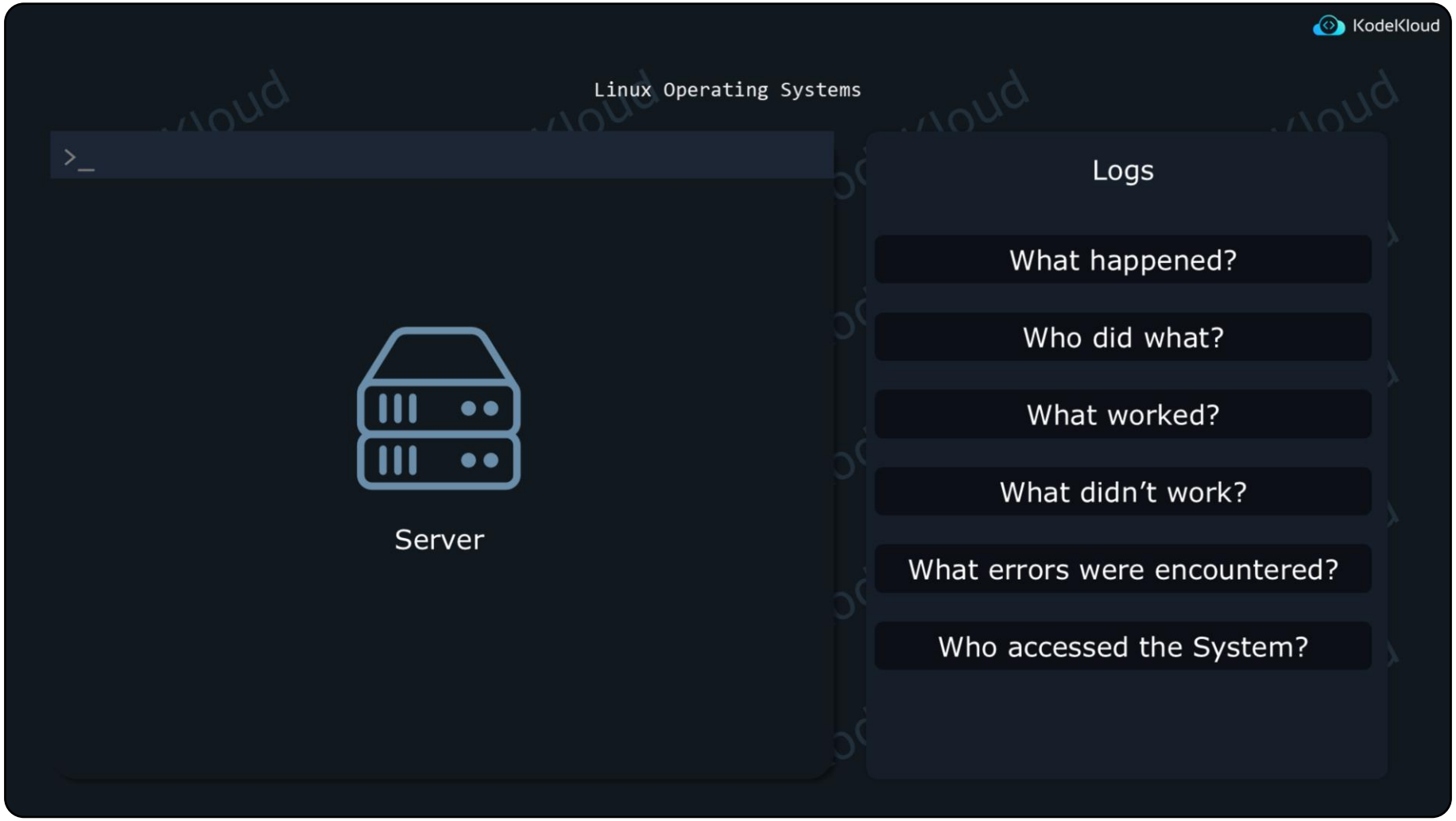
>_



Now, we'll look at how to locate and analyze system log files in Linux.



Linux operating systems are largely server-oriented.



And on a server, you always want to know what happened, who did what, what worked, what didn't work, what errors were encountered, who accessed the system, and so on. Everything important that happens on a Linux system is saved as a text message somewhere, in what are called logs.

The Linux kernel and most programs that run on the operating system generate status messages, error messages, warnings, and so on. There can be tens of different programs generating these messages all the time. So we need a way to collect all of these and organize them nicely into files. And this is the job of logging daemons. These are simply applications that collect, organize and store logs. The most popular one on Linux is rsyslog. Its name comes from "rocket-fast system for log processing".

rsyslog stores all logs in the /var/log/ directory.

ls /var/log/

Since these are usually regular text files, you can search through them with grep commands or any of the other text-oriented utilities that we learned about.

One important thing though, most of these files cannot be read by regular users. So before diving into log files, you might want to log in as the root user with a command like

su --login

followed by the password of root.

or

sudo --login

followed by the password of your current user.

Once logged in, let's say we have no idea where SSH logs are stored, to see who logged in through SSH. We can search all files and look for anything that contains the word "sshd", to find stuff belonging to the SSH daemon.

grep -r 'sshd' /var/log/

And we'll notice a few lines like this:

/var/log/auth.log:Mar 3 03:32:37 kodekloud sshd[1653]: Connection closed by authenticating user aaron 10.11.12.1 port 57660 [preauth]
/var/log/auth.log:Mar 3 03:32:39 kodekloud sshd[1655]: Accepted password for aaron from 10.11.12.1 port 52560 ssh2
/var/log/auth.log:Mar 3 03:32:39 kodekloud sshd[1655]: pam_unix(sshd:session): session opened for user aaron(uid=1000) by (uid=0)

This tells us that some SSH logs are stored in the /var/log/auth.log file.

We could then open up the file

less /var/log/auth.log

and start looking for "sshd" logs (press /, type "sshd" and press "n" to find the next match). Press q to quit.

In the /var/log/auth.log file we'll find logs related to secure authentication or authorization, things like

Who logged in through SSH. Who tried to log in and was rejected because of a bad password or anything similar.
Who used sudo to get root permissions. What command they used with sudo.
What app requested special, administrative privileges and what it did with them.

So it's not just stuff related to SSH, but also other authentication-related logs.

For example, when we change a password, Linux will log something like this to the /var/log/auth.log file:

Mar 3 03:37:38 kodekloud passwd[2130]: pam_unix(passwd:chauthtok): password changed for aaron

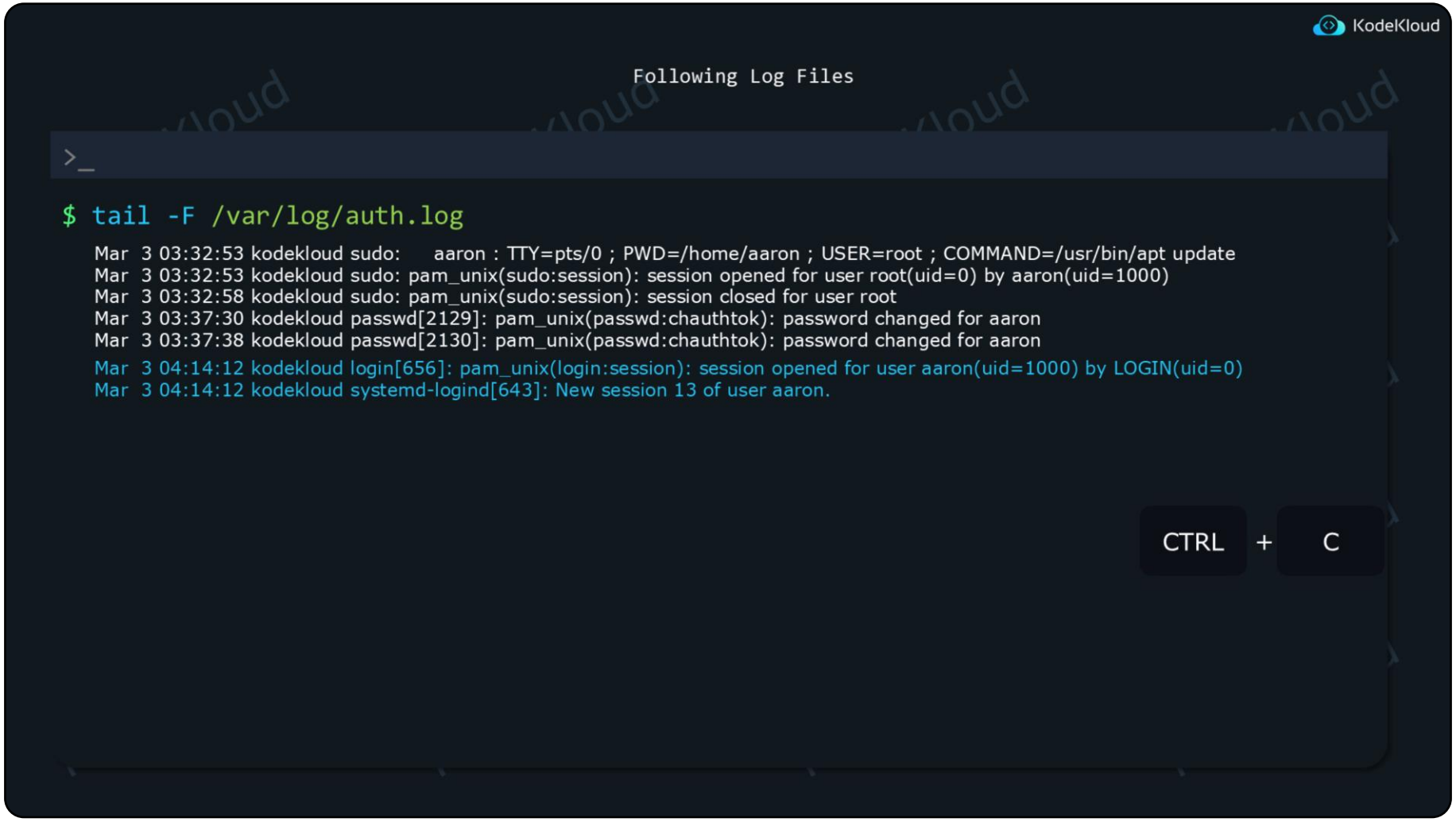
In such a log line we can see:

The date and time when this happened.
The hostname of the computer where this happened (kodekloud)
The application/source that generated this log message: passwd
And the log message itself

Many log entries follow the same format, especially system logs that keep track of what happened on the operating system.

And speaking of system logs, you'll find most of these in the /var/log/syslog file.

You might have noticed files with names like auth.log.1, or auth.log.2.gz. These are older logs. auth.log is the latest. auth.log.1 is the one that was logged before the latest auth.log. And auth.log.2.gz is the one created before auth.log.1. And it's also archived with the gzip utility to make the file smaller.



Most of the time you'll look at logs to see what happened in the past. But sometimes, you'll also want to take a look at what is currently happening. Imagine you're debugging an application. You want to try different things and see what errors it generates while you push its buttons. You can get a "live" view of a log file, see log entries as soon as they appear, with this command:

tail -F /var/log/auth.log

-F makes tail enter "follow" mode.

You can test this out by using that command and then switching to your virtual machine and logging in at the text console.

As soon as you enter your password, you'll see two new lines like this, showing information about your login:

Press CTRL+C to exit follow mode.

It's not hard to navigate through logs in /var/log/, but it could be easier.

For example, think about this. You want to see what your team did with the sudo command. You could open up /var/log/auth.log and filter results with grep or with less to only look for text entries containing the word "sudo". But why go through that trouble when there's an easier way?

We mentioned how rsyslog tracks logs and stores them in files in /var/log/. But modern Linux operating systems started using an additional way to keep track of logs. A program called the systemd journal daemon is a bit smarter at how it collects this data.

And the journalctl (journal control) command lets us analyze logs more efficiently.

For example, journalctl lets us filter for logs generated by a specific command.

First, we'd need to find out the full path to the sudo command:

Now when we type in which sudo

This will tell us that the sudo utility is located at /usr/bin/sudo. And now we can easily tell journalctl, "Hey, only show me logs generated by this command:"

journalctl /usr/bin/sudo

This output opens up in the less pager, so we can use the navigation keys, search keys, and everything else we learned we can do in the less utility. Press q to quit.

In a previous lesson we learned how the operating system is managing important system applications with the help of service units. For example, the SSH daemon is constantly running in the background and such a service unit controls and monitors its activity. If we know the name of the service, we can view logs for it with:

journalctl -u ssh.service

-u tells journalctl to display logs generated by the ssh.service unit.

If we simply type:

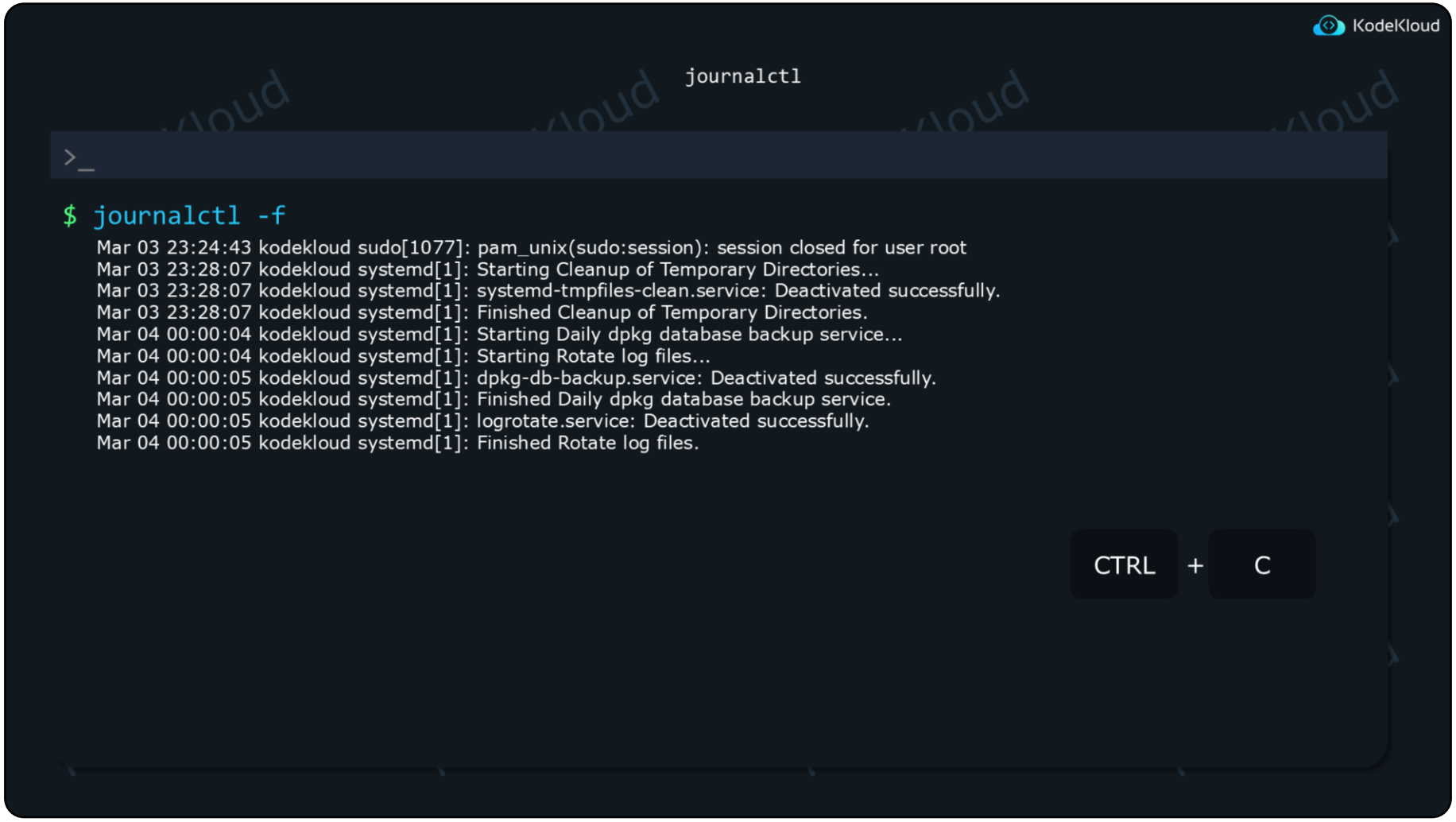
journalctl

we'll see all logs collected by the journal daemon. To go to the end of the output, we can press >.

If we want to open the journal and instantly go to the end of the output, we can use the -e, end option.

journalctl -e

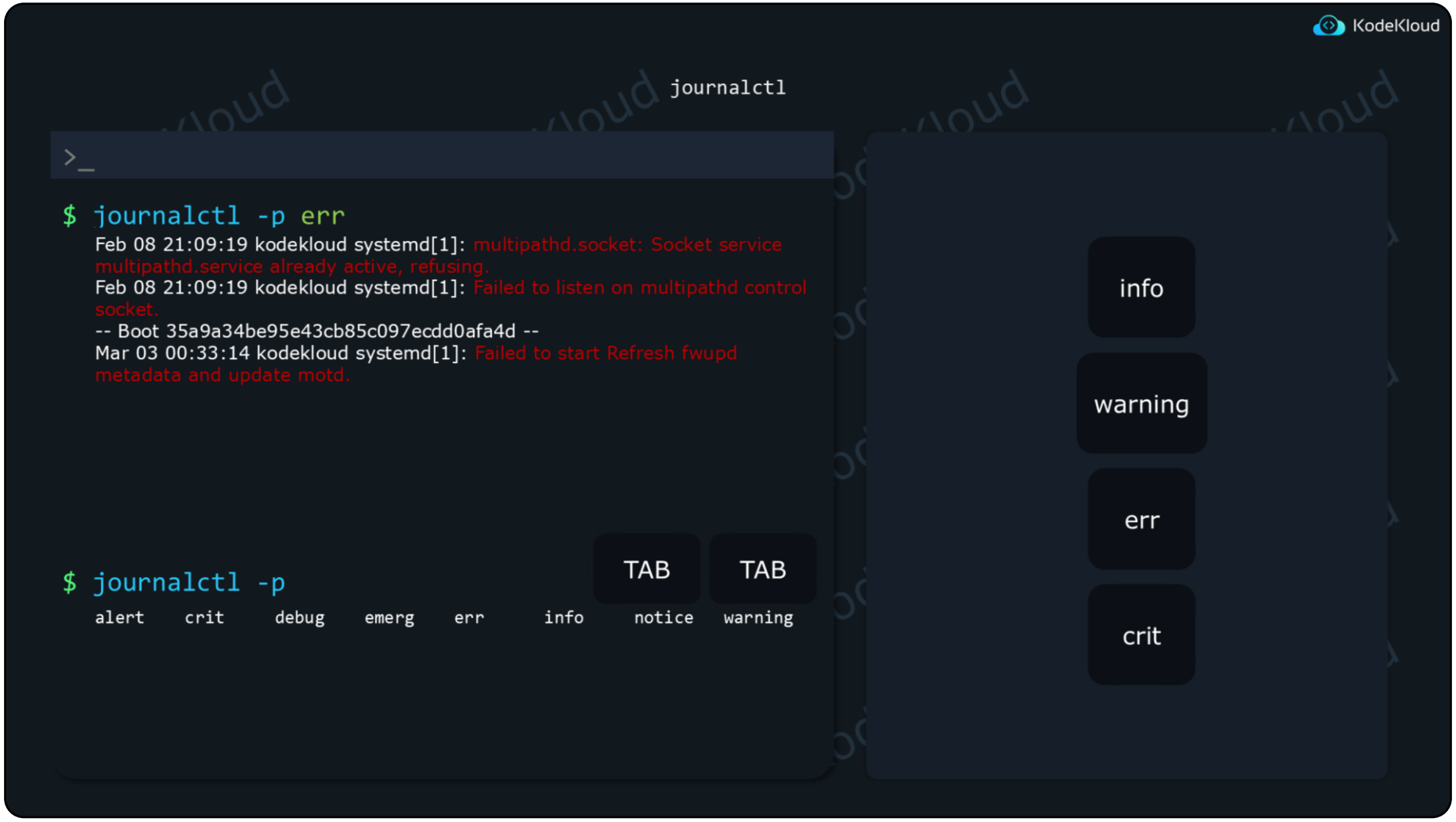
Just like we opened up a log file in "follow mode", journalctl also has a follow mode.



We can access that by passing the -f option to our command:

journalctl -f

And now we have a live view of all new logs that appear on the system. Press CTRL+C to exit this.



Most logs are just informative. Telling us about normal stuff that is happening on the system. But some are warnings about things not going quite well. Others are errors about things that failed. And others are huge alarm signals about things going critically wrong. These are tagged as

info

warning

err

crit

If we'd only want to look for moderately serious errors, we can use the -p, priority command line option:

journalctl -p err

This shows us messages tagged with the "err", or error priority level.

The list of priorities we can use with the -p option:

alert crit debug emerg err info notice warning

If you forget this list just write "journalctl -p ", then add one space at the end, and press TAB two times. You'll get a list of all priority codenames you can use.

We can also use grep expressions to filter output, with the help of the -g command line option. And, of course, we can combine multiple command line options. For example, we could filter out only logs with the "info" priority that have a message beginning with the letter "b".

