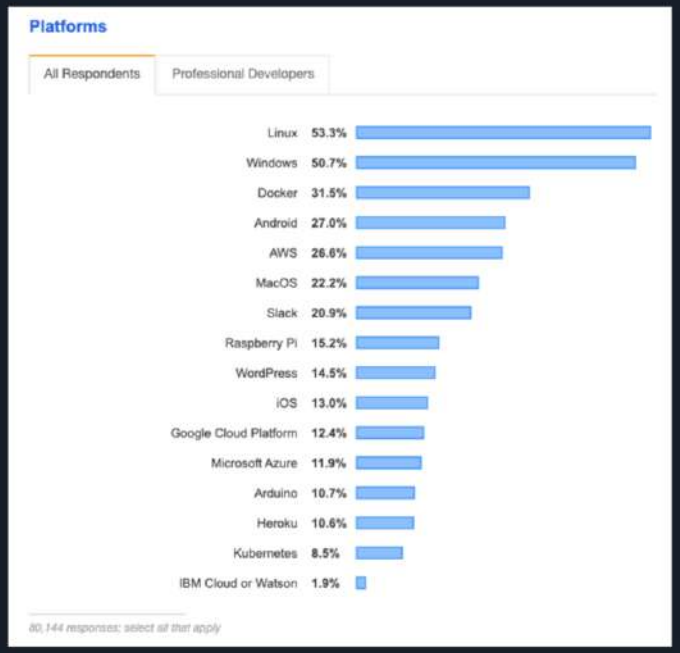
KodeKloud

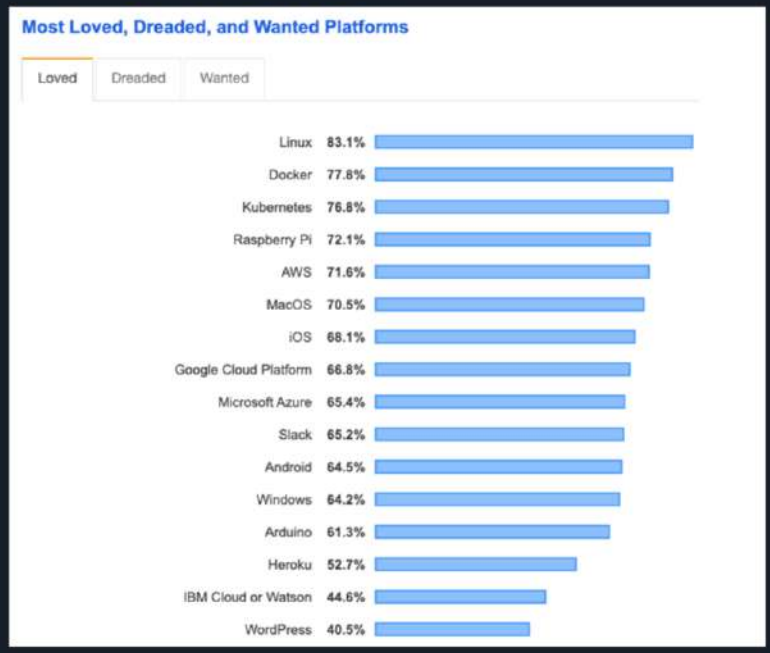
Visit www.kodekloud.com to discover more.

Hello, and welcome to the Linux Foundation Certified System Administrator preparation course.

I'm Aaron Lockhart, and I'll be your instructor for this course.

As per Stack overflow's insights the most common and the most loved platform used for development work happens to be Linux.





As per Stack overflow's insights the most common and the most loved platform used for development work happens to be Linux.

Linux Foundation Certified Sysadmin certification was developed by The Linux Foundation to help meet the increasing demand for Linux administration talent. LFCS helps you validate your system administration skill set and stand out in the market.

During the course, we'll cover each of the topics you'll need to know for the LFCS exam, from the six domains of knowledge it includes:

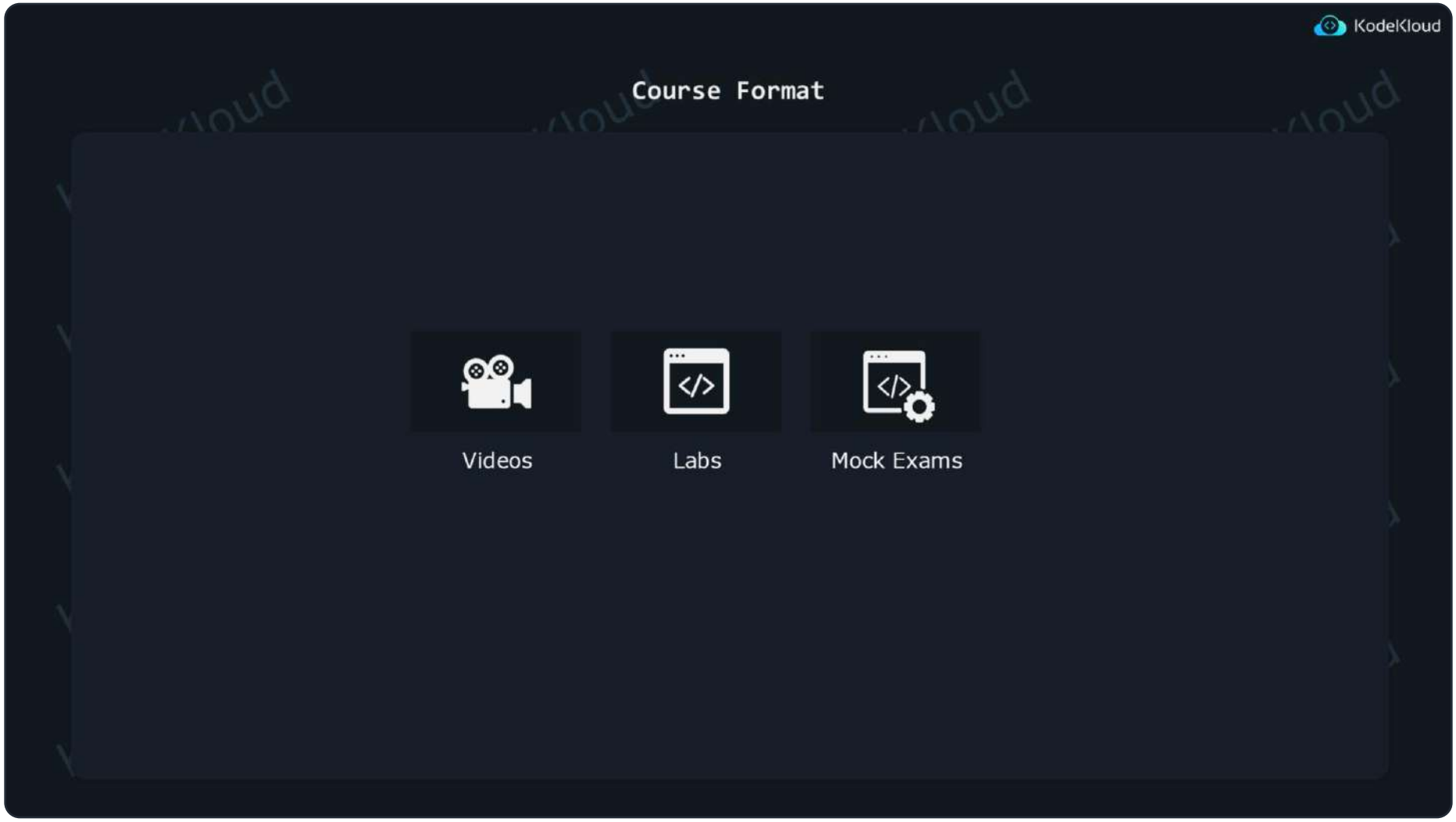
1: Essential commands Which will cover logging into Linux systems and working with files and directories.

2: Operations Deployment - Where we'll explore the system boot process, task automation, and managing critical resources and processes.

3: Users and Groups - Which will not only cover creating and managing user and group accounts, but also setting resource quotas and configuring advanced authentication options.

4: Networking Where we will learn to work with network services, routing, and packet filtering.

5: And storage management In which we will learn to set up logical volume management, RAID, and encrypted storage, as well as advanced file system permissions.



This is primarily a hands-on course with videos, labs and mock exams that will help you prepare for the certification. The LFCS exam is a hands-on exam and you need enough practical experience to pass it.

So, this course is designed accordingly. 30% of videos, 60% of labs and 10% mock exams. That is over 28 hours of lab time alone. And you can go through those as many times as you want.

KodeKloud

Visit www.kodekloud.com to discover more.

Now, let’s go over some details related to the Linux Foundation Certified System Administrator Exam.

First of all, there are no pre-requisites to attend the LFCS exam. Anybody with skills can attend the exam. With this course we will prepare you with the necessary skills you will need to prepare for and clear the exam.

Let's go over the exam objectives, in a bit more detail now.

1: The Essential commands section accounts for 20% of the possible points on the LFCS exam.

2: The section on Operation deployment accounts for 25% of the possible points on the LFCS exam.

3: The Users and groups section accounts for 10% of the possible points on the LFCS exam.

4: The Networking section accounts for 25% of the possible points on the LFCS exam.

5: And finally the storage section accounts for 20% of the possible points on the LFCS exam.

This course is divided into sections as shown above.

You will have two hours to complete the exam.

The exam is entirely performance-based. It simulates on-the-job tasks, and there are no multiple choice or true/false questions.

As of this recording, the cost for the exam is $395 US Dollars, and is valid for 2 years.

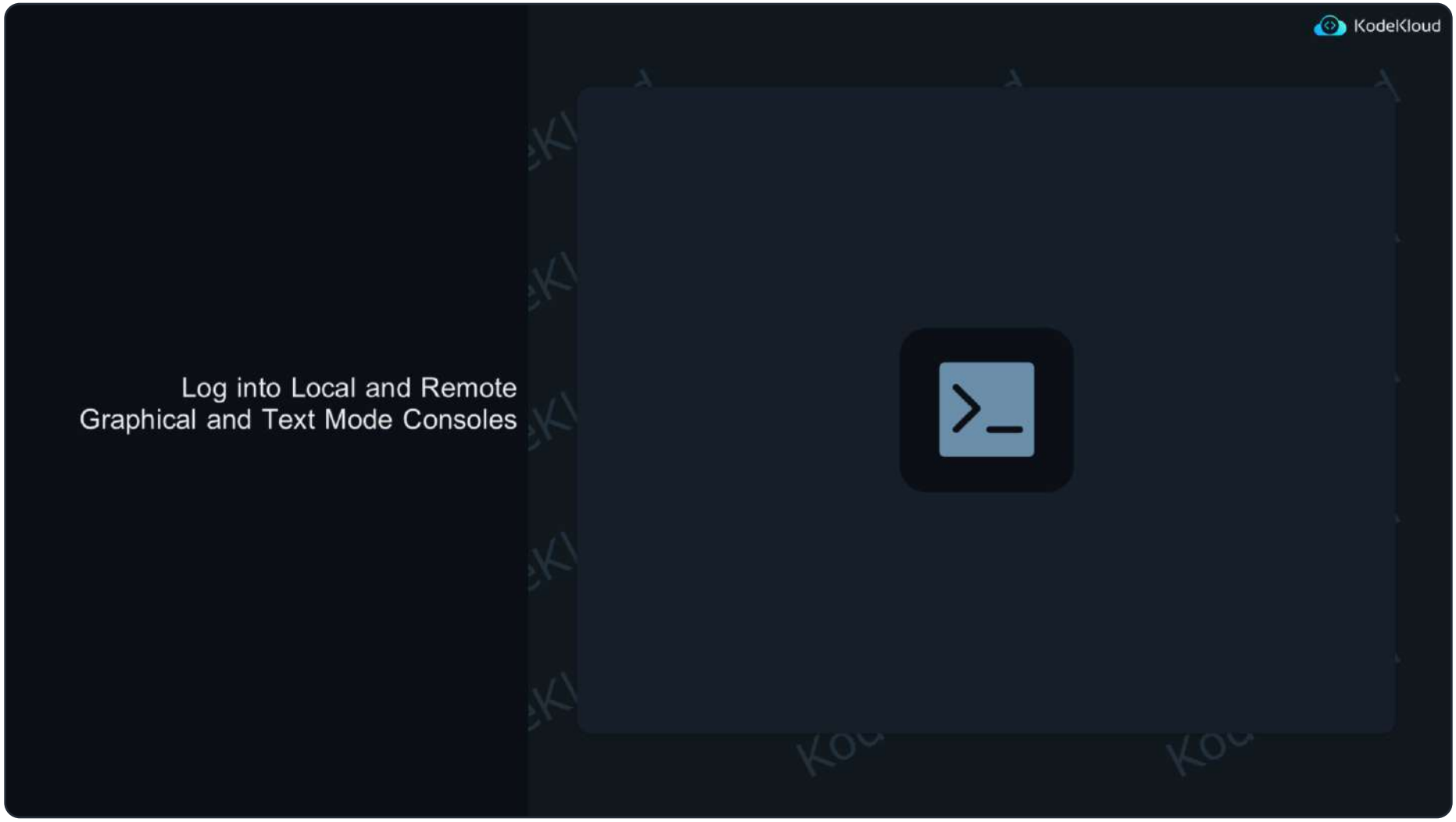
The exam is a proctored exam, and is available online through your browser, so you can take it from home.

We will discuss more details about registering for the exam towards the end of the course.

KodeKloud

Visit www.kodekloud.com to discover more.

Now, let’s go over some details related to the Linux Foundation Certified System Administrator Exam.



Now we'll look at how to create, delete, copy, and move files and directories in Linux.

Before we dive into this lesson, we need to understand a few basic things:

What is a filesystem tree?
What is an absolute path?
What is a relative path?

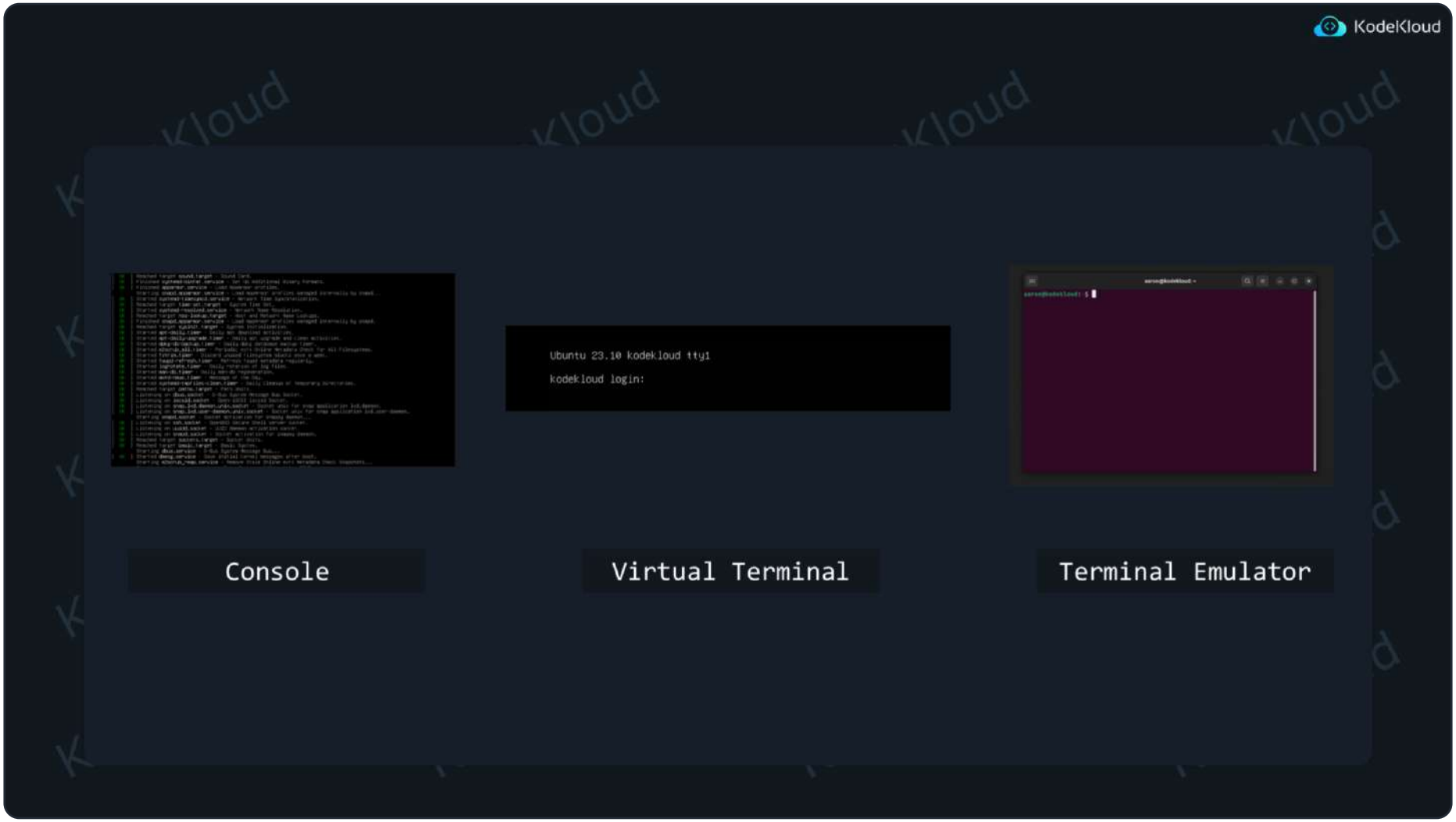
So, let's dive right in and start with some simple concepts.

We're all used to logging in to apps or websites by providing a username and password. Logging into a Linux system is pretty much the same, so there's not much mystery here. We'll look at four ways to log in:

Log into a local Linux system (local text-mode console).
Log into a local Linux system (local graphical-mode console).

Log into a remote Linux system (text-mode login)

Log into a remote Linux system (graphical-mode login)

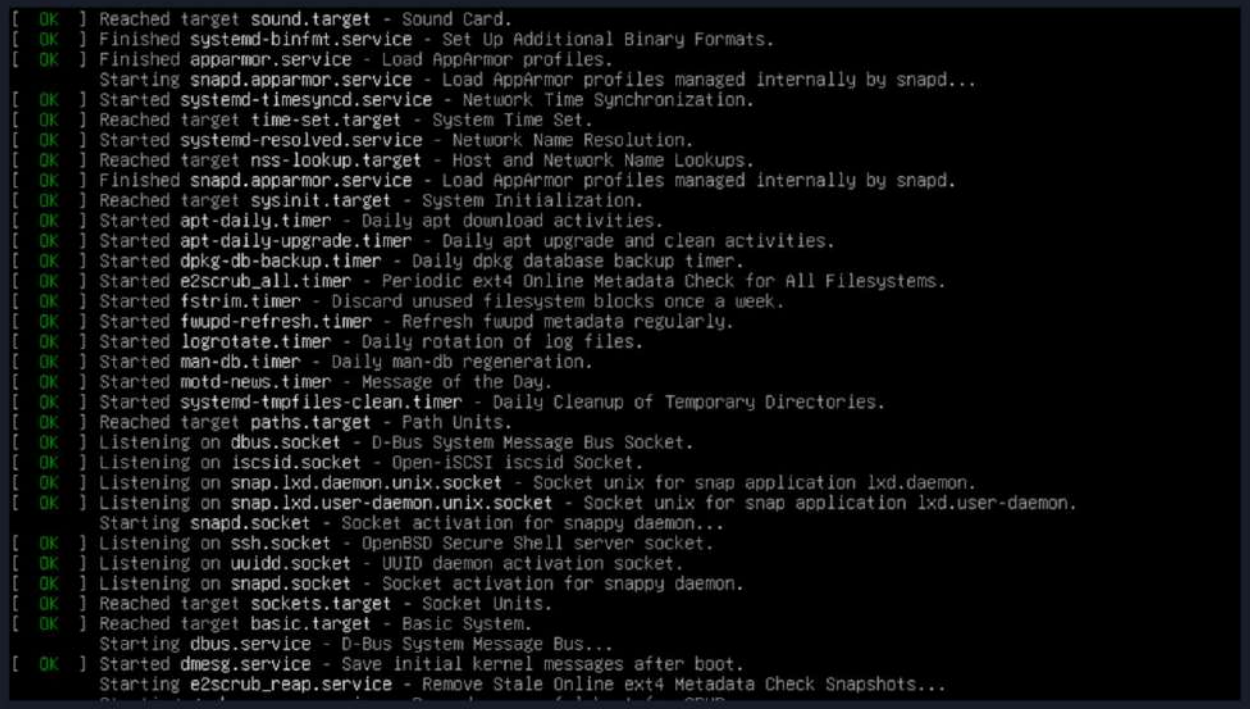


You'll often hear terms like "console", "virtual terminal", and "terminal emulator". And it may be hard to understand why they are called this way.

Nowadays, a "console" is just a screen where your operating system displays some text and where it allows you to log in or type commands. And a terminal emulator is nothing more than a graphical app that runs in a window and does a similar thing (shows you text output and allows you to type commands). These terms come from the old days of computing.

KodeKloud

Consoles



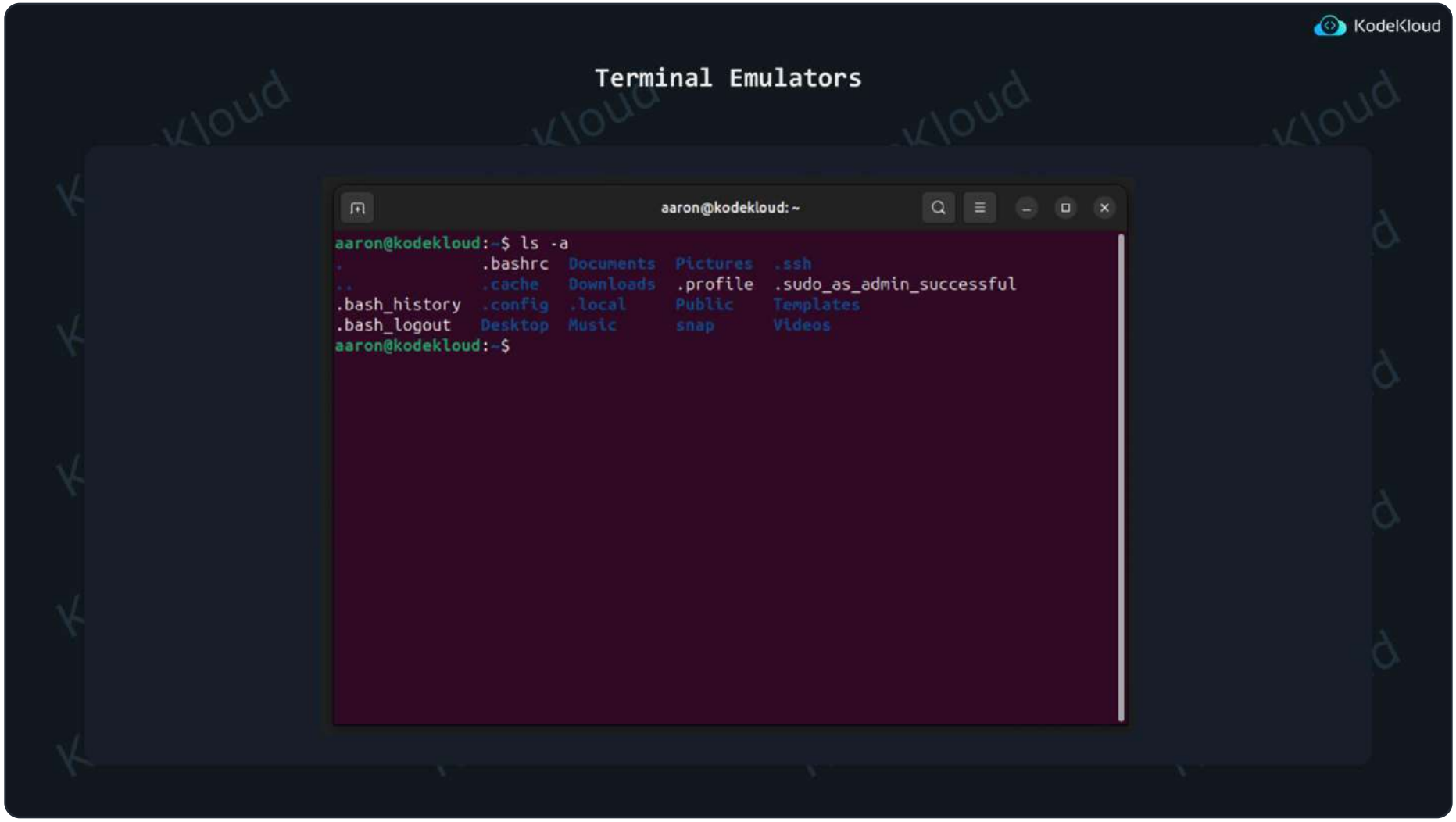
Computers were incredibly expensive, so a university may have had only a single computer available for their entire building.

But multiple people could connect to it and do their work by using physical devices that allowed them to type text and commands and also display on a screen what is currently happening. These devices were consoles or terminals. So instead of buying 25 super expensive computers, you could have just one, but 25 people could use it, even at the same time.

Nowadays, consoles and terminals, in Linux, are usually things that exist in software, rather than hardware. For example:

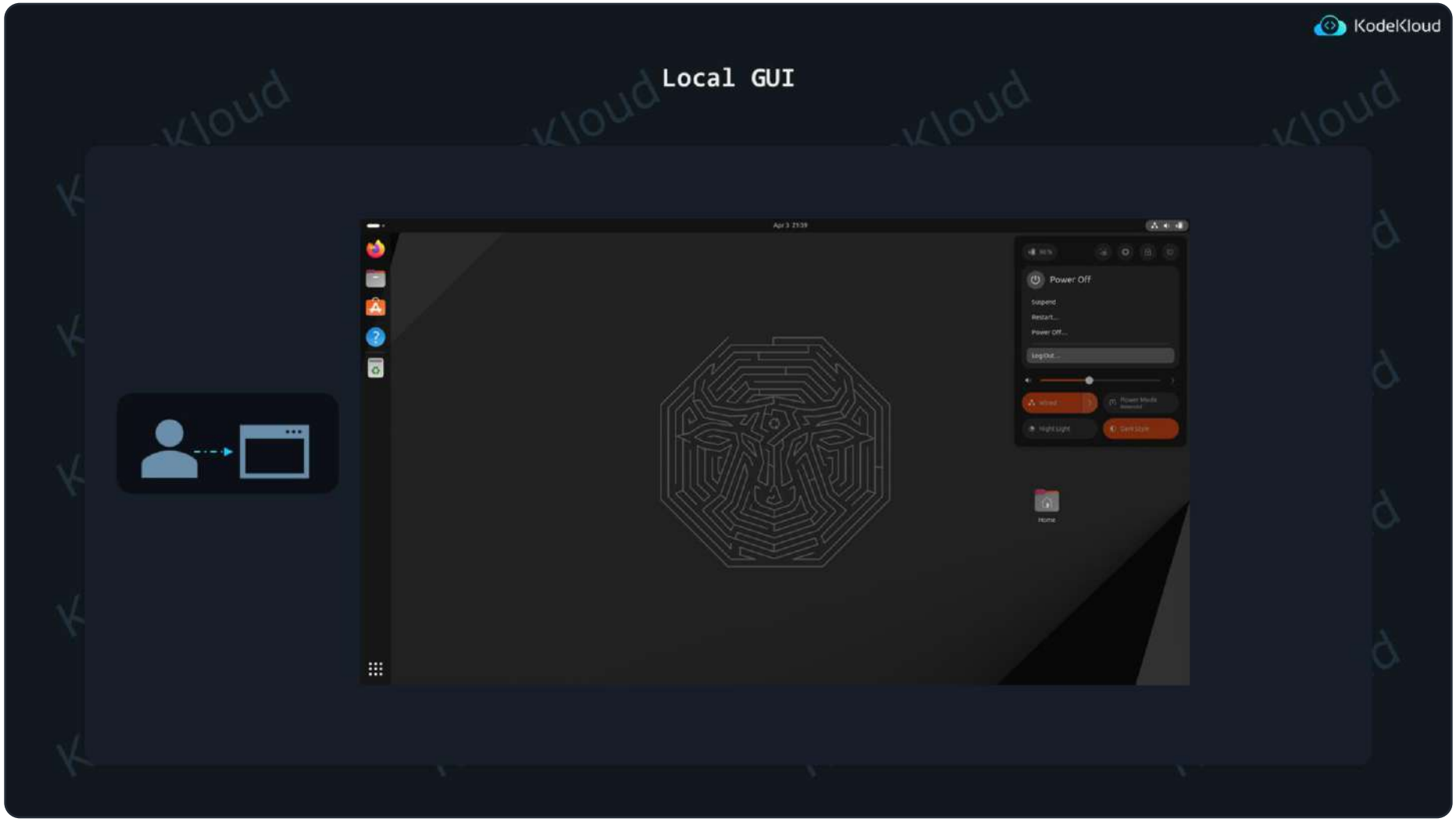
When you see Linux boot and a bunch of text appears on screen, telling you what happens as the operating system is loading - that's the console.

After a Linux machine has booted, if you press CTRL+ALT+F2 on the keyboard, you'll see a virtual terminal (vt2).



If you have Linux installed on your desktop, with a graphical user interface, when you want to type commands you open a terminal emulator.

Let's move back to logins. In practice, most often you'll log in to remote Linux systems. But let's start with the less common scenarios.



"Local" is just a tech word for "something that is in front of you" or "something you can physically access". A computer on your desk is local. A server running on Google Cloud is remote.