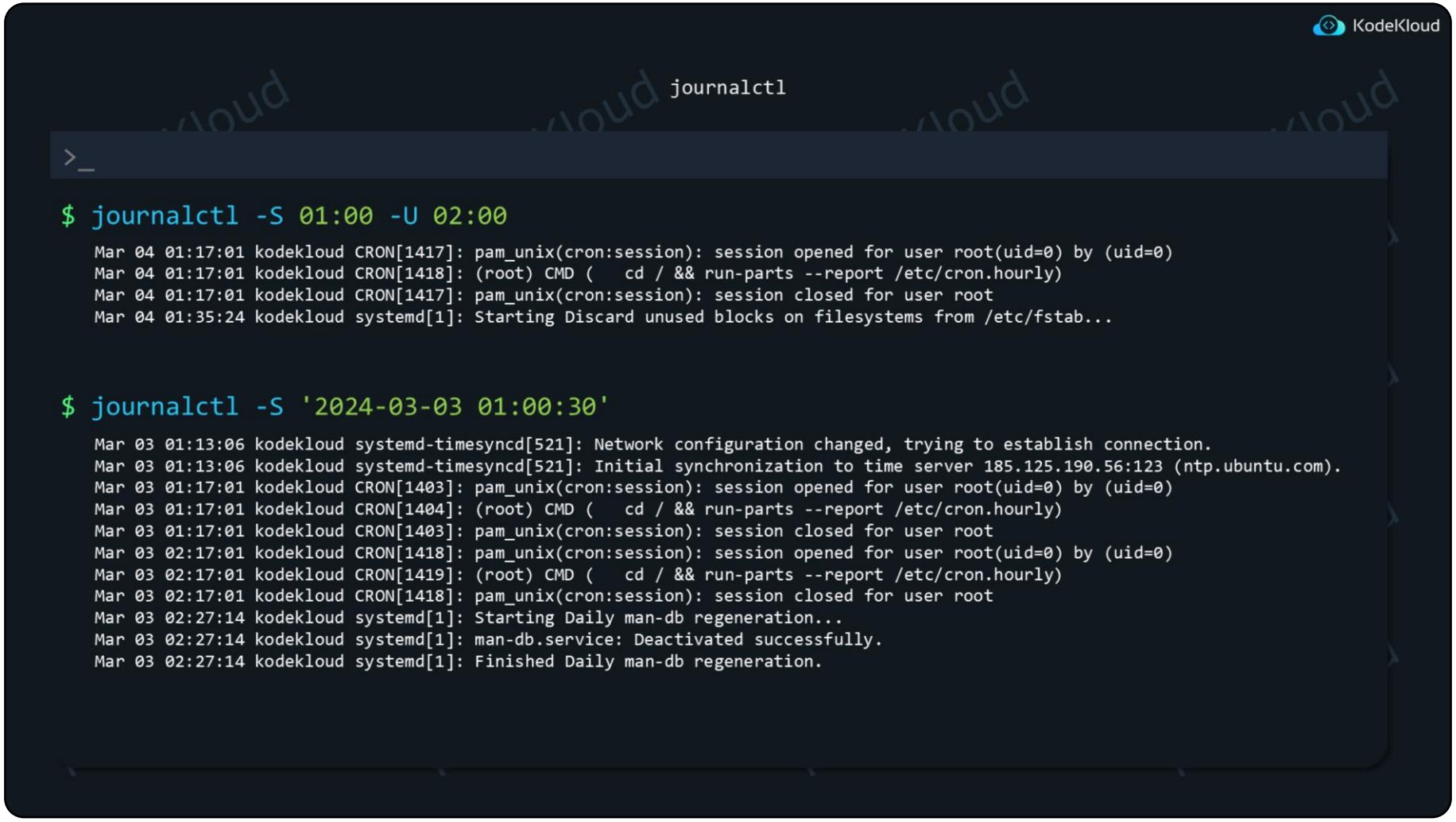
journalctl -p info -g '^b'

With -g '^b' we passed a grep expression to only look for lines that begin with "b".

We can also tell journalctl to only display logs recorded after a certain time, with the -S, "since" option:

journalctl -S 02:00

This uses the 24 hour format so 02:00 shows us logs that were generated since 2 AM.



We also have an until, -U option. So if we combine both options, we could see logs generated since 1AM and until 2AM, with:

journalctl -S 01:00 -U 02:00

The -S and -U options also support dates. So we can use a command like this to see logs generated since 2024, March 3rd, 23:00 and 30 seconds.

journalctl -S '2024-03-03 01:00:30'

Note how we can fine tune this to the second.

It's important to wrap this date format between ' ' single quotes.

Normally, journalctl will show us all logs collected on the system, starting with the first one available. But the first entries might be super old, for example, since two months ago. In most cases, we won't need such old data.

In a nutshell, the journal daemon also groups logs based on when the system booted. And often, we'll want to see logs generated since the last time the system booted. To see logs generated for this current boot we can use the -b boot option:

journalctl -b 0

In this case, "-b 0" picks our current boot, boot zero.

To see logs for the previous boot:

journalctl -b -1

To see logs generated two boots ago:

journalctl -b -2

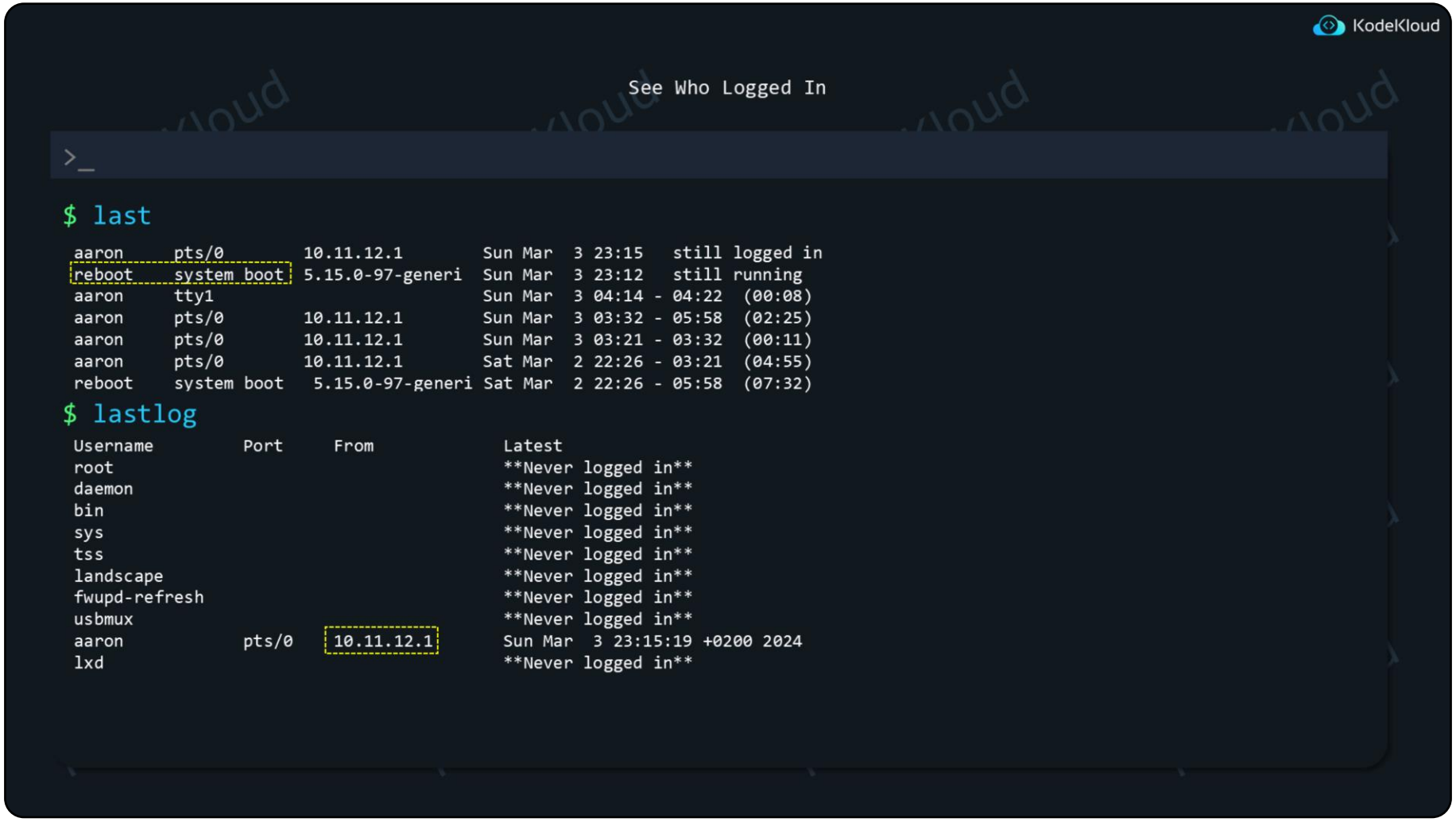
On some operating systems though, the journaling daemon is configured to only keep logs for the current boot. And only in memory, not on disk. So as soon as we reboot, the journal log is lost. And a command like "journalctl –b –1" might show us that "no persistent journal was found". Although text logs are still preserved in the old-fashioned logging format in the /var/log/ directory.

But, usually, the journaling daemon is configured to decide whether to save a persistent journal, depending on the existence of the directory: /var/log/journal. If it doesn't exist, it won't preserve journal logs on disk. But if it exists, it will start to preserve systemd journal logs in that directory. So by simply creating this directory, we can instruct the journaling daemon to preserve these logs there:

sudo mkdir /var/log/journal/

It's also worth noting that using the journalctl command requires you to be logged in as root, or your current user to be part of certain groups. Either "wheel" on operating systems from the Red Hat family, or "adm", or "sudo" on operating systems from the Ubuntu family.

If a simple "journalctl" command does not show you the logs, just prepend that command with "sudo" to temporarily get the required privilege.



To see a history of who logged in to the system, you can use the last command:

last

The newest entries are at the top, so you might have to scroll up to see them.

The lines beginning with "reboot system boot" also show you when this system was powered on.

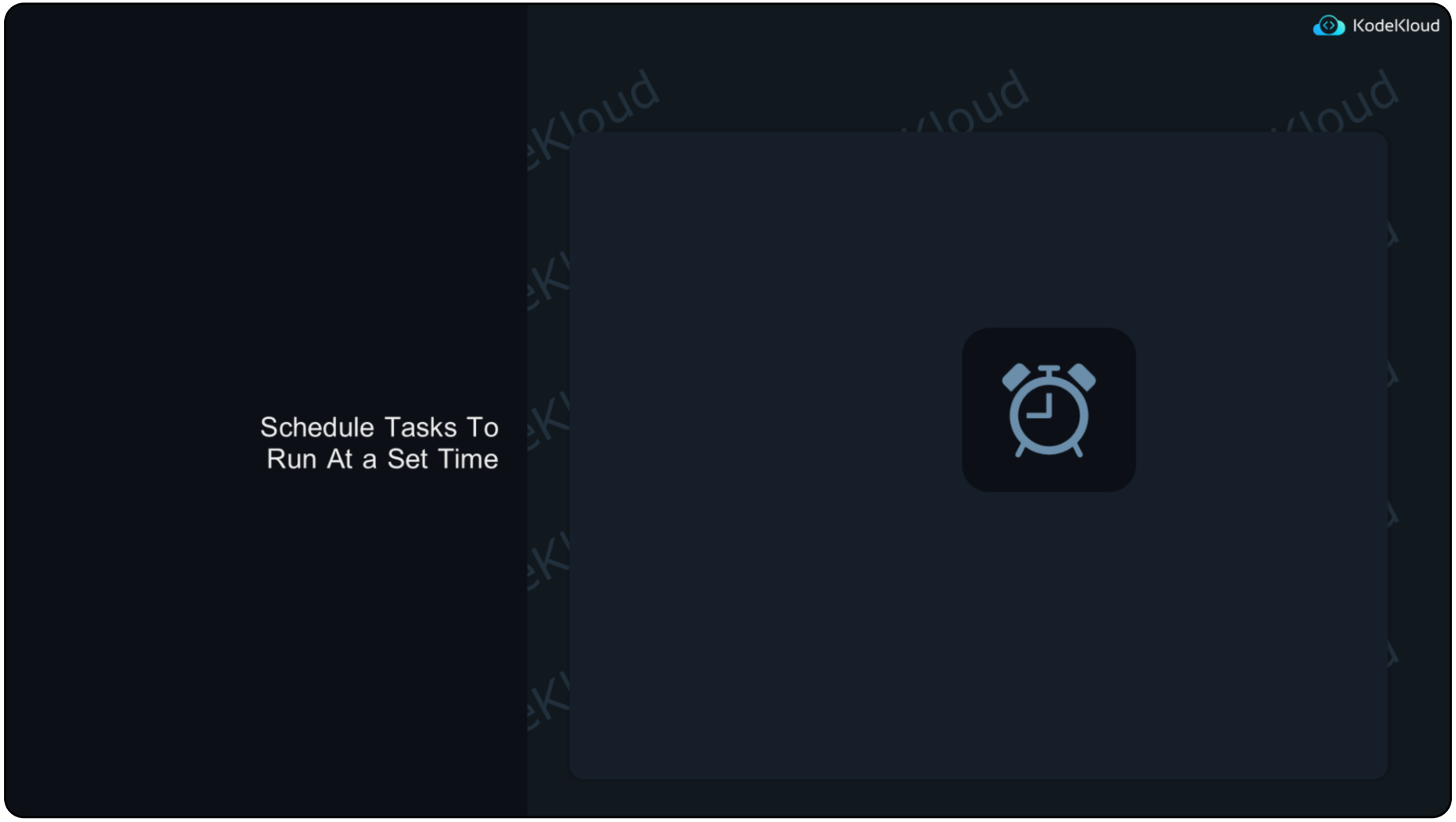
An alternative command is:

lastlog

This shows you when each user on the system logged in the last time. For remote logins, for example through SSH, you can also see the IP address from which they logged in.

KodeKloud

Visit www.kodekloud.com to discover more.



Now, we'll explore how to schedule tasks to run at set times in Linux.

On servers, we'll sometimes need to set up some tasks to run automatically. For example, we could create a job that automatically backs up the database, next, or every Sunday at 3 AM. We have three main ways to set up such tasks:

With the cron utility.

With anacron.
Or with the utility called "at".

Here's the main purpose of each tool:

The cron utility is well-suited for repetitive jobs that execute once every few minutes, or hours, or on specific days, and specific times.

anacron is also used to create repetitive jobs. But with a few differences. One is that it cannot repeat those jobs every few minutes or hours. The smallest unit it can work with is a day. So it can run a job every day, or every 3 days, or every week, even every month, or year. But it cannot run a job multiple times per day, just once per day. This utility has been created because cron can miss jobs if the computer is not powered on all the time. For example, if a job is set to run daily at 12:00 and the computer is powered on at 12:01, that job won't run for that day. anacron, on the other hand, will check if the job for today ever got a chance to run. If it didn't, it will execute it, no matter when the system is powered on.

Now for the "at" utility. We saw that cron and anacron are focused on repetitive automated tasks. In contrast, at is focused on tasks that should only run once.

KodeKloud

Setting tasks automatically

Server

Automated Tasks

KodeKloud

Example

Server

Automated Tasks

Task:

Database Back-up every Sunday at 3:00 AM

Cron

- Minute
- Hour
- Day
- Other specific times

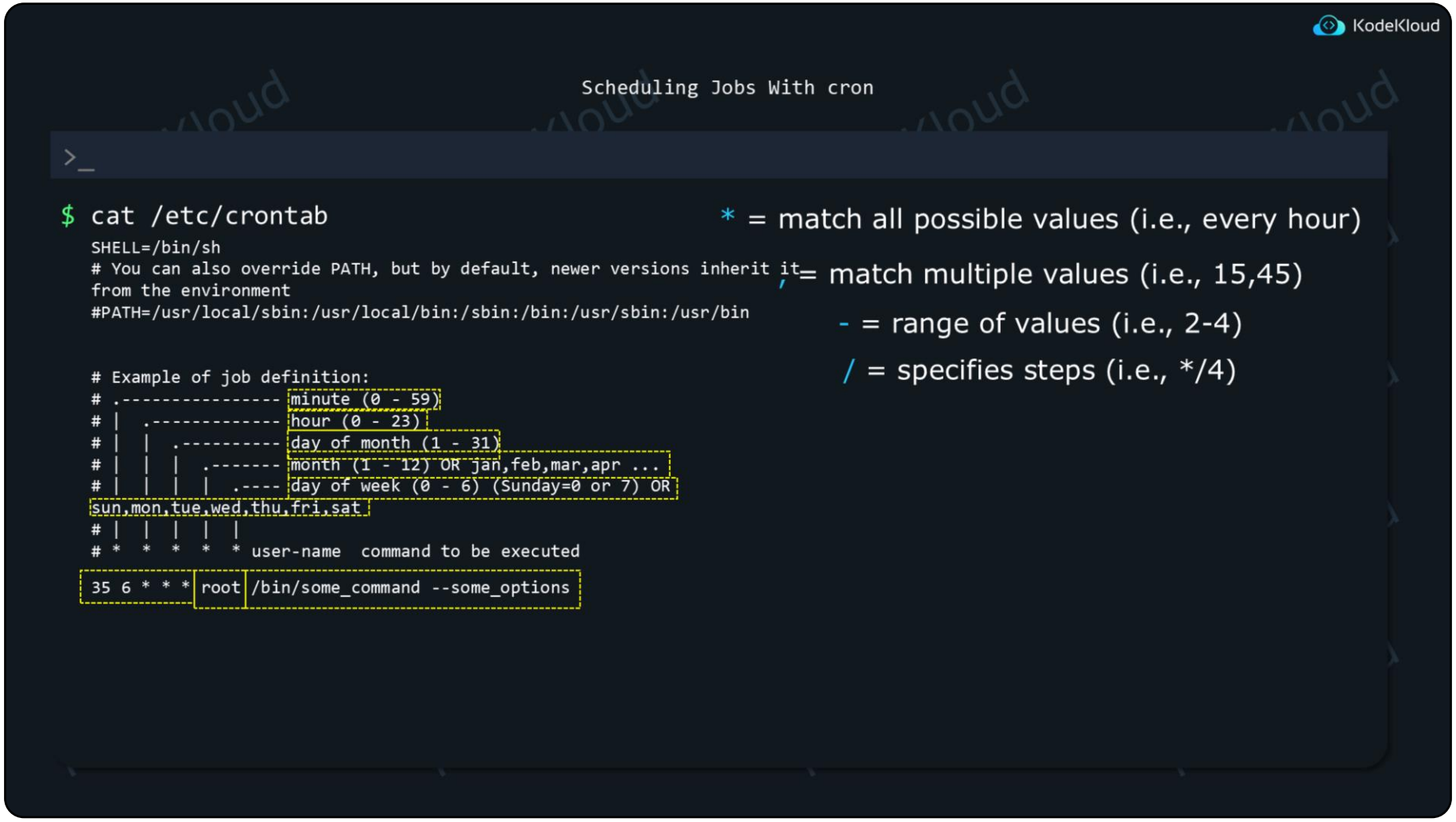
(Miss jobs when power is off)

Anacron

- Day
- Few Days
- Week
- Month
- Year

at

For tasks that only run once



The syntax of a cron job might be hard to remember the first few times you try to set one up. But you can get a quick reminder by taking a look at this file:

cat /etc/crontab

This is the default, system-wide cron table. And the commented lines tell you about the syntax you should use. If you ever need to add a cron job here, add a new line to the end of the file. For example, you could use a line like this:

35 6 * * * root /bin/some_command –some_options

The first five values tell cron when this should run. The sixth value is an username. And this is followed by the command that should run. cron will execute this as if the user specified in the username field launched that task.

The thing about adding jobs to this default system-wide table is that in certain cases, the package manager might overwrite this file. And we might lose the changes we've added. For example, this could happen when the package manager upgrades cron.

So in our lesson we'll learn how to add cron jobs by editing a user's personal cron table, instead of the system-wide table. Just keep in mind when you are managing servers to first look up cron jobs in the /etc/crontab system-wide table. And afterwards look at the cron table belonging to the root user. We'll soon learn how to do that.

But first, let's dive into the first five values here that establish when the cron job should run.

In the first two fields we pick the time:

Minute - between 0 and 59
Hour - between 0 and 23

In the next two fields we pick the:

Day of the month - between 1 and 31
Month - between 1 and 12

And in the last field we pick the:

Day of the week - between 0 and 6, 0 being Sunday, 1 being Monday, 2 Tuesday and so on. This can be a bit confusing since we may feel it should naturally be a number between 1 and 7. So cron accepts the alternate notation too, with 1 being Monday and 7 being Sunday.

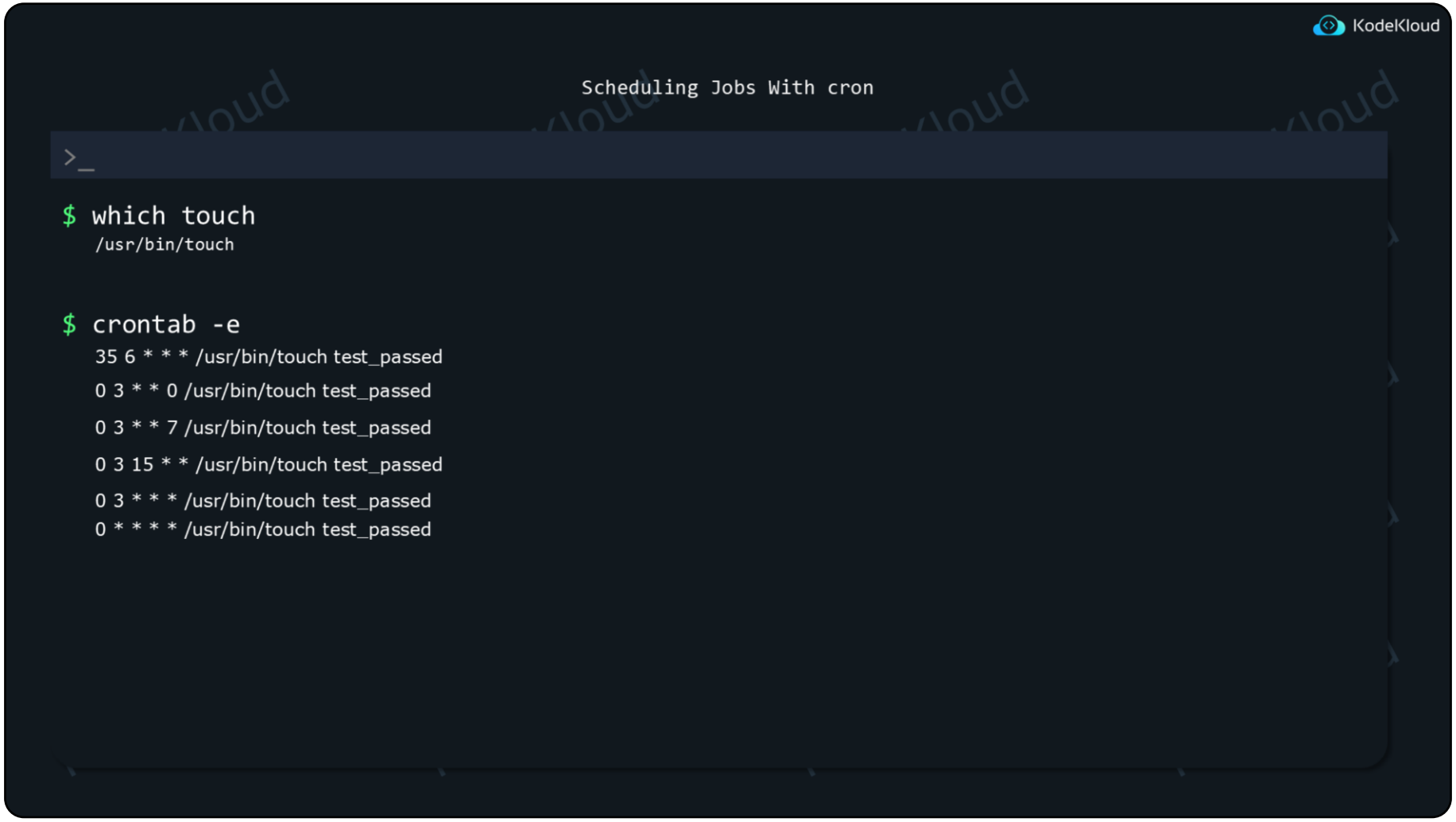
Instead of fixed numbers, in these fields we can also use alternate characters:

* matches all values possible in that field. So a * for the hour field means run at every hour, 0, 1, 2, all the way to 23.

, can be used to specify, or enumerate multiple values for that field. For example, if we'd want something to run at minute 15 and minute 45, we could type "15,45" (NO SPACES) in the first field.

- can be used to specify a range. For example, if we'd want to schedule a job to run at night, at hours 2, 3 and 4 AM, we could type this in the second field: "2-4".

/ specifies a "step". For example, if you have * in the hour field, cron will step through hours, one by one. So, by default, the step is one. First, the job will run at 0 AM, then 1 AM, then 2 AM, and so on. But if we want it to run every four hours, we need to change the step to 4. To do this, we can type "*/4". The job will now run at 0AM, then 4AM, then 8AM, 12AM, and so on. We can also combine ranges with steps. For example, let's say we want to run something at only these three hours: 0AM, 4AM and 8AM. We could type "0-8/4" in the hour field. "0-8" specifying the range from 0AM to 8AM. And /4 telling cron to step through this range in 4 hour increments.



With theory alone this kind of stuff can look a bit abstract, and hard to digest. So let's make it crystal clear by looking at some practical examples.

We'll add a simple cron job that just creates a file. This way, if we see the file was created at a certain time, we'll know that our job executed correctly.

In cron jobs, we can execute either commands, or shell scripts.

We'll use the touch command here. First thing required is to find the full path to this command. We can do this by typing:

which touch

It's important to remember that in cron jobs we should always use the full path to a command.

We mentioned we'll edit the user's cron table instead of the system-wide cron table at /etc/crontab. To edit a user's cron table we can use the -e, edit option in the crontab command:

crontab –e

This will edit the table of your current user; whoever you're currently logged in as.

To run the touch command, and create a file called "test_passed", every day, at 6:35 AM, we could add this line:

35 6 * * * /usr/bin/touch test_passed

You might notice that there's no field specifying an username now (as we had in the system-wide cron table). This is not necessary here because this time cron sees who the table belongs to and knows to run commands as that user.

Now we can save our file and exit the text editor and that's all there is to adding a cron job.

To give you ideas about alternate time settings for a cron job, here are some other lines we could have used:

To run this every Sunday at 3AM:

0 3 * * 0 /usr/bin/touch test_passed

or

0 3 * * 7 /usr/bin/touch test_passed

To run it at the middle of every month, on the 15th, at 3AM:

0 3 15 * * /usr/bin/touch test_passed

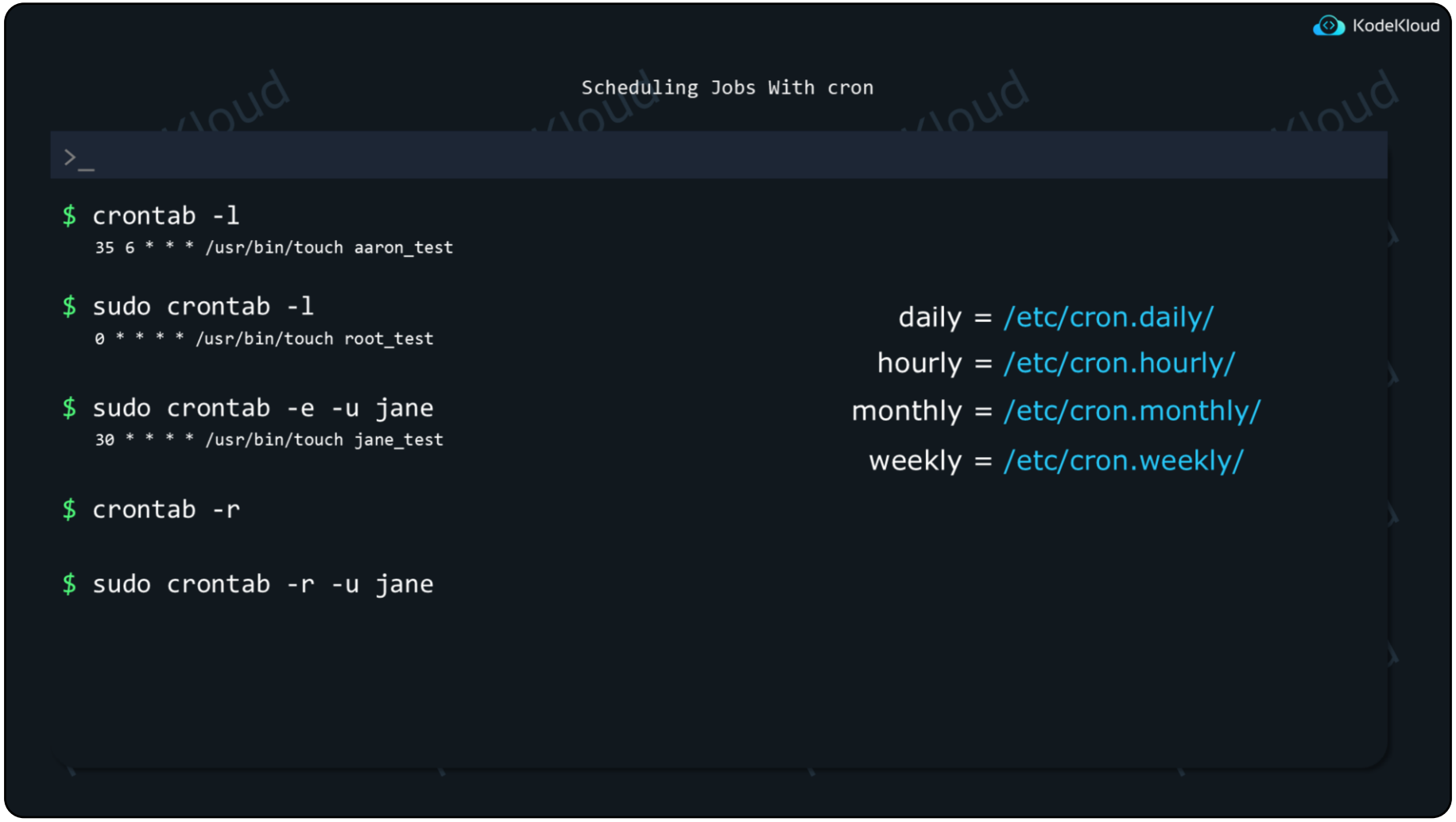
To run it every day at 3 AM:

0 3 * * * /usr/bin/touch test_passed

To run it once at every hour (when minute is :00):

0 * * * * /usr/bin/touch test_passed

If you want to practice and get used to writing cron expressions, you can visit a website with the address: "crontab.guru"



To see the crontab of the current user, use the -l, list option

crontab –l

If you want to see the cron table of the root user, just add sudo in front of the command:

sudo crontab -l

Same if you want to edit the table of the root user:

sudo crontab -e

If you want to edit the cron table of a different user, you have two options. One is to simply log in as that user and then use the regular crontab commands. The other is to use the -u, user option, at the end of your command. For example, to edit the cron table of the user "jane":

sudo crontab -e -u jane

Only root can edit the cron tables of other users, that's why you also have to add sudo before this command.

To remove your user's crontab entirely, use the -r, remove option:

crontab -r

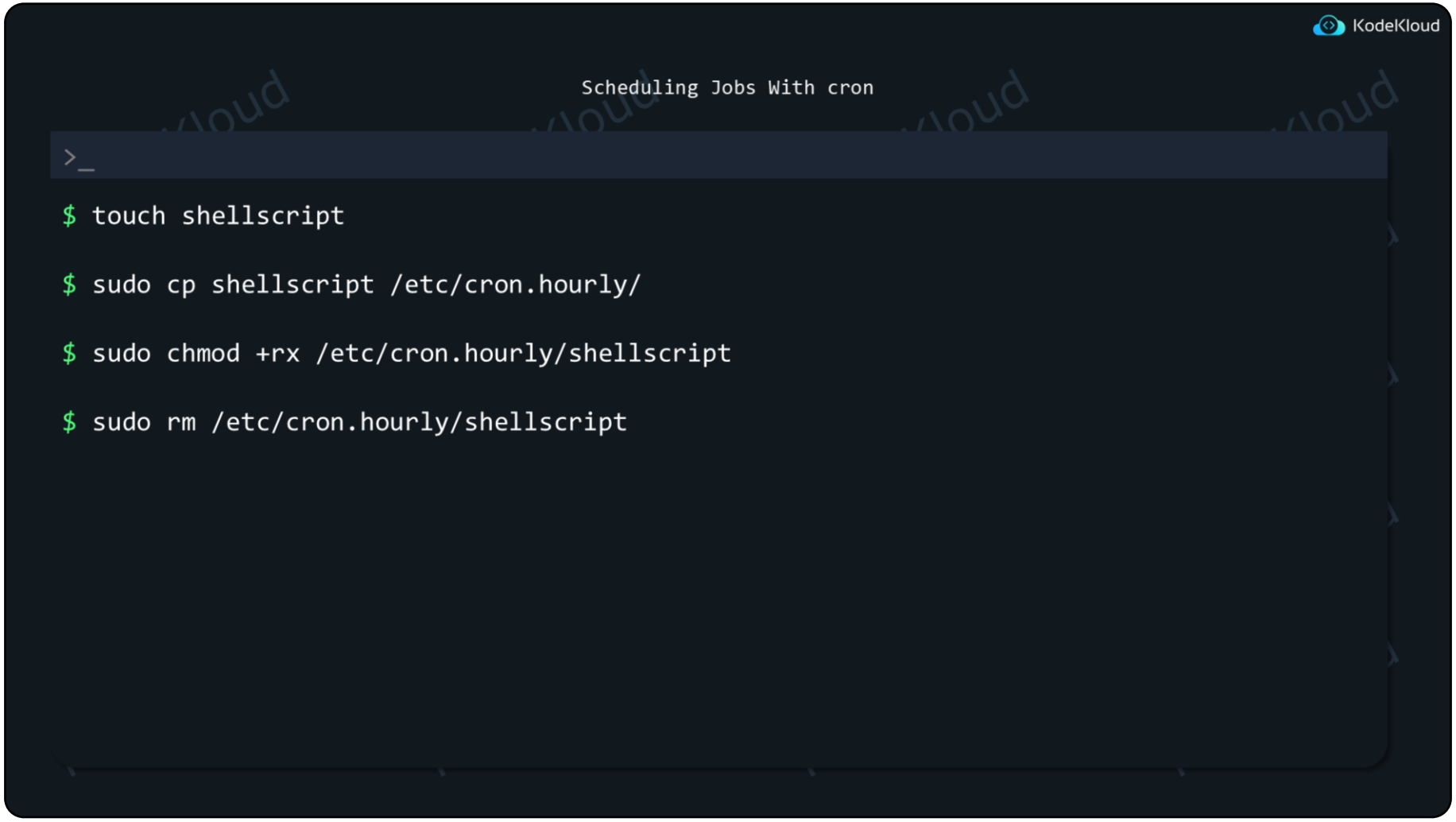
To remove the cron table of a different user:

sudo crontab -r -u jane

An alternative way to set up cron jobs is through these special directories:

/etc/cron.daily
/etc/cron.hourly
/etc/cron.monthly
/etc/cron.weekly

Whatever cron finds in here, it will run daily, hourly, monthly or weekly.



Let's see how that works.

Imagine we have a shell script in our local directory, called simply "shellscript":

touch shellscript

The shell script file should have no extension, if we want to use it as a cron job. For example, don't use a name such as shellscript.sh if you intend to use this with cron.

To make this run hourly, we would follow this procedure.

First, copy the script to /etc/cron.hourly/. We need root privileges since only the root user can write to this directory.

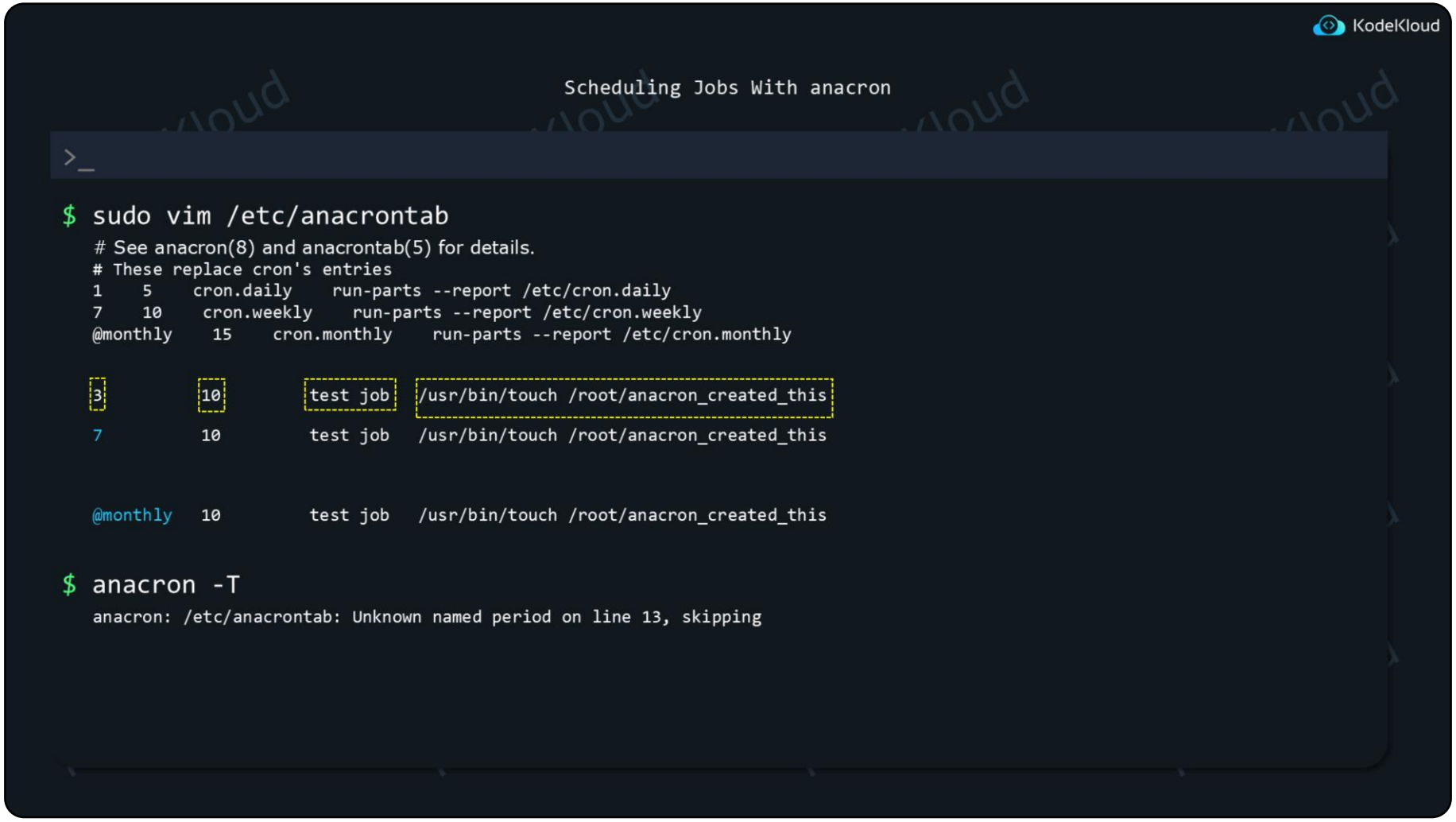
```
sudo cp shellscript /etc/cron.hourly/
```

Next, make sure the script is readable and executable, so cron can run it:

```
sudo chmod +rx /etc/cron.hourly/shellscript
```

If later on you need to remove this job, simply remove the file.

```
sudo rm /etc/cron.hourly/shellscript
```



Now let's move on to the next utility for scheduling jobs. On some Ubuntu systems "anacron" might not be installed by default, but we can install it with the command:

sudo apt install anacron

When using anacron, we don't really care at what time the job will run. We just want it to run daily, monthly, or once every few days, no matter the time of the day.

To schedule a job with anacron, we edit the /etc/anacrontab file:

sudo vim /etc/anacrontab

There are a few helpful hints in this file. First is that we can consult the anacrontab manual in section 5 to see more details about the syntax here. So we can get some help with a command like:

man 5 anacrontab

Second, we already have three anacron entries at the end of this file, showing us some live examples.

The syntax is quite simple. We have 4 fields in an anacrontab line:

#period in days delay in minutes job-identifier command

First, we have a number that decides how often this should run. The period in the first field is specified in days. Typing 3 here would mean the job will run once every 3 days. There's also an alternate format where we can specify this in natural language. For example, instead of 30, or 31, we can just type @monthly.

When anacron checks to see what needs to run, maybe 10 different jobs have been missed because the machine was powered off during that day. It wouldn't be healthy to start all 10 jobs at the same time. So in the second field, we can pick a different delay for each one, and make anacron run each job after waiting a specific number of minutes. This way jobs would be spaced out, starting at different times, not all at once.

The third field, the job identifier, is useful for logging. You could name your job any way you want. If you'd name it test_job, you'd see a line like this in your logs: anacron[1947]: Job `test_job` started. So you could identify each scheduled task more easily.

Finally, the fourth field is where we specify the command we want to run. Just like with cron, we should specify the full path to the command here, or the full path to the script we want to run.

So, to add a job that uses the touch command to create a file at the path /root/anacron_created_this, we can add this line at the end of anacrontab:

3 10 test_job /usr/bin/touch /root/anacron_created_this

Such a line makes the job run once every three days, with a delay of 10 minutes and the job identifier for it, or its name, would be set to "test_job".

The command in this job will run as the root user.

If we'd want it to run every week, we could write:

7 10 test_job /usr/bin/touch /root/anacron_created_this

And if we'd want it to run every month:

@monthly 10 test_job /usr/bin/touch /root/anacron_created_this

After we save our file, we should verify if we used the correct syntax. We can do that by passing the -T, test option to the anacron command:

To verify if the syntax of our anacron job is correct, we can use the -T, test option:

anacron -T

We get a message if there are errors. But if it's correct, we get no message.

As mentioned in the intro, cron and anacron are meant to run jobs that repeat periodically. But the utility called "at" is used to schedule jobs that only need to run once.

On Ubuntu systems it might not be pre-installed, but it can be installed with the command:

sudo apt install at

With the "at" command, it's much easier to schedule jobs. We just specify the time when we want the job to run.

For example, to run something at 15:00, we just type:

at '15:00'

Then we type the commands that we want to run at that time. For example:

/usr/bin/touch file_created_by_at

Note how we also use the full path to the command here. Although in the case of "at" this is not an absolute requirement. We could just type "touch" without preceding that with the /usr/bin/ path.

After each command we want this job to run we press ENTER. If we need to, we could add a second command, and a third one, and so on. After pressing ENTER after the last command, on the next empty line, we press CTRL+D to save this job.

We can also schedule this to run on a specific date:

at 'August 20 2024'

If we want to run on a specific date, but also a specific time:

at '2:30 August 20 2024'

We can also use relative times/dates. To run this 30 minutes later:

at 'now + 30 minutes'

To run it three hours later:

at 'now + 3 hours'

As you may have guessed, we could do the same with days, weeks, or months:

at 'now + 3 days'
at 'now + 3 weeks'
at 'now + 3 months'

Note how we have to wrap our time/date specifications between ' ' single quotes.

To see what jobs are currently scheduled to run with at, we can use the atq command (at queue).

atq

1	Wed Mar 6 15:00:00 2024 a aaron

The first number in the output is the job id, which will be used in our next commands.

Later on, we might forget what a job was supposed to do. We can use the -c, "cat" option (like the cat command that prints contents of a file) to display what an at job contains:

at -c 1

We'll see a lot of output, but the commands that we added to this job are included in the last lines.