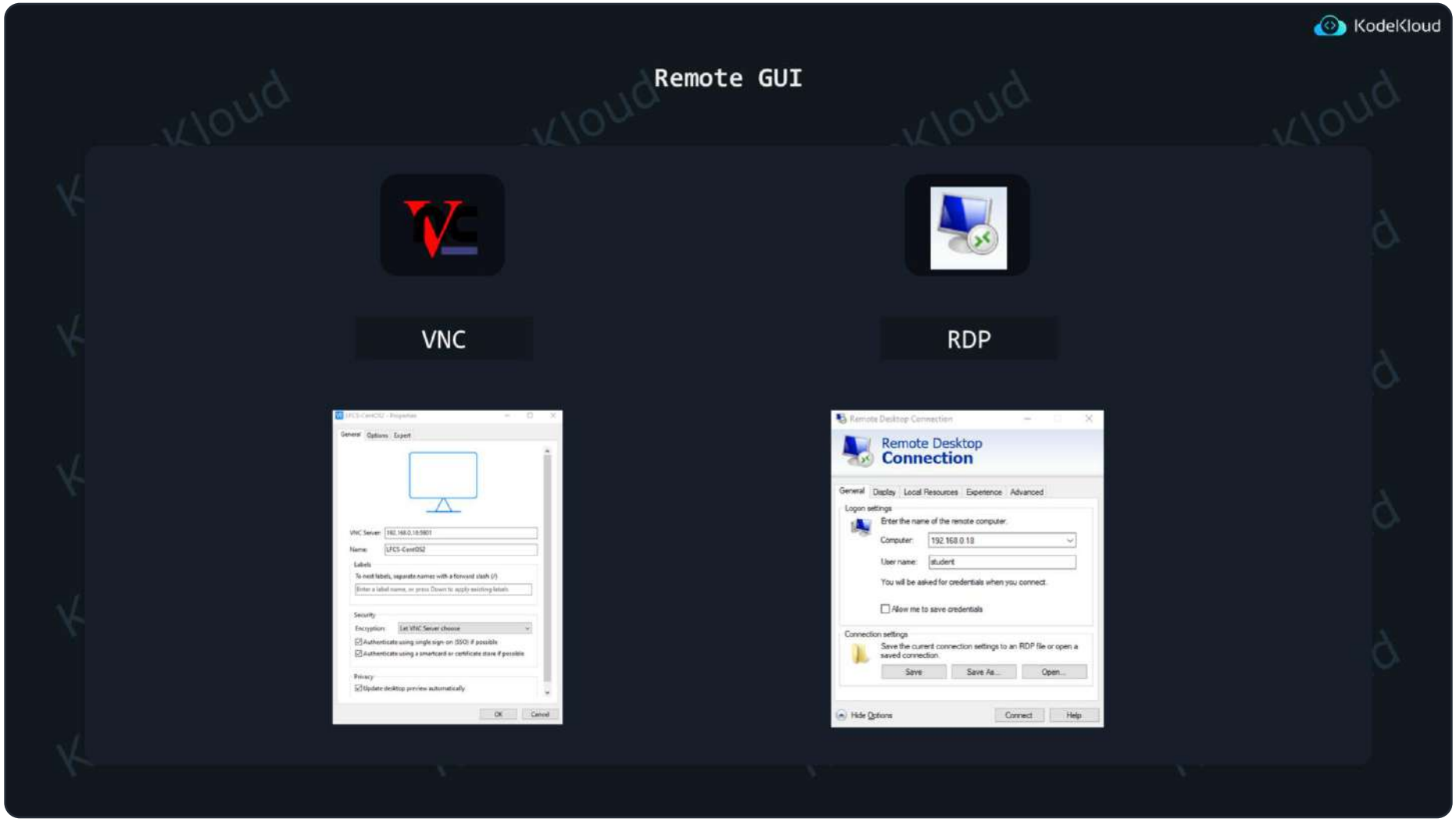
Usually, when Linux is installed on servers, it is installed without GUI (Graphical User Interface) components. There's no mouse pointer, no buttons, no windows, no menus, nothing of that sort, just text. But you might sometimes run across servers that include this GUI. Logging in is super easy, as it's all "in your face". You'll see a list of users you can choose from and you can then type your user's password.

Don't forget to log out when you've finished your work.

If the device has the typical server-oriented Linux OS installed, without any GUI components, logging in (locally) is also easy. You'll usually see something like this on your screen:

There's no list of users this time, but you can just type your username and then your password. Note that you won’t see your password as you type.

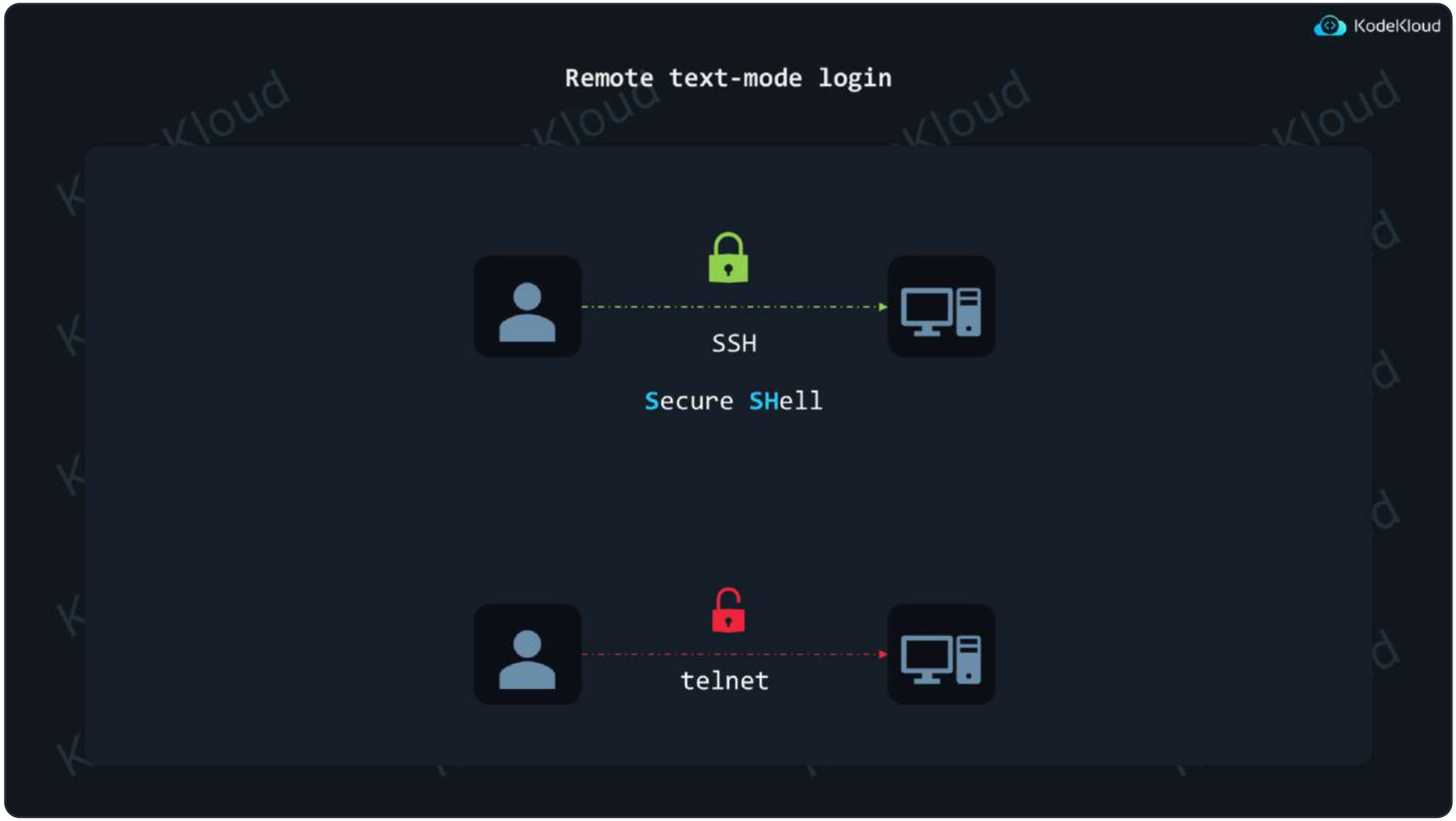
When your work is done, you should type exit to log out.



Again, most Linux operating systems running on servers will have no GUI components installed. But you will sometimes run into exceptions. Connecting to a remote server, to its graphical user interface is slightly more tricky. First of all, there is no standard set in stone. Whoever configured that server chose their preferred way of dealing with these remote graphical logins. They could have chosen to install a VNC (Virtual Network Computing) solution. In this case, you'd need to download the proper VNC client (also called "VNC viewer") to connect to it. This might be TightVNC or RealVNC or something else entirely. It all depends on the VNC server running on the remote system and what VNC clients your local operating system supports.

If the administrator of that server wanted to let Windows users connect easily, it might mean that they used a solution allowing for RDP connections (Remote Desktop Protocol). This means you can just click on Windows' start button, type "Remote Desktop Connection", open that app and then enter the username and password you've been provided.

Whatever it might be, connecting to a remote graphical console is pretty easy. It all boils down to downloading the application that lets you do that, entering the remote system's IP address, followed by an username and a password.



Initiating a text-based remote connection to a Linux system is pretty standard. That's because almost every Linux server uses the same tool that allows for remote logins: the OpenSSH daemon (program that runs in the background, on the server, all the time). SSH comes from Secure SHell. Until SSH, something called telnet was the standard. telnet was highly insecure as it did not encrypt communication between you and the server you were connecting to. This meant that anyone on the same network with you could steal your Linux user password and see everything you did on that server, during your telnet session.

The SSH protocol uses strong encryption to avoid this and the OpenSSH daemon is built carefully to avoid security bugs as much as possible. Long story short, OpenSSH is used by millions of servers and has stood the test of time, proving to be very hard to hack. For these reasons everyone happily uses it and trusts that it can do a pretty good job at only letting authorized people

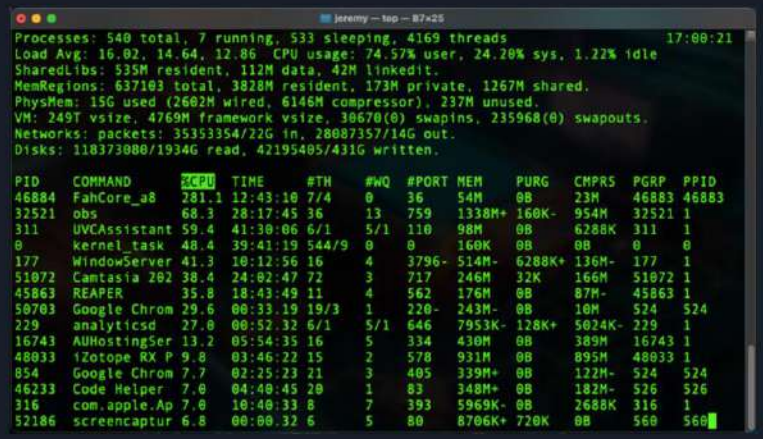
log into their operating systems, while keeping bad people out.

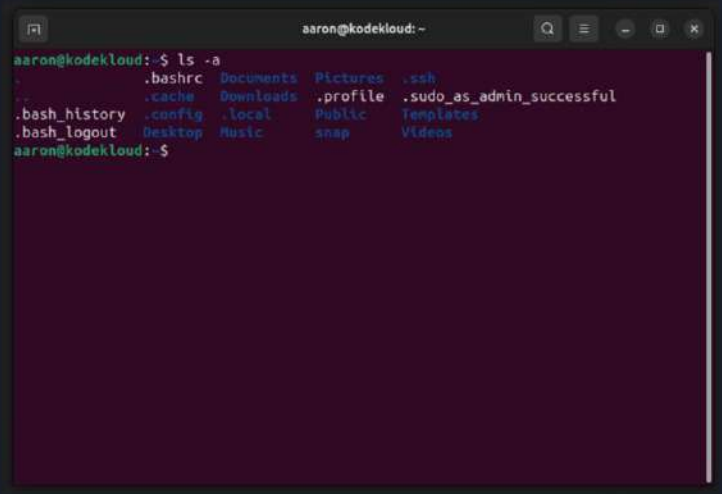
In case you're following along on your virtual machine, log in locally (directly from the virtual machine window) and then enter this command: (ip a) You'll see what IP your machine uses. I've outlined the information we're looking for in yellow.

We'll use this IP – in our case 192.168.0.17 -- to simulate a situation where we have a server in a remote location.

Now to recap. We have an SSH daemon (program) running on the server. This listens for any incoming connections. To be able to connect to this remote SSH daemon, we'll need something called an SSH client (yet another program). This client will run on our current laptop/desktop computer.

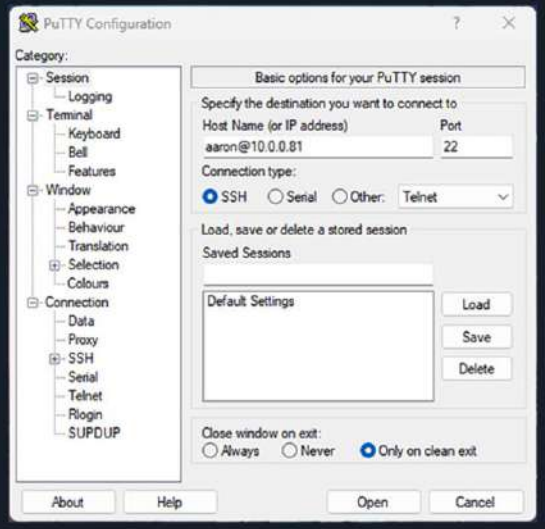
MacOS & Linux

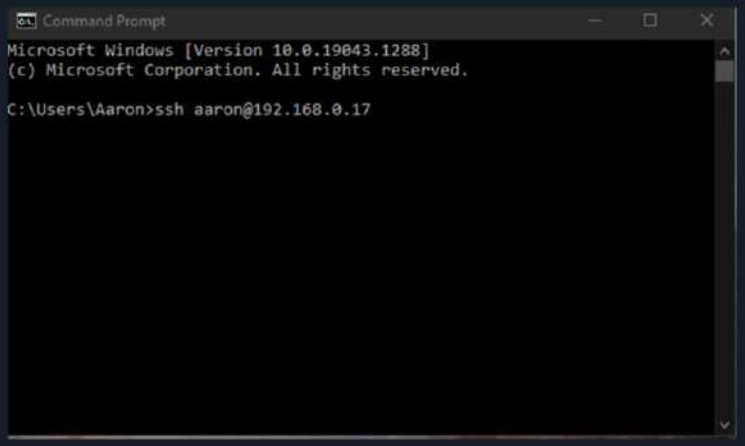




MacOS systems and Linux-based operating systems, such as Ubuntu, already have an SSH client preinstalled. If you're on MacOS or Linux, open a terminal emulator window.

Windows





In the past, if you were running Windows, you needed to install an SSH client like PuTTY. On the latest Windows 10 (and 11) this is no longer necessary as an SSH client is also preinstalled. If you're on Windows, click the Start Menu and type "cmd" to open up Command Prompt.

To connect to a remote Linux system through SSH, type:

Of course, replace "aaron" with the actual username you created inside your Linux OS running in the virtual machine. Same with the IP address. From here on, we'll stay inside this SSH session to go through all the exercises in the upcoming lessons. Please join me in the demonstration video to see each of these login methods. I'll see you there.

KodeKloud

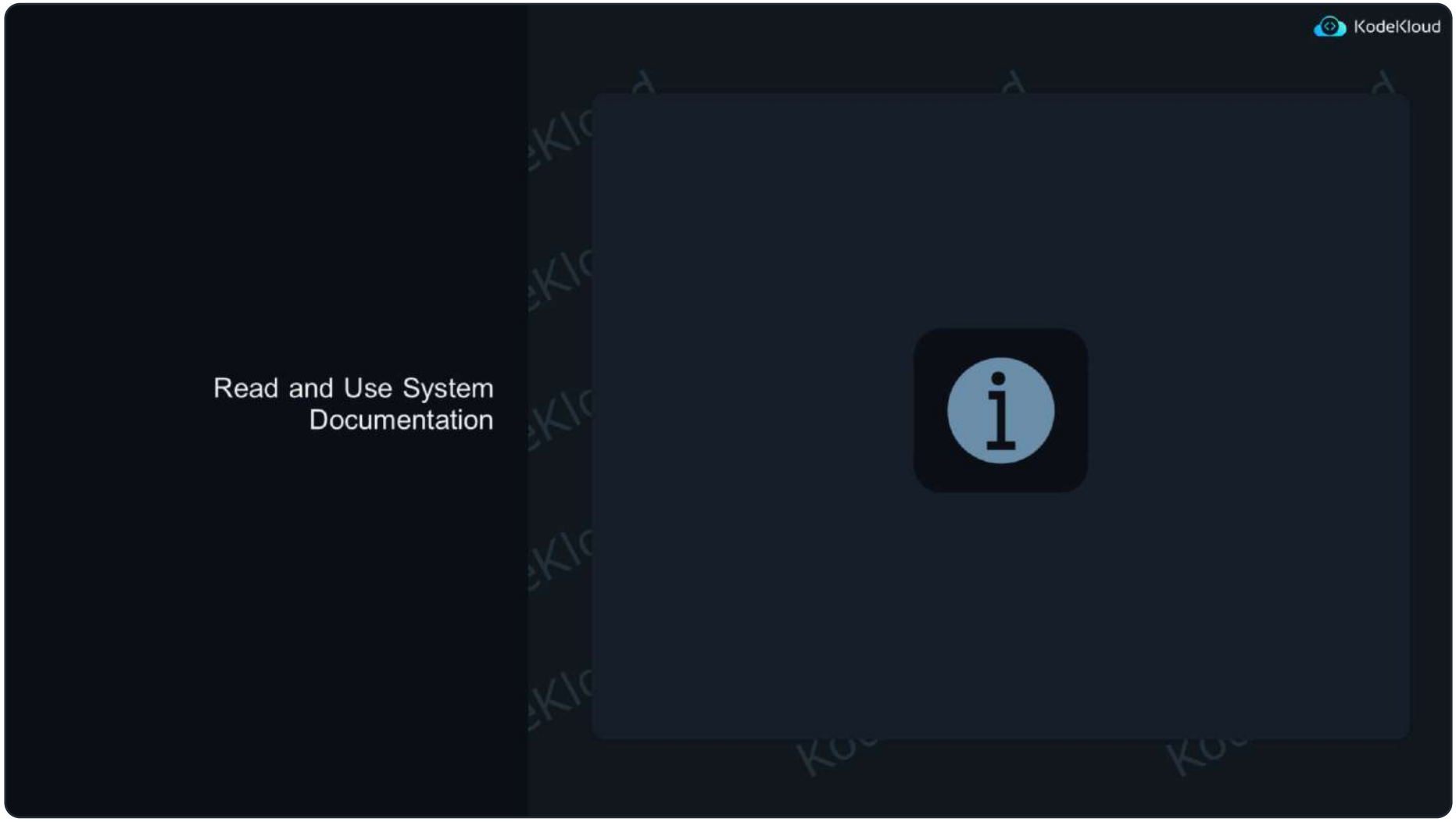
Visit www.kodekloud.com to discover more.

KodeKloud

Demo

Log into Local and Remote

Graphical and Text Mode Consoles



There will be many commands we will use in Linux. And each command has a lot of command line switches. How are we supposed to remember them all?

As we use a command repeatedly, we'll learn everything about it and memorize what each option does. But in the beginning, we might forget about these options after just one or two uses. That's why Linux gives you multiple ways to access "help manuals" and documentation, right at the command line.

Let's say you want to see that long listing format with ls, to get a look at file permissions. But you forgot what the correct option was. Was it -p for permissions? We can get a quick reminder with:

ls --help

This will show us a lot of output. But if we scroll up, we'll find what we're looking for: the –l flag, in this case.

You can see how command line options are sorted alphabetically and described with short text. That's why the --help option for commands will very often be helpful when we forget about these options (and we will, as there are so many of them for each command).

--help will usually show a condensed form of help, with very short explanations. For ls, that's ok, as it's a very simple command. Other commands, however, are very complex and we need to read longer explanations to understand what they do and how we use them.

Let's take journalctl as an example, a command that lets us read system logs.

journalctl --help

will show us this:

We'll notice that this opens in a slightly different way (look at "lines 1-27") in the bottom left corner. This opened in what Linux calls a "pager". It's simply a "text viewer" of sorts that lets us scroll up and down with our arrow keys or PAGE UP, PAGE DOWN. To exit this help page, press q.

All important commands in Linux have their own manuals or "man pages". To access a command's manual enter "man name_of_command". In our case, we'd use:

man journalctl

Now we get:

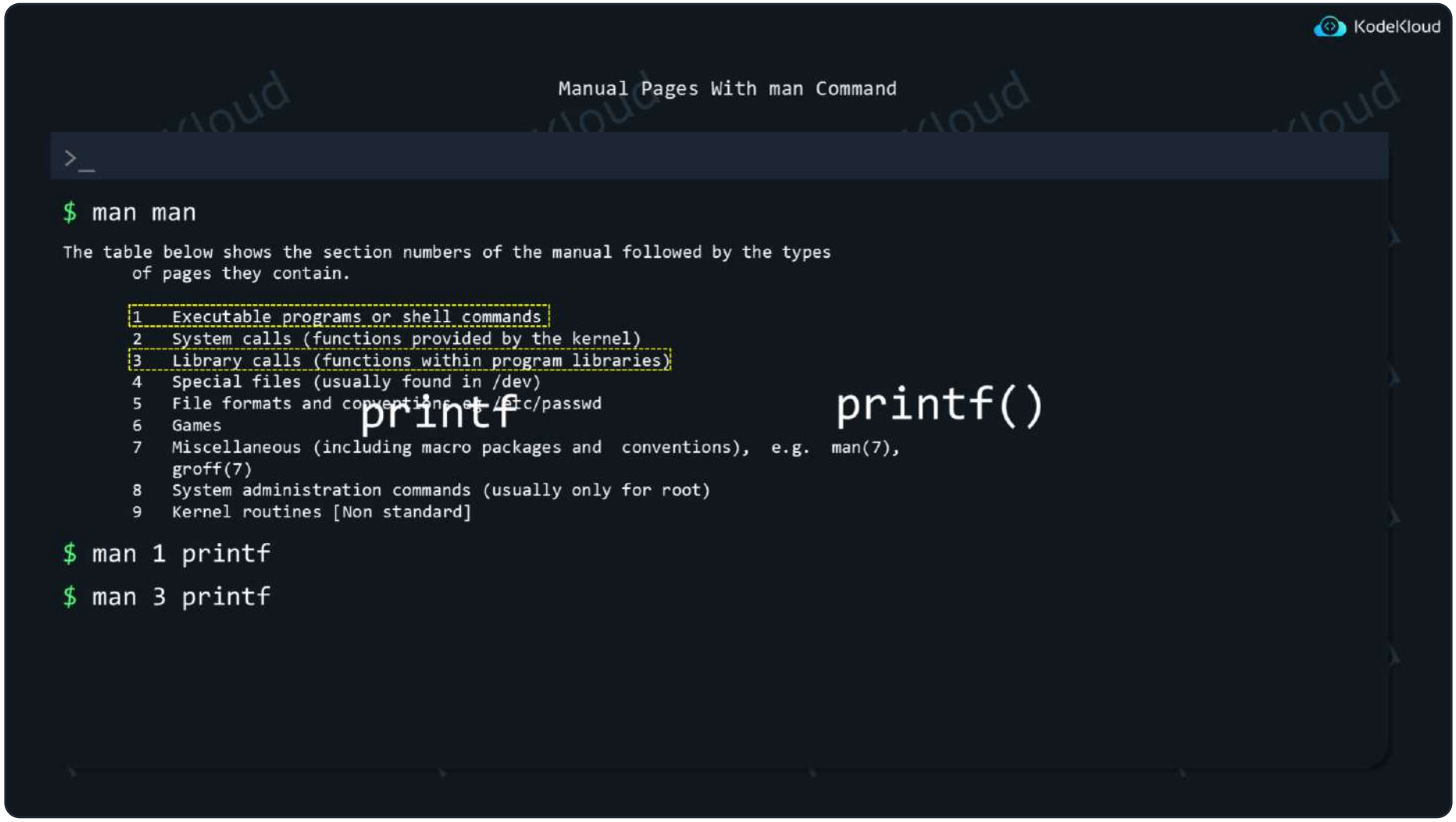
Short description of what the command does in NAME.

General syntax of command in SYNOPSIS

Detailed description of command, how it works, and so on, in DESCRIPTION.

Detailed descriptions of command line options in OPTIONS.

And some manual pages even have some EXAMPLES near the end of the manual.



Sometimes, you will have two man pages with the same name. Example:

printf is a command. But printf is also a function that can be used by programmers.

Manual pages can fall into one of these categories (sections), and we can see these by looking at the man page for man itself, by typing man man:

If you want to read the man page about printf, the command, you tell man you want to consult printf from section 1, like this

man 1 printf

If you want to read about printf, the function, you tell man you want to look at section 3

man 3 printf

It's useful to know that during online exams, the Linux Foundation will let you use man and --help. Try to use --help if you forgot a command line option as that gives you the fastest results. Diving deep into a manual page will eat up more time.

But this is all well and good when we know what command we want to explore. But what if we can't even remember the name of the command that we need to use?

Imagine you forgot the name of the command that lets you create a new directory. How would you search for it?

apropos is a command that lets you search through man pages. It looks at the short descriptions of each man page and tries to see if it matches the text we entered. For example, with the next line we can search for all man pages that have the word "director" in their short descriptions. We'll use "director" and not "directory". "director" will match commands that contain the word "directory" but also the ones that contain "directories". So, we keep it more generic this way.

The first time we would run apropos director, we'd get an error.

That's because apropos relies on a database. A program must refresh it periodically. Since we just started this virtual machine, the database hasn't been created yet. We can create it manually with:

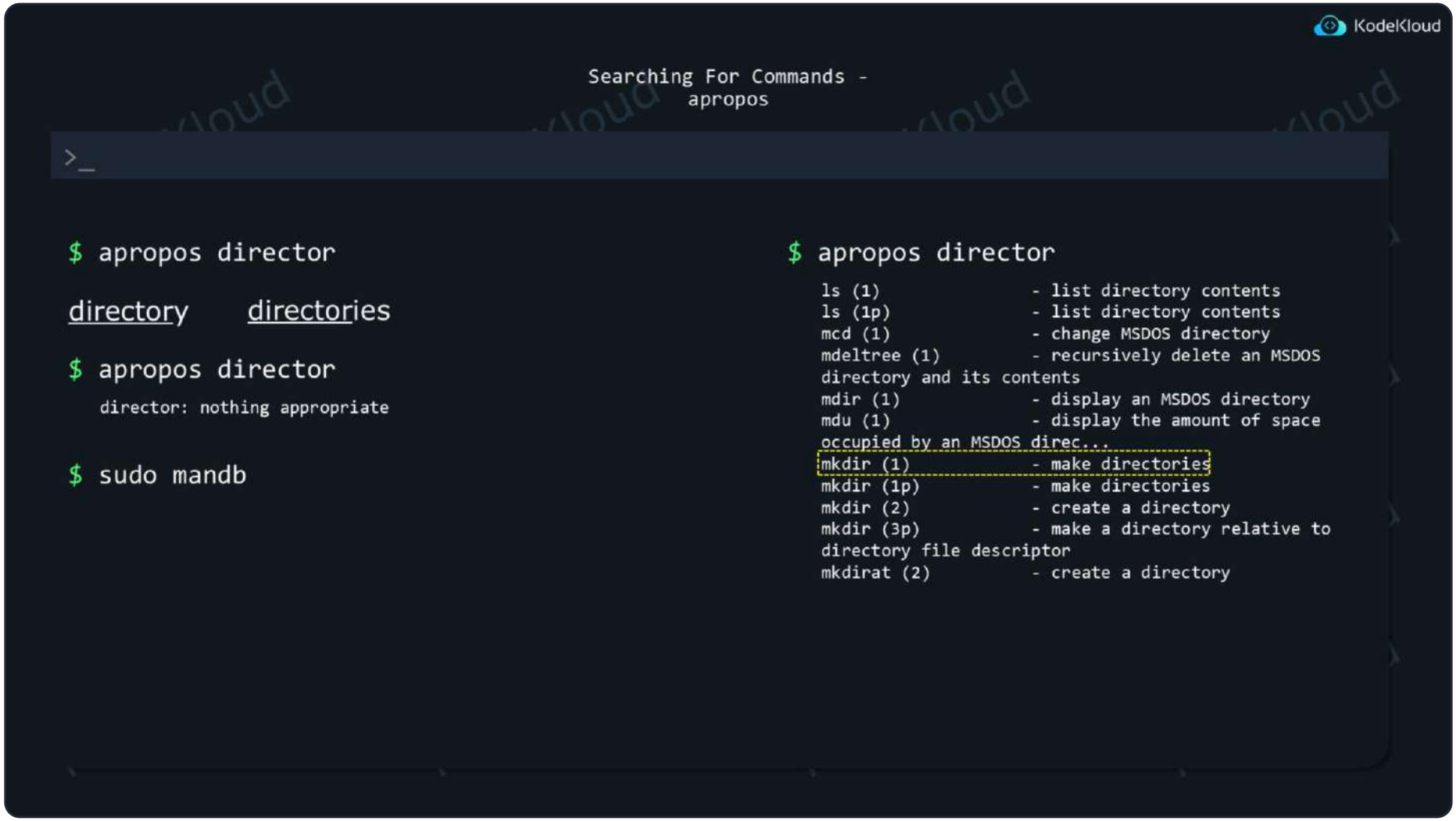
sudo mandb

On servers that have already run for days, there should be no need to do this, as it will be done automatically.

Now the apropos command should work:

apropos director

If we scroll up, we can see the entry we're looking for: mkdir.



Imagine you forgot the name of the command that lets you create a new directory. How would you search for it?

apropos is a command that lets you search through man pages. It looks at the short descriptions of each man page and tries to see if it matches the text we entered. For example, with the next line we can search for all man pages that have the word "director" in their short descriptions. We'll use "director" and not "directory". "director" will match commands that contain the word "directory" but also the ones that contain "directories". So, we keep it more generic this way.

The first time we would run apropos director, we'd get an error.

That's because apropos relies on a database. A program must refresh it periodically. Since we just started this virtual machine, the database hasn't been created yet. We can create it manually with:

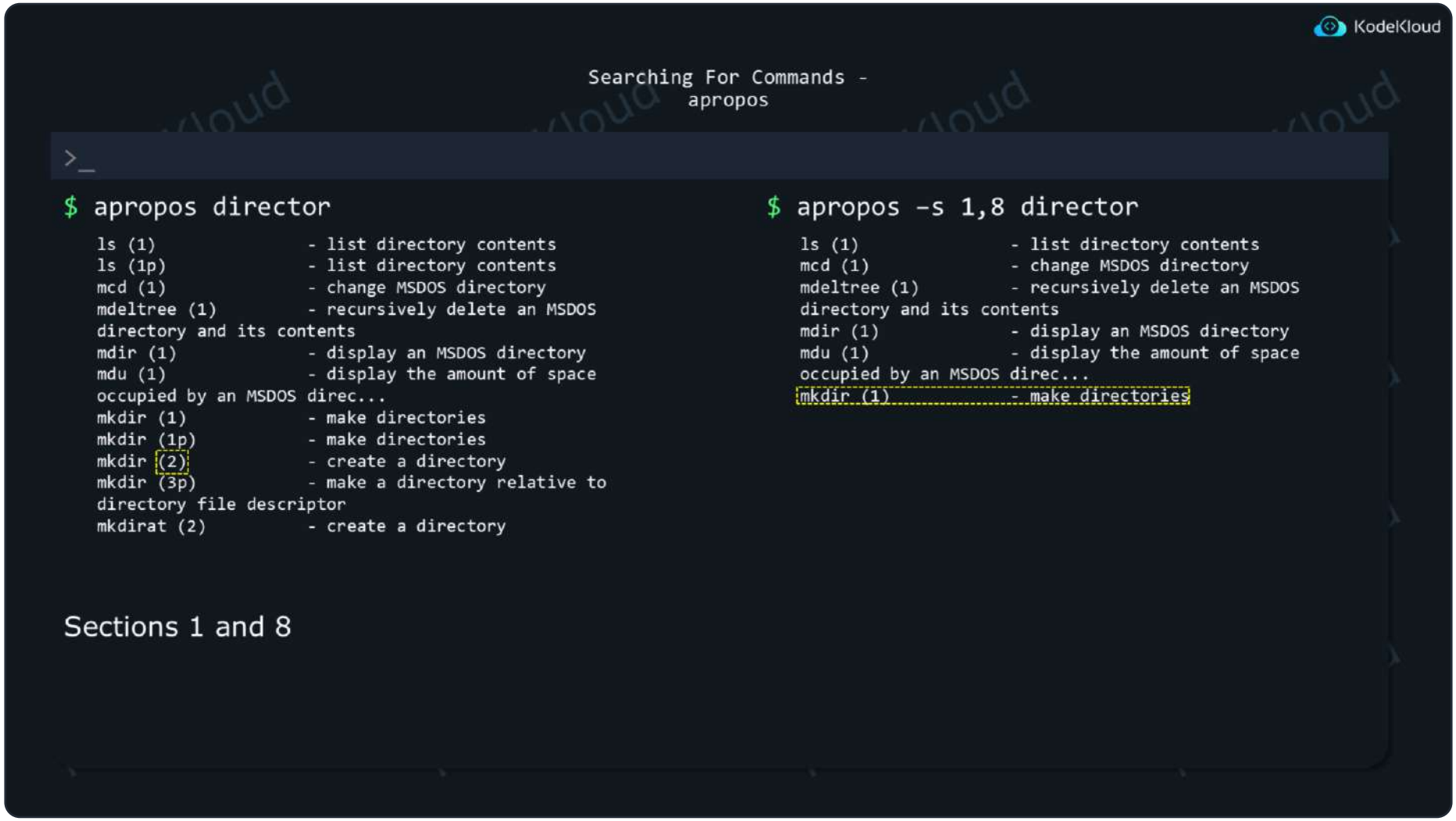
sudo mandb

On servers that have already run for days, there should be no need to do this, as it will be done automatically.

Now the apropos command should work:

apropos director

If we scroll up, we can see the entry we're looking for: mkdir.



But those are a lot of entries. Makes it hard to spot what we're looking for. You see, apropos doesn't just list commands. It also lists some other things we don't need, currently. We see stuff like (2). That signals that that entry is in section 2 of the man pages (system calls provided by the Linux kernel). That's just too advanced for our purposes. Commands will be found in sections 1 and 8. We can tell apropos to only filter out results that lead to commands from these categories. We do this by using the -s option, followed by a list of the sections we need.

apropos -s 1,8 director

And we can spot what we were looking for more easily.

Notice how mkdir's description contains the word "directories". If we'd have used the word "directory" in our apropos search, this command wouldn't have appeared since "directory" wouldn't have matched "directories". This is something to keep in mind when you want to make your searches as open as possible and match more stuff.

Another thing that'll save a lot of time is autocompletion. Type

systemc

press TAB

you get:

systemctl

Although this is not technically system documentation, it can still be helpful. Many commands have suggestions on what you can type next. For example, try this. Type

systemctl

add a space after the command (don't press ENTER) and now press TAB twice.

You get a huge list of suggestions. This can help you figure out what your options for that command are. Although you should not always rely on it. It's not necessary that absolutely all options are included in this suggestion list.

now add to that:

systemctl list-dep

press TAB

endencies will get added at the end and you get: systemctl list-dependencies. This is TAB autocompletion and many commands support it. When you press TAB once, if your command interpreter can figure out what you want to do, it will automatically fill in the letters. If there are many autocomplete options and it can't figure out which one you want, press TAB again and it will show the list of suggestions we observed earlier. These will be huge timesavers in the long-run, and they might even help you in the exam, to shave off a few seconds here and there, which might add up and let you explore an extra question or two.

TAB suggestions and autocompletions also work for filenames or directory names. Try

ls /u TAB

ls /usr/ TAB TAB

Now we can see directories available in /usr/ without even needing to explore this directory with "ls" beforehand. And if we have a long filename like "wordpress_archive.tgz" we might be able to just type "wor", press TAB and that long name will be autocompleted.

Recommendation

While manuals and --help pages are super useful, the first few times you use them, it might be hard to figure out how to do something, with that info alone. We recommend you take a command you know nothing about and try to figure out with just man and --help, how to do something. This practice will help you develop the ability to quickly look for help when you're taking the LFCS exam. There will be questions about theory you either don't know about, or you just forgot. If you know how to quickly figure out the answer with a man page or --help, you'll be able to pass the exam much more easily.

TAB suggestions and autocompletions also work for filenames or directory names. Try

ls /u TAB

ls /usr/ TAB TAB

Now we can see directories available in /usr/ without even needing to explore this directory with "ls" beforehand. And if we have a long filename like "wordpress_archive.tgz" we might be able to just type "wor", press TAB and that long name will be autocompleted.

Recommendation

While manuals and --help pages are super useful, the first few times you use them, it might be hard to figure out how to do something, with that info alone. We recommend you take a command you know nothing about and try to figure out with just man and --help, how to do something. This practice will help you develop the ability to quickly look for help when you're taking the LFCS exam. There will be questions about theory you either don't know about, or you just forgot. If you know how to quickly figure out the answer with a man page or --help, you'll be able to pass the exam much more easily.

KodeKloud

Visit www.kodekloud.com to discover more.

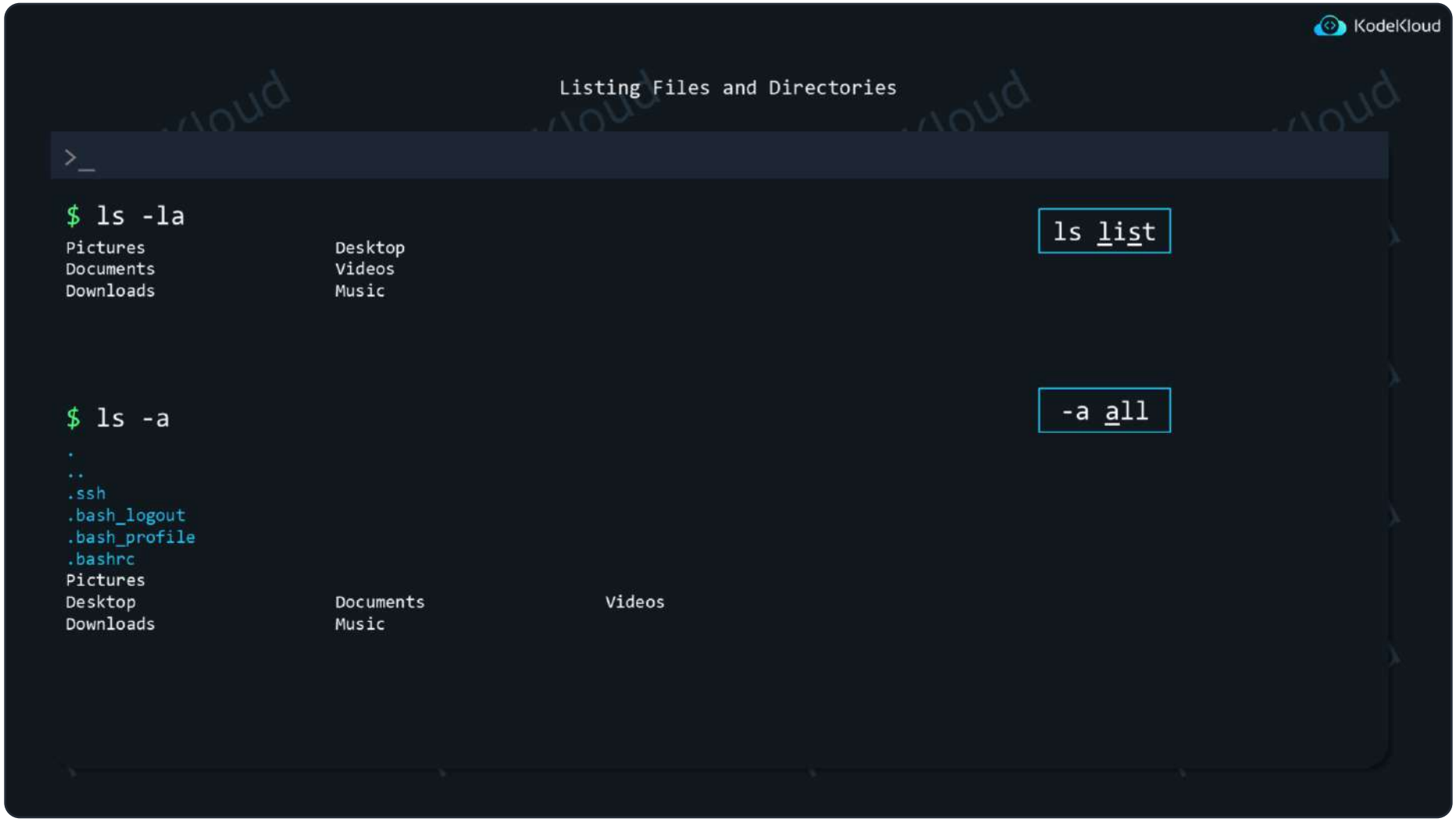
Now we'll look at how to create, delete, copy, and move files and directories in Linux.

Before we dive into this lesson, we need to understand a few basic things:

What is a filesystem tree?

What is an absolute path?

What is a relative path?

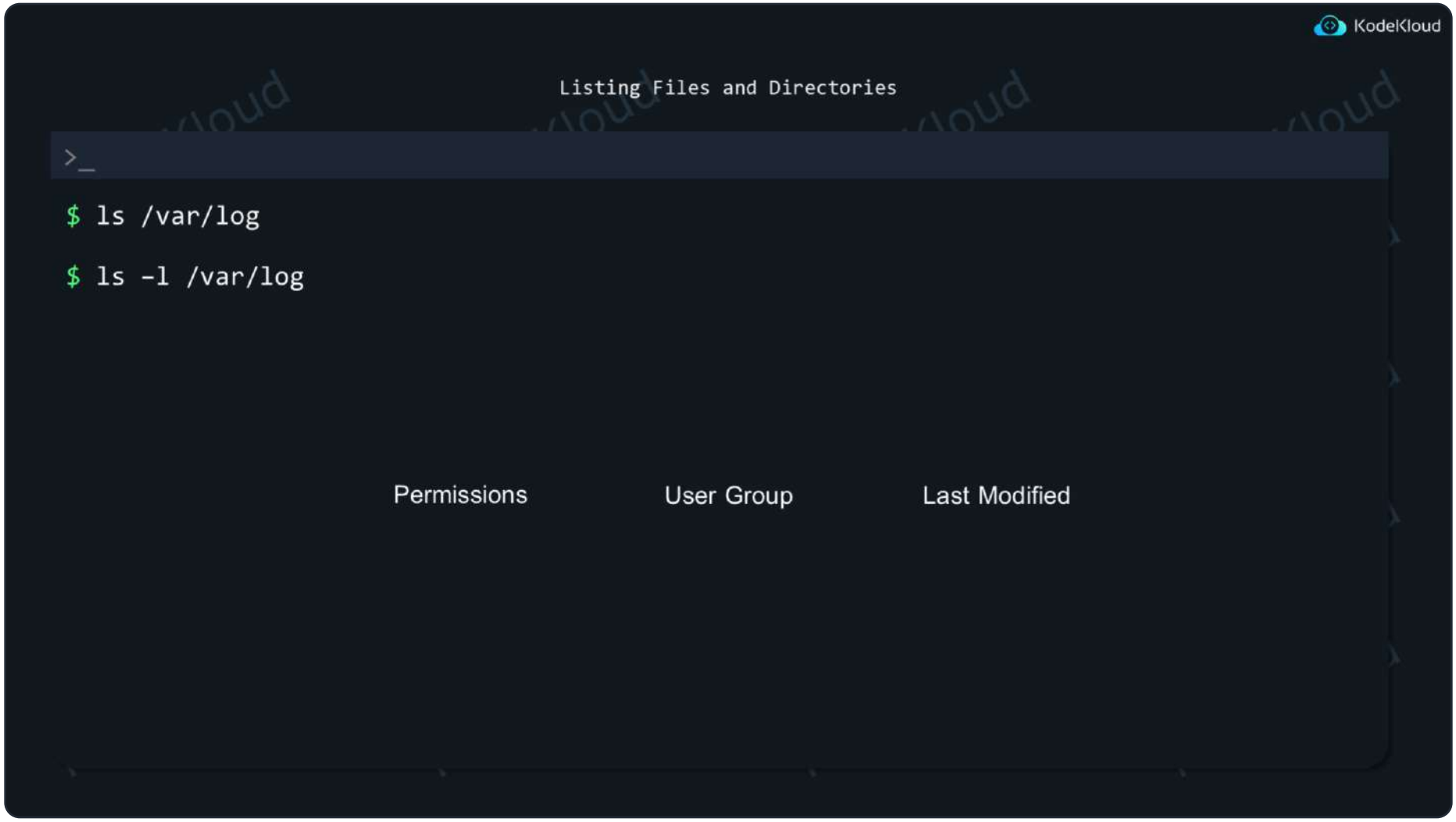


To list files and directories in your current (working) directory, we use the ls command in Linux. Using ls in your home directory might look like this:

ls comes from list.

On Linux, files and directories can have a name that begins with a . Example: the ".ssh" directory. These won't be displayed by a simple ls command. They are, in a way, hidden.

To list all files and directories, even the ones beginning with a ., use ls –a (the –a flag comes from the word all.)

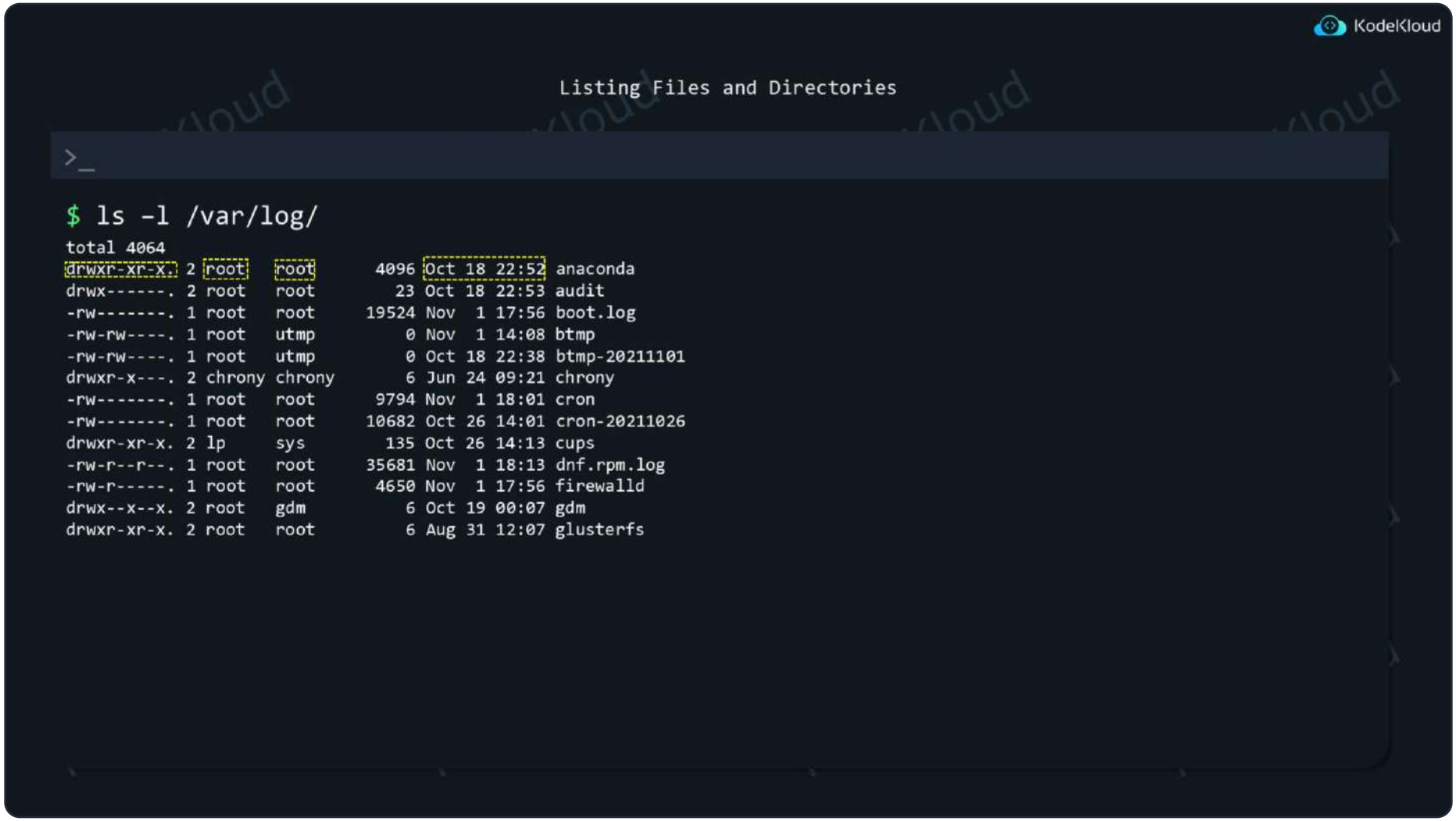


To list files and directories in your current (working) directory, we use the ls command in Linux. Using ls in your home directory might look like this:

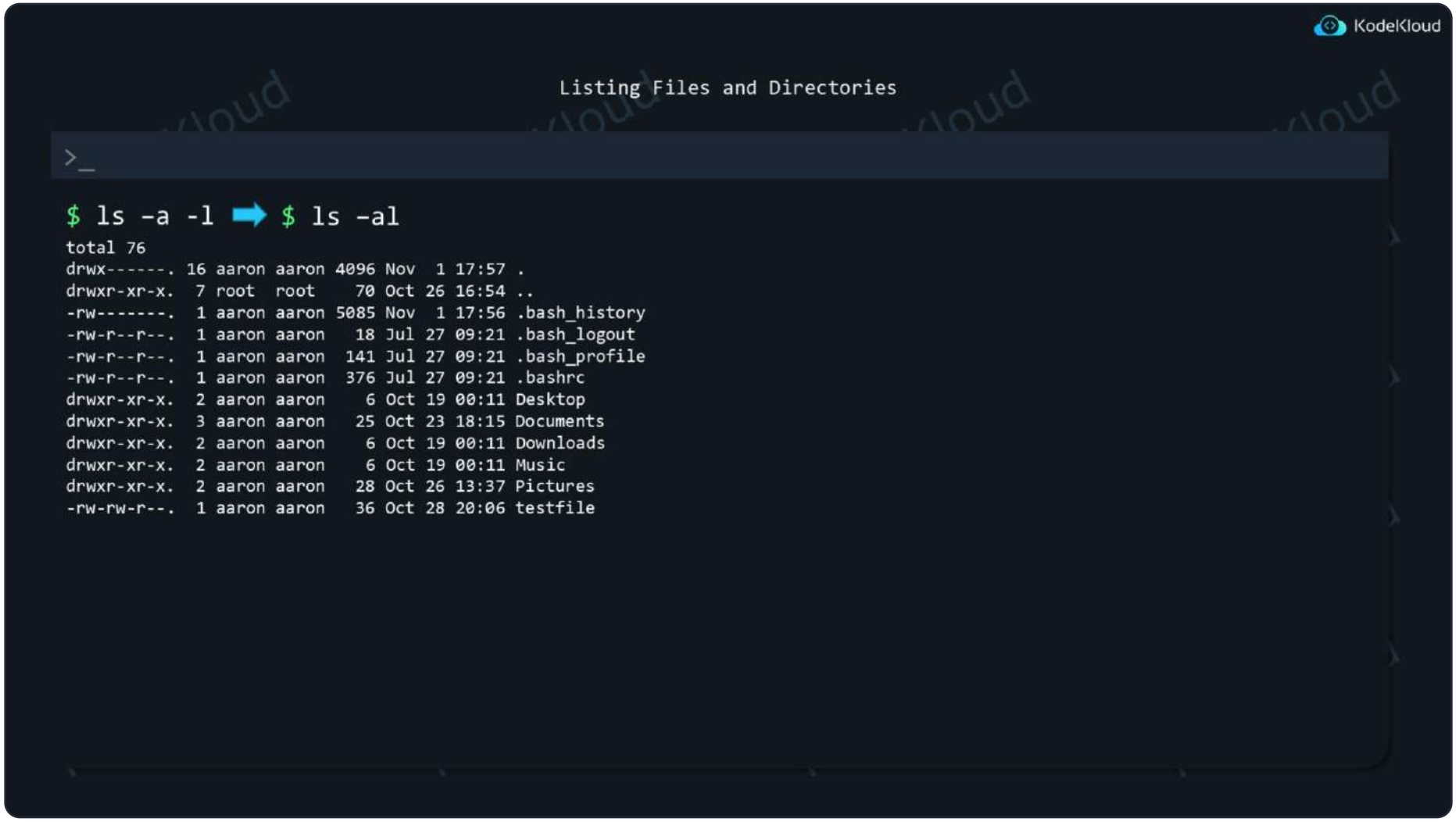
ls comes from list.

On Linux, files and directories can have a name that begins with a . Example: the ".ssh" directory. These won't be displayed by a simple ls command. They are, in a way, hidden.

To list all files and directories, even the ones beginning with a ., use ls –a (the –a flag comes from the word all.)



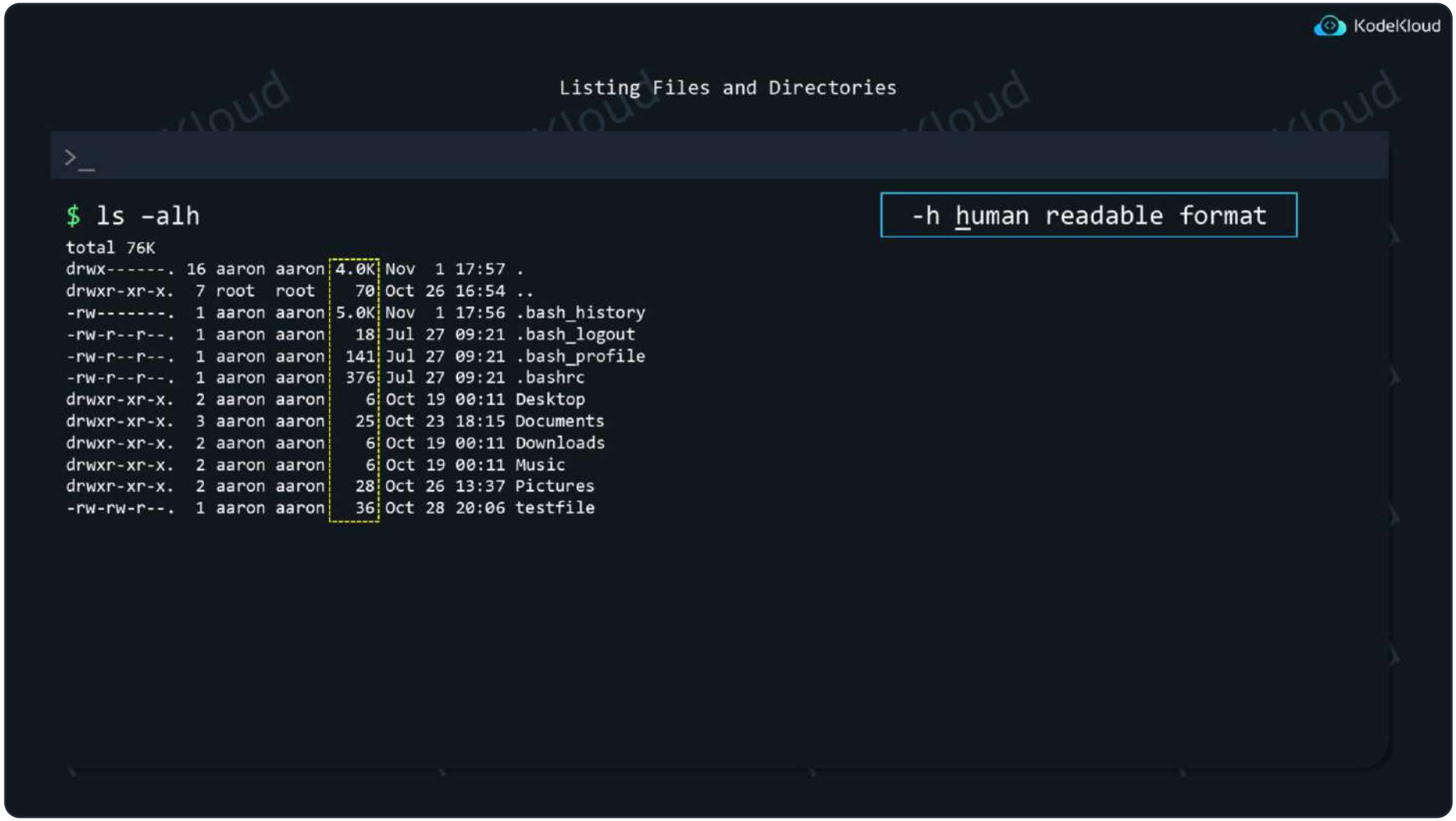
Of course, to list files and directories from a different location, we just type the directory path at the end of ls, like ls /var/log/ or ls -l /var/log/ to list files and directories in a different format, called a "long listing format," which shows us more details for each entry, like the permissions for a file or directory, what user/group owns each entry, when it was last modified.



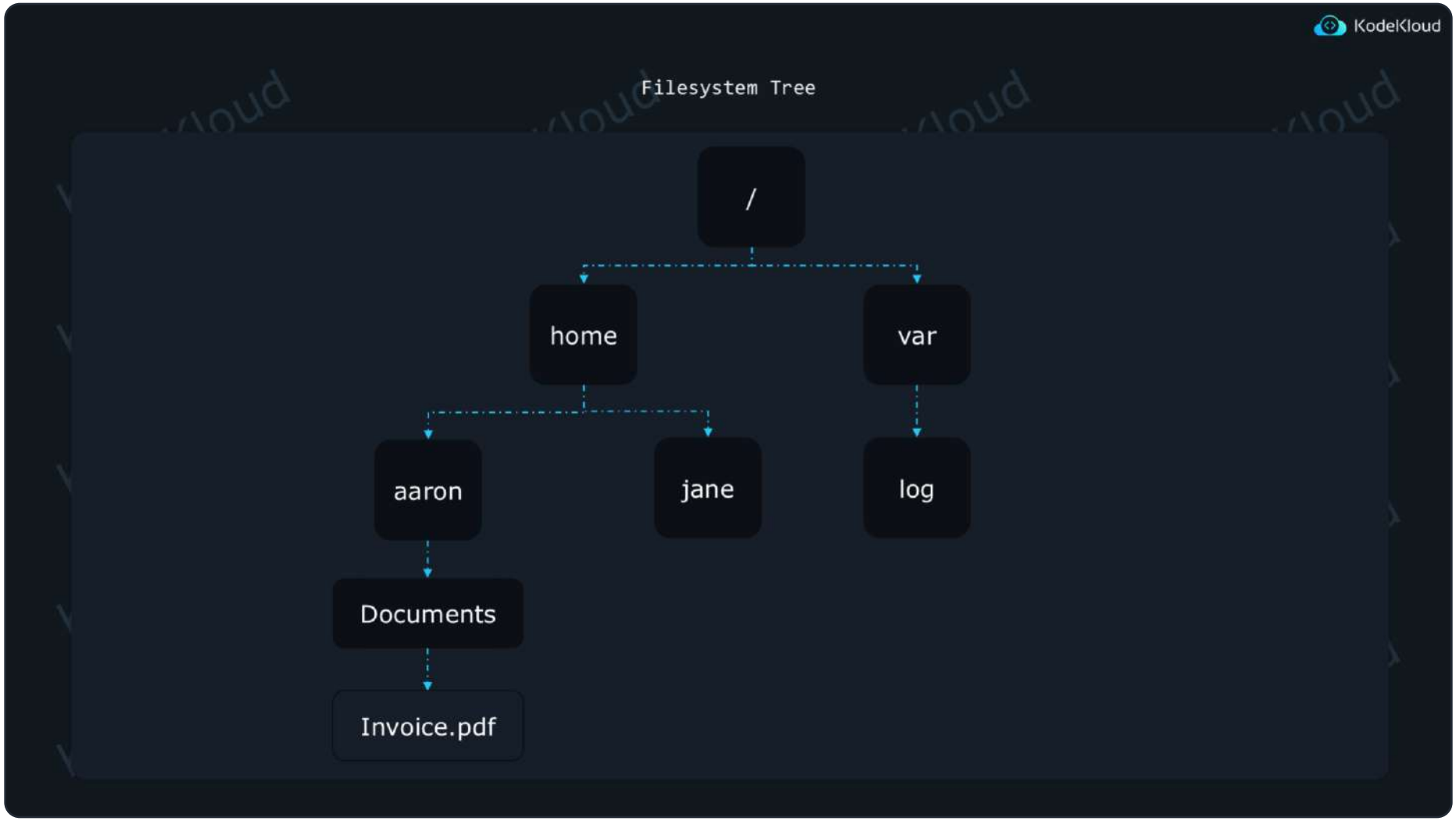
We can combine the -a and -l command line options like this:

ls -a –l or like this as ls –al.