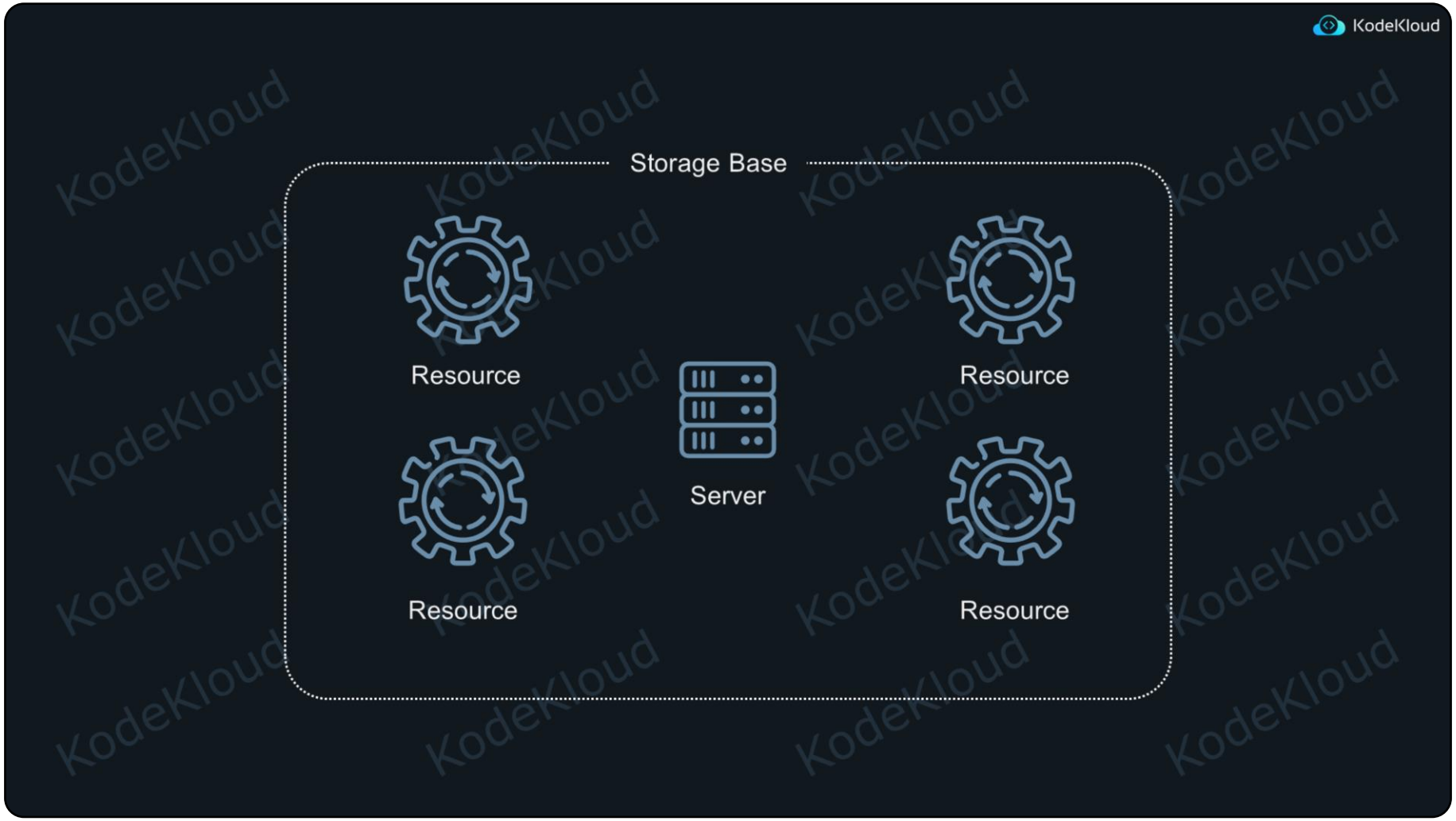
To remove a scheduled at job, we can use its job identifier number at the end of the atrm command. Example:

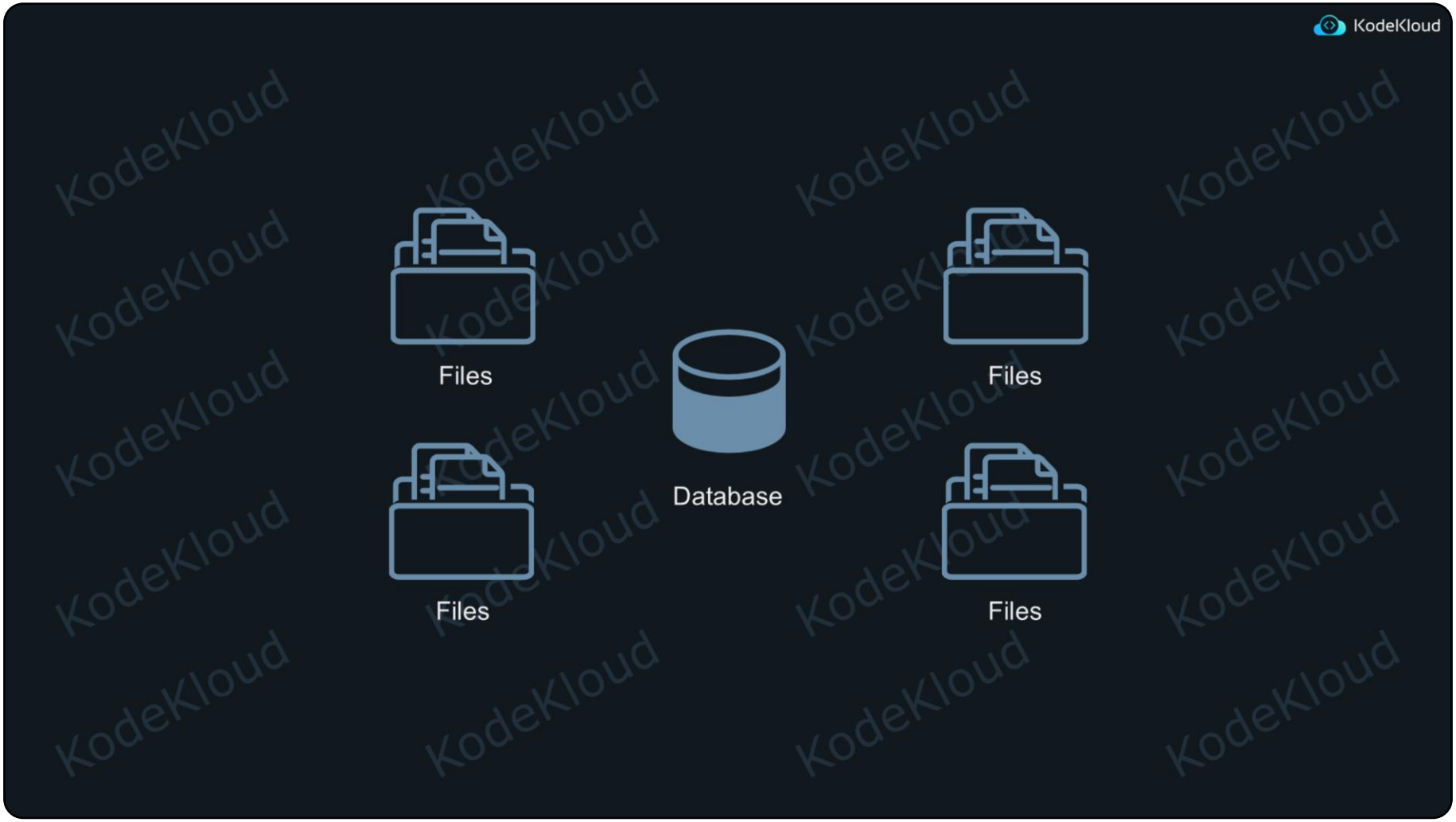
atrm 1

KodeKloud

Visit www.kodekloud.com to discover more.

In this lesson we'll take a look at how to verify the integrity and availability of important resources and processes.


KodeKloud
Storage Base
Resource
Resource
Server
Resource
Resource


KodeKloud
Files
Files
Database
Files
Files

As time goes by, a server will usually use more and more resources. Storage space is a good example. As a database grows, as users store more files, more and more gigabytes of storage space are used. In the cloud, it's relatively easy to add more disks or SSDs to a server. But how do we know when we are running out of storage space? We can use the df (disk free) utility.

df

But this output is a bit hard to read. For example, we see how much space is used on a disk, but we see this in 1Kilobyte blocks. We can tell df to make this output a bit more "human readable", with the -h option:

df -h

Now sizes are displayed in megabytes, gigabytes and terabytes instead, where we see the letters M, G or T.

We can ignore the filesystems that contain the word tmpfs. Those are virtual filesystems that only exist in the computers' memory, not on the storage devices. In our case, we see we have two real filesystems. The one Mounted on / is the root filesystem, where our Linux operating system is installed. The one mounted on /boot is a small filesystem where boot files are installed.

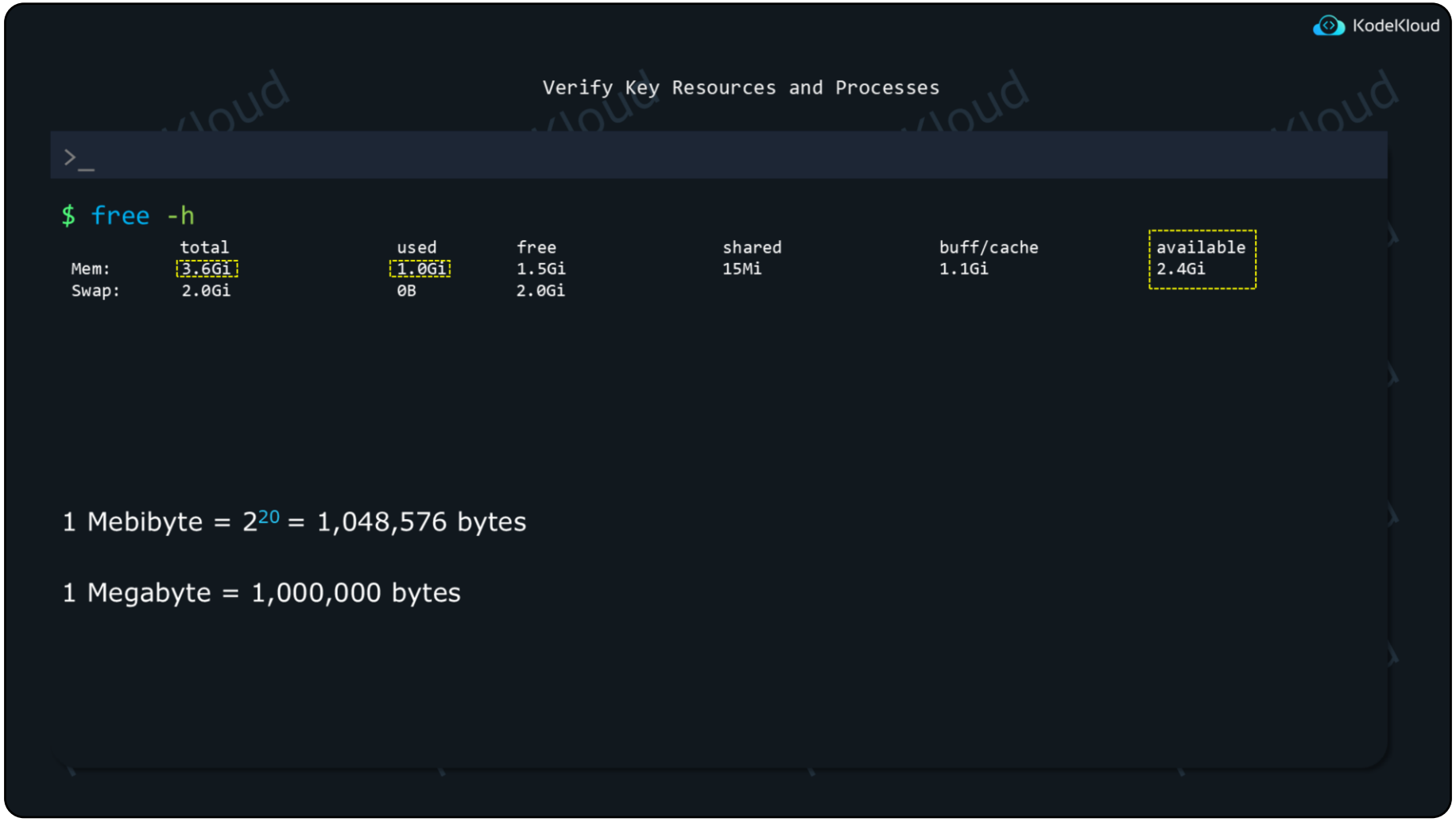
We'll learn what /dev/mapper and /dev/vda2 mean in one of our later lessons where we discuss how to manage storage devices.

We can see that 3.9GB are used on our root filesystem. And we have 5.4GB free. Also, since 43% of our filesystem is used, it means we still have plenty to spare, around 57% of free space.

To see how much disk space a specific directory is using, we can use the du (disk usage) utility:

du -sh /usr/

-s is the summarize option. This makes du display disk space used by the entire /usr/ directory, alone. Without the -s option, du would show us disk space used by that directory and also every other subdirectory it contains.
-h makes it display sizes in human readable format, megabytes, gigabytes, and so on.

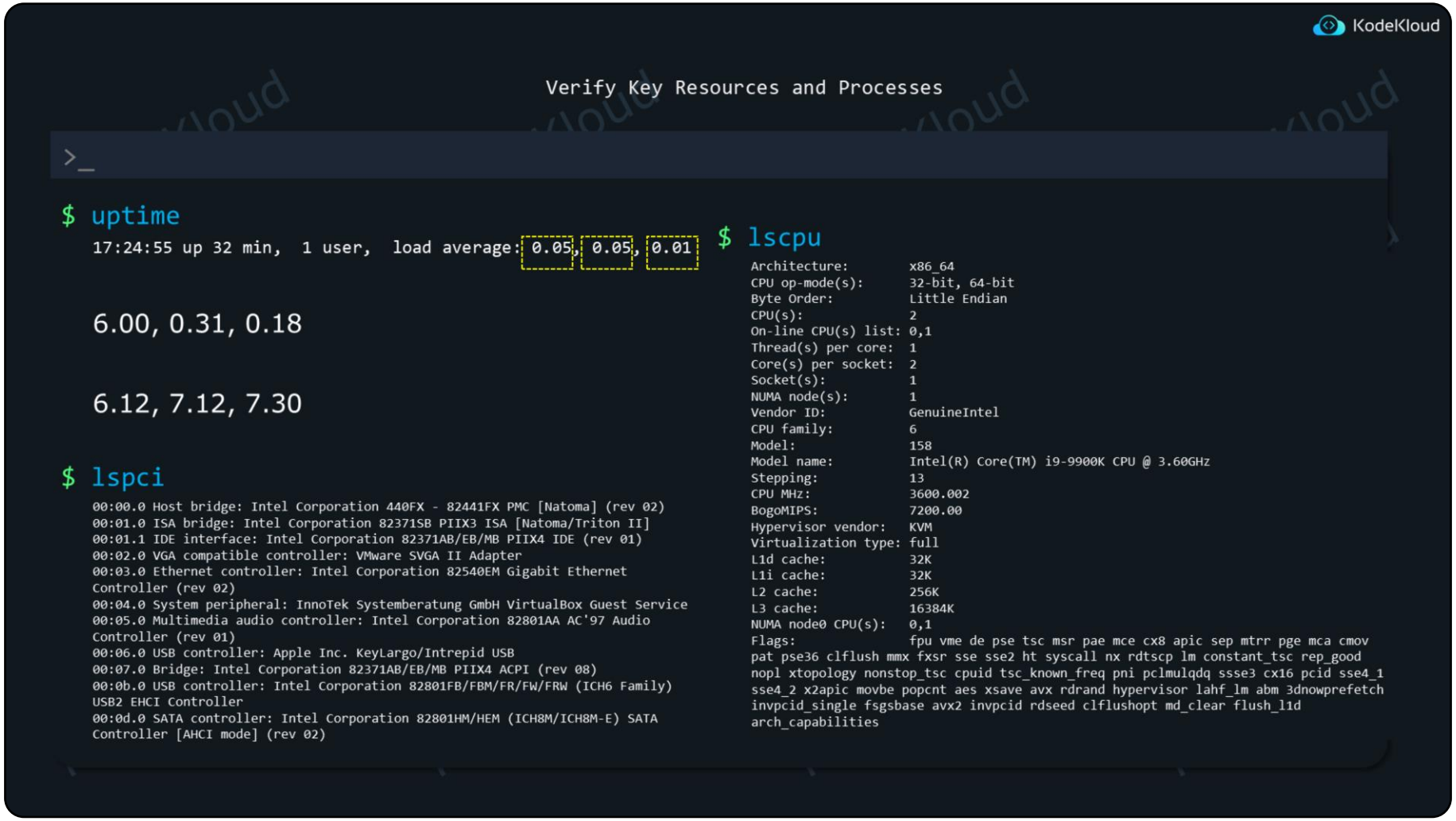


Now let's move on to the server's RAM, Random Access Memory. To see how memory is utilized, we can use the free command:

free -h

Once again, we used the -h option to make the command display sizes in megabytes, gigabytes, and so on. Technically speaking, these are actually Mebibytes and Gibibytes, that's why we see the letters Mi and Gi here. One Mebibyte is equal to 2^20 = 1,048,576 bytes. And one Megabyte is equal to 1,000,000 bytes.

In this output we see that we have around 3.6Gibibytes total usable memory and 1.0Gibibytes are currently used. But how much memory is free, ready to be used by programs? That's the amount we see in the column titled available, not the one named free. Which can look a bit confusing. But the technical explanation is that some memory that is used, hence not actually free, the operating system can make available at any time. For example, if we read a 1 Gigabyte file, the operating system will temporarily store it in memory. For faster access the next time we need it. But if a program needs that 1 Gigabyte of memory, the operating system will free it up immediately. So even though that 1GB was used, and not free, it's still available any time it's needed.



Now let's look at how the processor is used. With the uptime command we can see how the CPU cores were used by programs running on our server.

uptime

The important information is in the three numbers we see after load average:

The first number tells us the load average for the last 1 minute. The second one tells us the load average for the last 5 minutes. Finally, the third number refers to the last 15 minutes.

So, if we see a load average of 1.0 as our first number, it means that one CPU core has been used at full capacity in the last one minute, on average. Otherwise said, 100% of that CPU core was used intensely in the last minute. Since it's an average, this could mean 100% utilization for an entire minute. But it could also mean that for 30 seconds 0% was used, and for the other 30 seconds, 200% was used. That is, two cores. That would still lead to an average of 100% utilization, displayed as 1.0 in this field.

If we see 2.0 as our last number it means that, on average, two CPU cores were fully utilized, at 100%, in the last 15 minutes.

Now let's say we have 8 CPU cores on this system. And we see this load average:

6.00 0.31 0.18

This would mean that 6 CPU cores were used intensely in the last minute. But in the last 5 and 15 minutes, the CPU cores were barely used. So, it's just something that happened recently. Some programs worked hard to do something, for a very brief time. But, overall, the system is not pushing the CPU too much, so we shouldn't be too concerned. But if we would see averages like this:

6.12 7.12 7.30

Then this would tell us that some programs use our CPU cores heavily, almost all the time. It's time to upgrade to a more powerful server or optimize our setup so that it requires fewer CPU resources.

If we need to see some details about the CPU used on this system, we can run:

lscpu

And if we need to see some details about other hardware on this system, we can type:

lspci

Integrity of File Systems

Note: To check file systems for errors, one must unmount it

Redhat OS ----------> xfs file systems (default)

Ubuntu OS ----------> ext4 file systems (default)

Now let's jump to the integrity of filesystems. To check a filesystem for errors, we first must unmount it, in case it is mounted. We'll learn more about filesystems, partitions, mounting and unmounting in the Storage section of this course.

Operating systems from the Red Hat family tend to use XFS filesystems by default. And operating systems from the Ubuntu family use the ext4 filesystem. But this can change in the future for both of them.

To verify an XFS filesystem, we would use a command like this:

```
sudo xfs_repair -v /dev/vdb1
```

-v turns on verbose output. Verbose means that more details will be displayed, about what the command is doing. -v is not required, but it's a useful option to throw in there to see what's going on.

/dev/vdb1 points to the partition where this filesystem is stored on. In this case, it points to the first partition on the second virtual storage device. In a real scenario, this could be something entirely different, such as /dev/vda2 or /dev/sdc3, depending on where the filesystem that you want to scan is located. And we'll learn how to decode these device names later on.

To check and repair an ext4 fileystem we can use a command like this:

sudo fsck.ext4 -v -f -p /dev/vdb2

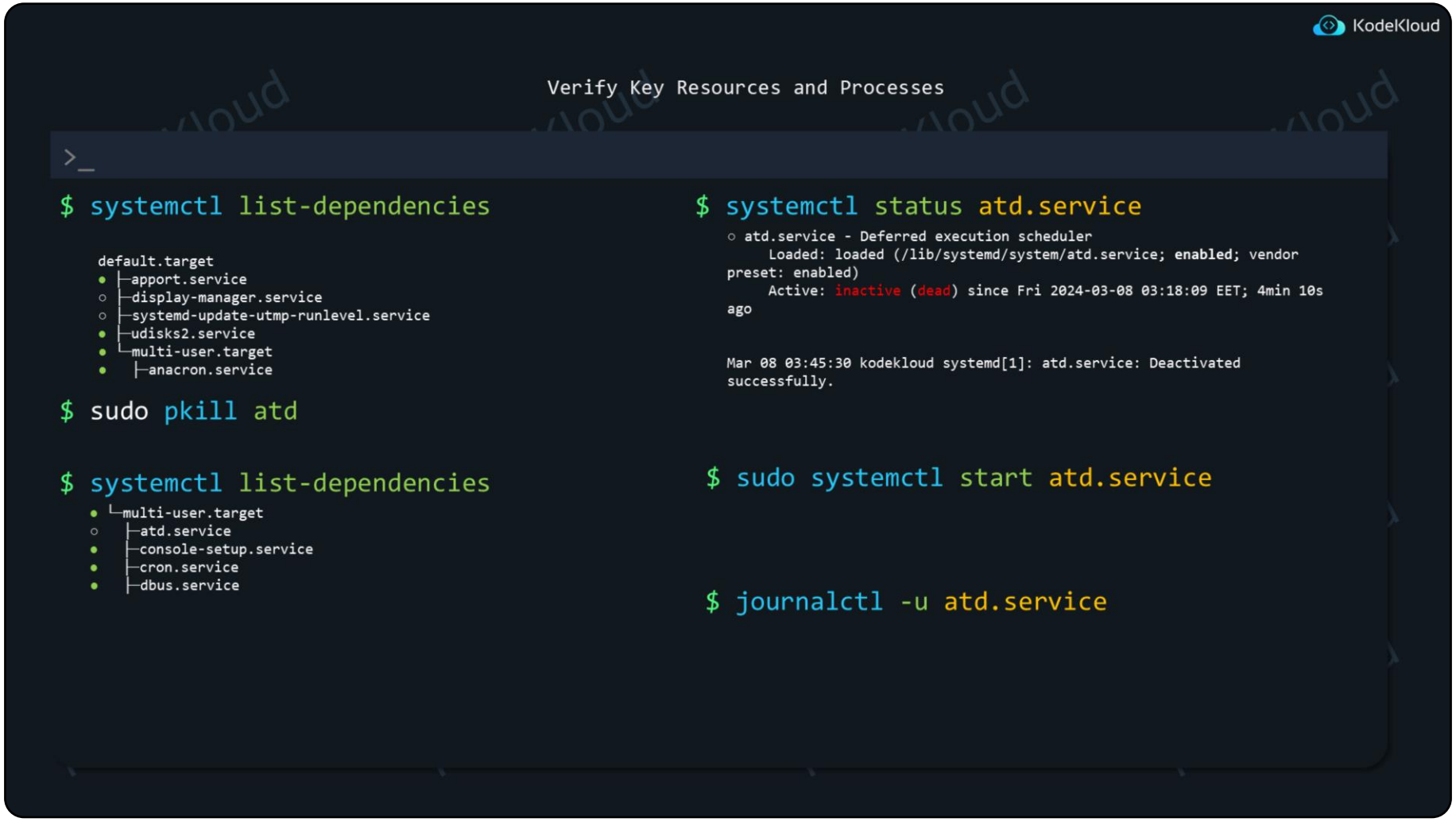
-v turns on verbose output.
-f forces a check even if the filesystem reports that it's healthy.