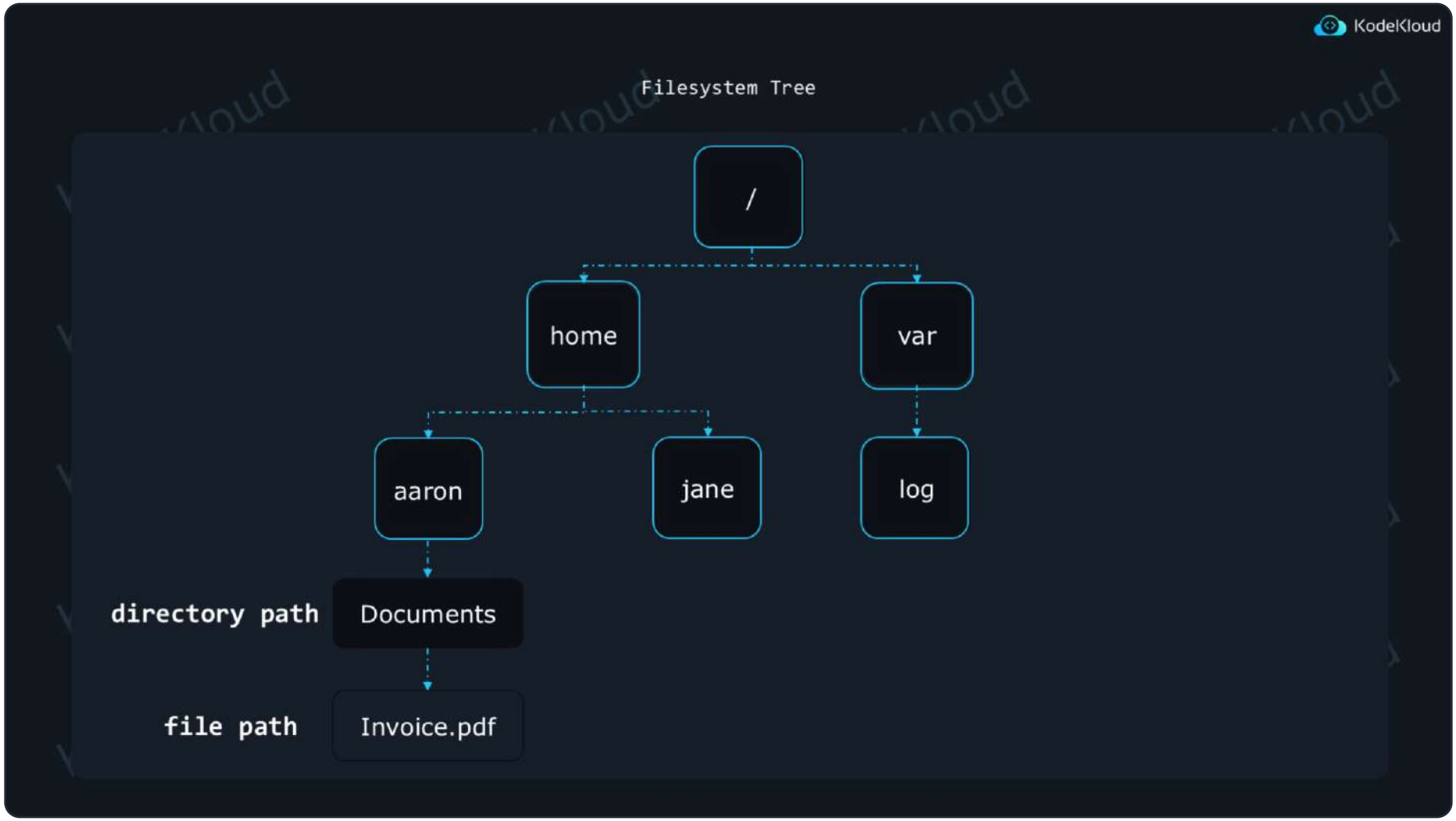
This will display entries in long listing format and also show us "pseudo-hidden" files and directories which have a name beginning with a . It doesn’t matter which order you put the flags, and you don’t have to put a – in front of each of them. However, the last form is preferred as it's faster to write it.



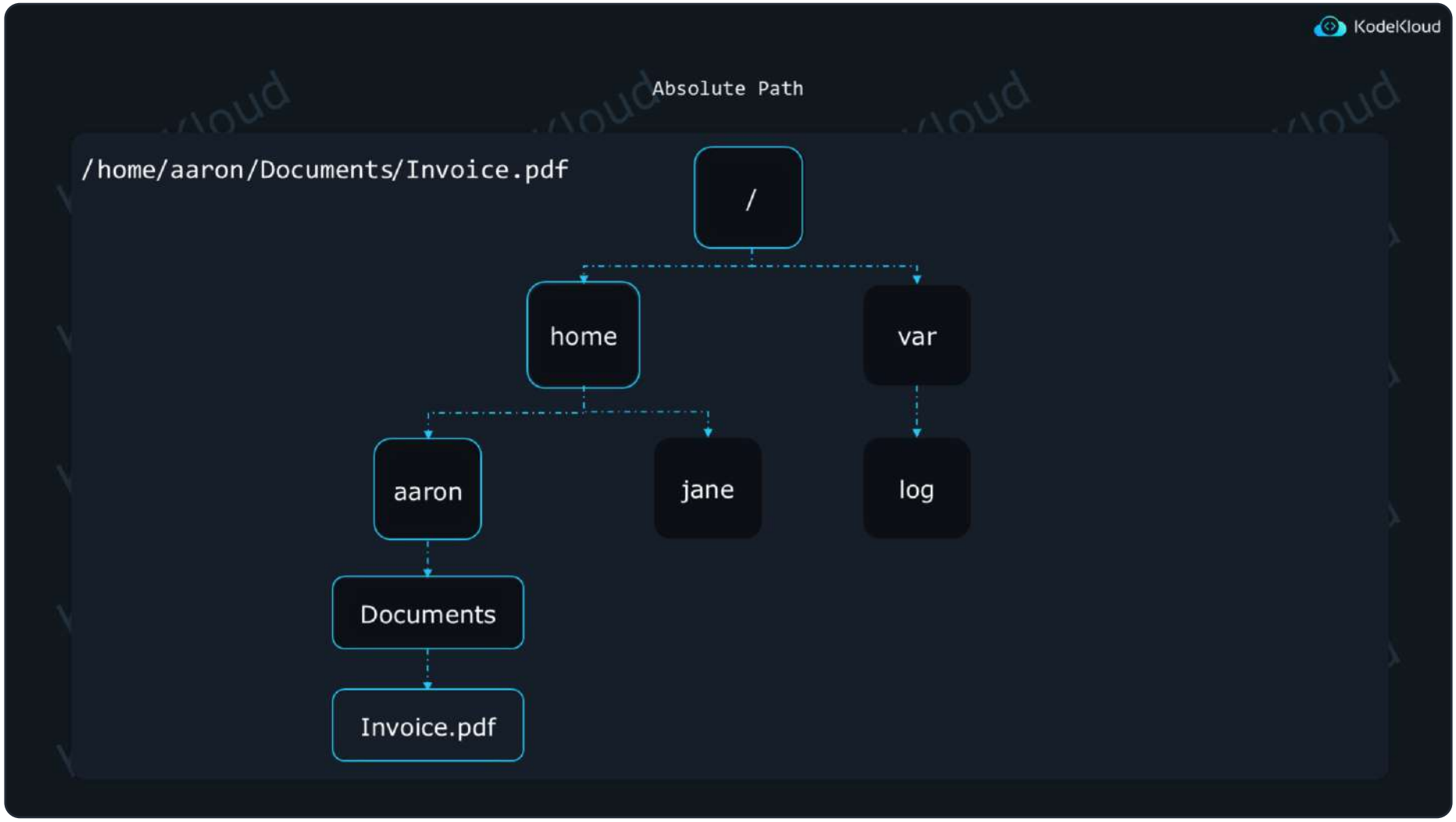
There's also a command line option, -h, that shows sizes in "human readable format": bytes, kilobytes, megabytes, and so on. This must be combined with the -l option. If we want to use three options, we could use ls –alh.



Linux organizes files and directories in what it calls the filesystem tree. Why is it called the filesystem tree? Because just like a tree we'd see in nature, this also has a root, branches and leaves. Except, Linux's filesystem tree is inverted. The root is at the top and its branches and leaves "grow" downward.



The root directory is /. This is the top-level directory, there can be no other directories above it. Under / there are a few subdirectories like home, var, etc, and so on. These subdirectories may also contain other subdirectories themselves. To access a file or directory on our command line, we must specify its file path or directory path. This path can be written in two different ways:

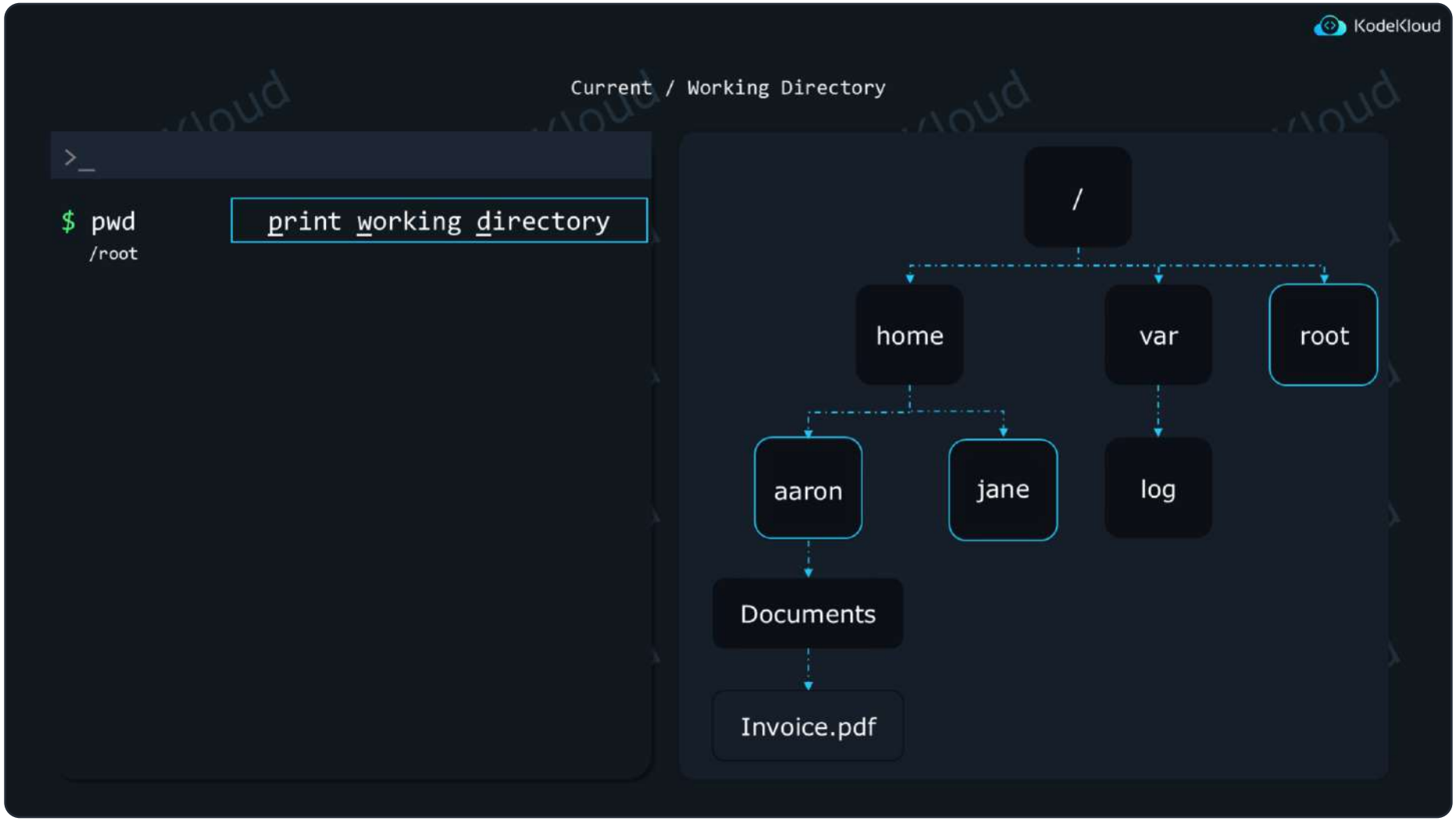


The easiest to understand is the absolute path.

/home/aaron/Documents/Invoice.pdf is an example of such a path.

Absolute paths always start out with the root directory, represented by /. Then we specify the subdirectories we want to descend into, in this case, first home, then aaron, then Documents. We can see the subdirectory names are separated by a /. And we finally get to the file we want to access, Invoice.pdf.

An absolute path can end with the name of a file, but also with the name of a directory. If we'd want to refer to the Documents directory, we'd specify a path like /home/aaron/Documents



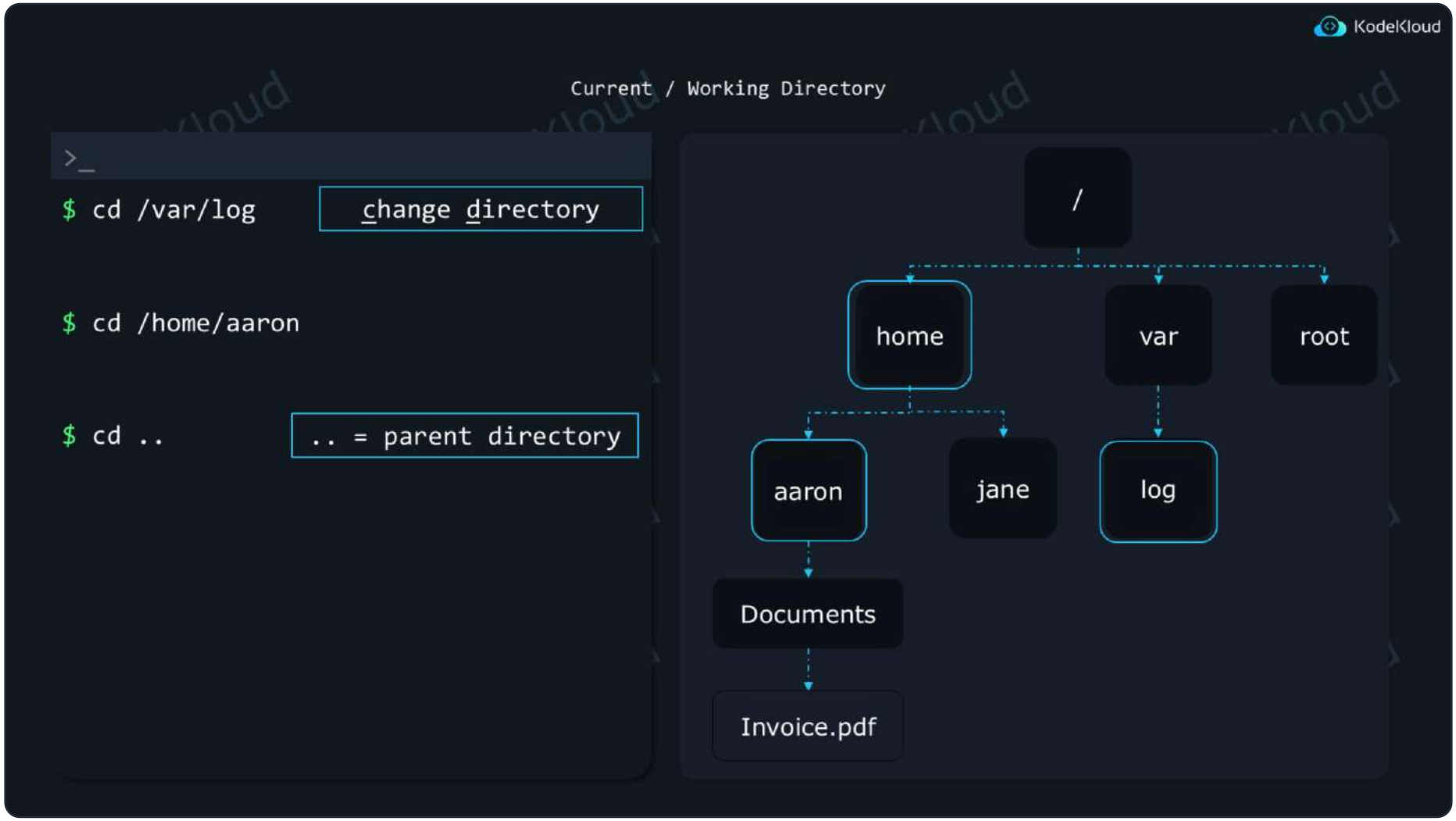
To understand a relative path, we first must explore what the current directory means. This is also called the working directory.

To see our current (working) directory we can type

pwd

pwd = Print Working Directory

When we're working at the command line, we're always "inside" a directory. For example, if we log in as the user "aaron" on some server, our starting current directory might be /home/aaron. Every user starts in its home directory when they log in. jane might have it at /home/jane, and root (the super user/administrator) has it at /root.



To change our current directory, we use the cd command (change directory).

cd /var/log

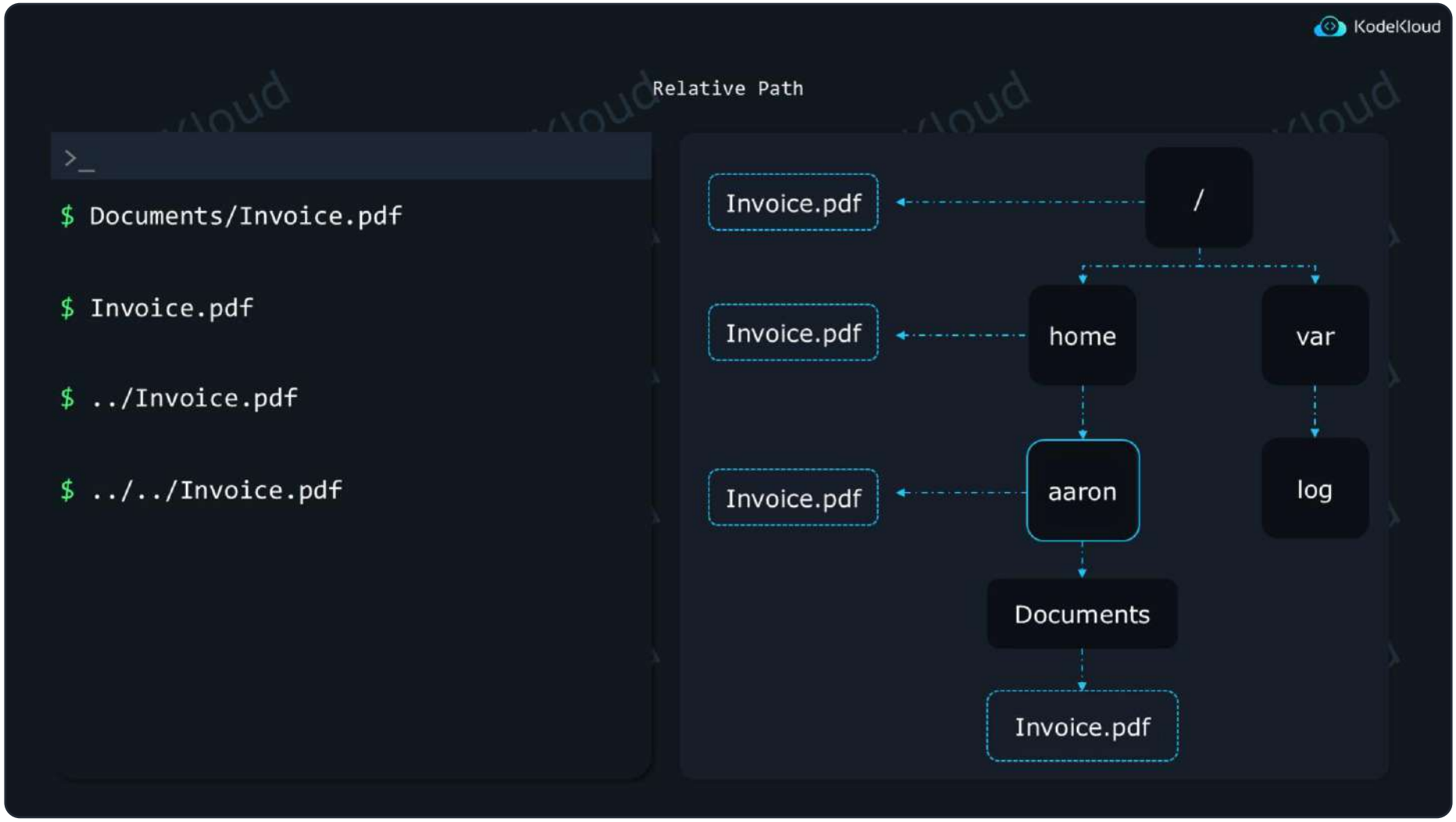
would change our current directory to /var/log. We used an absolute path here. But we can also change directory this way:

cd ..

This will take us one directory UP.

If we were cd into /home/aaron, running “cd ..” would take us into /home, which becomes the new current directory.

“..” always refers to the parent directory of our current directory. This was an example of using a very simple relative path. Let's dive deeper.



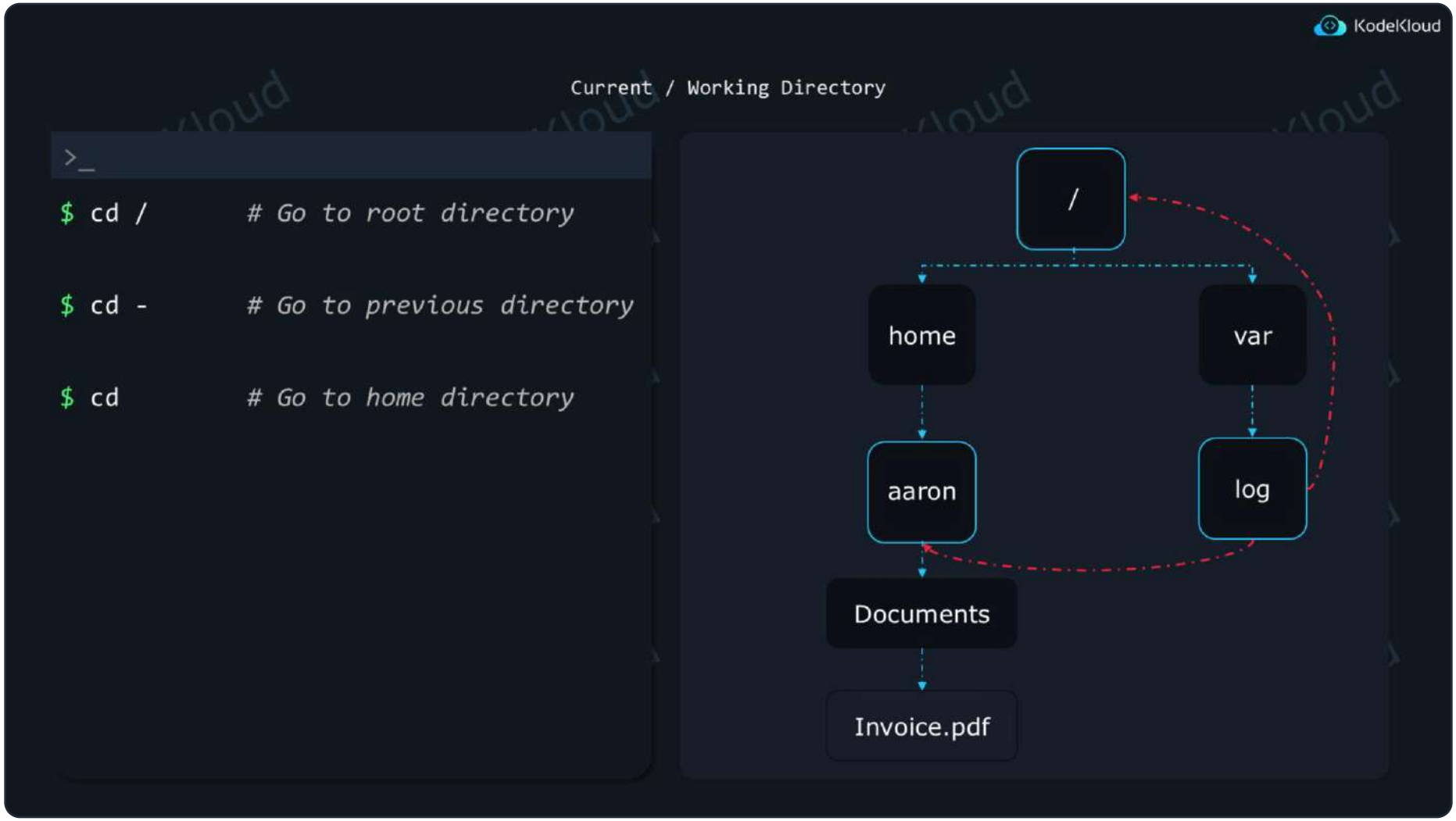
Let's imagine our current directory is /home/aaron. With relative paths we can refer to other places in one of three main ways

Locations "under" our current directory. E.g., Documents/Invoice.pdf Since we're in /home/aaron, typing a path like Documents/Invoice.pdf is like typing /home/aaron/Documents/Invoice.pdf. Our relative path "gets added" to our current directory and we get to our PDF file.

Locations in our current directory. Typing Invoice.pdf will access the file at /home/aaron/Invoice.pdf

Locations above our current directory. Typing ../Invoice.pdf points to the file at /home/Invoice.pdf. Since we used ../ we basically said, "go one directory up".

We can use .. multiple times. ../../Invoice.pdf points to the file at /Invoice.pdf. The first .. "moved" the relative path at /home, the next .. moved it at /.



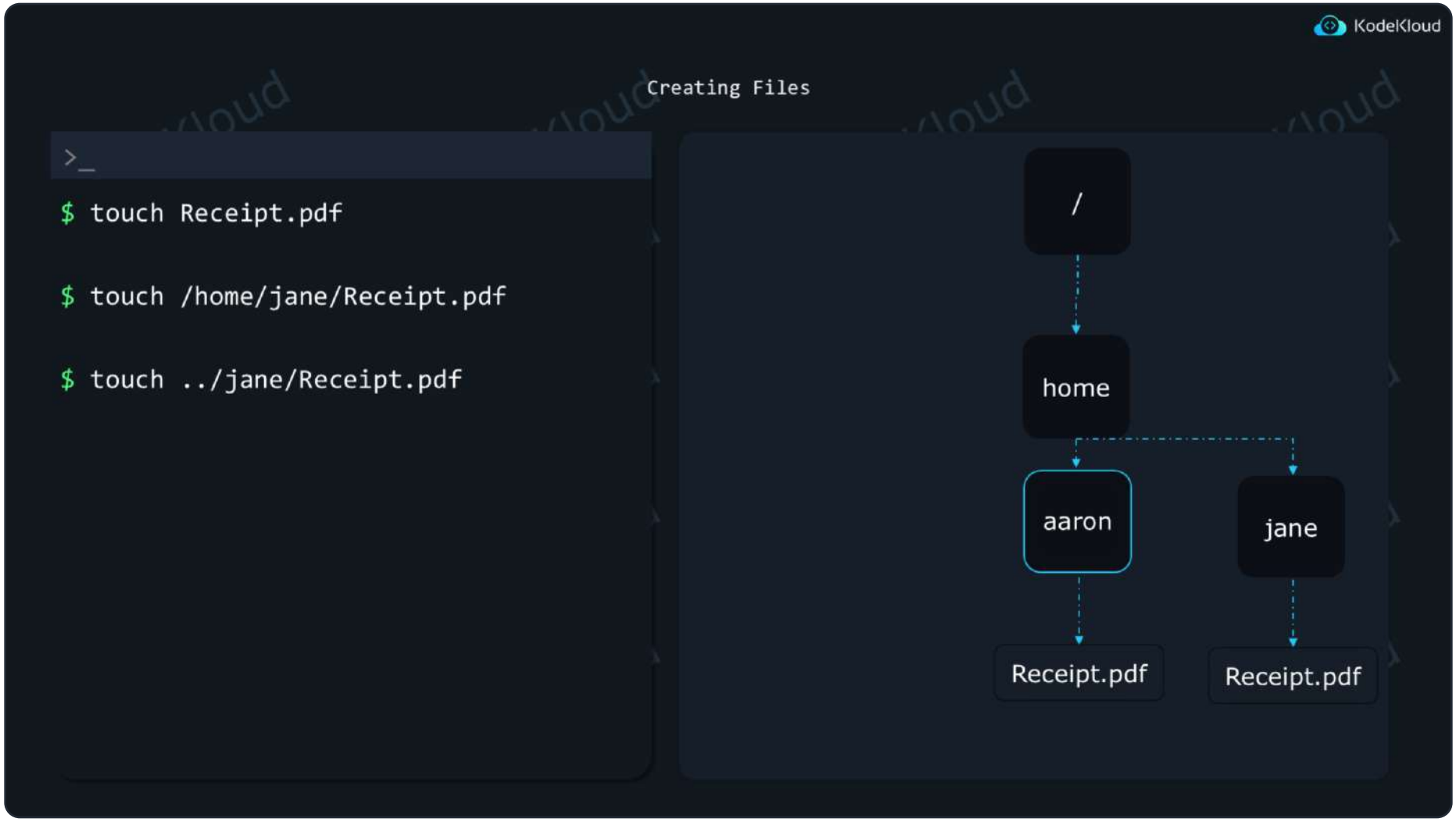
Extra tips:

If you're in /var/log currently and you move to /, you could run the command cd / and it will take you to the root directory.

You can return to your previous working directory with the cd - command. It will take you back to /var/log.

If you're in /var/log and you want to return to your home directory – in our case, /home/aaron – use cd.

cd without any options or paths after it will always take you back to the home directory.

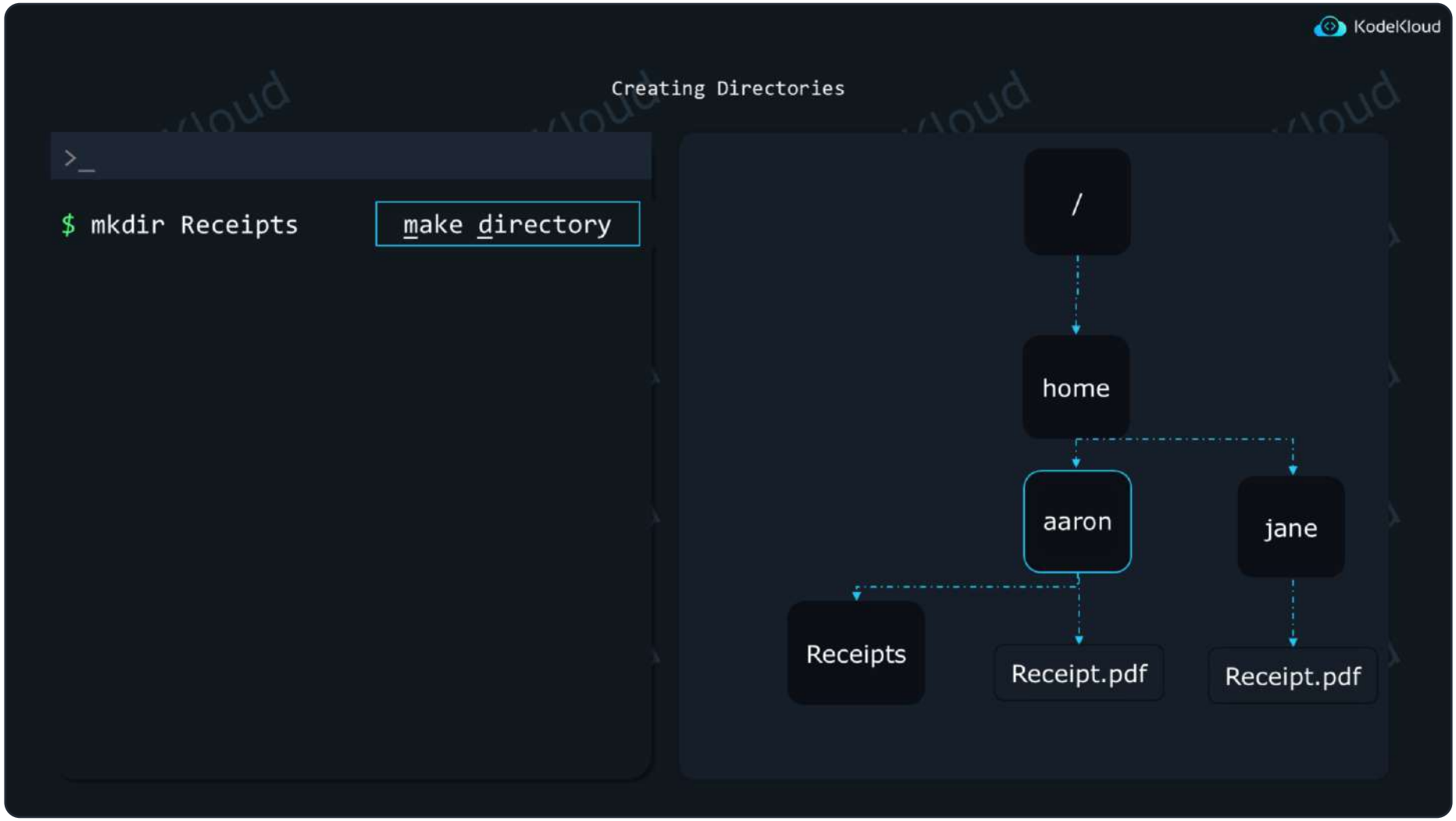


Let’s assume we’re in our home directory, and we want to create a new file. To do this, we can use touch. For example, to create a file named “Receipt.pdf,” we would type touch Receipt.pdf.

This will create it inside the current directory. To create it at another location, we could use touch /home/jane/Receipt.pdf

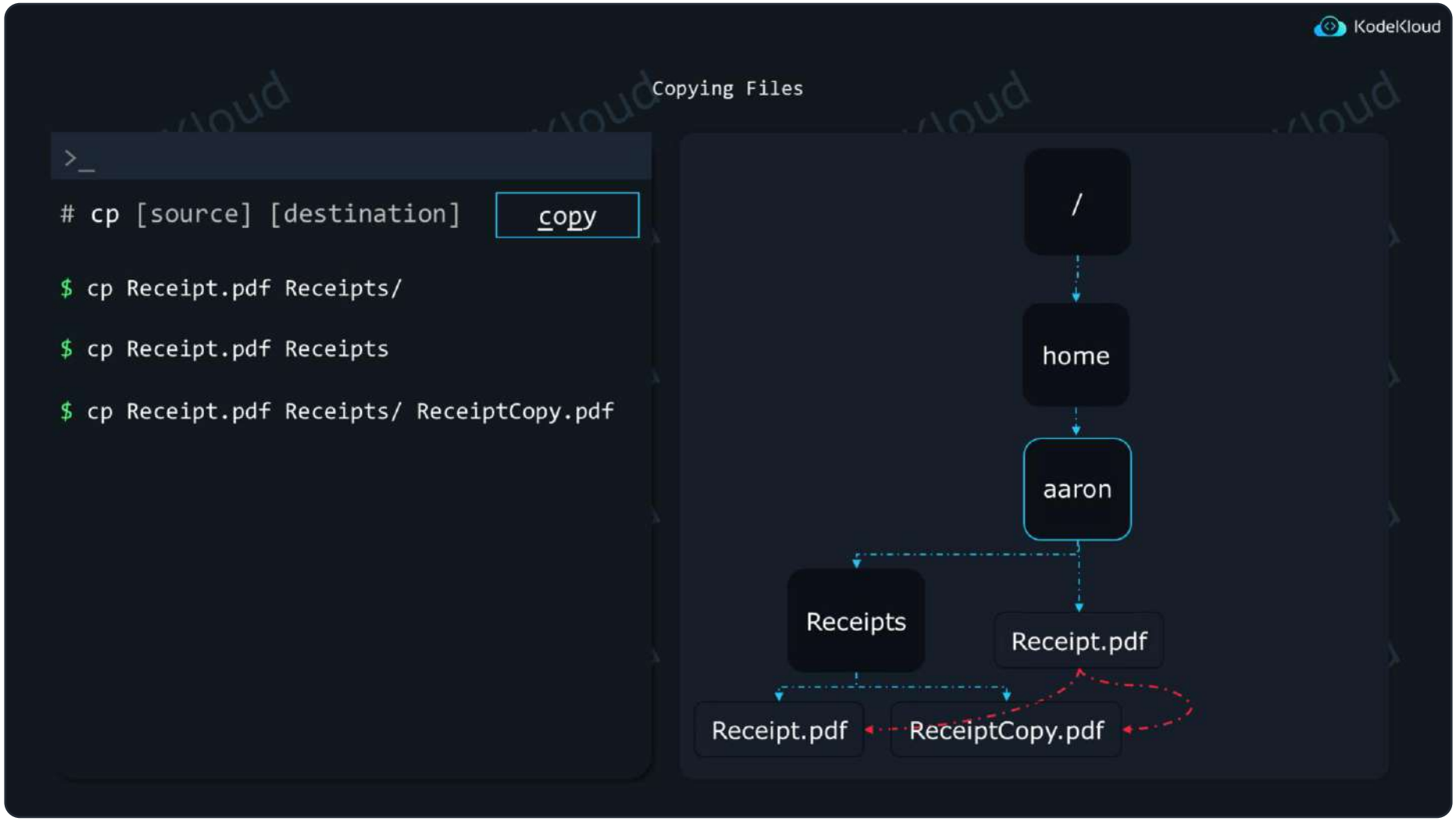
Since we're in /home/aaron, we could also use the relative path to create file in /home/jane by typing touch ../jane/Receipt.pdf.

Both commands would work the same because all the commands we'll discuss accept both absolute, and relative paths, so we won't mention these alternatives for each one. Just know that after the command, you can use any kind of path you want.



To create a new directory, use mkdir; for example: mkdir Receipts

mkdir comes from make directory



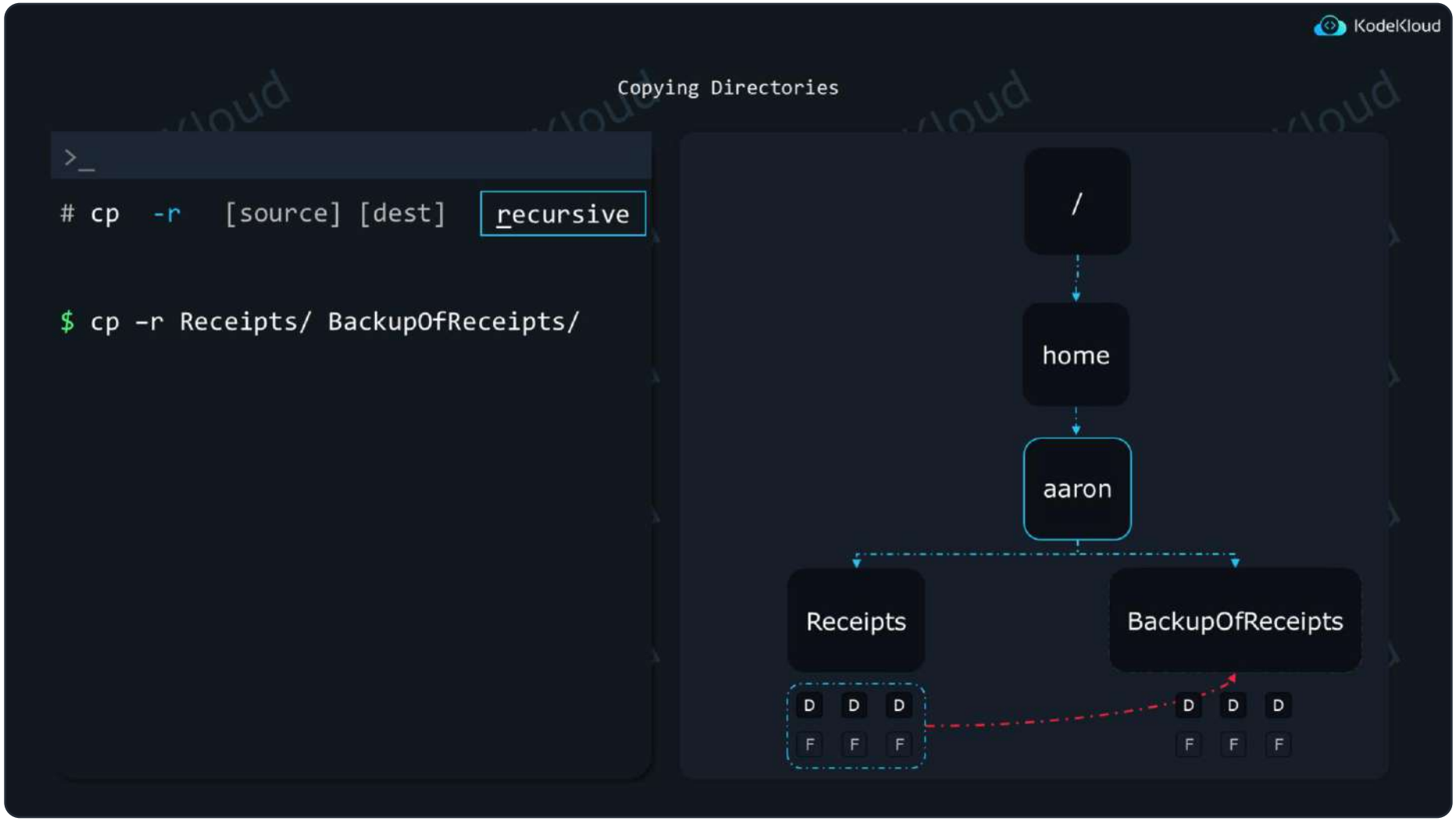
To copy a file, we use the cp command, which is short for copy. cp is followed by the path to the file we want to copy (source), then the path to the destination where we want to copy it. "cp source destination"

To copy Receipt.pdf to the Receipts directory, we'd use cp Receipt.pdf Receipts/

Notice how we terminated the path to the Receipts directory with a /, to make it Receipts/? Without the / would have worked too. But it's good practice to end your directories with a /. This way,

you'll form a healthy habit and get a visual indicator that tells you when Receipts (without /) might be a file, and Receipts/ might be a directory.

To copy Receipt.pdf to the Receipts directory, but also choose a new name for it, we could use cp Receipt.pdf Receipts/ReceiptCopy.pdf.



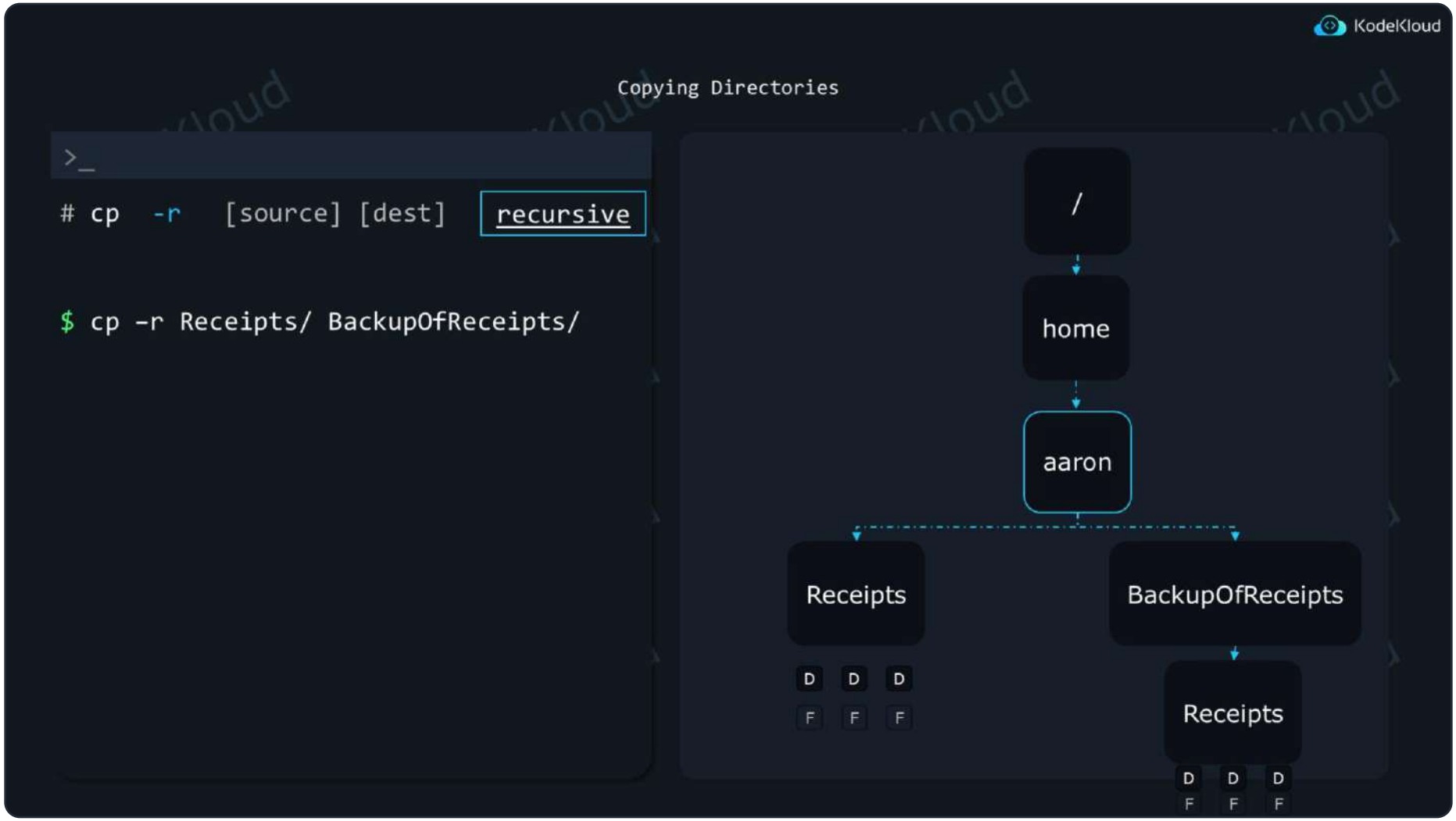
To copy a directory and all its contents to another directory run the cp command as before but with the –r option.

The -r is a command line option (also called command line flag) that tells cp to copy recursively. That means, the directory itself, but also descend into the directory and copy everything else it contains, files, other subdirectories it may have, and so on.

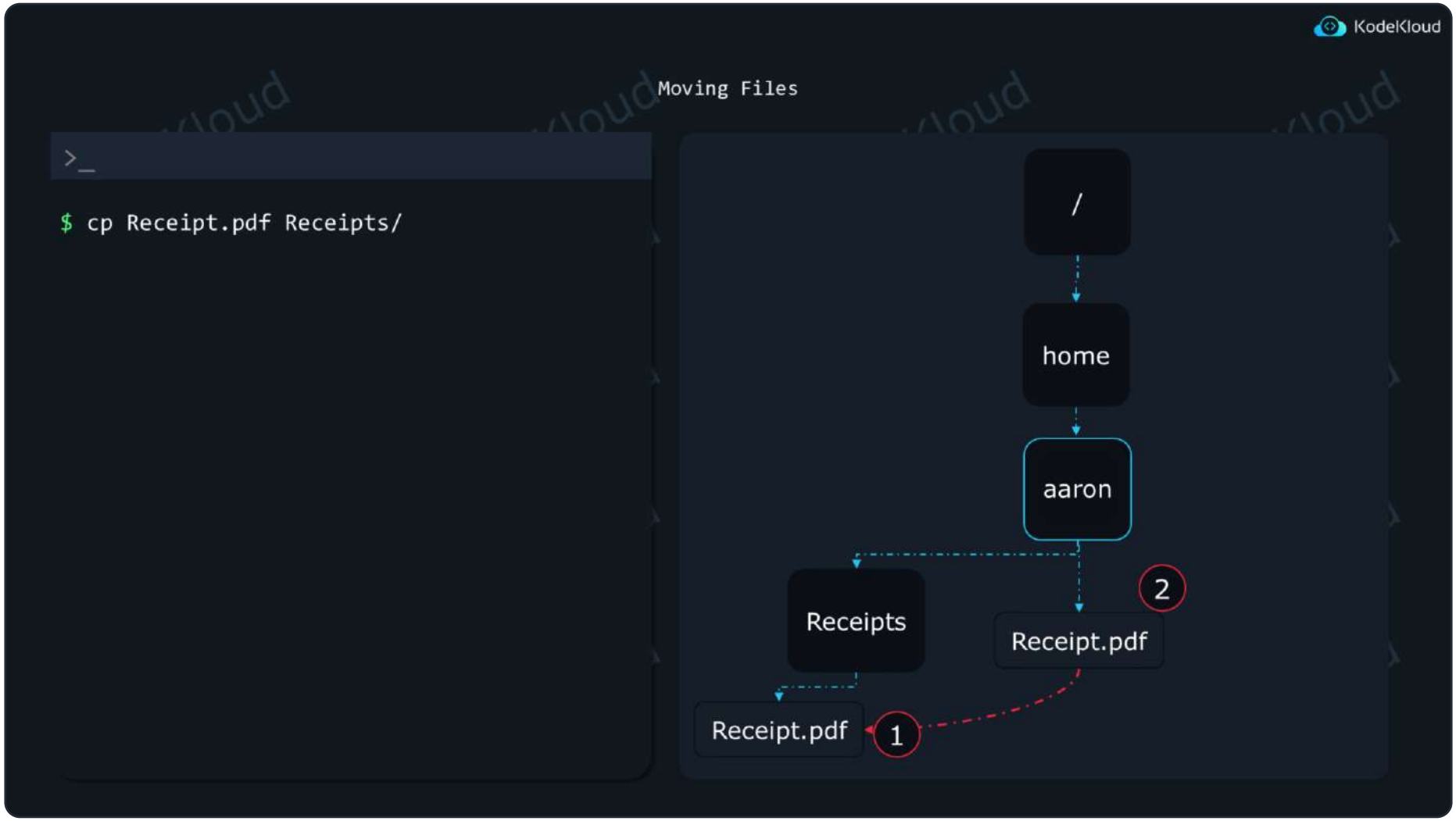
For example, say I have a lot of directories, subdirectories and files under the receipts directory. And I'd like to back up all the contents into a backup directory named BackupOfReceipts.

Run the command – cp –r Receipts/ BackupOfReceipts/

This copies all subdirectories and files from the receipts folder into the backpupofreceipts folder.

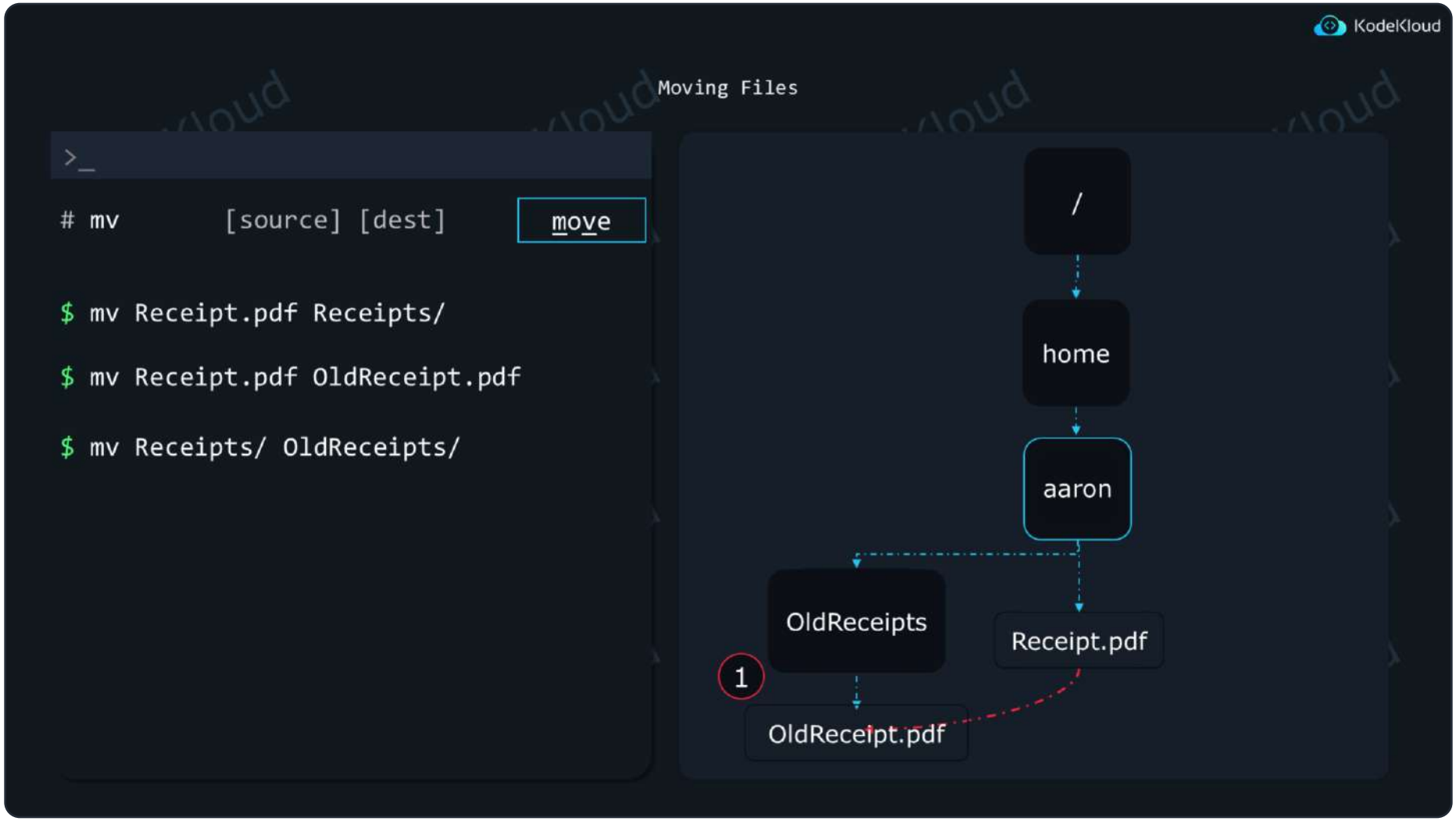


The name you choose for your cloned directory must not exist at your destination. For example, if we'd already have a directory at /home/aaron/BackupOfReceipts, this will just move Receipts there and it would end up at Documents/BackupOfReceipts/Receipts/.



So, we saw that the copy operation copies a file from one place to another, resulting in 2 copies of files – the original one and the new one in the new location.

But what if we want to move a file from one location to another? So that the file is not present in the original location but is only present in the new location?



For this use the mv command. Mv stands for move.

Run the command mv Receipt.pdf Receipts/ to move the file from Receipt.pdf to the Receipts folder. The file is moved and there is only 1 copy of file available.

To rename a file, we can use: mv Receipt.pdf OldReceipt.pdf

To rename a directory, we can use the new name as the destination, such as: mv Receipts/ OldReceipts/.

Notice that we did not have to use the -r flag with mv to recursively work with directories? Mv takes care of that for us.

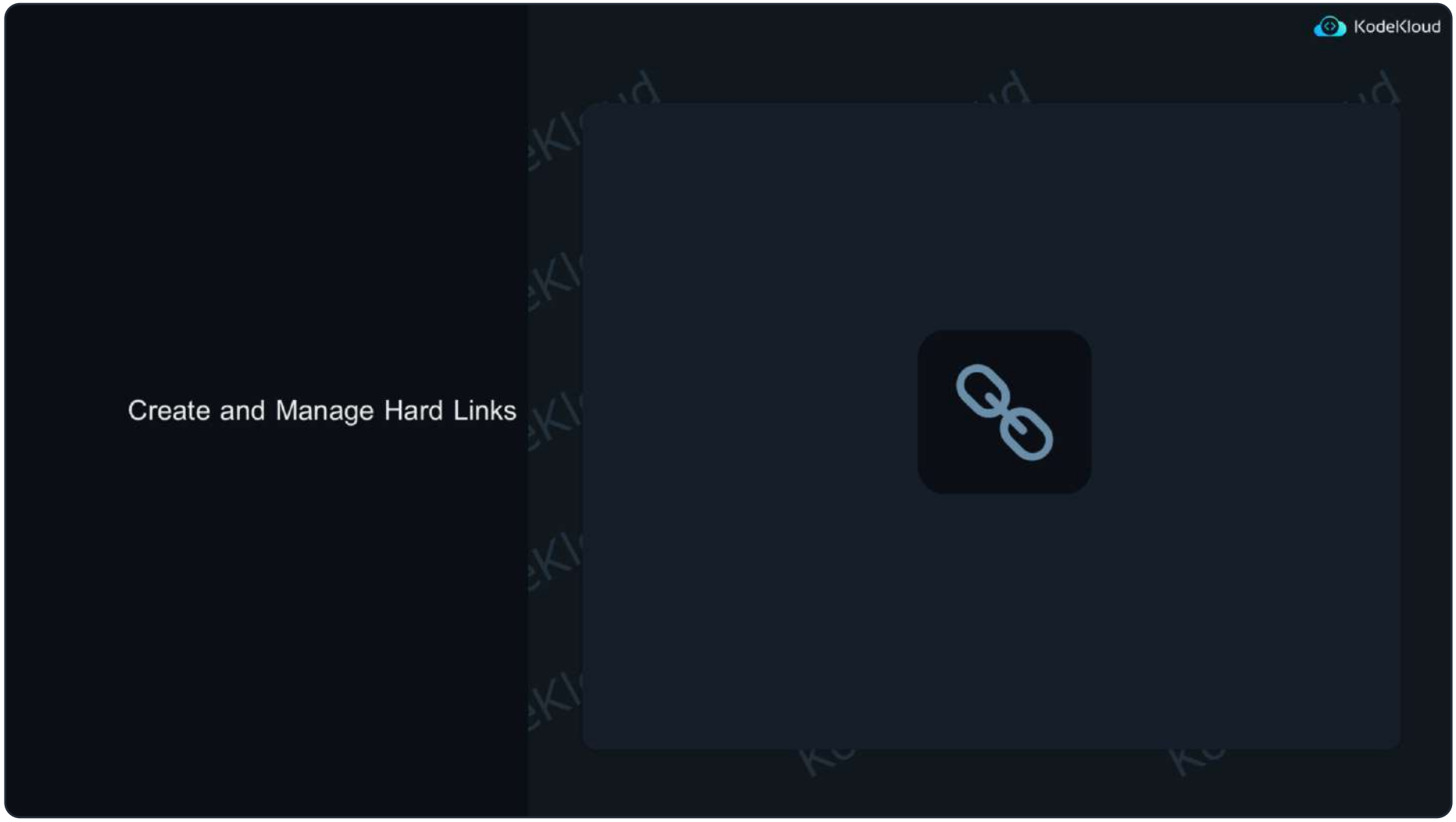
To delete a file, we use the rm command. rm comes from remove. To delete the file Invoice.pdf, we can use rm Invoice.pdf

To delete a directory like the Invoices directory, we would use : rm -r Invoices/

Once again, the -r option was used to do this recursively, deleting the directory, along with its subdirectories and files. When you copy or delete directories, remember to always add the -r option.

KodeKloud

Visit www.kodekloud.com to discover more.



In this lecture, we'll look at how Linux manages hard links.

Aaron

Username Aaron

Password **********

Same Computer

Different Desktops

Different Program Settings

Jane

Username Jane

Password **********

Aaron

/ home / aaron / pictures / family_dog.jpg

To understand hard links and soft links we first must learn some very basic things about filesystems.