-p turns on the so-called "preen" mode. This lets the utility fix some simple problems, automatically, without asking us any questions. But if serious errors are encountered, yes/no questions will still be asked.

-v and -f are optional. If you can't remember all these options, at least remember to use the -p option as that is more important if you have a filesystem with many errors. It could help you avoid typing yes to dozens of questions, and instead, they will be fixed automatically.

sudo fsck.ext4 -p /dev/vdb2

You could think about "ext4 filesystem problems" to remember to add the -p option.



Finally, how do we make sure that key processes, important programs, are working correctly on our system? First, we can use this command:

systemctl list-dependencies

This will show us a tree-like structure of important systemd units that are active and inactive.

A transparent, or white circle next to the unit name means that this particular unit is currently inactive. Usually, that's no issue. Some units, or services run once at boot time and then exit. Others are not meant to automatically start at boot time, just when needed.

But there are some exceptions. Some service units are very important and should actually run at all times. An example is the ssh.service unit that controls the lifecycle of the SSH daemon. If this service shows up as inactive, it would mean the SSH daemon is also inactive. Which would mean that administrators wouldn't be able to connect to this operating system remotely to manage it.

A green circle next to a unit name shows us that it's currently active. So we generally want to see this green circle next to all important services meant to run all the time. For example, besides the ssh.service, we also want the cron.service, and atd.service to be active. Otherwise scheduled cron jobs and at jobs wouldn't run automatically.

Let's purposely terminate our "at" daemon to simulate an issue with a service that should generally be active at all times:

sudo pkill atd

Now in this command:

systemctl list-dependencies

we'll see a white light next to the atd.service. Which signals a problem for such an important system service. Scheduled at jobs won't be executed.

To start investigating what went wrong, we can begin with this command:

systemctl status atd.service

This will confirm that the service is currently inactive, the process that should run is dead, or non-existent. And the last log line will show us when the atd.service was deactivated. So no serious errors that we need to fix. And, also, we can see the service is enabled to automatically start each time the system boots. Which means that our job is rather easy. All we need to do is start the service again:

sudo systemctl start atd.service

In a real scenario, things might be slightly more complex. But the log lines in the "systemctl status" command usually point to what went wrong, and why the service cannot start. Which means that first, the problem pointed in the logs should be fixed, and only afterwards we can try to start the service again.

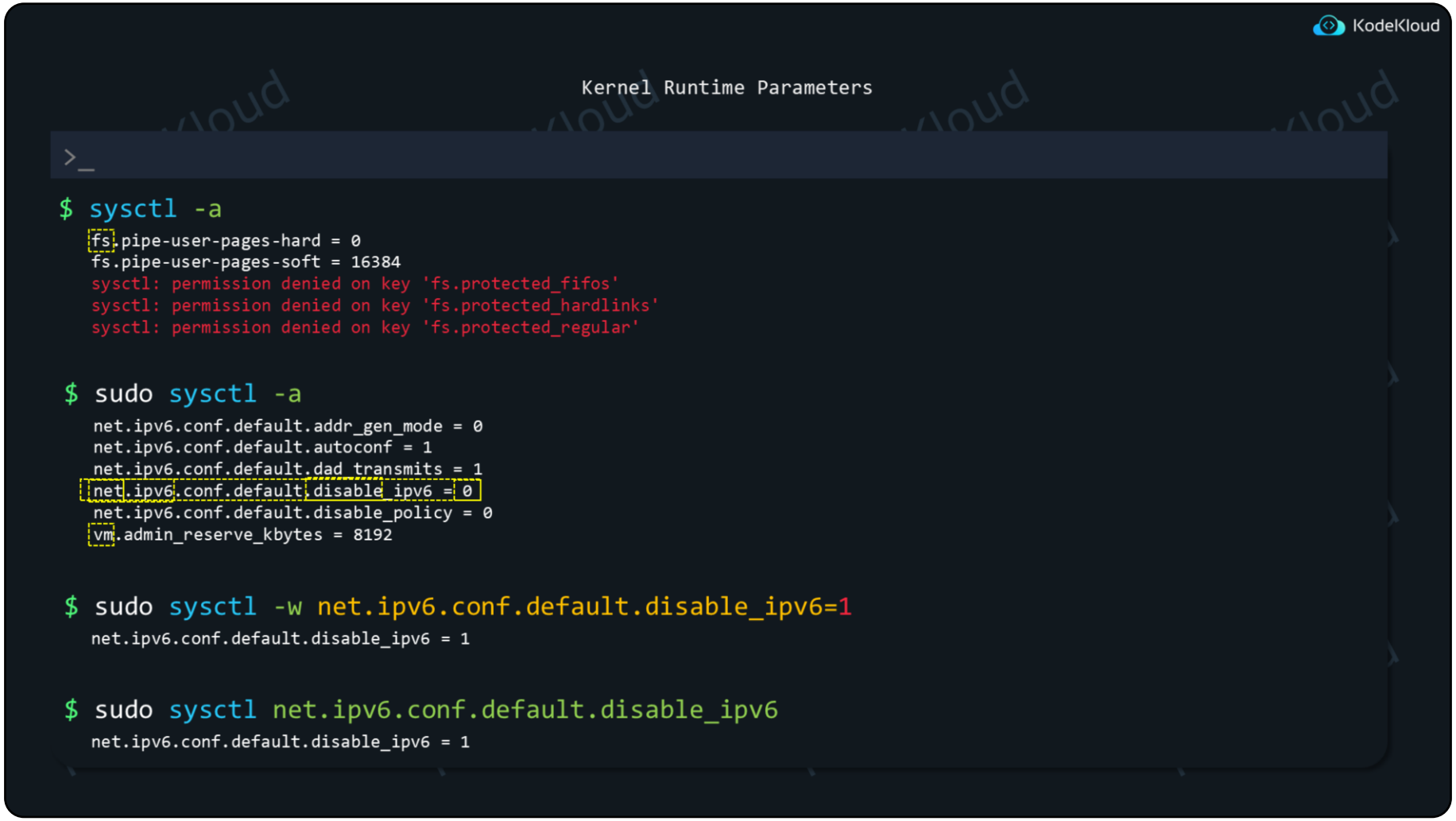
If the logs in the "systemctl" command don't show the real cause of the problem, remember from our previous lesson: We can dive deeper and see all logs generated by this service with a simple journalctl command. For example, to see logs for the atd.service unit, we just run:

journalctl -u atd.service

KodeKloud

Visit www.kodekloud.com to discover more.

In this lesson, we'll explore how to change both persistent, and non-persistent runtime parameters for the Linux kernel.



"Kernel runtime parameters" is just a fancy term for what are basically settings for how the Linux kernel does its job, internally. Since the kernel deals with low-level stuff like allocating memory, handling network traffic, and so on, most of these settings revolve around such kinds of things.

To see all kernel runtime parameters currently in use, enter this command:

sysctl -a

Not all parameters can be read. We'll occasionally see lines like this:

sysctl: permission denied on key 'fs.protected_fifos'

If we want to make sure that all parameters are displayed, we can run the previous command with root privileges:

sudo sysctl -a

Let's take a look at a parameter and its value.

net.ipv6.conf.default.disable_ipv6 = 0

This naming convention makes it quite easy to figure out what the parameter does. Everything that starts with net. is some kind of network-related parameter. Memory-related stuff will start with vm. (virtual memory). Filesystem settings in the Linux kernel start with fs.

So we can figure out that our parameter is related to IPv6 (Internet Protocol version 6) and it disables something. With a value of zero, it means that this does NOT disable anything. 0 basically means "false" in this case, while 1 means "true". To make it disable this, we would need to set it to 1. And to set the value of this paremeter, we can use this command:

sudo sysctl -w net.ipv6.conf.default.disable_ipv6=1

-w is the option to write a value to a parameter. Make sure there is no space before or after the = sign.

But this is a non-persistent change to the parameter. It will be active from now on, if our operating system continues to run. But as soon as we reboot it, net.ipv6.conf.default.disable_ipv6 will be set to zero once again, its default value. So a non-persistent change like we did here does not survive across reboots. We'll soon learn how to make this change persistent.

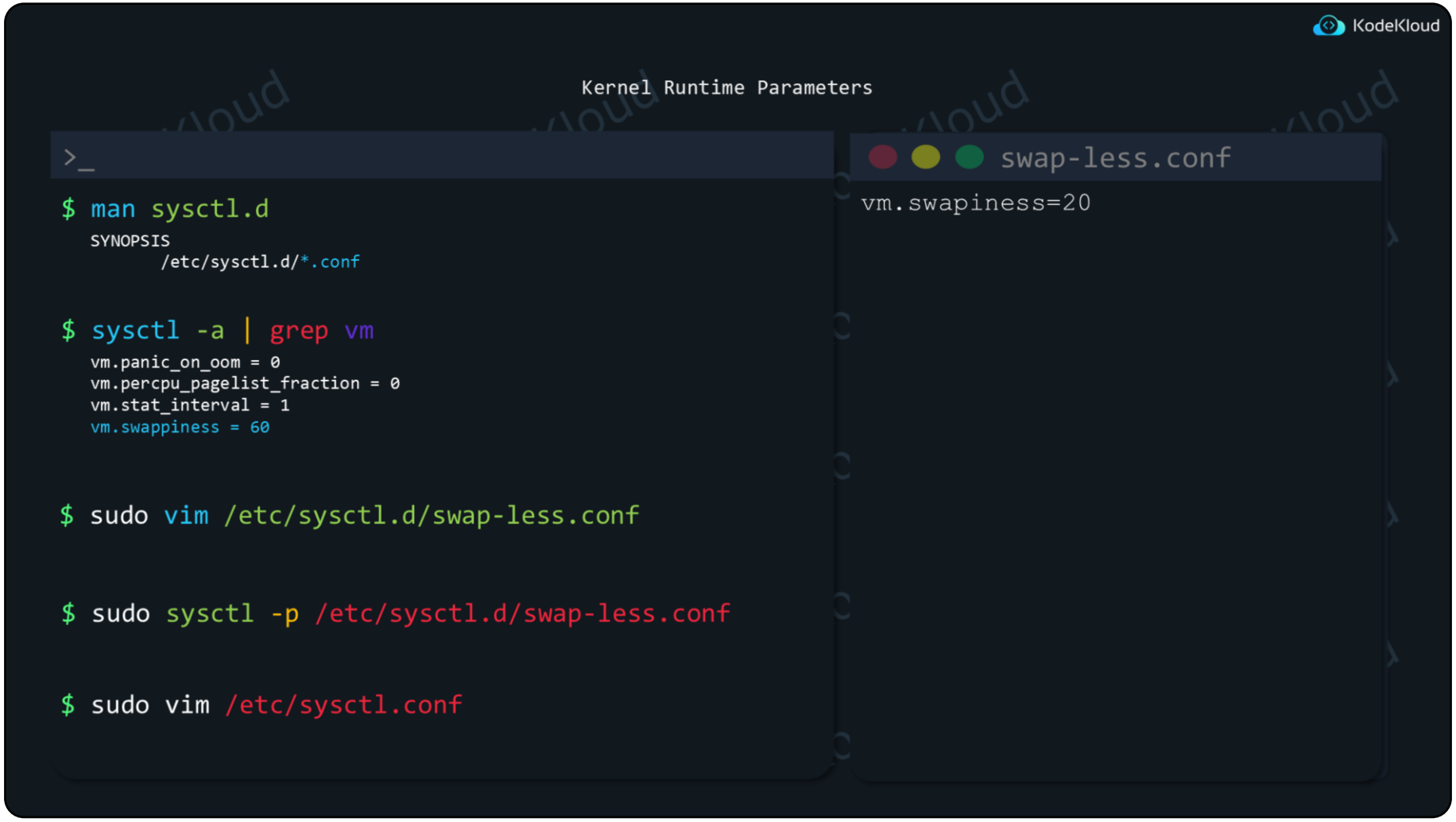
To check the current value for a parameter, we just type "sysctl" followed by the exact name of that parameter:

```
sysctl net.ipv6.conf.default.disable_ipv6
```

If the value cannot be read, just add sudo to get root privileges:

```
sudo sysctl net.ipv6.conf.default.disable_ipv6
```

So how do we make a change persistent? We can add a file to the /etc/sysctl.d/ directory. But it's important to remember that this file has to end with the .conf extension. It's easy to forget that so to get a refresh on how we use this directory



we can type:

man sysctl.d

and we'll immediately see that the .conf extension is required. We can also run the "ls" command on that directory. And we'll notice a few .conf files already in there. Reminding us about the required extension, and also serving as inspiration on how to use these files. Since we can look at their contents and see how they're built.

Now let's say we want to look only at memory-related parameters. We mentioned they start with vm. So, we could use the "sysctl -a" command to display all parameters, and then filter with grep to only show stuff that contains the text "vm":

```
sysctl -a | grep vm
```

We'll see this:

```
vm.swappiness = 60
```

Users often configure this parameter when they want to change how Linux behaves when it starts to run out of free memory. A technique called "swapping" can move data from memory to a storage device like an SSD. This can temporarily free up some memory so it can be used by other programs.

The swappiness value can be anything between 0 and 100 on older kernels. And between 0 and 200 on newer kernels. With a higher value we basically tell the Linux kernel: "Hey, swap whenever you want to, at even the slightest sign you're running low on memory.". And with a lower value we tell it: "Hey, avoid swapping at all costs. Only do it when absolutely necessary."

Now let's say we want to change this value to 20 and make the change persistent. First, we should create a .conf file in /etc/sysctl.d/. We can give the file any name we want. We'll call it "swap-less" to indicate to other possible administrators of this system that we made this .conf file to make our system swap less often.

So we'll run a command like:

```
sudo vim /etc/sysctl.d/swap-less.conf
```

And in this file we simply add this content:

```
vm.swappiness=20
```

Then save our file.

Now, every time our Linux operating system boots, this parameter will be set to the value of 20. The parameter value is now persistent.

But there's a catch. Until the next boot, swappiness will still be using the old value of 60. If we want to make Linux immediately adjust parameters, in the way we defined them in our file, we should also run this command:

sudo sysctl -p /etc/sysctl.d/swap-less.conf

The man page for sysctl does not really say why this option is called "-p" but you can think of it as reading settings from permanent file, to remember this letter.

It's worth noting that another place where we can edit kernel parameters is in the /etc/sysctl.conf file. So we can run a command like:

sudo vim /etc/sysctl.conf

to edit that directly. What's interesting about this file is that it includes explanations about important, and often-changed parameters. So it can give us a bit of extra context on what some parameters do, before we actually change them. Plus, it conveniently places those parameters in that file, and they're easy to turn on by just uncommenting their respective lines.

But keep in mind that that file can be accidentally overwritten when upgrading the operating system. So if you intend to make persistent changes of your own, place them in the /etc/sysctl.d/ directory instead.

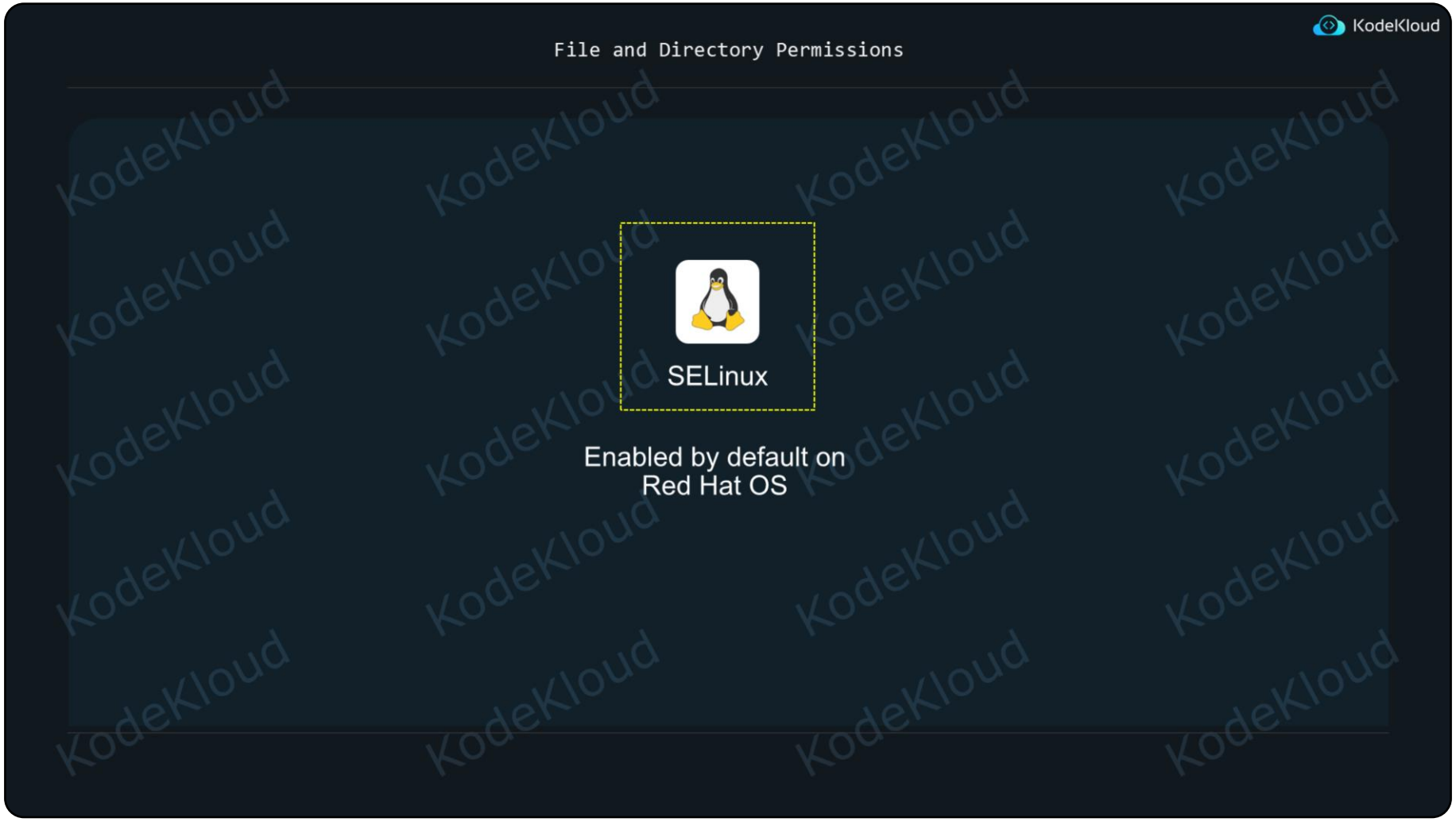
KodeKloud

Visit www.kodekloud.com to discover more.

In this lesson, we'll explore SELinux contexts applied to files, and processes.

We saw that Linux supports some rudimentary forms of access control. Files and directories have permissions that establish who can read, write, or execute content. Also, regular users are not allowed to do certain things that the root user can do. But these forms of security are not enough for today's world of sophisticated cyber attacks. They're just too generic, not fine-tuned enough. And a simple read, or write permission gives a user, or process, too much power over a particular area of the system.

Fortunately, the Linux kernel can be easily extended with so-called modules. And there is a security module called SELinux that adds very advanced capabilities of restricting access.



SELinux is pre-configured and enabled by default on operating systems from the Red Hat family. But it's not enabled by default on Ubuntu. And the initial configuration on Ubuntu can be a bit tricky. But we'll take a deeper look at SELinux and Ubuntu in one of our next lessons.

For now, let's see how this Linux security module works at a surface level.

SELinux allows for very fine-grained control of what actions should be allowed or denied. But how does it make such decisions?

We saw how a command like

ls -l

shows us what permissions are enabled/disabled for a file or directory.