Let's imagine a Linux computer is shared by two users: aaron and jane. Aaron logs in with his own username and password, Jane logs in with her own username and password. This lets them use the same computer, but have different desktops, different program settings, and so on. Now Aaron takes a picture of the family dog and saves it into /home/aaron/Pictures/family_dog.jpg.

Let's simulate a file like this.

echo "Picture of Milo the dog" > Pictures/family_dog.jpg

With this, we created a file at Pictures/family_dog.jpg and stored the text "Picture of Milo the dog" inside.

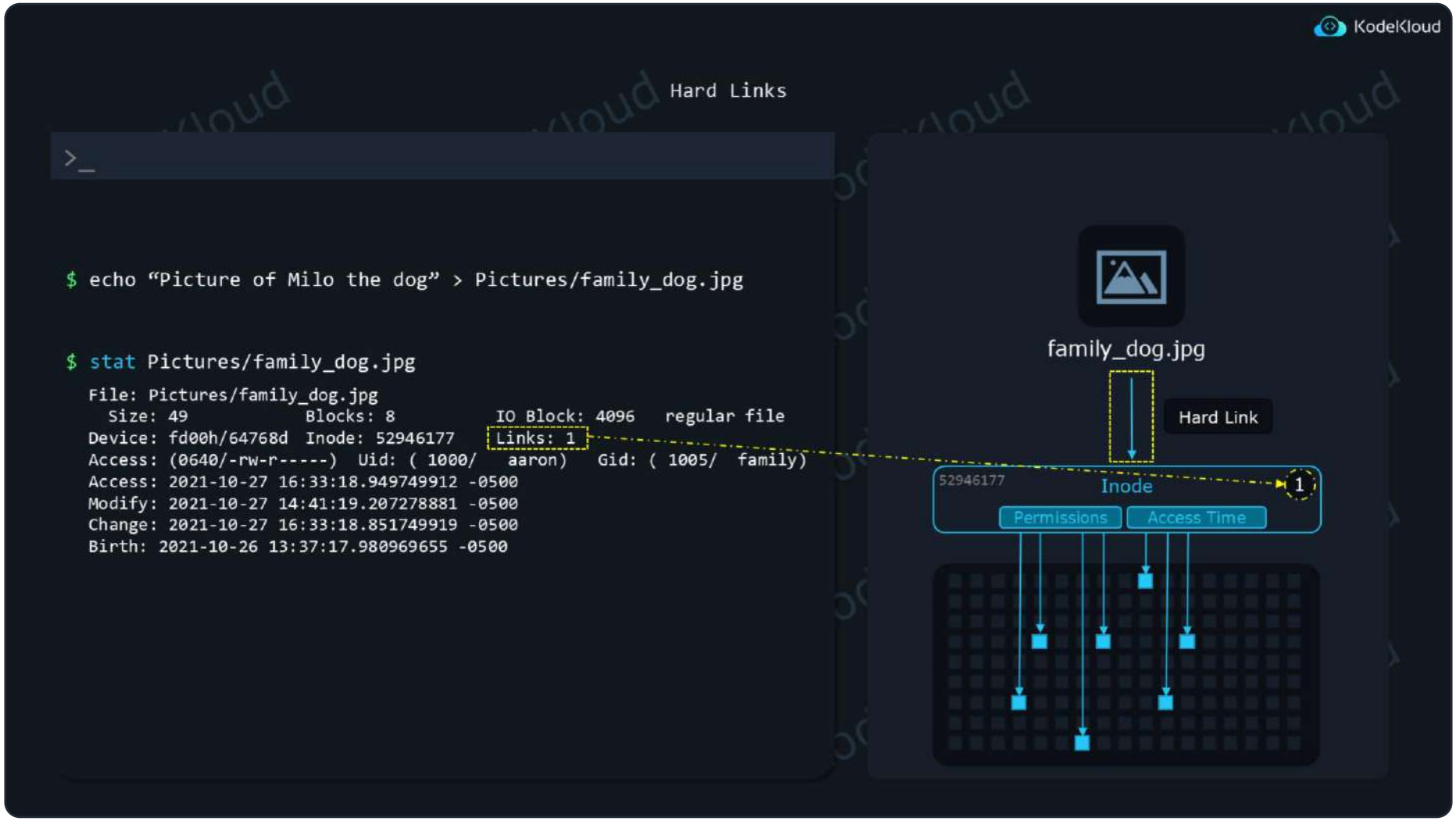
There's a command on Linux that lets us see some interesting things about files and directories.

stat Pictures/family_dog.jpg

We'll notice an Inode number. What is this?

Filesystems like xfs, ext4, and others, keep track of data with the help of inodes. Our picture might have blocks of data scattered all over the disk, but the inode remembers where all the pieces are stored. It also keeps track of metadata: things like permissions, when this data was last modified, last accessed, and so on. But it would be inconvenient to tell your computer, "Hey, show me inode 52946177". So, we work with files instead, the one called family_dog.jpg in this case. The file points to the inode, and the inode points to all the blocks of data that we require.

And we finally get to what interests us here.



We notice this in the output of our stat command.

There's already one link to our Inode? Yes, there is. When we create a file, something like this happens:

We tell Linux, "Hey save this data under this filename: family_dog.jpg"
Linux says: "Ok, we'll group all this file's data under inode 52946177. Data blocks and inode created. We'll hardlink file "family_dog.jpg" to Inode 52946177.

Now when we want to read the file:

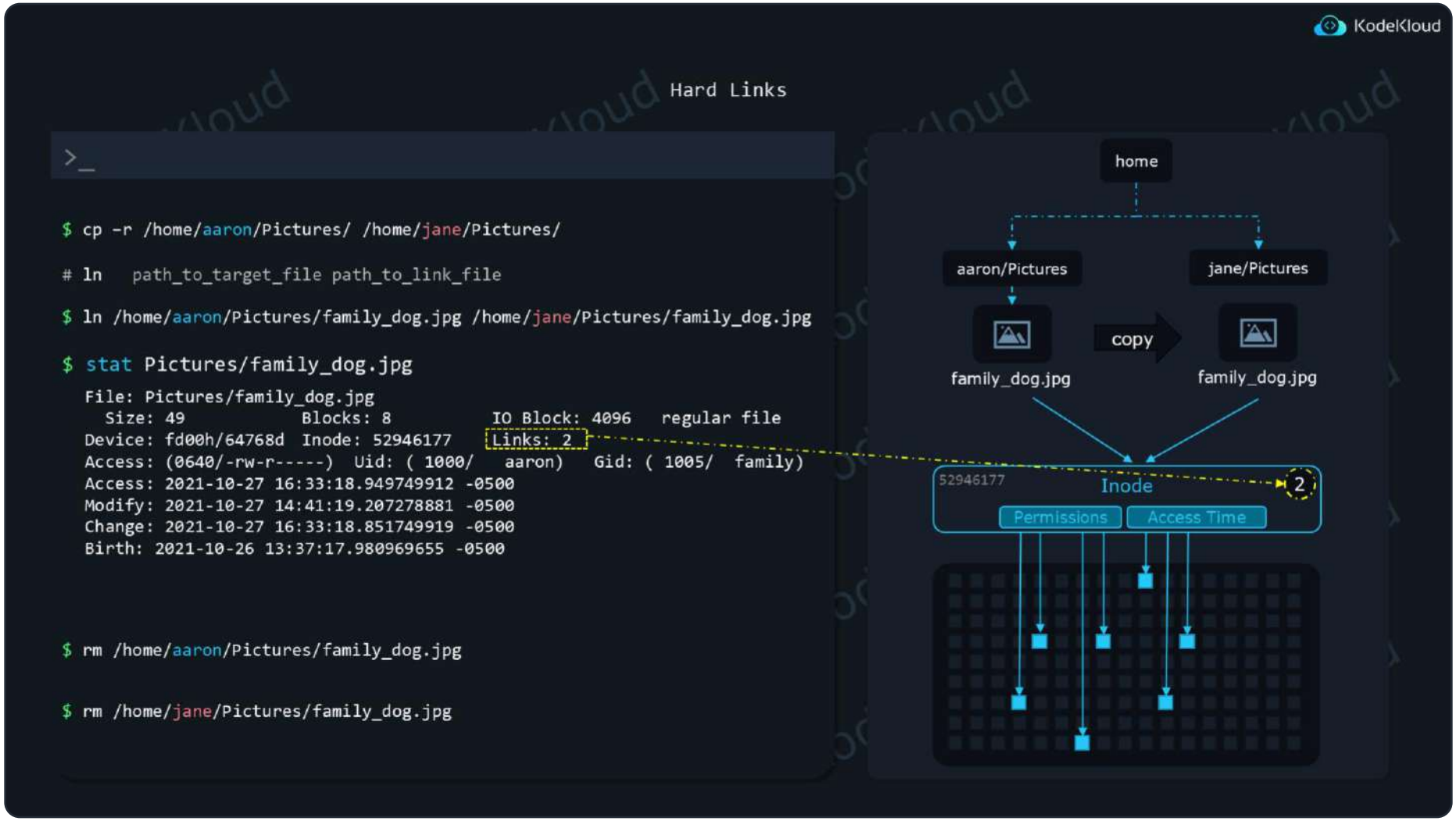
"Hey Linux, give me data for family_dog.jpg file"

And linux goes: "Ok, let me see what inode this links to. Here's all data you requested for inode 52946177"

family_dog.jpg -> Inode 52946177

So the number shown as Links in the output of the stat command is the number of hard links to this inode from files or filenames.

Easy to understand. But why would we need more than one hard link for this data?



Well, Jane has her own folder of pictures, at /home/jane/Pictures. How could Aaron share this picture with Jane? The easy answer, just copy /home/aaron/Pictures/family_dog.jpg to /home/jane/Pictures/family_dog.jpg. No problem, right? But now imagine we must do this for 5000 pictures. We would have to store 20GB of data twice. Why use 40GB of data when we could use just 20GB? So how can we do that?

Instead of copying /home/aaron/Pictures/family_dog.jpg to /home/jane/Pictures/family_dog.jpg, we could hardlink it to /home/jane/Pictures/family_dog.jpg.

The syntax of the command is:

ln path_to_target_file path_to_link_file

The target_file is the file you want to link with. The link_file is simply the name of this new hard link we create. Technically, the hard link created at the destination is a file like any other. The only special thing about it is that instead of pointing to a new inode, it points to the same inode as the target_file.

In our imaginary scenario, we would use a command like:

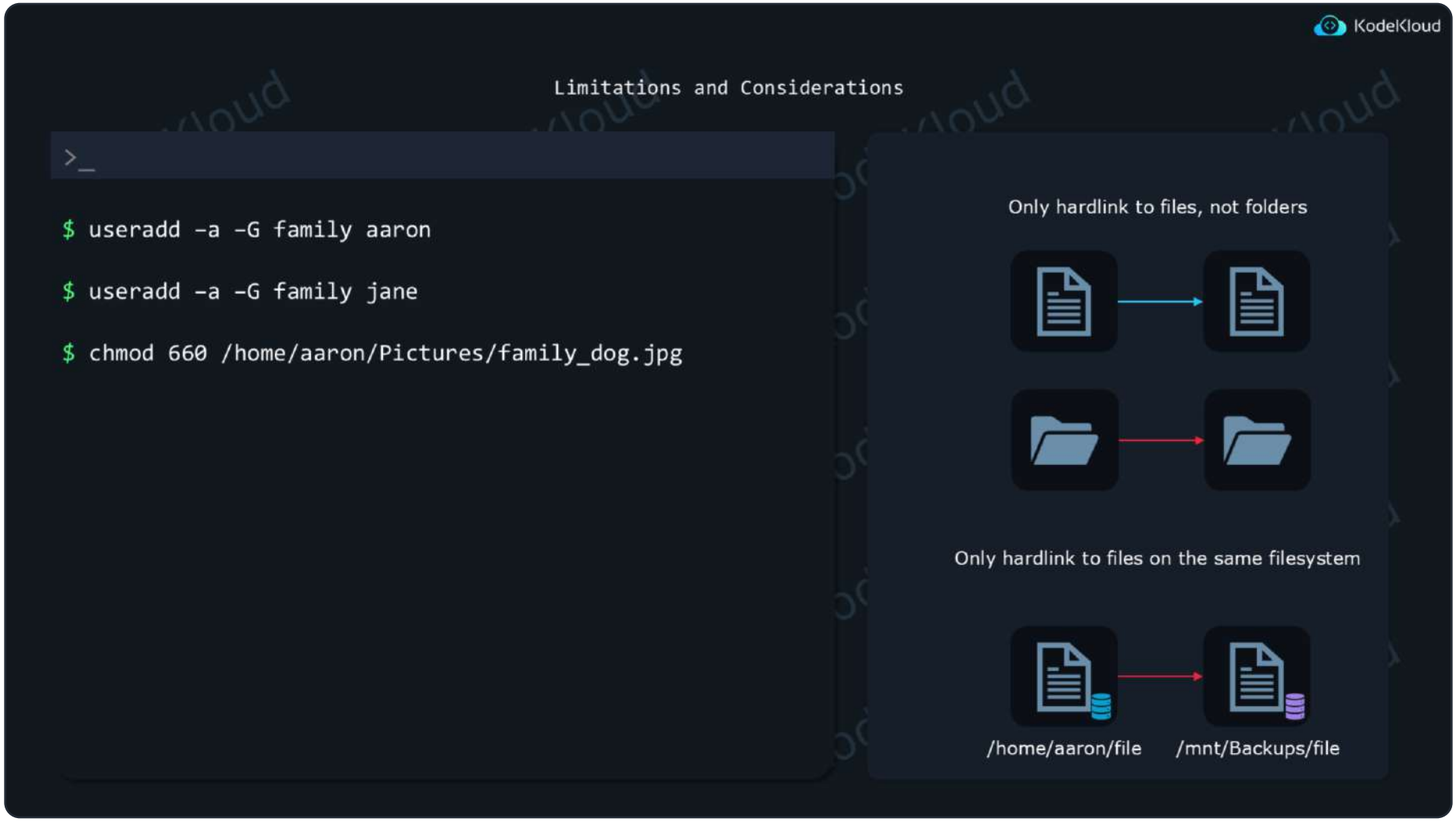
ln /home/aaron/Pictures/family_dog.jpg /home/jane/Pictures/family_dog.jpg

Now our picture is only stored once, but the same data can be accessed at different locations, through different filenames.

If we run the stat command now we see the Links are now 2. This is because this Inode now has 2 hard links pointing to it.

Another beautiful thing about hard links is this: Aaron and Jane share the same 5000 pictures through hardlinks. But maybe Aaron decides to delete his hardlink of /home/aaron/Pictures/family_dog.jpg. What will happen with Jane's picture? Nothing, she'll still have access to that data. Why? Because the inode still has 1 hard link to it (it had 2, now it has 1). But if Jane also decides to delete her hard link /home/jane/Pictures/family_dog.jpg, the inode will have 0 links to it. When there are 0 links, the data itself will disappear from the filesystem.

The beauty of this approach is that people that share hard links can freely delete what they want, without having a negative impact on other users that still need that information. But once everyone deletes their hard links to that data, the data itself will be deleted. Technically, the data blocks are not actually erased. They are just marked as unused, so the system can overwrite them with new data. But from the user's perspective the data is gone. So, it is "intelligently removed" only when EVERYONE involved decides they don't need it anymore.



Limitations of hard links:

You can only hardlink to files, not directories.

You can only hardlink to files on the same filesystem. If you had an external drive mounted at /mnt/Backups, you would not be able to hardlink a file from your SSD, at /home/aaron/file to some other file on /mnt/Backups since that's a different filesystem.

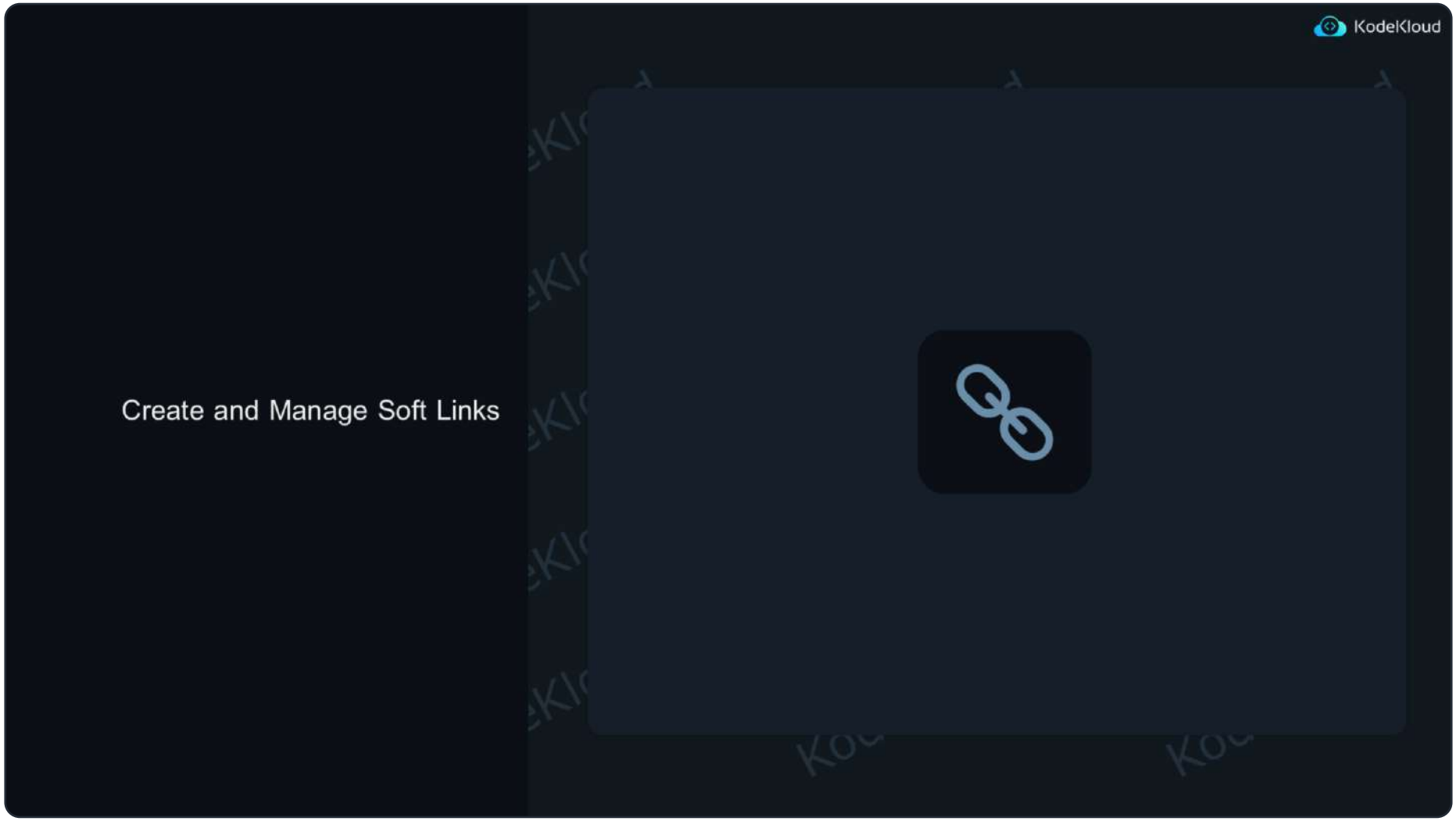
Things to take into consideration when you hardlink:

First, make sure that you have the proper permissions to create the link file at the destination. In our case, we need write permissions at: /home/jane/Pictures/.

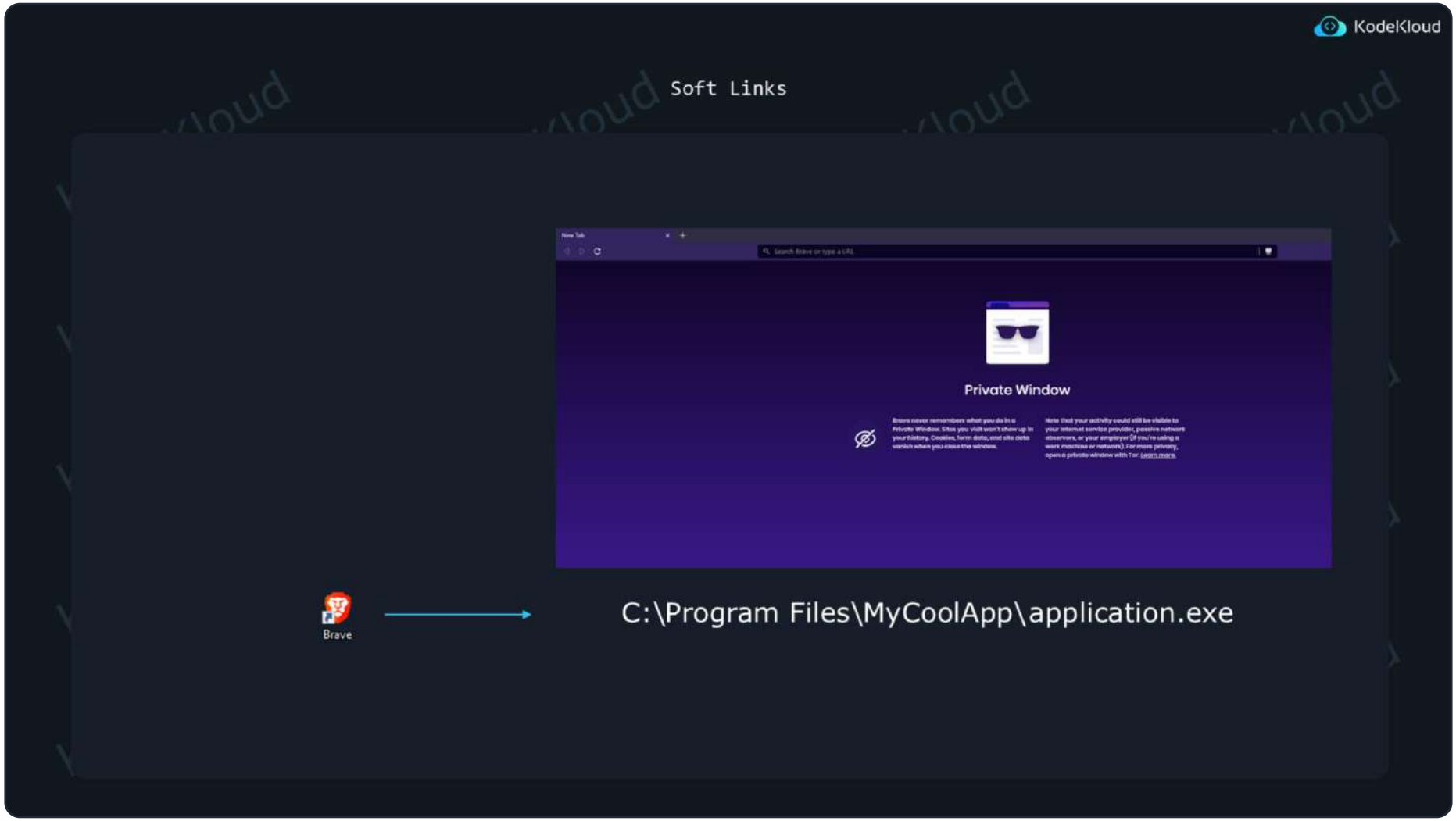
Second, when you hardlink a file, make sure that all users involved have the required permissions to access that file. For Aaron and Jane, this might mean that we might have to add both their usernames to the same group, for example, "family". Then we'd use a command to let the group called "family" read and write to this file. You only need to change permissions on one of the hardlinks. That's because you are actually changing permissions stored by the Inode. So, once you change permissions at /home/aaron/Pictures/family_dog.jpg, /home/jane/Pictures/family_dog.jpg and all other hard links will show the same new sets of permissions.

KodeKloud

Visit www.kodekloud.com to discover more.



Let's look now at how Linux manages soft links.



Know how when you install a program on Windows, you might get a shortcut on your desktop? You double click on that shortcut and that application gets launched. The application is obviously not installed on your desktop. It may have its files stored in C:\Program Files\MyCoolApp directory. And when you double click the shortcut, this only points to an executable file at C:\Program Files\MyCoolApp\application.exe. So, the double click on that shortcut basically redirects you to the file C:\Program Files\MyCoolApp\application.exe, which gets executed.

Soft links in Linux are very similar. A hard link pointed to an inode. But a soft link is nothing more than a file that points to a path instead. It's almost like a text file, with a path to a file or directory inside.

The syntax of the command to create a soft link (also called symbolic link) is the same as before, but we add the -s or --symbolic option:

ln -s path_to_target path_to_link_file

path_to_target = our soft link will point to this path (location of a file or directory)

path_to_link_file = our soft link file will be created here

For example, to create a symbolic link that points to the Pictures/family_dog.jpg file, we can run the command:

ln -s Pictures/family_dog.jpg family_dog_shortcut.jpg

Now if we list files and directories in long listing format with the ls –l command, we'll see an output like this:

The l at the beginning shows us that this is a soft link. And ls -l even displays the path that the soft link points to.

If this path is long, ls -l might not show the entire path. An alternative command to see the path stored in a soft link is:

readlink path_to_soft_link

So, in our case, it would be:

readlink family_dog_shortcut.jpg

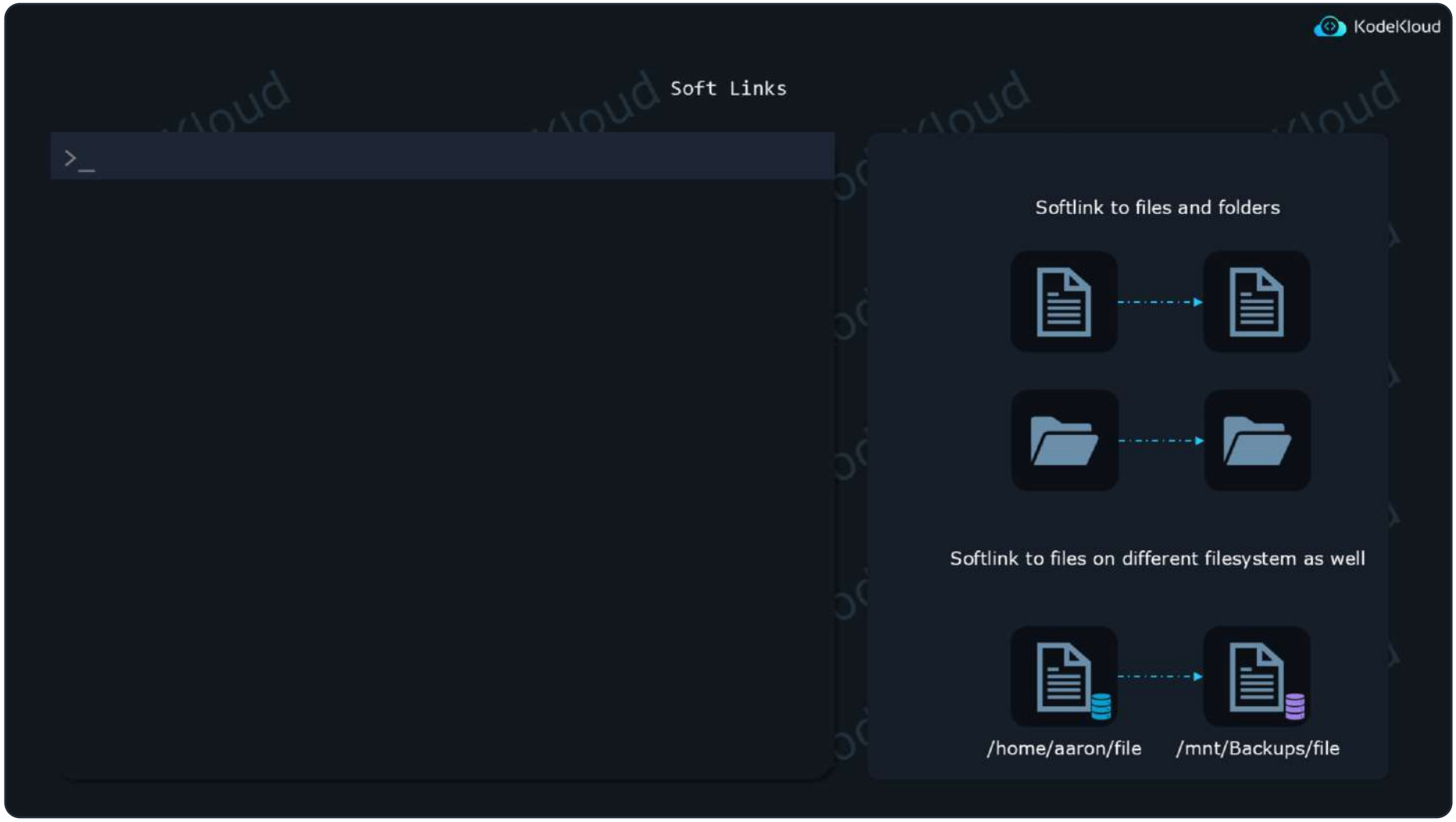
You may also notice that all permission bits, rwx (read, write, execute) seem to be enabled for this file. That's because the permissions of the soft link do not matter. If you'd try to write to "fstab_shortcut", this would be denied because the permissions of the destination file apply and /etc/fstab does not allow regular users to write here.

In our first command we used an absolute path - /home/aaron/Pictures/family_dog.jpg.

if we ever change the directory name "aaron" in the future, to something else, this soft link will break. You can see a broken link highlighted in red in the output of the ls –l command.

To tackle this you could create a soft link with a relative path. Say for example you were in the home directory of aaron, you could create a soft link using the relative path of the family_dog file instead of specifying the complete path.

When someone tries to read relative_picture_shortcut, they get redirected to Pictures/family_dog.jpg, relative to the directory where the soft link is.