In output like this

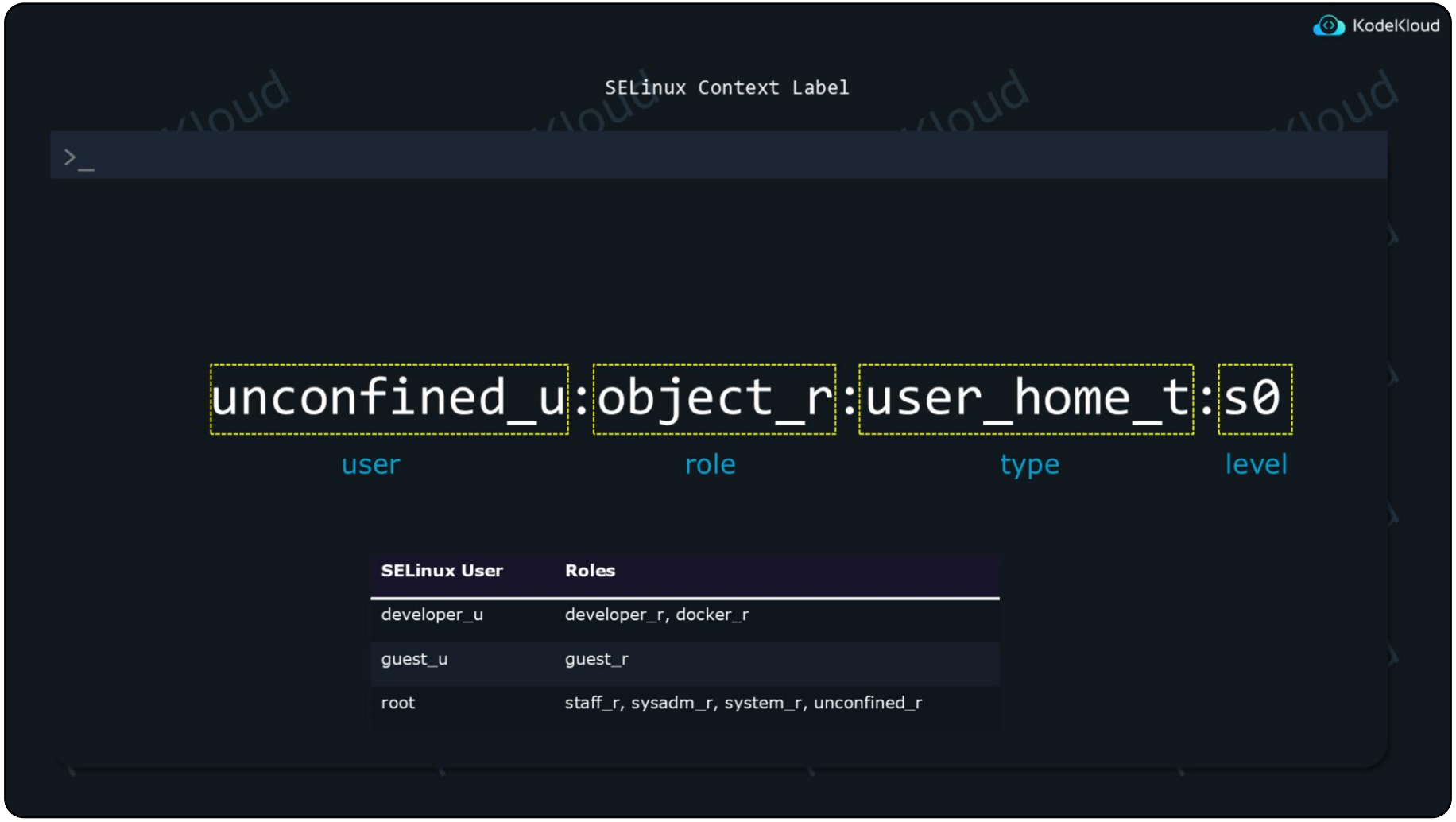
we can tell from the string rw-rw-r-- how permissions are set for this file. SELinux employs a similar mechanism.

Just like we can see permissions for files and directories with the -l option, we can use a different option to see SELinux file/directory labels. We do that by passing the -Z option to our ls command.

ls -Z

We'll see output like this:

unconfined_u:object_r:user_home_t:s0 archive.tar.gz



This part unconfined_u:object_r:user_home_t:s0 is called the SELinux context (also called simply label sometimes). Because at a filesystem level this is simply a label applied to the file. Just some additional metadata glued on top of it. And by looking at the label, SELinux can decide what the security context is for this file, and what kind of actions it should allow.

Although these vaguely resemble file/directory permissions, they're much more complex. So, let's take a quick look at what this context tells us.

First, these context components are always displayed in this order:

user:role:type:level

So, in our case, unconfined_u is the user, object_r is the role, user_home_t is the type and s0 is the level.

To allow or deny some action, SELinux roughly follows this logic:

First, it looks at the user part of the label. This is not the same as the username you log in with. It's an SELinux user. Every user who logs in to a Linux system is mapped to an SELinux user as part of the SELinux policy configuration.

After the user is identified, a decision is made to see if it can assume the role. Each user can only assume a predefined set of roles. For example, we could imagine how a developer_u user should only be able to enter roles like developer_r or docker_r that allow them to read and write application code, launch Docker containers, and so on. But they should not be able to enter roles like sysadm_r that let them change system settings.

If the role can be entered, next, a check is performed to see if this role can transition to the type. The name of this field comes from "type enforcement". This "type" is like a protective software jail. Once something enters the type enforcement defined here, it can only do certain things and nothing else. This restriction bubble is called "type" when dealing with files and "domain" when dealing with processes. We'll soon see how processes also have SELinux contexts applied to them, just like files and directories.

So, we notice how we basically go from user, to role, to type, to enter into some kind of restricted, secure box. Going through this three-step path, SELinux achieves three important things:

It makes sure that only certain users can enter certain roles and certain types. So it keeps unauthorized users and processes from entering types/domains that are not for them.

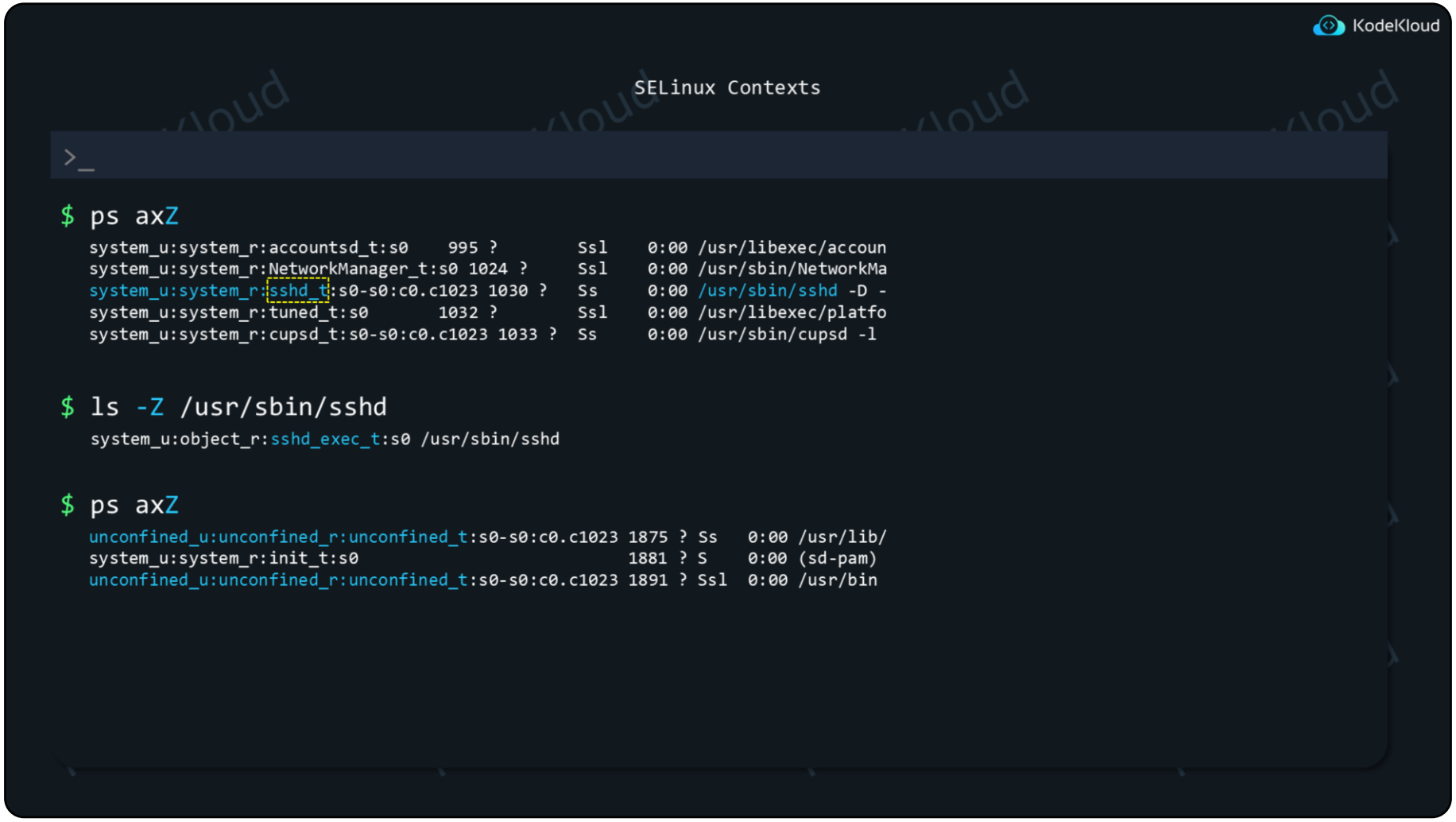
It lets authorized users and processes do their job, by granting the permissions they need.

However, at the same time, even authorized users and processes are allowed to take only a limited set of actions. Everything else is denied.

Why is this important? Here's just one example: it can protect against a hijacked program. Imagine a hacker takes control of the Apache program that serves web pages to our website visitors. Even though the malicious user now controls the program, he, or she is still restricted by SELinux. The hijacked Apache application is confined to a certain security domain applied to that process. So it can only take actions allowed for that security domain. Hence the hacker isn't able to "step out" of those boundaries. The attacker is effectively in a virtual jail and the damage that can be done is greatly reduced.

Usually, the most important part of a SELinux context is the type/domain, the third part in our label here.

The fourth part, the level, is almost never actively used on regular systems. That's a better fit for complex organizations that require multiple levels of security. An example of this: Imagine Jane is a high-ranking military officer. She has a high clearance level and will be able to read documents classified from level 0 to level 4, level 4 being top secret files. On the other hand, a user with level 1 access can only access stuff on level 1 and level 0, but access to anything above level 1 is restricted.



And since we mentioned processes also get contexts, just like files, how do we explore what SELinux contexts are applied to them? With the same Z option. To see all processes, we use the ps ax command, and to add security labels to the output, we add the Z option, so we get:

ps axZ

We can see our SSH daemon running here:

system_u:system_r:sshd_t

and spot that it's restricted to the sshd_t type enforcement domain. This domain contains a strict set of policies that define what this process is allowed to do. Also, important to note, not anything can enter the sshd_t domain. In the default system-wide SELinux policy, only files that have the sshd_exec_t SELinux type in their label can enter this domain. And sure enough, if we check the file containing the sshd program:

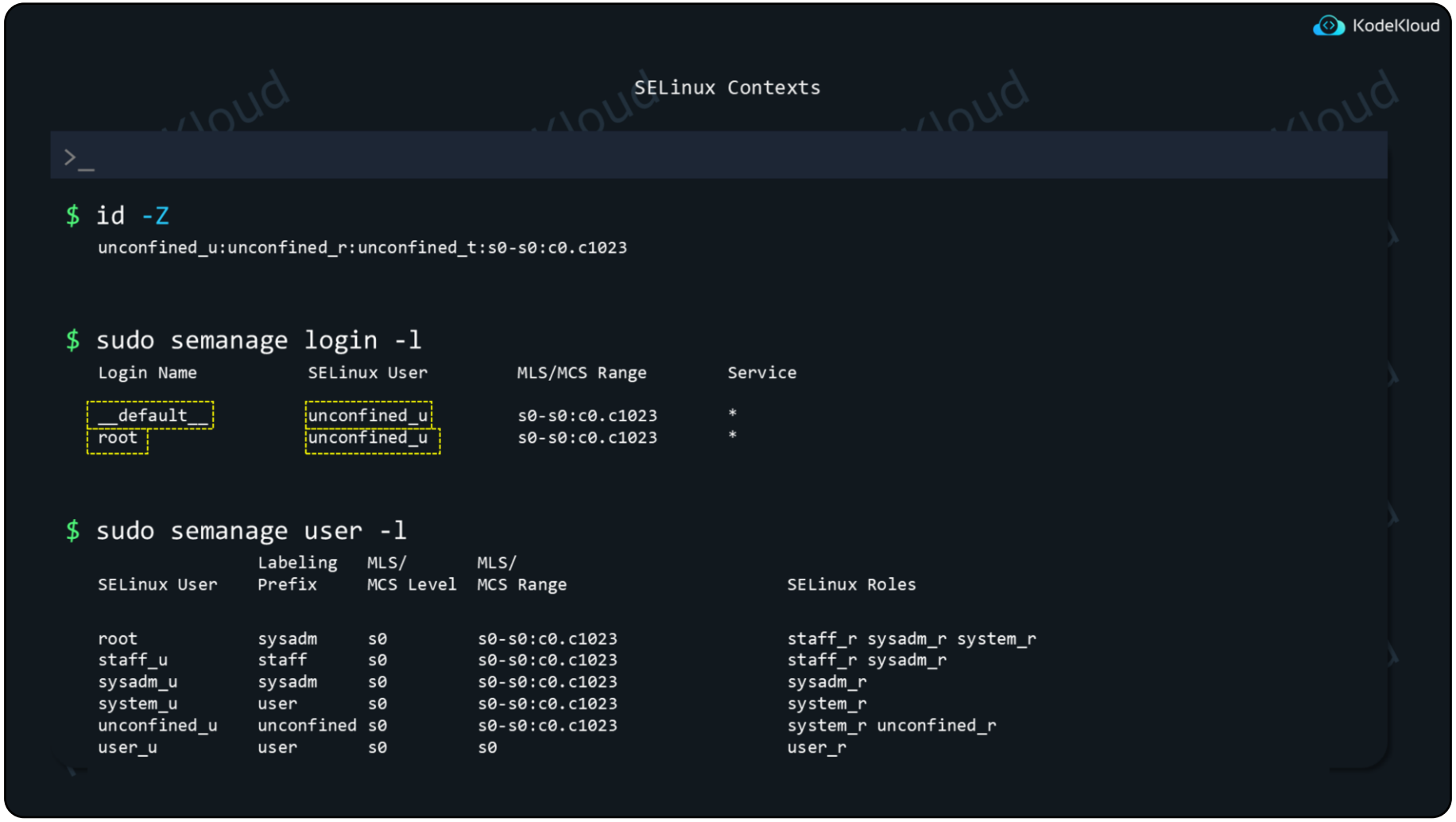
ls -Z /usr/sbin/sshd

system_u:object_r:sshd_exec_t:s0 /usr/sbin/sshd

we'll see it's indeed labeled with this type.

To recap: Only a file marked with a certain type can start a process that can shift into a certain domain. The type establishes permitted actions for a file. The domain establishes permitted actions for a process.

Back to the output of our "ps axZ" command. Anything labeled with unconfined_t is running unrestricted. SELinux lets those processes do almost anything they want.



Here are some other useful commands we can use.

SELinux puts a security bubble around everything. Even a user, not just a file, or process.

To see the security context assigned to our current user (what got applied to our user when we logged in), we can run:

id -Z

When we log in, the user we log in as is automatically mapped to a SELinux user. To see how this mapping is done, we can use this command:

sudo semanage login -l

Login Name SELinux User MLS/MCS Range Service

__default__ unconfined_u s0-s0:c0.c1023 *
root unconfined_u s0-s0:c0.c1023 *

Output like this means that when the root user logs in, they'll be mapped to the SELinux user called "unconfined_u". If any other user logs in, not defined in this list, they'll be mapped to whatever we see on the "__default__" line. In this case, also unconfined_u.

We can see that, by default, SELinux doesn't initially restrict the user in any way, since they are transitioned to this unconfined user. On most systems it's usually the role and type components of a context that start to restrict access. But, for advanced use-cases, things can be set up so that restrictions begin at the user level.

To see the roles that each user can assume on this system, we can type:

sudo semanage user -l

Finally, to see if SELinux is enabled and actively restricting actions (enforcing security), we can use this command:

getenforce

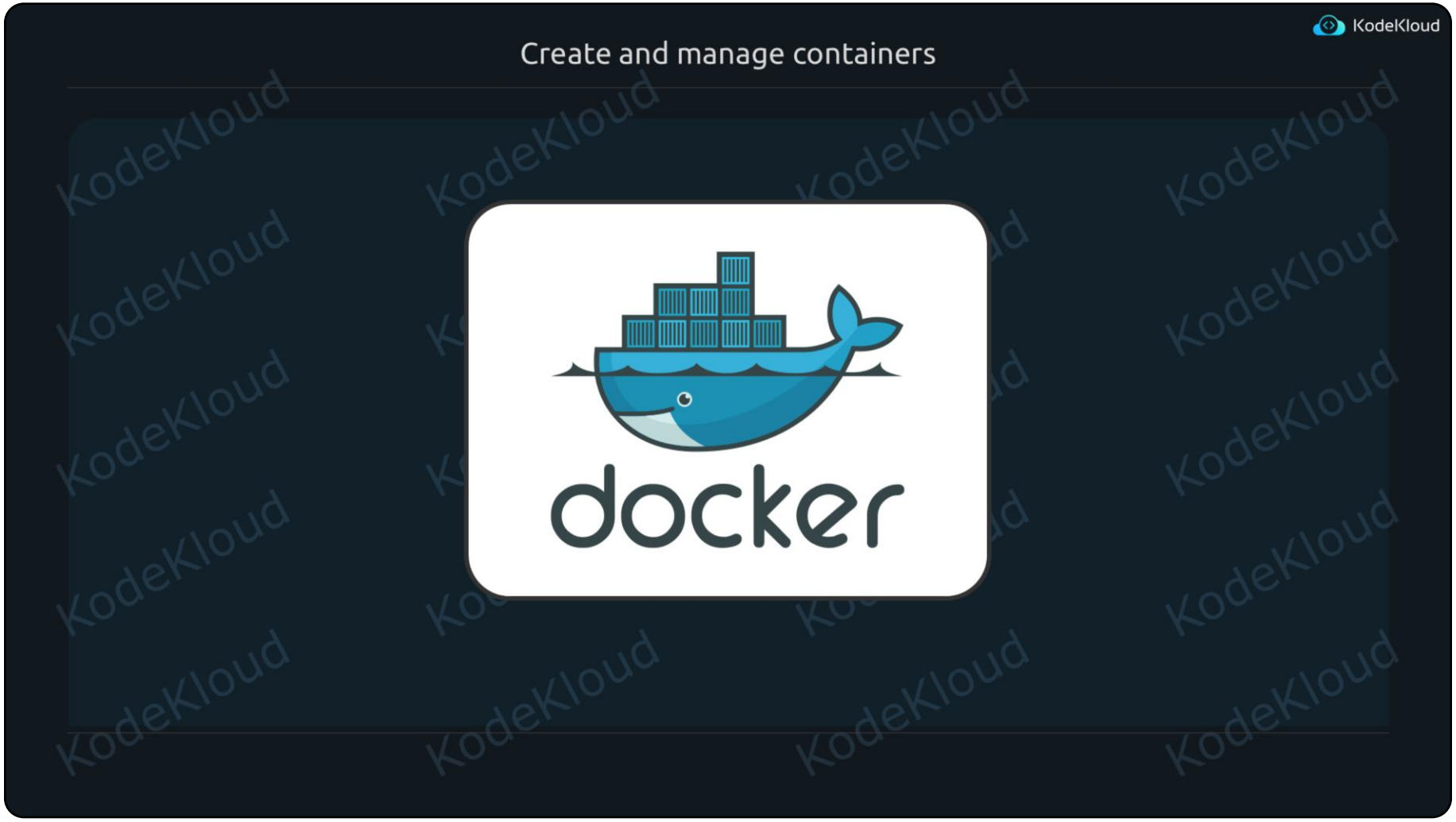
Only if we see "Enforcing" here it means it's doing its job and denying unauthorized actions. "Permissive" means it's allowing everything. It's just logging actions that should have been restricted, but it's not denying anything. Finally, if we see "Disabled" here, it means that SELinux is not doing anything. It isn't enforcing security, and it isn't even logging.

KodeKloud

Visit www.kodekloud.com to discover more.

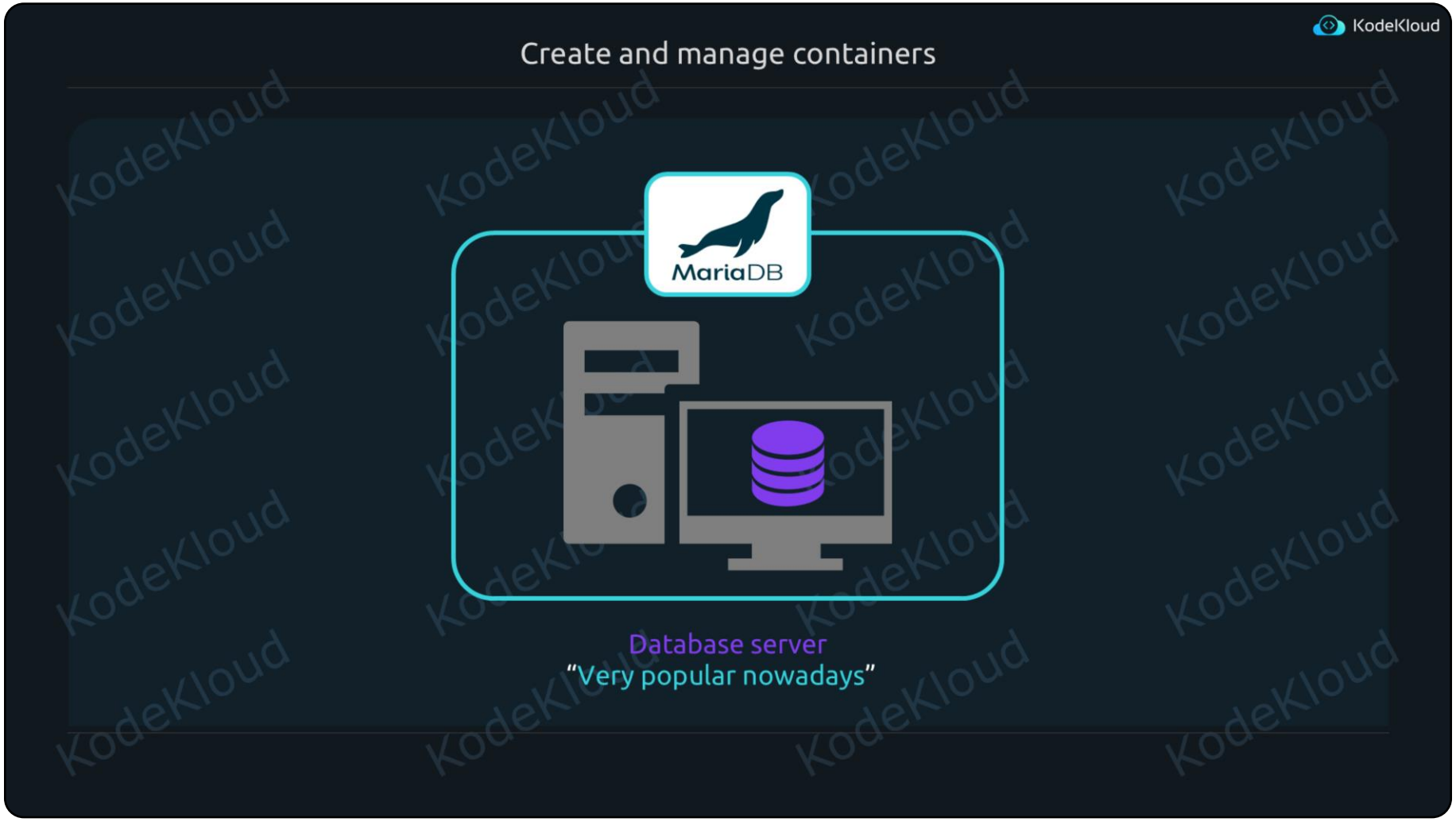
Docker containers are very popular nowadays, but why? To get a glimpse of their usefulness just think about a database server we can create with MariaDB, the classic way.