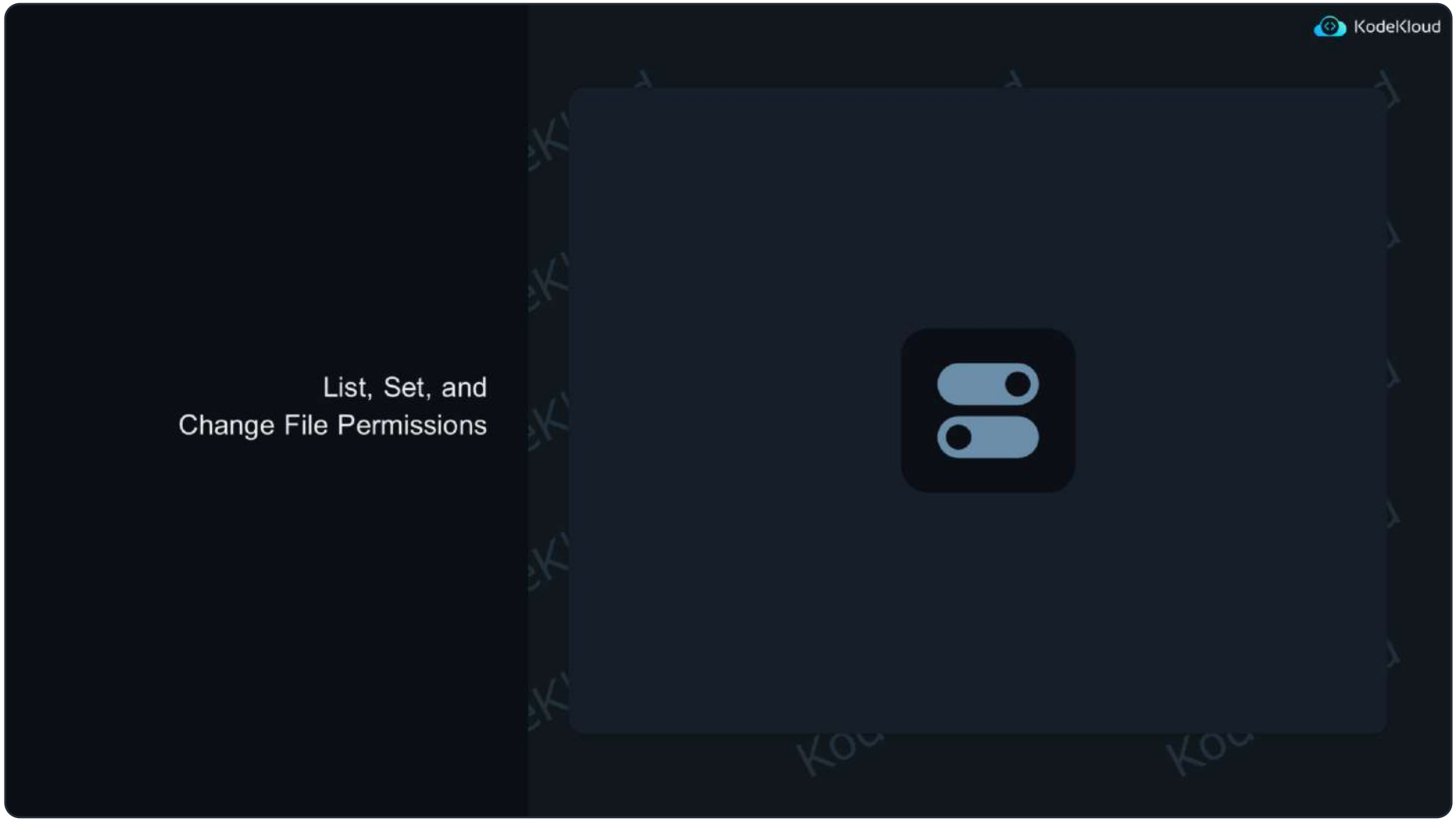
Since soft links are nothing more than paths pointing to a file, you can also softlink to directories:

ln -s Pictures/ shortcut_to_directory

Or you can softlink to files/directories on a different filesystem.

KodeKloud

Visit www.kodekloud.com to discover more.



We'll now discuss how to list, set, and change standard file permissions in Linux.

To understand how file and directory permissions work on Linux we must first look at file/directory owners.

If we type

ls -l

we'll see something like this:

Any file or directory is owned by a user. In this case, we see that the file "family_dog.jpg" is owned by the user called aaron. Only the owner of a file or directory can change permissions, in this case, aaron. The only exception is the root user (super user/administrator account), which can change permissions of any file or directory.

In the second field we can see that this file also has a group associated with it, the family group. We'll see later what the role of the group is.

To change the group of a file/directory, we use the chgrp command (change group).

Syntax:

chgrp group_name file/directory

For example, to change this file's group to "sudo" we'd use:

chgrp sudo family_dog.jpg

If we do another ls –l, we can see that the group has now changed to sudo.

We can only change to groups that our user is part of.

We can see to what groups our current user belongs with:

groups

This means we can change the group of our file to: aaron, sudo or family.

Again, the root user is the exception, which can change the group of a file or directory to whatever group exists on the system.

There's also a command to change the user owner of a file or directory: chown (change owner).

The syntax is:

chown user file/directory

For example, to change ownership of this file to jane, we'd use:

chown jane family_dog.jpg

But only the root user can change the user owner, so we'd have to use the sudo command to temporarily get root privileges:

sudo chown jane family_dog.jpg

With another ls –l, we can see the user has now changed to jane.

We can change both user owner and group with a different syntax of chown:

chown user:group file/directory

And since only root can change user ownership, let's set user to aaron and group to family to revert all our changes:

sudo chown aaron:family family_dog.jpg

One last ls –l will show us that the owner is aaron again, and the group is family.

Our

ls -l

command also shows us the permissions of all files and directories in our current directory

first character on that line shows us what type of entry this is: a file, a special file, a directory and so on. For example, we'd see "d" for a directory, "l" for a soft link, or "-" for a regular file. Here’s a table that shows the different identifiers and what they stand for.

We will learn about some of these file types later in this course.

The next 9 characters show us permissions:

First 3: permissions for the user that owns this file.

Next 3: permissions for the group of this file.

Last 3: permissions for other users (any user that is not aaron or not part of the family group).

Let's see what r, w and x mean in two different contexts, because they act in a certain way for files and have slightly different behavior for directories.

For a file:

r means the user, group, or other users can read the contents of this file. - means they cannot read it.

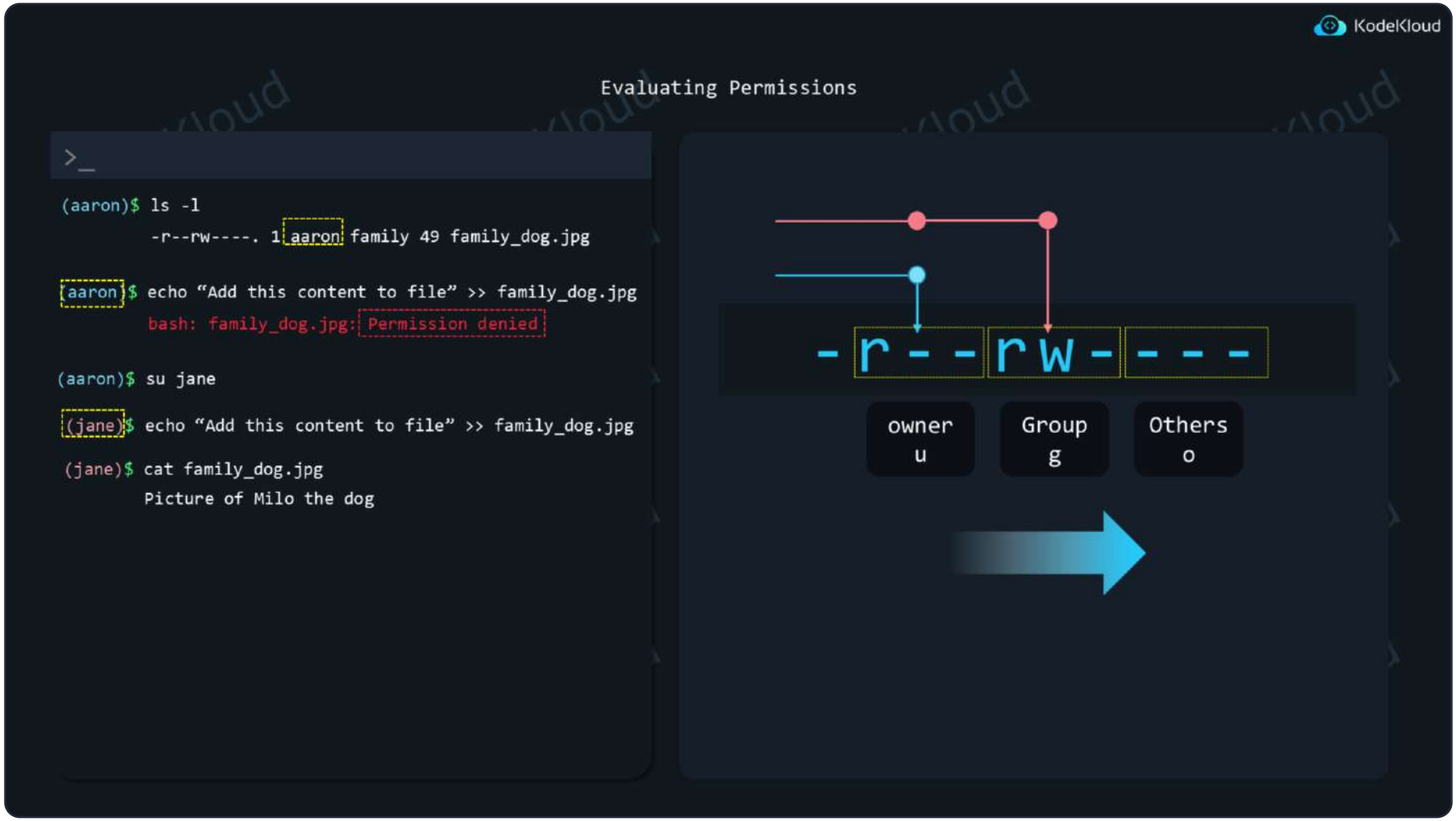
w means the user, group, or other users can write to this file, modify its contents.

x means the user, group, or other users can execute this file. Some files can be programs or shell scripts (instructions we can execute). To be able to run this program or shell script, we must have the x permission. A - permission here means the program or shell script cannot be executed.

•w means the user, group, or other users can write to this directory. We need w to be able to create or delete entries in that directory (add/delete files or subdirectories), as when we use mkdir.

- x means we can "execute" into this directory. We need x to be able to do "cd Pictures/" and "enter" into the Pictures/ directory.

When directories are meant to be accessible, you'll normally find both the r and the x permissions enabled.



Whenever you're on a Linux system, you're logged in as a particular user.

We've changed permissions in an interesting way to make this easier to understand.

<c> Look at the permissions for the family_dog.jpg file. It’s set to <c> read only for owner, read write for group and no permissions for others.

<c> We see the current owner of the file is aaron. And we know aaron is part of the family group.

Can aaron write to this file considering the fact that the owner has read permissions only? It might seem that he should be able to do that, as he is part of the family group, and that group has rw- (read/write) permissions.

<c> But if we try to add a line of text to this file, it fails.

Why is that? Because permissions are evaluated in a linear fashion, <c> from left to right.

With these permissions in mind:

let's see how the operating system decides if you're allowed to do something.

It goes through a logic like this:

Who is trying to access this file? <c>aaron
Who owns this file? <c> aaron
Ok, current user, aaron, is the owner. <c> Owner permissions apply: r--. aaron can read the file but cannot write to it. <c> Write permission denied!

It does not evaluate the permissions of the group because it already matched you to the first set of permissions: the ones for the owner of the file.

<c If you'd be logged in as a different user, for example jane, the logic would be like this:

Who is trying to access this file? <c> jane
Who owns this file? aaron
Ok, owner permissions do not apply, <c> moving on to group permissions
Is jane in the family group? Yes. Ok, <c> group permissions apply: jane has rw- permissions so she can read and write to file.

If the user trying to access the file is not the owner and is also not in the "family" group, the last three permissions would apply, the permissions for other users.

Now that we have a basic understanding of permissions, let's move on to how we can change them to suit our needs.

To change permissions, we use the chmod command. The basic syntax of the chmod command is:

chmod permissions file/directory

We can specify these permissions in many ways. Let's start out with simple examples.

We saw that our owner, aaron, cannot write to this file. Let's fix that. To specify what permissions we want to add, on top of the existing ones, we use this syntax:

To add permissions for the user (owner): u+[list of permissions]. Examples: u+w or u+rw or u+rwx.
To add permissions for the group: g+[list of permissions].
To add permissions for other users: o+[list of permissions].

In our case, we want to add the write permission for our user owner of the file:

chmod u+w family_dog.jpg

Now the old r-- becomes rw- with the newly added "w" permission. So we fixed our problem and aaron can write to this file.

To remove permissions for the user (owner): u-[list of permissions]. Examples: u-w or u-rw or u-wx.

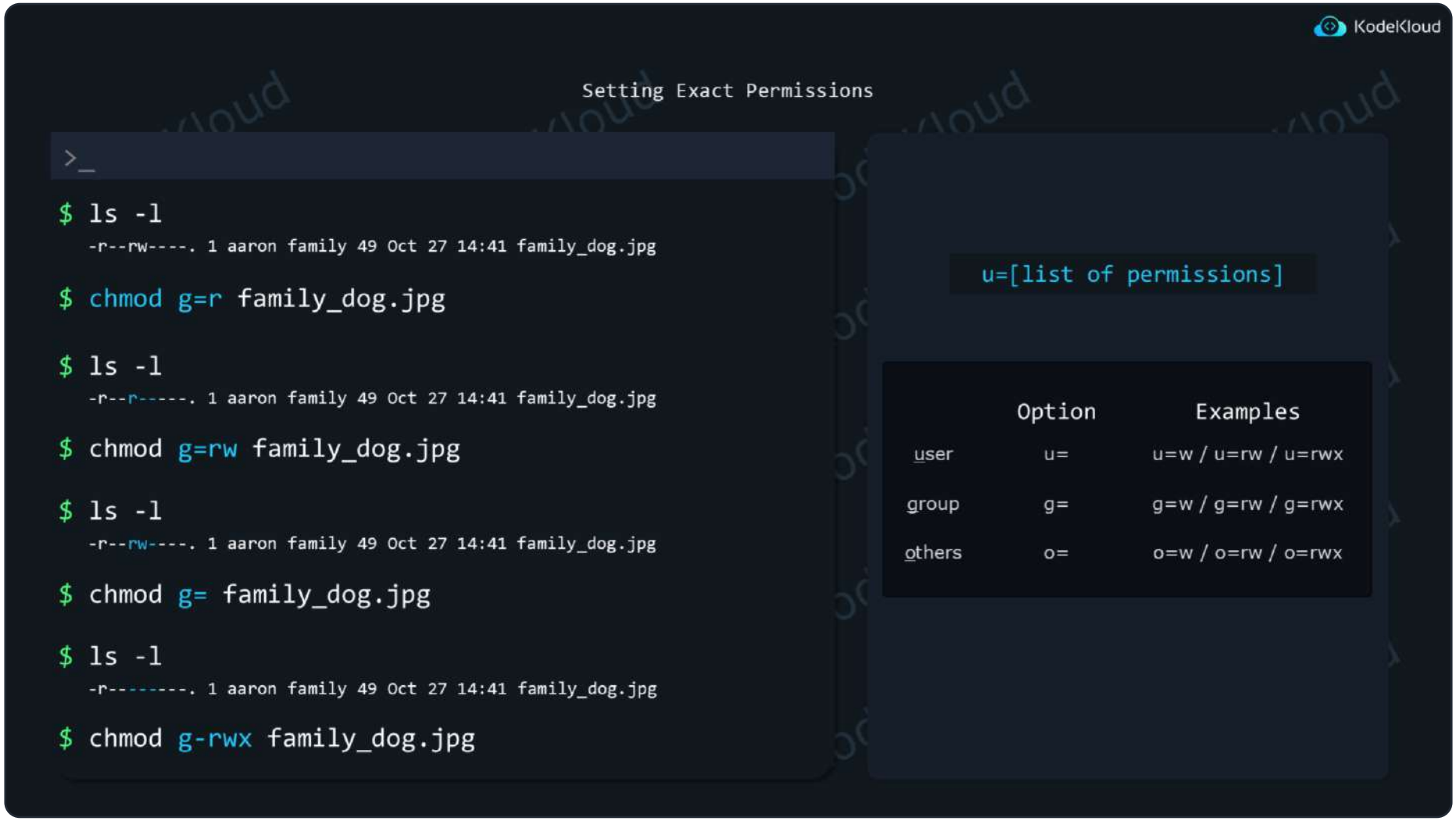
To remove permissions for the group: g-[list of permissions].

To remove permissions for other users: o-[list of permissions].

At this point, we have the permission r-- for other users. That means anyone on this system can read our family_dog.jpg file. If we want only the user owner and group to be able to read it, but hide it from anyone else, we can remove this r permission.

chmod o-r family_dog.jpg

Now only aaron or the family group can read this file, no one else.



With + and - we saw that we can add permissions on top of the preexisting ones or remove some of them from the preexisting ones.

If a file has rwx and we remove x, we end up with rw-. If another file has r-x and we remove x, we end up with r--. If we only care about removing the execute permission and we don't care what the other permissions are, this is perfect. But, sometimes, we'll have a different requirement. We'll want to make sure that permissions are set exactly to certain values. We can do this with the = sign.

Just like before, this is done with the format: u=[list of permissions] or g=[list] or o=[list].

Example: we want to make sure that the group can only read this file, but not write to it or execute it. We can run

chmod g=r family_dog.jpg

We can see that, before, group permissions were rw-. We didn't tell chmod to actually remove the w permissions, but by saying g=r, we told it to make the group permissions exactly: r--. This only affects the group permissions and not the user or other permissions.

If we'd want to let the group read and write, but not execute, we'd use:

chmod g=rw family_dog.jpg

We can see that whatever letter is missing, will make chmod disable permissions for that thing. No x here means no execute permission will be present on the file.

Which leads us to the next thing. What if we omit all letters? No r, no w, no x. This would disable all permissions for the group:

chmod g= family_dog.jpg

This is like saying "make group permissions all empty". Another command that does the same thing is

chmod g-rwx family_dog.jpg

It does the same thing, but following another logic - remove all these permissions for the group: r, w, and x.

We saw how to
add permissions with +
remove with -
set exactly to: with =

We can group all these specifications in one single command by separating our permissions for the user, group and others, with a "," comma.

For example, let's consider this scenario:
We want the user to be able to read and write to the file; don't care if execute permission is on or off.
We want the group to only be able to read (exactly this permission).
And we want others to have no permissions at all.

Our command could be:

chmod u+rw,g=r,o= family_dog.jpg

Or, let's say:
We want the user to only be able to read and write.
But we want to remove the write permissions for the group and leave all other group permissions as they were.
We don't care about permissions that apply to other users.

We would use:

chmod u=rw,g-w family_dog.jpg

chmod supports another way to set/modify permissions: through octal values.

First, let's look at another command that shows us permissions:

stat family_dog.jpg

Here's the list of permissions displayed by stat.

We can see rw-r----- has an octal value of 640 (ignore the first 0, that's for special permissions like setuid, setgid and sticky bit, which we'll discuss in a later lesson). If we break this down, 640 means the user/owner permissions are 6, the group permissions are 4 and the permissions for other users are 0. How are these calculated?

Let's take a closer look at this permission. We have rw for user, r for group and none set for others. Each permission is represented in binary. If it's set the binary is set to 1 or else its set to 0. In this case the first part has 110, the second part is 100 and the third part is 0. Converting this binary to decimal would give us 6 for the first part, 4 for the second part, and 0 for the third part. Here's a quick binary table for your reference.

Let's take another example. This time rwx r-x and r-x. So, the binary format would be 111, 101, 101. The decimal of which is 755.

In the last example it’s read write execute for all, so its 1 for all bits, and so the decimal value is 777.

if you find binary difficult another approach would be to use the octal table. It's much simpler. For each permission assign an octal value. For example 4 for read, 2 for write and 1 for execute. Then whichever permission is set, consider the respective value for that and for the permission bit not set consider 0. Once done, add up numbers within each group. 4 + 2 = 6 and 4 + 0 + 0 is 4 and the last group is 0.

Let's look at using the same approach for the other examples as well. rwx r-x and r-x gives us 755

and rwxrwxrwx gives us 777.

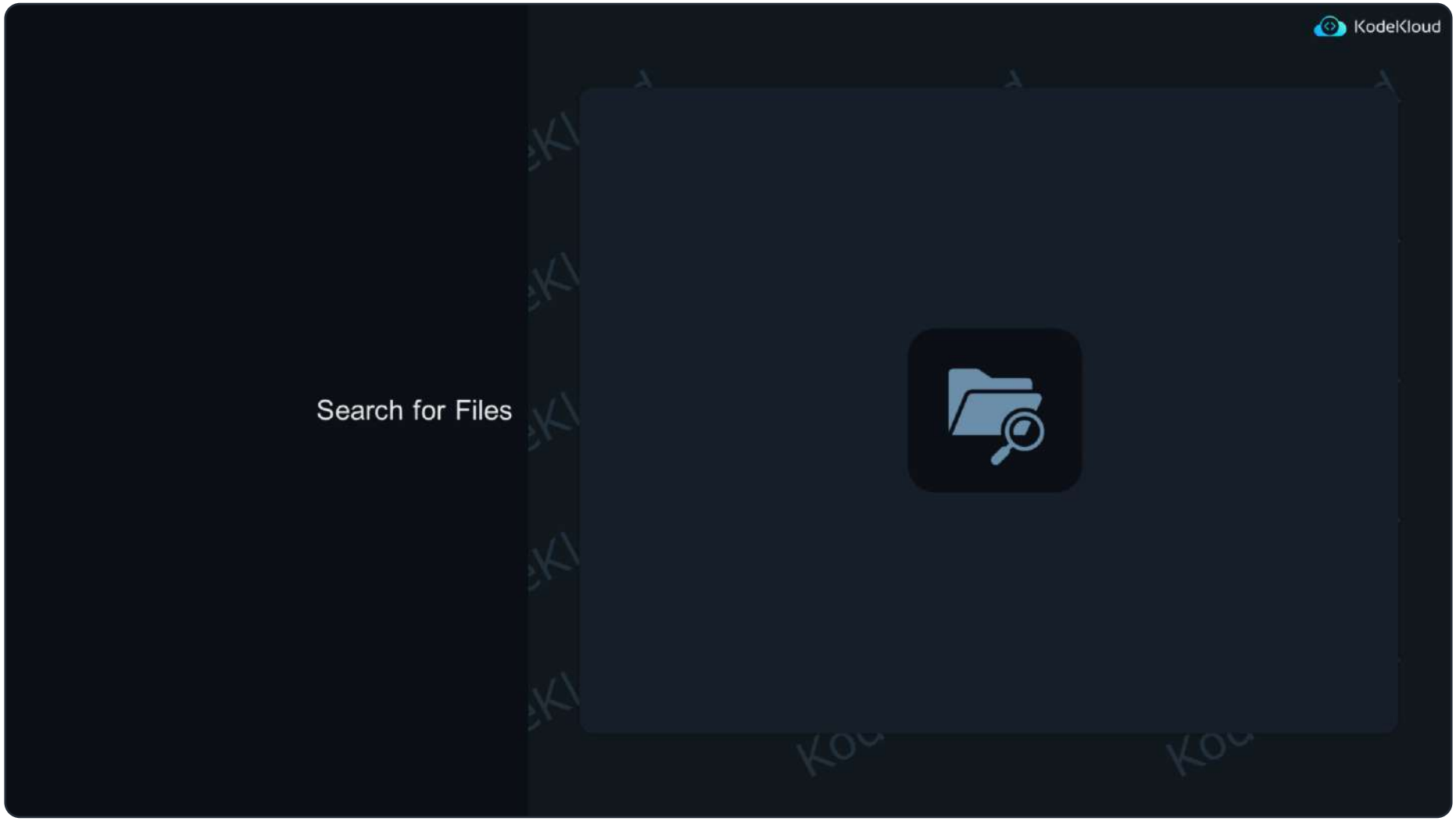
Once we identify the number we want to set to, we can use the same in chmod commands as well. Instead of specifying the permissions for each group, we could just provide a number like this.

chmod 640 family_dog.jpg

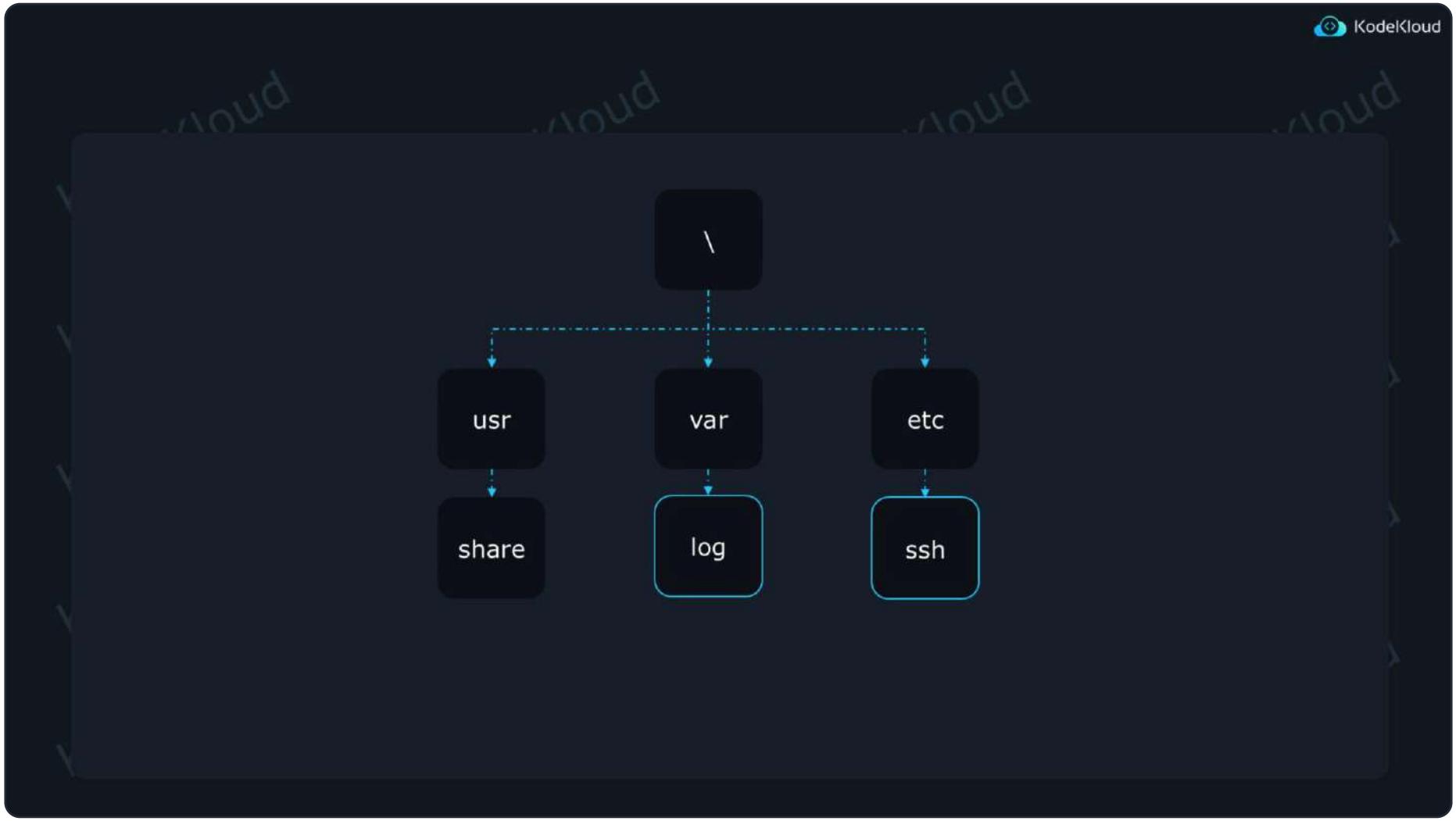
Well, that's all for now, I will see you in the next one.

KodeKloud

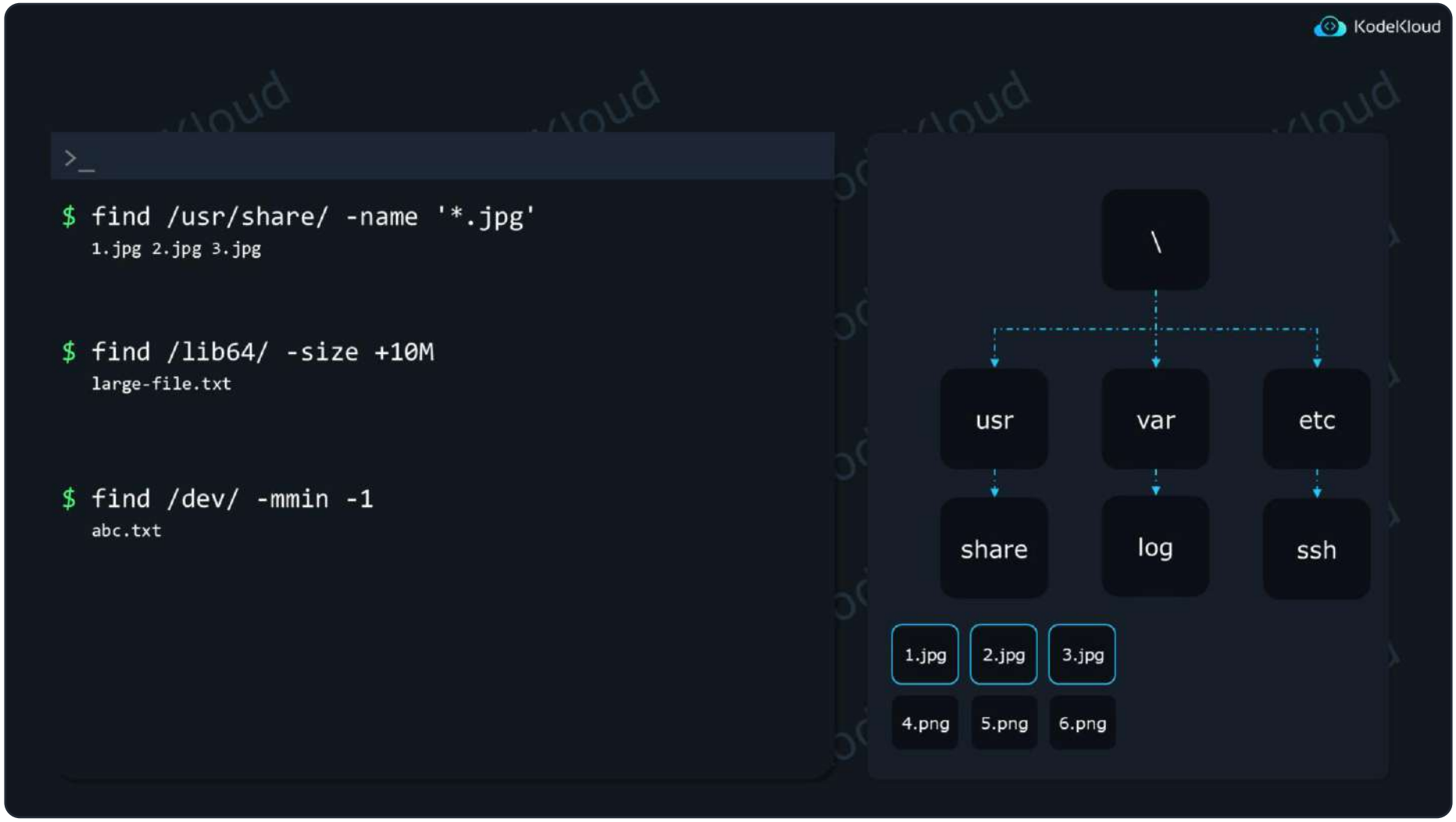
Visit www.kodekloud.com to discover more.