We install it with a "sudo apt install mariadb-server" command. Then we configure its settings by modifying files in the "/etc/mysql/" directory. We create a few databases, we configure database users, assign privileges, and so on.



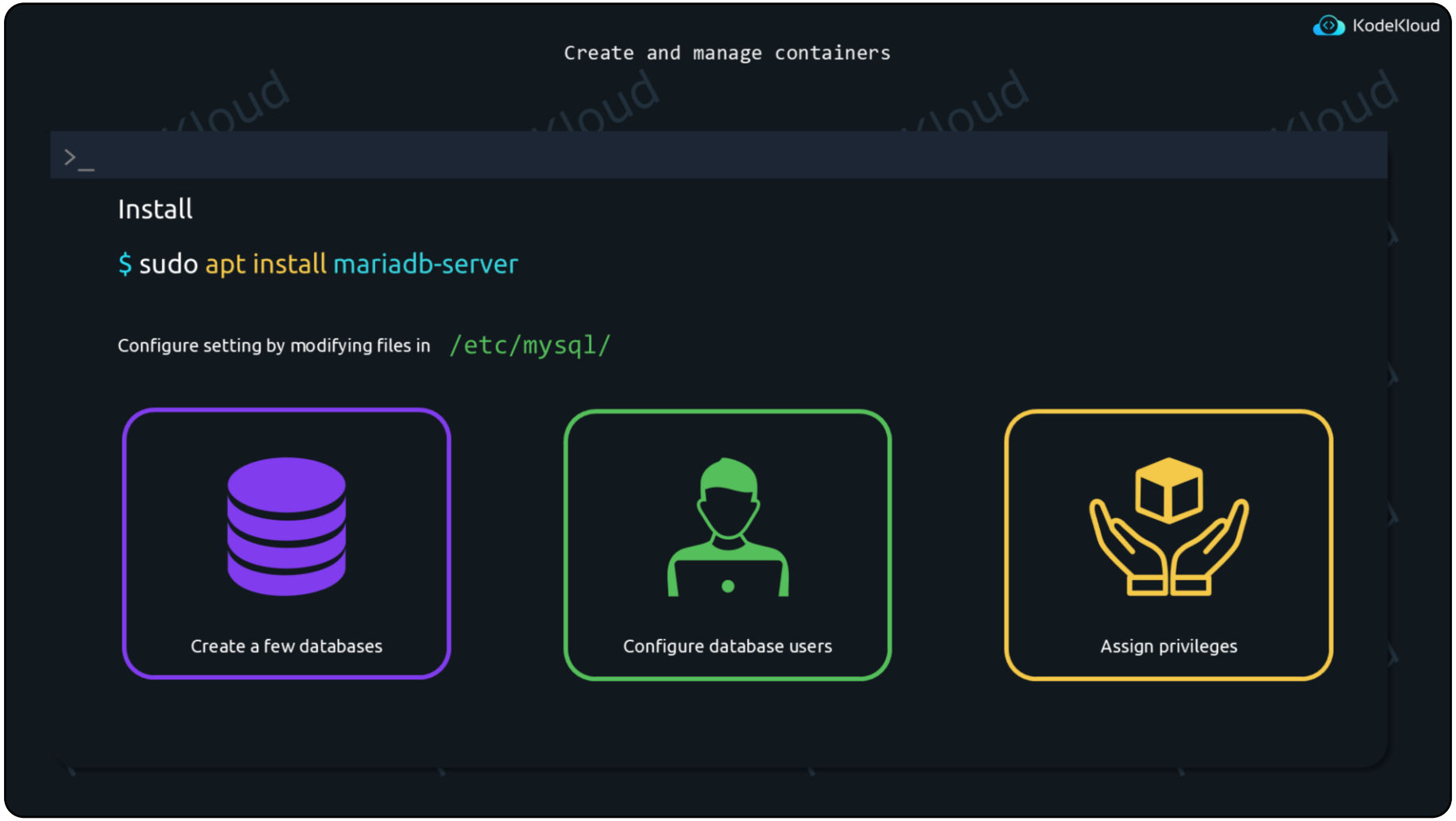
Docker containers are very popular nowadays, but why? To get a glimpse of their usefulness just think about a database server we can create with MariaDB, the classic way.

We install it with a "sudo apt install mariadb-server" command. Then we configure its settings by modifying files in the "/etc/mysql/" directory. We create a few databases, we configure database users, assign privileges, and so on.



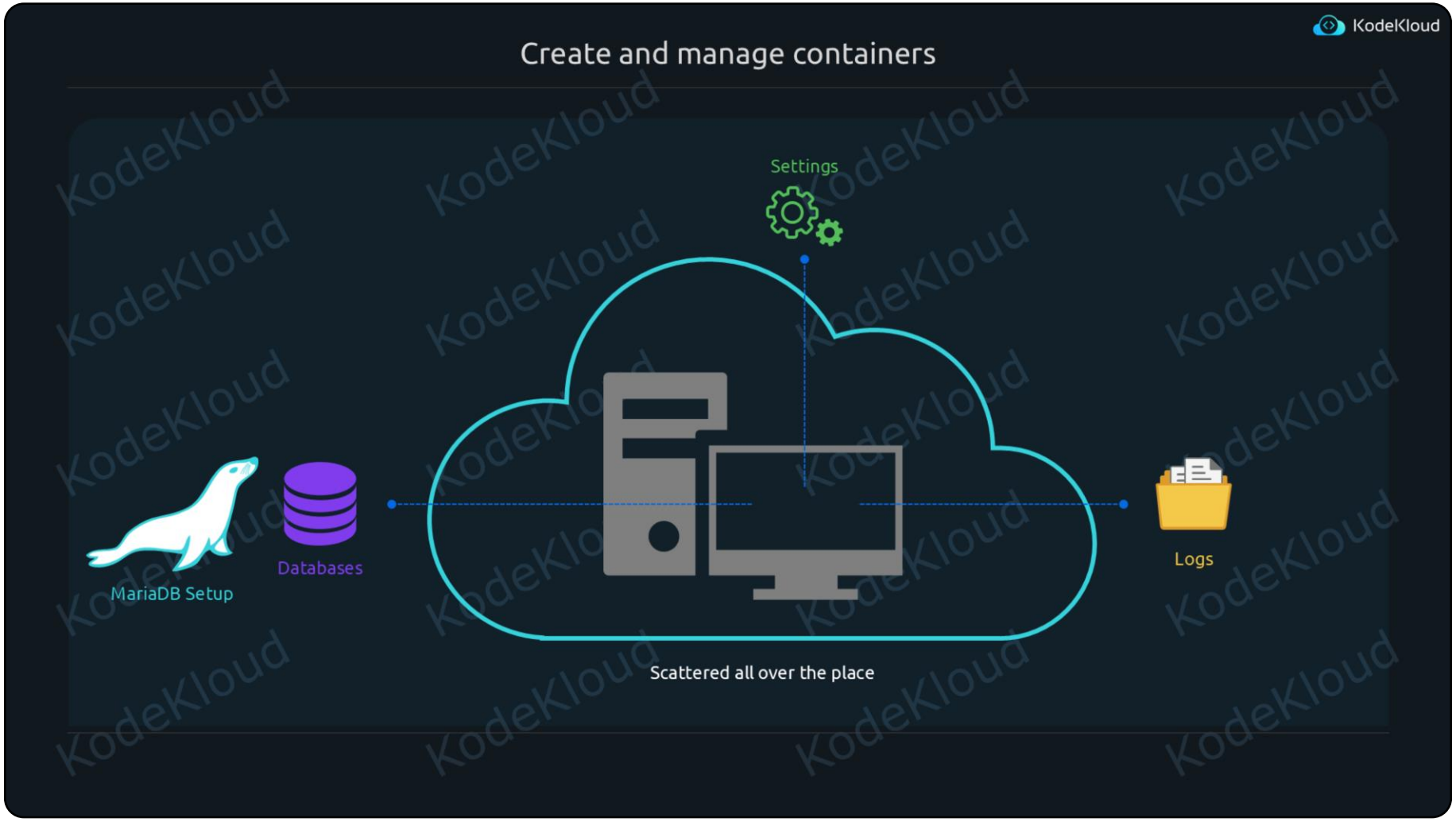
Docker containers are very popular nowadays, but why? To get a glimpse of their usefulness just think about a database server we can create with MariaDB, the classic way.

We install it with a "sudo apt install mariadb-server" command. Then we configure its settings by modifying files in the "/etc/mysql/" directory. We create a few databases, we configure database users, assign privileges, and so on.



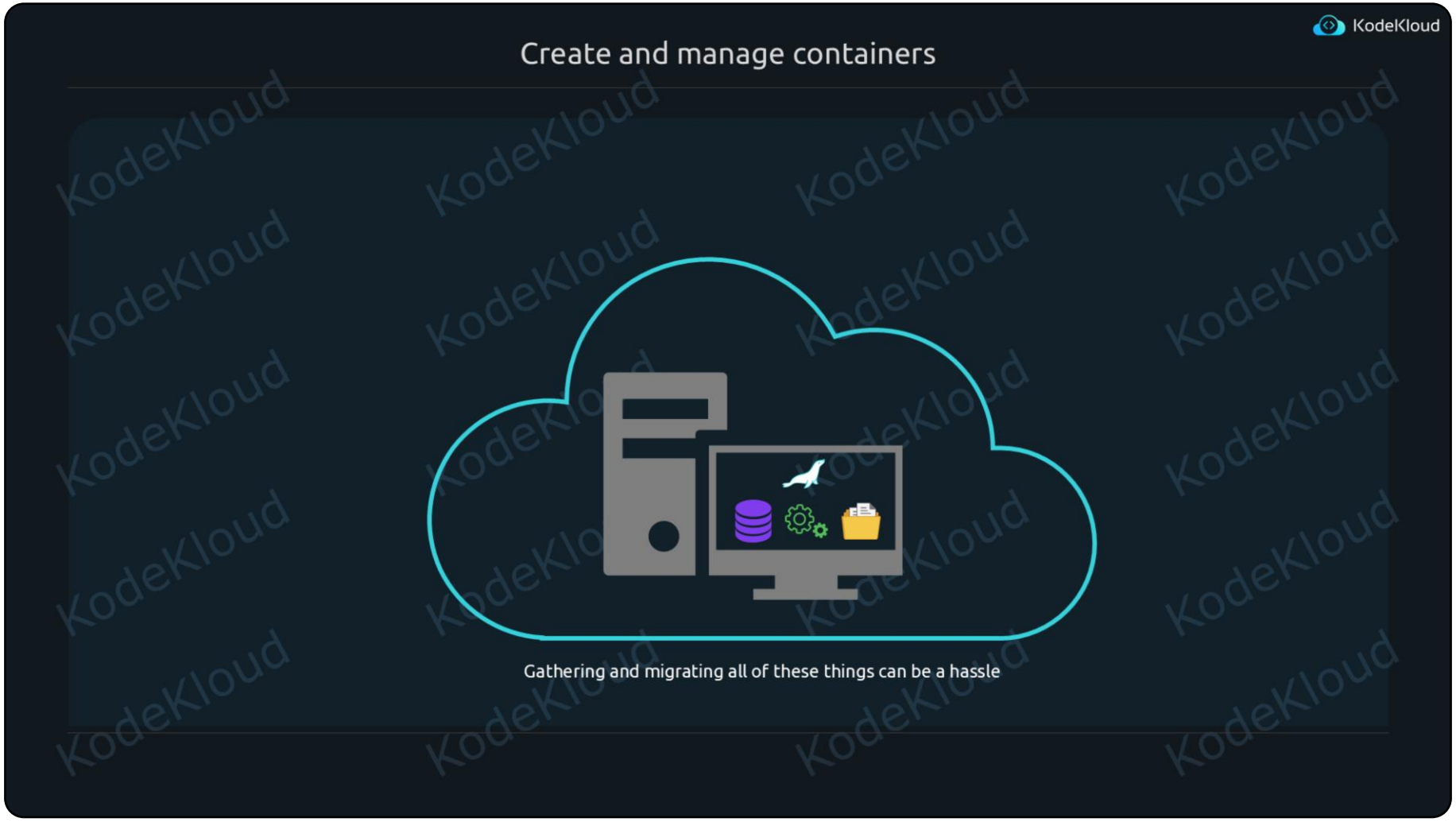
Docker containers are very popular nowadays, but why? To get a glimpse of their usefulness just think about a database server we can create with MariaDB, the classic way.

We install it with a "sudo apt install mariadb-server" command. Then we configure its settings by modifying files in the "/etc/mysql/" directory. We create a few databases, we configure database users, assign privileges, and so on.



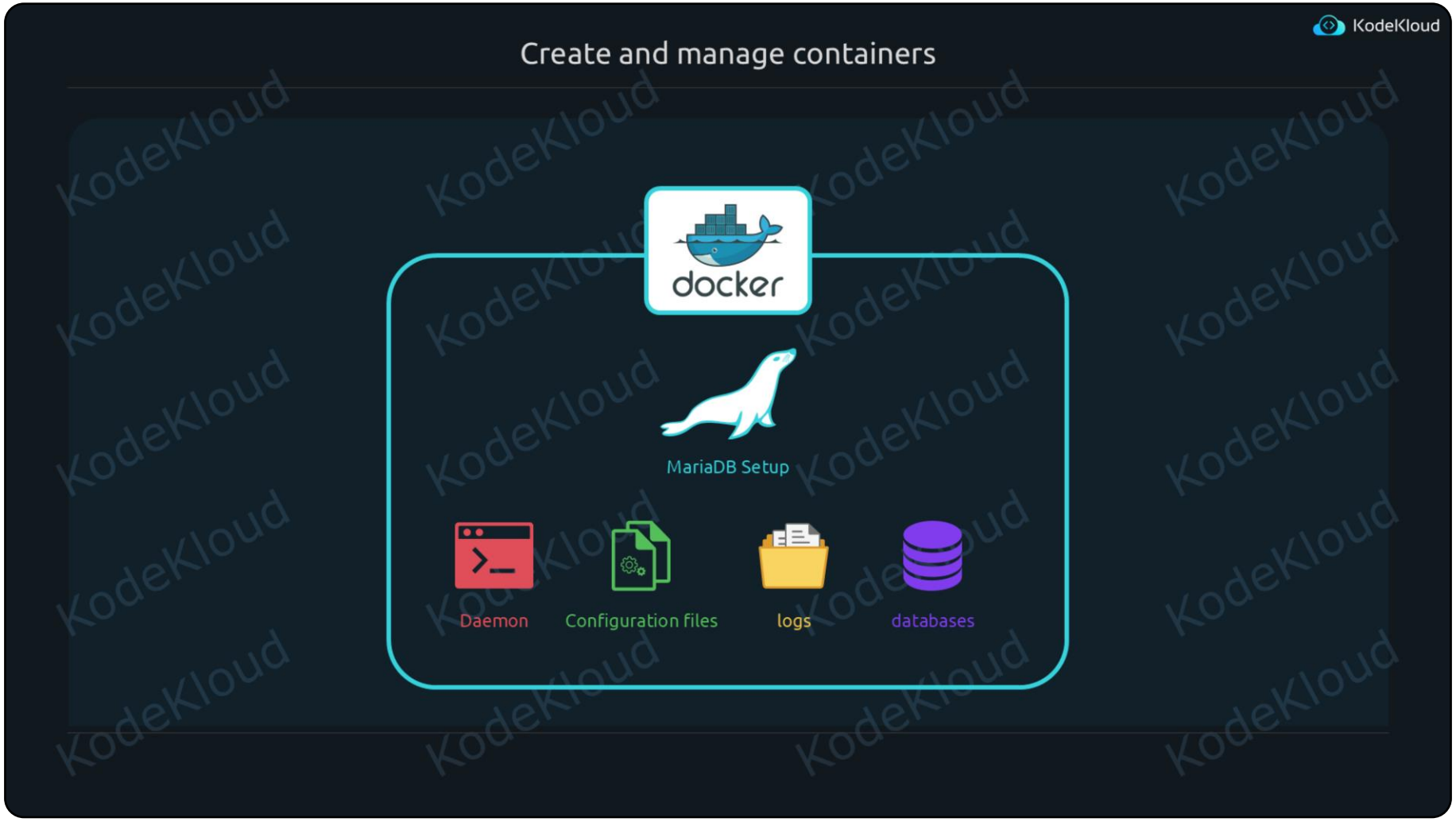
Let's say we're happy with our setup and we want to move it to a more powerful computer, somewhere in the cloud. The problem is that all of its components are scattered all over the place. The settings for MariaDB are in one directory. The databases are in some other directory, logs in another one. Gathering and migrating all of these things can be a hassle.

But what if our MariaDB setup would have been created in a Docker container instead? Now everything would exist inside this container: the daemon, the configuration files, the logs, the databases. All we would need to do is copy this container to some other computer and everything will work exactly the same as it did before.



Let's say we're happy with our setup and we want to move it to a more powerful computer, somewhere in the cloud. The problem is that all of its components are scattered all over the place. The settings for MariaDB are in one directory. The databases are in some other directory, logs in another one. Gathering and migrating all of these things can be a hassle.

But what if our MariaDB setup would have been created in a Docker container instead? Now everything would exist inside this container: the daemon, the configuration files, the logs, the databases. All we would need to do is copy this container to some other computer and everything will work exactly the same as it did before.

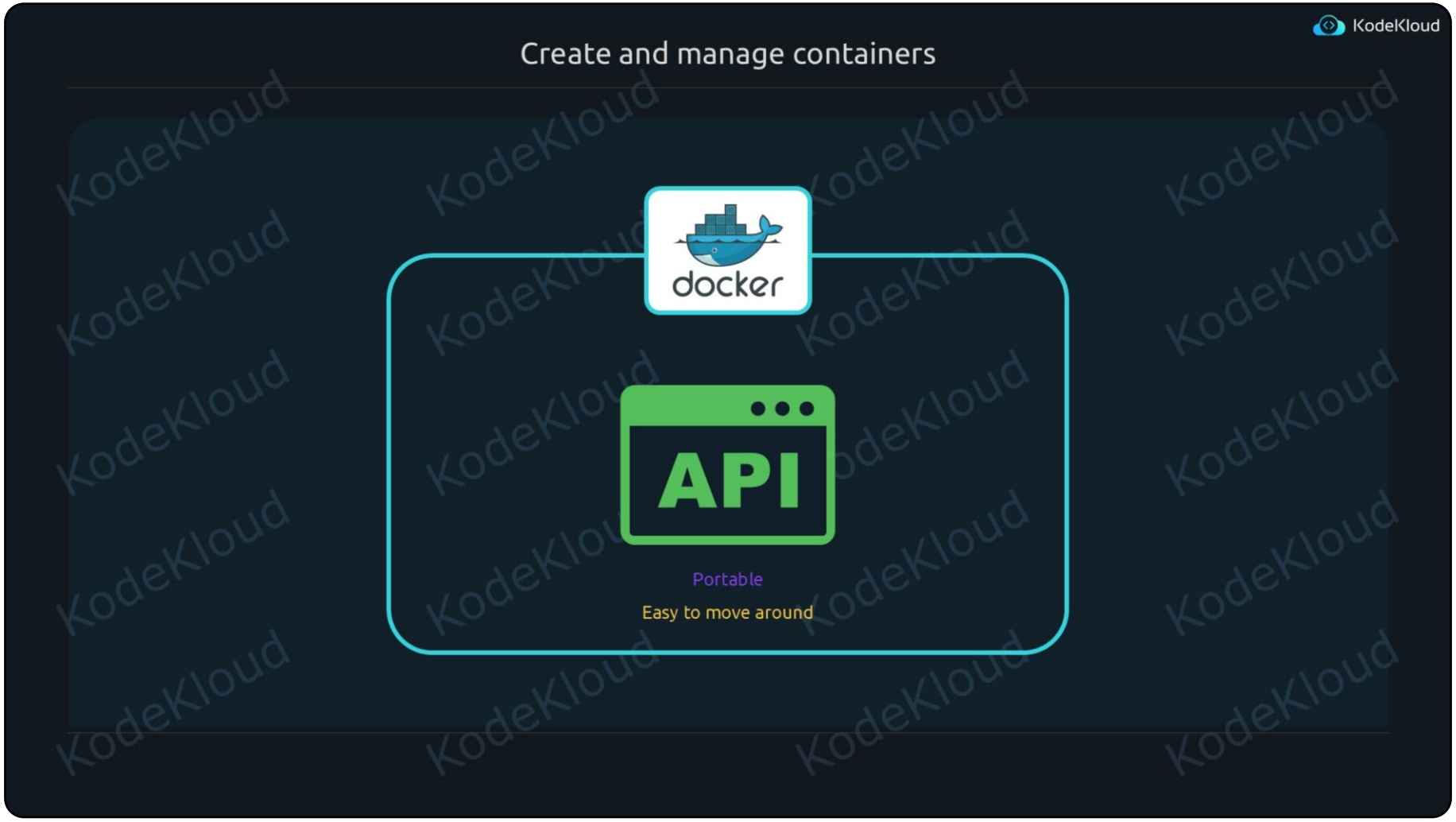


Let's say we're happy with our setup and we want to move it to a more powerful computer, somewhere in the cloud. The problem is that all of its components are scattered all over the place. The settings for MariaDB are in one directory. The databases are in some other directory, logs in another one. Gathering and migrating all of these things can be a hassle.

But what if our MariaDB setup would have been created in a Docker container instead? Now everything would exist inside this container: the daemon, the configuration files, the logs, the databases. All we would need to do is copy this container to some other computer and everything will work exactly the same as it did before.

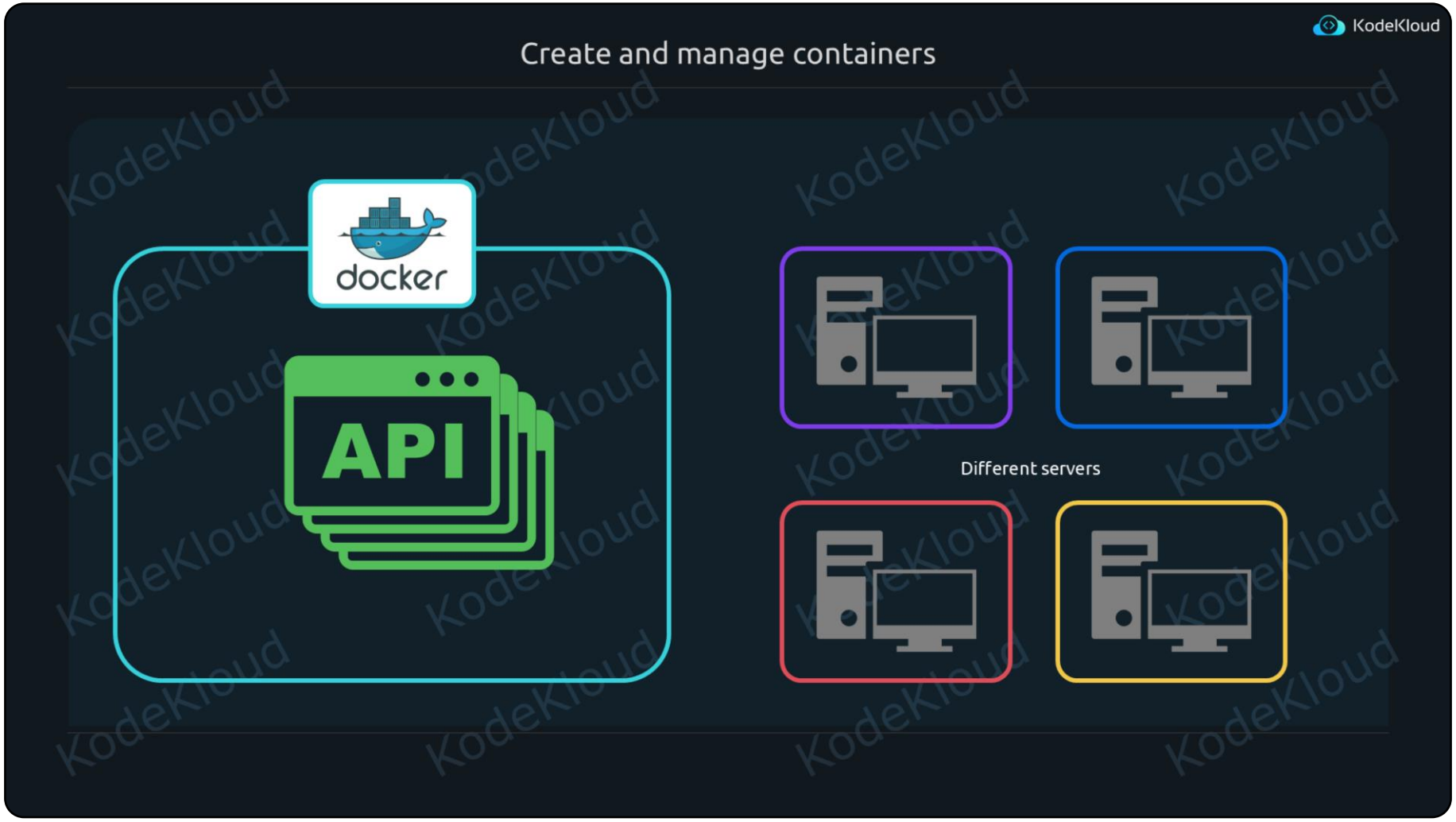
Let's say we're happy with our setup and we want to move it to a more powerful computer, somewhere in the cloud. The problem is that all of its components are scattered all over the place. The settings for MariaDB are in one directory. The databases are in some other directory, logs in another one. Gathering and migrating all of these things can be a hassle.

But what if our MariaDB setup would have been created in a Docker container instead? Now everything would exist inside this container: the daemon, the configuration files, the logs, the databases. All we would need to do is copy this container to some other computer and everything will work exactly the same as it did before.



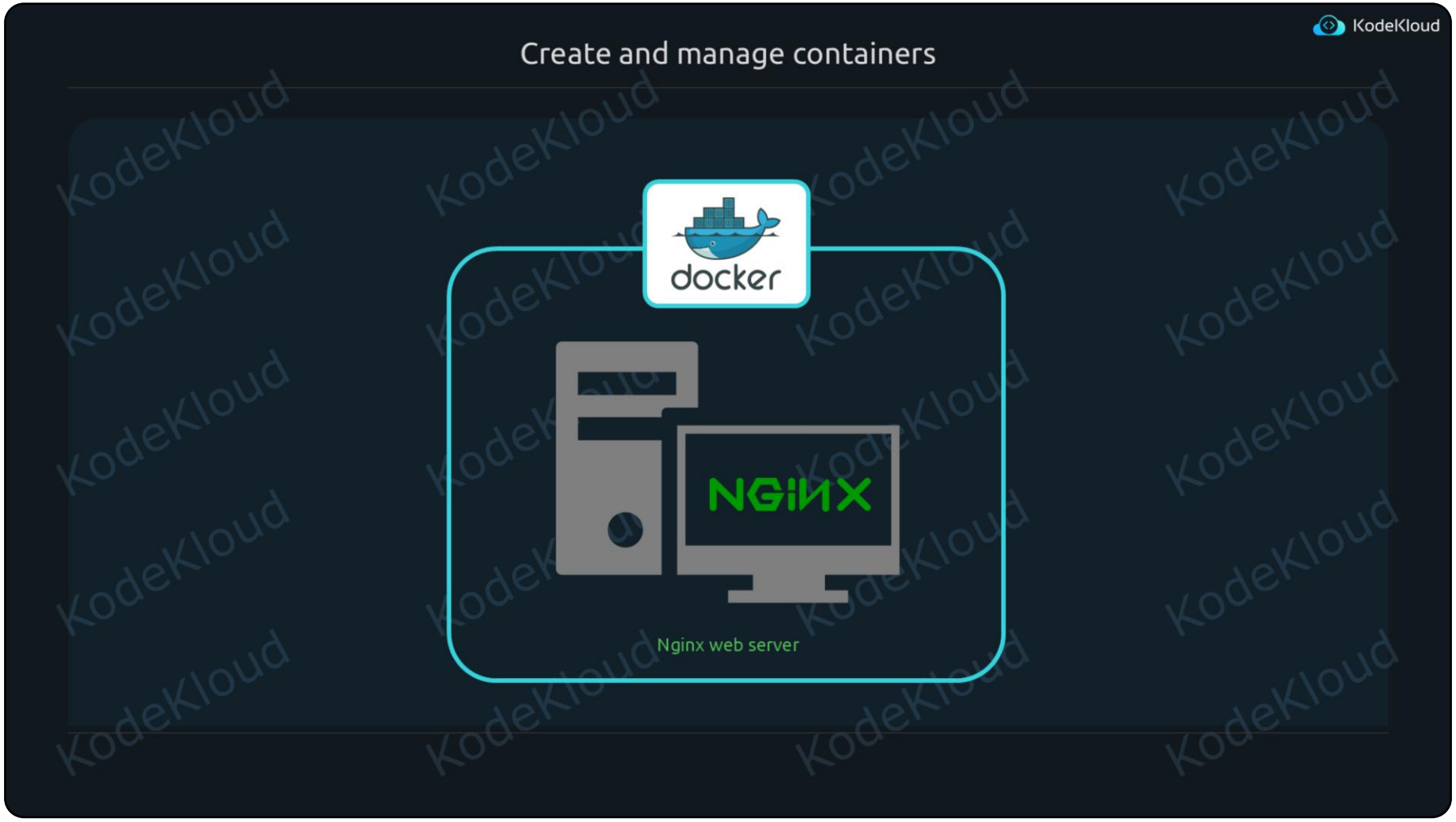
Containers encapsulate applications. This makes them portable, easy to move around. As easy as copy/pasting a file from one location to another.

Once we're happy with a containerized application, we can deploy it to tens of different servers, at scale. The same thing that Kubernetes does, and part of the reason why it's so popular.



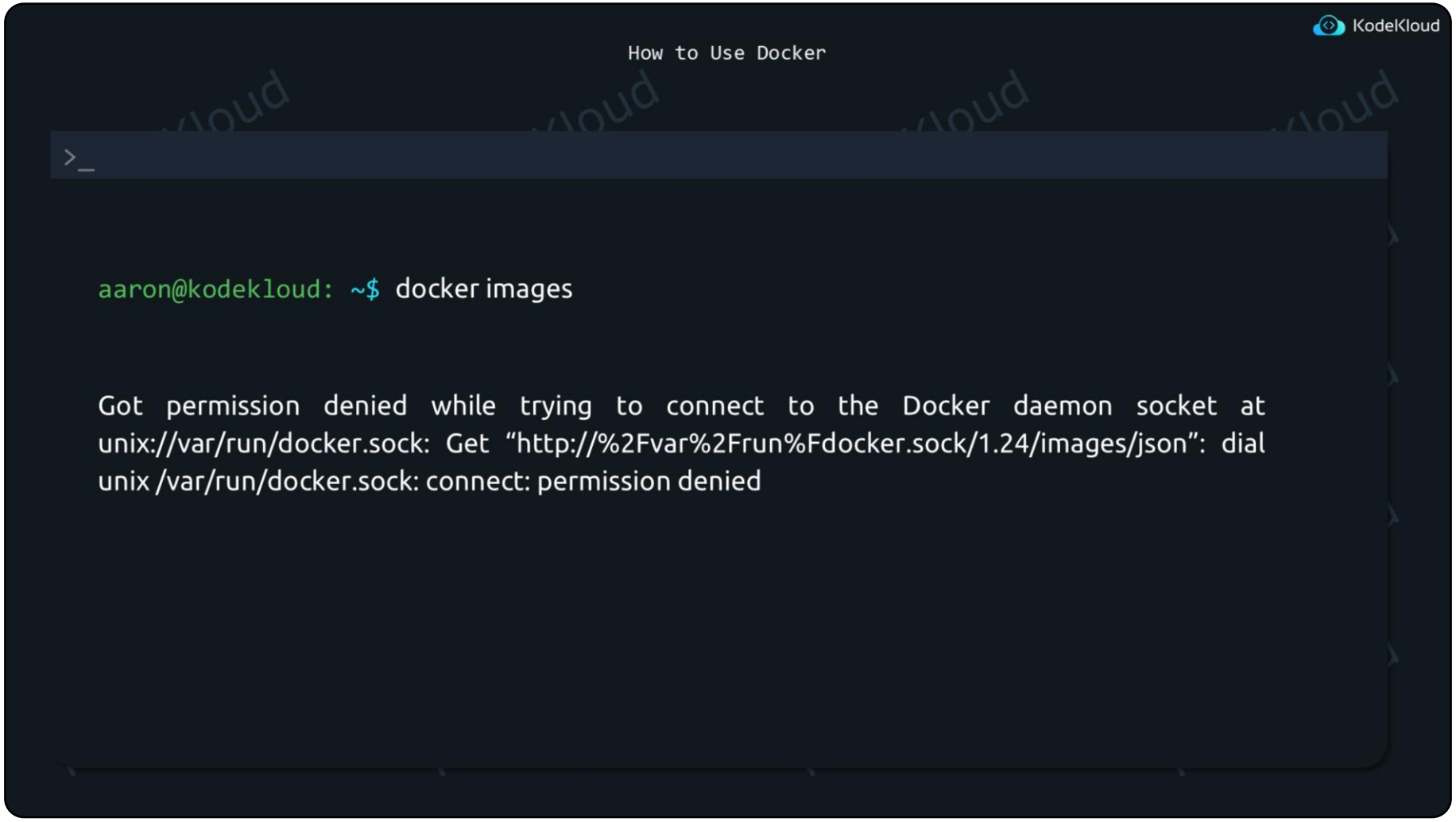
Containers encapsulate applications. It makes them portable, easy to move around. As easy as copy/pasting a file from one location to another. Furthermore, we can clone this container 100 times, and move it to 100 different locations if we want to.

Once we're happy with a containerized application, we can deploy it to tens of different servers, at scale. The same thing that Kubernetes does, and part of the reason why it's so popular.



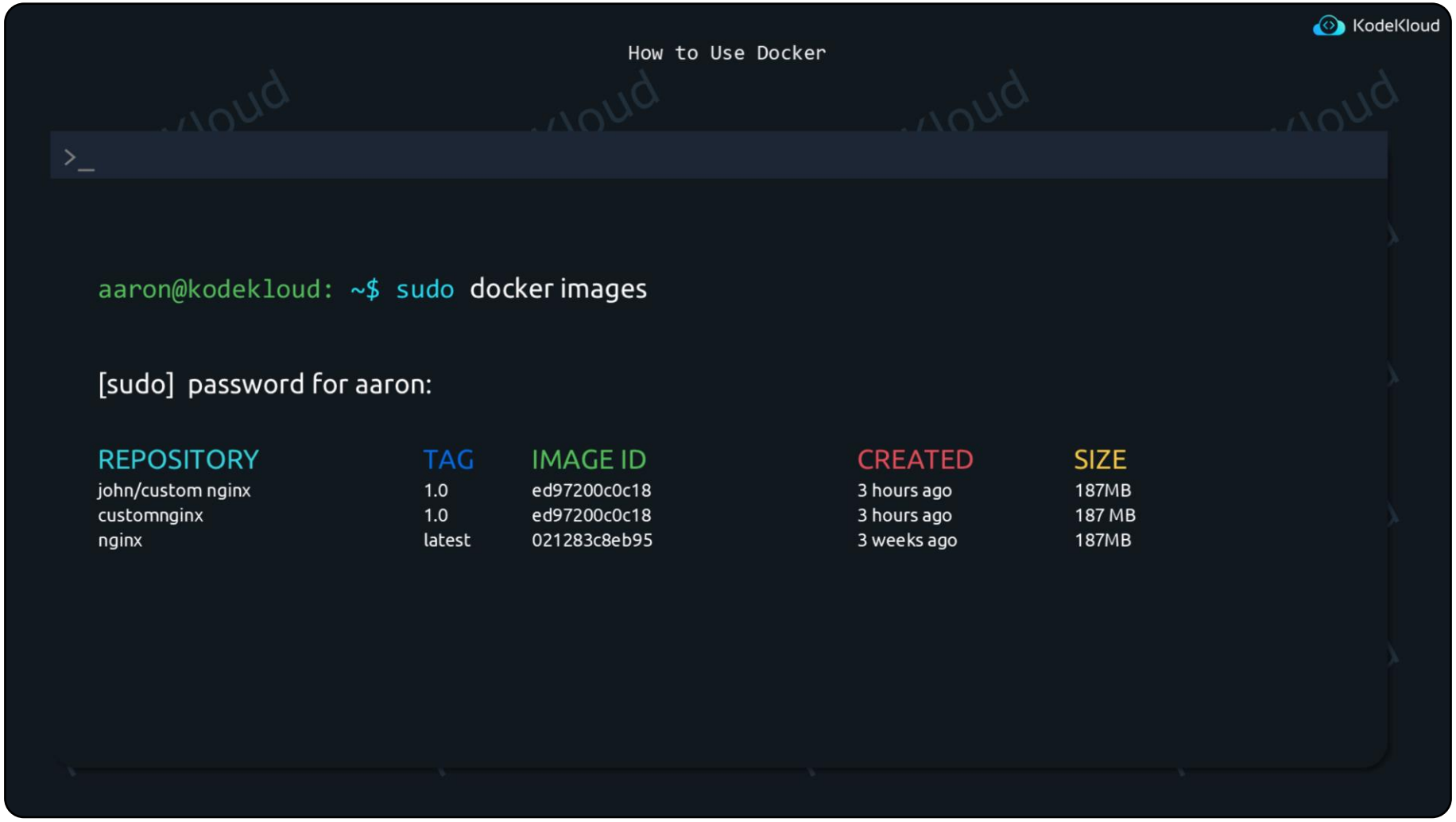
To understand how to work with Docker, let's jump right into some practical examples. It's easier to make sense of how containers work if we see them in action. We'll explore how to run an Nginx web server in a container.

But first, a quick note. Normally, you can run docker commands without "sudo" if your Linux user is part of a group called "docker". In some environments you might see "permission denied" errors when you try to run docker commands.



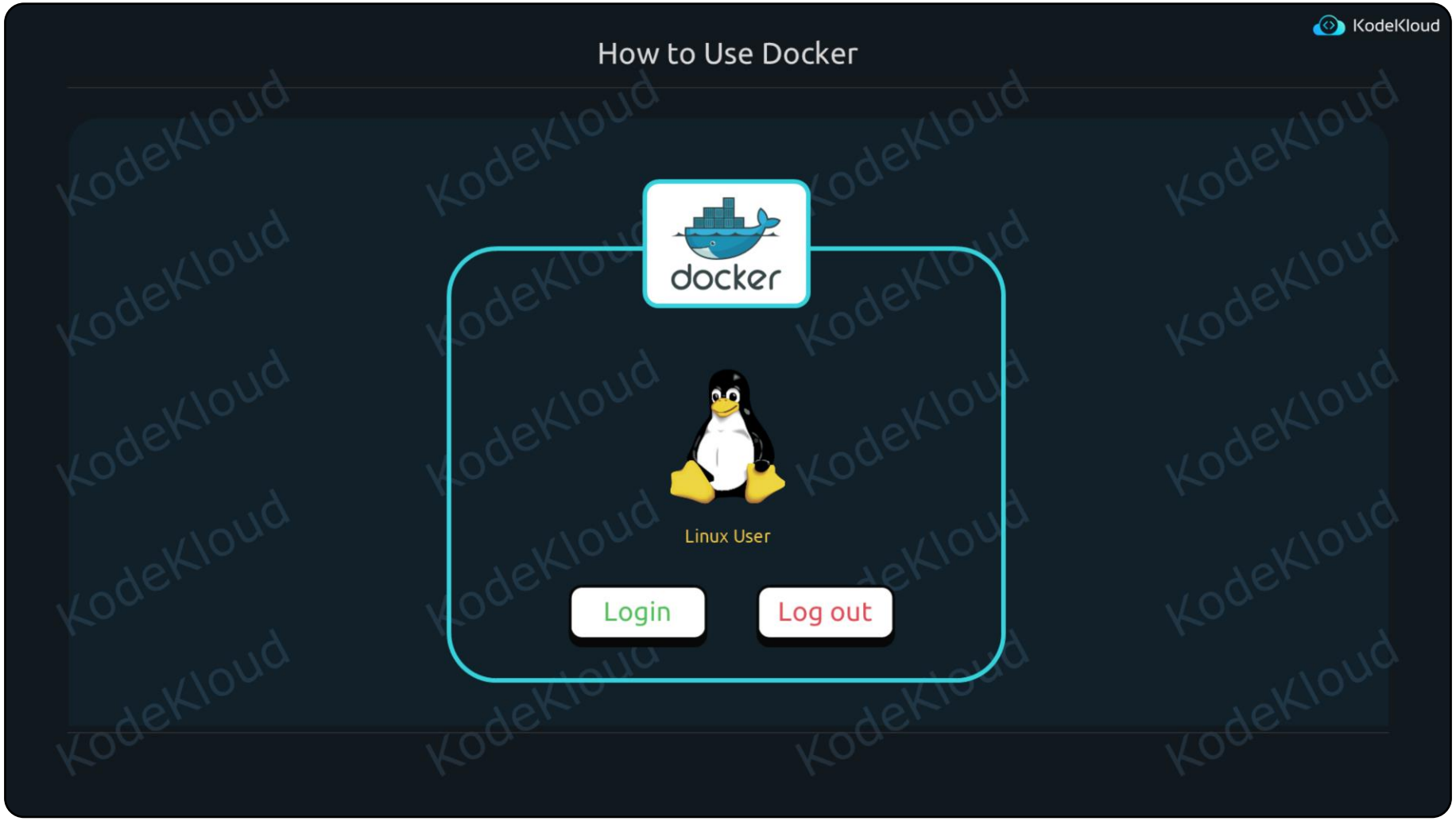
To understand how to work with Docker, let's jump right into some practical examples. It's easier to make sense of how containers work if we see them in action. We'll explore how to run an Nginx web server in a container.

But first, a quick note. Normally, you can run docker commands without "sudo" if your Linux user is part of a group called "docker". In some environments you might see "permission denied" errors when you try to run docker commands.



If you encounter such issues, just add "sudo" in front of your docker commands.

Or, if possible, add your Linux user to the "docker" group, log out, and log back in.



If you encounter such issues, just add "sudo" in front of your docker commands.

Or, if possible, add your Linux user to the "docker" group, log out, and log back in.

KodeKloud

Pre-installed Operating Systems

Cloud Images with Pre-installed Operating Systems

Installing an Operating System

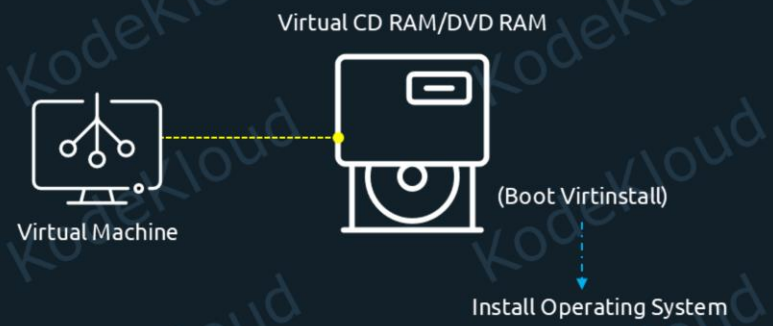
Cloud Images with Pre-installed Operating Systems

Virtual Machine

Virtual CD RAM/DVD RAM

Create an empty Disk Image

Installing an Operating System



KodeKloud

Visit www.kodekloud.com to discover more.