Let’s now look at how to search for files in Linux.



After you get a little bit familiar with a Linux OS you learn that files are very nicely organized. If you want to configure your SSH daemon, you'll know you'll find relevant config files in /etc/ssh/. Need to find logged errors? You go to /var/log. Most of the time, you'll know where everything is, at least, approximately. So why would you need to search for files? Let's look at some typical scenarios.



Imagine you have a website. You may want to find all your image files. If your website's directory would be /usr/share/, you could quickly get a list of all .jpg files with a command like:

In a different scenario, you're almost running out of disk space. This server is hosting virtual machines. You notice that most of the virtual machines require files under 20GB. You figure that you can search for files that are larger than 20GB to filter out the abnormally large ones.

We don't have such large files available, but here's how we would look for files larger than 10 megabytes:

Or let's say you've just updated an application and you're curious to see what files were changed. You can quickly look at all files that have been modified in the last minute, with a command like:

Of course, this applies to many other scenarios. Like you could use a similar command to see what configuration files your system administration team changed in the last hour.

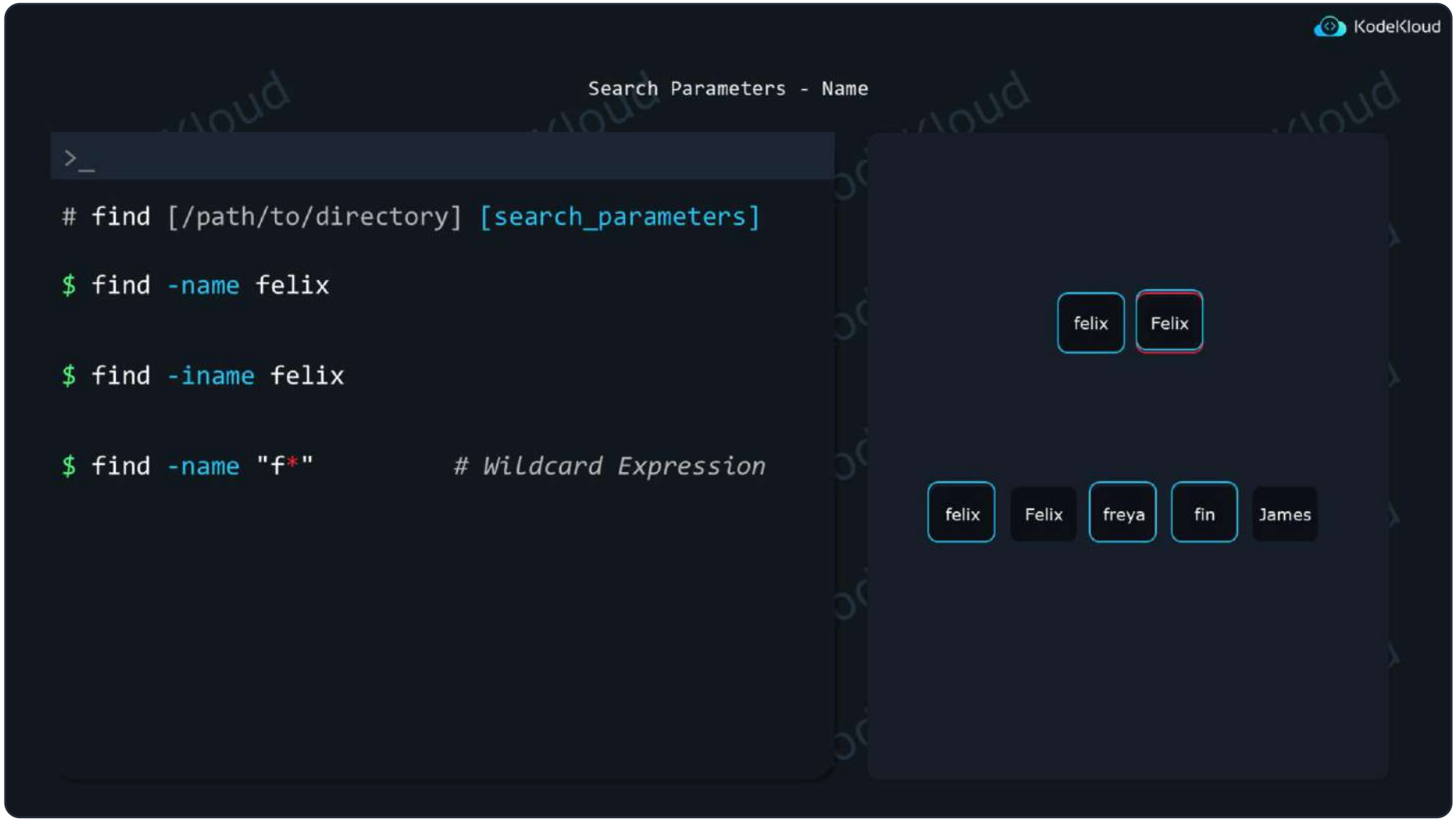
From these examples, it's clear that the command to search for files is find. Let's take a look at the syntax we'll use throughout this lesson:

For example to find a file named file1.txt in the directory /bin run the command find /bin/ –name file1.txt . –name is the search parameter used to specify the name of the file you are looking for.

You can sometimes skip specifying the path to the directory you want to search through. And when you do that it searches in the current directory.

The first few times you'll use this command, it may happen quite often that you mix up the directory path with the search parameters. Meaning, instead of writing find /bin/ -name file1.txt, you may write find –name file1.txt /bin/. If you find yourself falling into this trap, just think about it this way, "First I have to go there, then I will find it". You have to enter your room, and only after you can search for your keys. This will remind you that you first have to specify the search location and then the search parameters.

With this basic knowledge out of the way, let's focus on what makes the real magic happen, the search parameters.



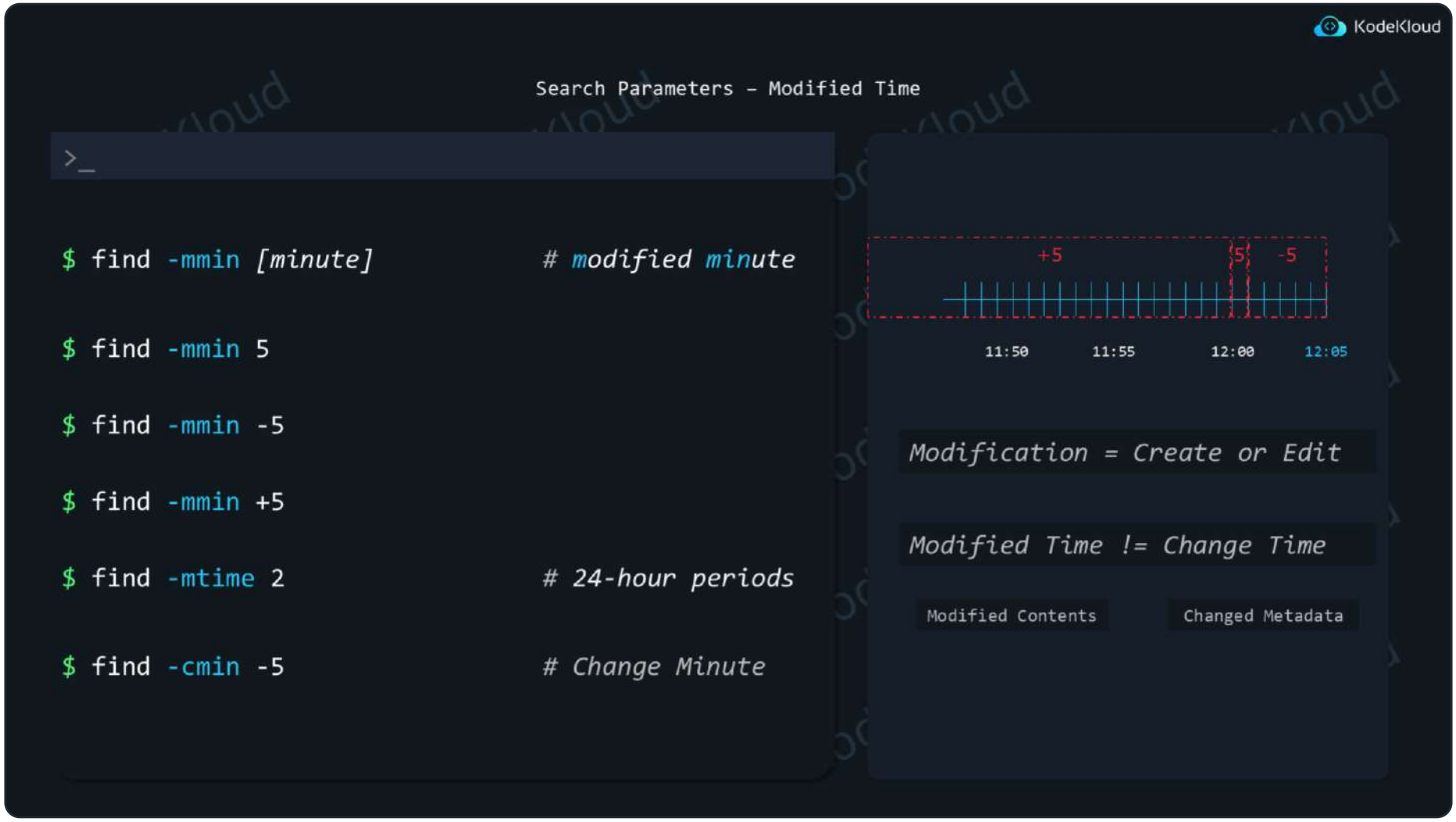
Let’s look at some other parameters.

We just saw the name parameter being used already. It is used to find files with a specific name in this case felix.

This however is case sensitive. Meaning it won’t find a file named Felix with a capital F.

If you'd like the find command to not be case sensitive, or case insensitive add an i infront of the option to make it iname.

At times you may want to find multiple files that have a pattern in their names. For example, I want to find all files that start with a lowercase f. For this use a wildcard expression, which is a starting expression, followed by a star. The * is like a joker card, for text. It will match anything even if it's 0 characters or 100. In this case it matches all names starting with f.



We already saw, in the examples, a command that looks for files modified in the last minute: find /dev/ -mmin -1. To remember "-mmin" think about "modified minute". But let's try to understand what "-1" means here, as we can use things such as "1" "-1" and "+1" and they mean different things.

It's a bit non-intuitive how find works with timestamps, so let's make it easier to understand. First, let's imagine the time is 12:05. With -mmin 5, we'd see files created at exactly 12:01, and not 12:00 as we'd expect. That's because, when we backtrack from the current time, we step through 12:05, 12:04, 12:03, 12:02, and end up at 12:01, exactly five values.

With -mmin -5 we'd find files modified in the last 5 minutes. That means, files modified at 12:05, 12:04, 12:03, 12:02 and 12:01. 12:00 not included, for the same reason as before. From 12:05 to 12:01 we have exactly five values. To 12:00, it would be six values, so we'd need to bump up -mmin to "-6" to also include that timestamp.

With -mmin +5, we'd see files created from 12:00, to 11:59, 11:58 and beyond (in the past). You might notice there's a twist this time. We see 12:00 here instead of 12:01. When you use a +, the count skips the current time. So it starts counting from 12:04 instead of 12:05. And we have five minutes from 12:04 to 12:00. So another way to think about this "find –mmin +5" command is that it shows us any object that was created more than 5 minutes ago. And 12:01 is not more than 5 minutes ago, but rather exactly 5 minutes ago. So more than five minutes is 12:00 or earlier.

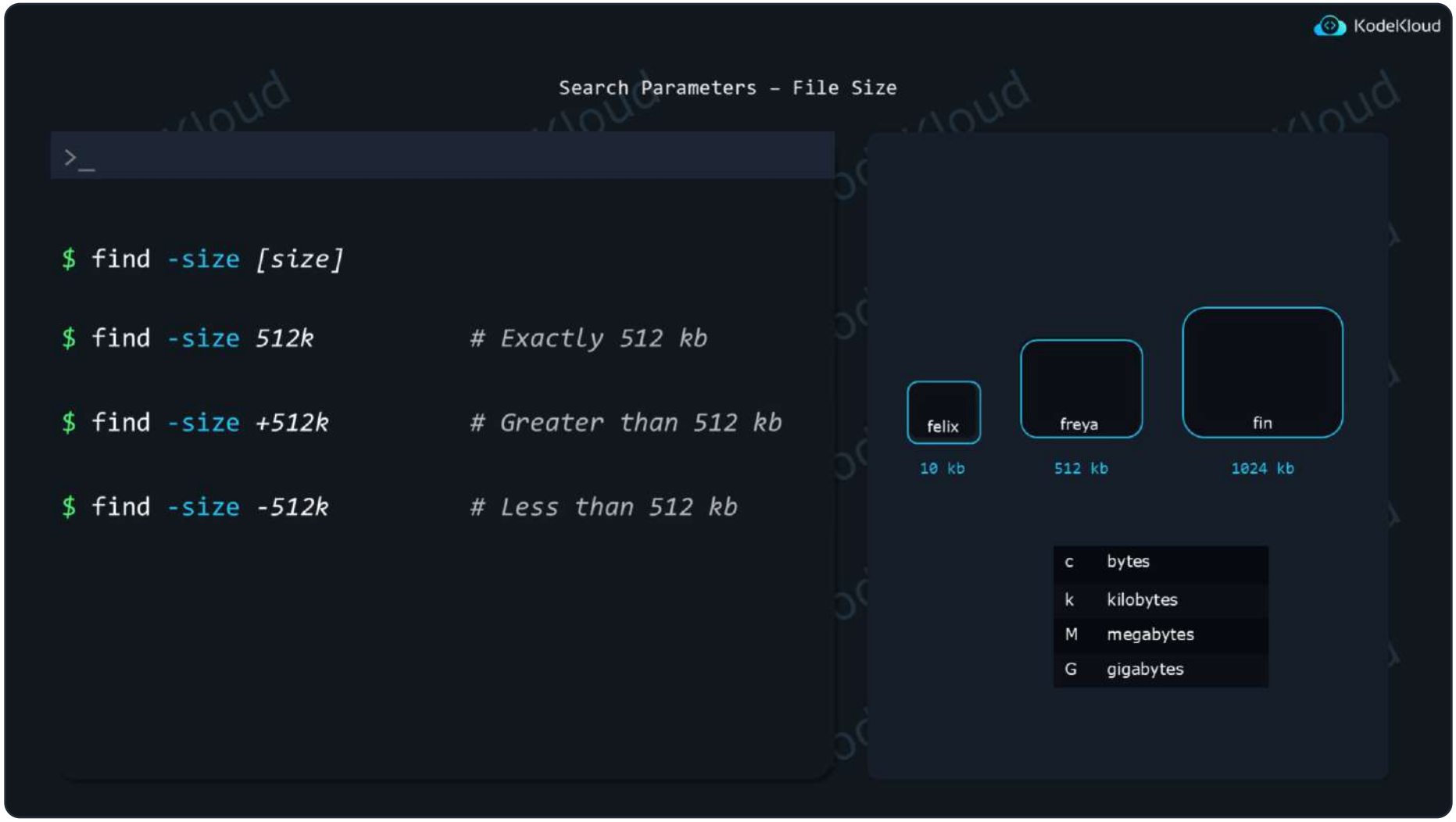
Another similar option is mtime. But instead of working with minute units, it works with 24 hour units. -mtime 0 matches what was modified in the last 24 hours. -mtime 1 matches what was modified between 24 and 48 hours ago, and so on.

[NOTE TO INSTRUCTOR: it might be useful to include an extra slide after this one and show multiple –mtime options and their effects. See this comment: https://docs.google.com/document/d/1pgSmA4FpwSQzzqku08o5pcXz68ybkyeTsfPExiFLUKA/edit?disco=AAABJEorlug]

It's worth noting that "modification" might mean either when the file was last edited, but also when it was created. If we edit a text file at 12:05, it will show up as modified at 12:05. But if we create a new file at 12:06, this will also count as a modification time. So it will show up as "modified" at 12:06, even though that's when it was created.

Linux also has a thing called "change" time for files. Which might sound like the same thing as a "modify" time, but it's actually different. Modify time refers to the time when the contents have been modified. Change time refers to the time when the metadata was changed. Metadata is "data about data", so in this case, "data about your file". This might mean something like file permissions, and other similar information. And this is where change time could be useful. Imagine you suddenly get errors in some application and you suspect it's because someone changed some file permissions in the wrong way. You could find files with permissions changed in the last 5 minutes, with a command like:

find –cmin -5



In our initial examples, we used -size to search for files based on their size.

To find files of a size of exactly 512 KB we can run the find command with the size parameter set to 512k. k stands for kilobytes. Here's a quick table showing the different values. c stands for bytes, k for kilobytes, M for megabytes and G for gigabytes. Note that M and G must be capital letters.

To search for files greater than 512 kb we can use +512 kb and for files less than 512 kb we can use –512 kb.

We saw how we can find all files (or directories) that start with the letter "f" with a command like find –size "f*". We also learned how we could find files of a certain size with a command like find –size 512k. -size 512k and –name "f*" are search parameters. But in the command's manual we'll also see that this part, where we tell find what to look for, is called a "search expression". That's because just like in mathematics, we can combine multiple parameters and build a more complex search expression.

For example, assume we want to find files that start with the letter "f" but also have a size of exactly 512k. We can combine our previous two parameters to build this search expression. And to do that we simply enumerate both of them:

find –name "f*" -size 512k

When we write it like this, there's an implied AND operator between the –name parameter and the –size parameter. That means both conditions must be true for the files to show up in the results.

But what if we want an OR operator between these parameters? Files or directories that start with the letter "f", but also files that have a size of exactly 512k. Not necessarily meeting both conditions at the same time. All we need to do is add "-o" where we want our OR operator to be. So we'd end up with this command:

find -name "f*" -o -size 512k

Another interesting thing we could insert into a find expression is the NOT operator. To make it easier to understand, let's look at another example. Say we want to find all files that do not begin with the letter f. To exclude files beginning with the letter f from our results, we would use the “-not” option before the “-name” option.

find –not -name "f*"

This would return a list of file names that do not begin with the letter f.

Another way to write the NOT operator is to use the exclamation sign “!”. One important note, however. For our command interpreter, Bash, the exclamation sign has a special meaning. And when it sees that in a command, it will interpret it a certain way, thinking we want to perform a special action in that spot. To tell Bash to ignore the special meaning of this character, and just consider it a regular character, we can do what is called "escaping it". To escape it we just add a backslash \ in front of the exclamation sign.

find \! -name “f*"

We can also search for files (or directories) based on their permissions. We'll use the "664" octal notation for our permissions. "664" means this set of permissions: user can read and write, group can read and write, others can read (u+rw,g+rw,o+r).

To search for files based on their permissions, we can use commands like these:

-perm 664 to look for files which have exactly these permissions

-perm -664 to look for files which have at least these permissions. Which means that even if the file has some extra permissions set, it will still show up in the search results. But if it has less than these permissions, it won't show up. For example, 664 denotes that a user should have read and write permissions. If they only have read permissions but no write, then find will not show this in the search result. Think of it as "bare minimum permissions are these:"

-perm /664 to look for files which have any of these permissions. Unlike the "bare minimum" condition above, this is more inclusive. For example, if a user can read the file, but cannot write to it, it will still show up in search results, as one permission has been matched, u=r, so it does not matter if other permissions exist or don't exist.

An alternative way to write each of these is:

-perm u=rw,g=rw,o=r
-perm -u=rw,g=rw,o=r
-perm /u=rw,g=rw,o=r

Now imagine we have a group of files.

We want to find files which only the owner can read and write, and no other permissions are set. To do that we would run find –perm 600. This would match the files, “felix,” “james,” and “bob.”

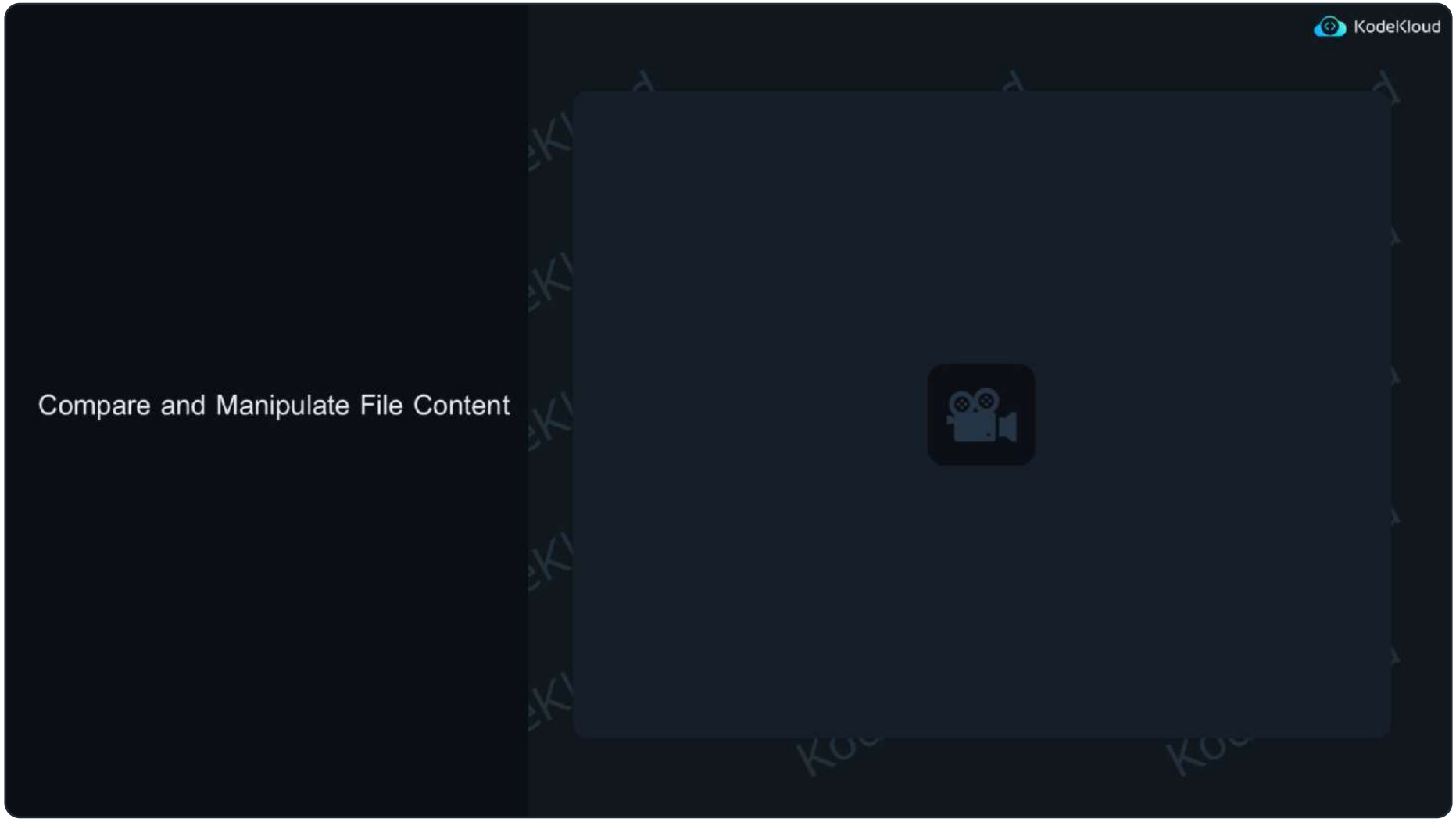
To find files that the owner can execute, at least, but rest of permissions can be anything, we would run find –perm -100, which would match only “freya” and “jacob.”

Now, imagine we want to make sure that nobody else can read these files, except users and groups that own them. In this case, we use the NOT operator. To look for files that others can NOT read, we would run find \! –perm –o=r, which matches “felix,” “james,” “bob,” “freya,” “john,” and “bean.”

Finally, imagine we want to find files that can be read by either the user, or the group, or others -- does not matter who it is -- but at least one of them should be able to read. To do this, we would run find –perm /u=r,g=r,o=r. In this case, all our files match the condition. If no one can read it, it won't show up in the results.

KodeKloud

Visit www.kodekloud.com to discover more.



As we can see, when it comes to Linux, we deal with a lot of text. Our SSH session is all text. We see text output and we input commands as text. And later on, we'll find out that configuring applications, or the system itself, is also often done through text files. This means we'll have to know how to:

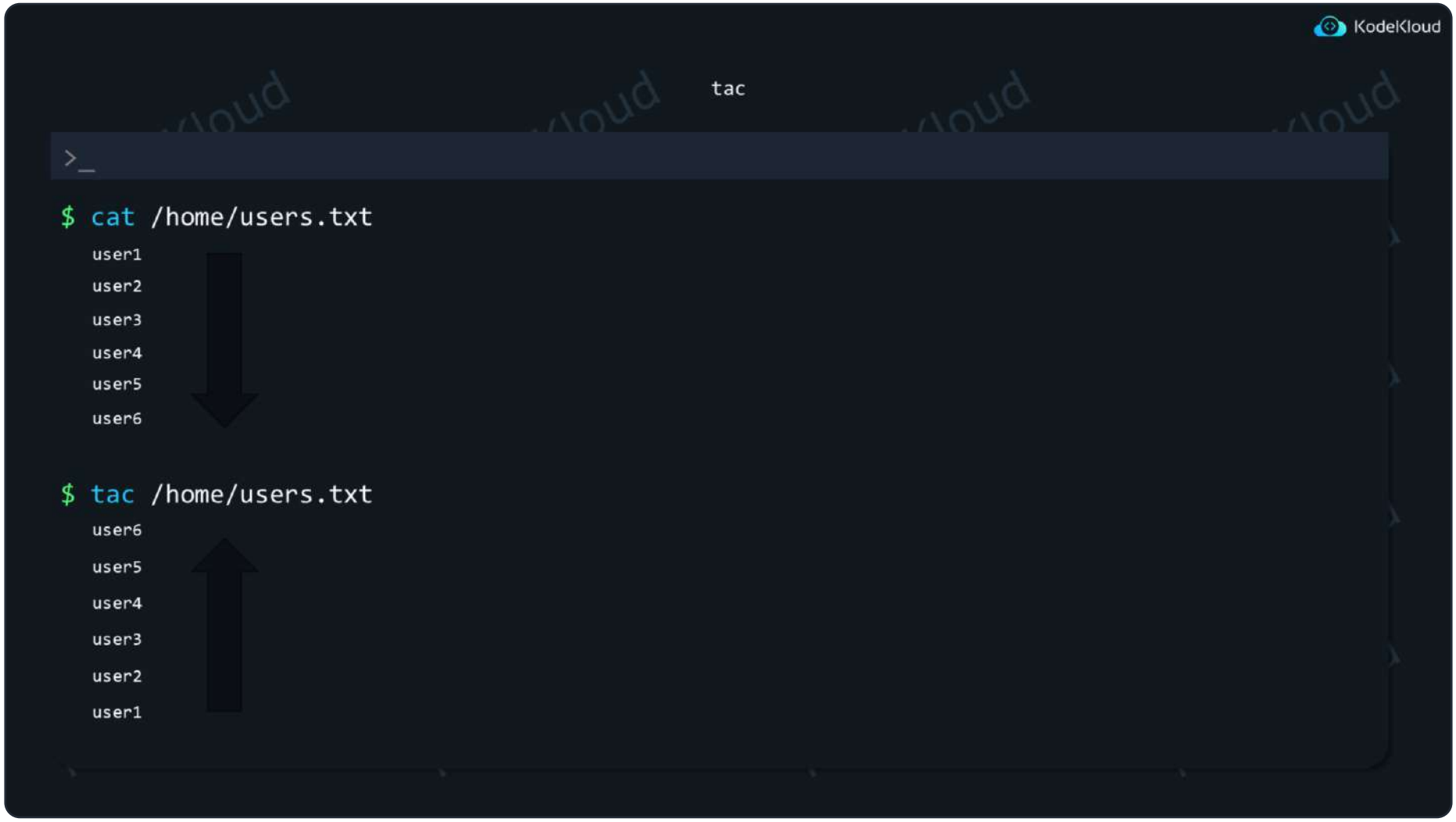
View text files

Edit text files

Transform text files

Compare text files

Let's start out with the most basic thing.



When the file you are trying to view is very small, you can use the cat command, followed by the name of the file you want to inspect.

cat /home/users.txt

To view the file reversed, from bottom, up, so that

line1
line2
line3

becomes

line3
line2
line1

tac /home/users.txt

But what if the file is very long? This might happen when you want to view a log file, where a program records all status messages about what it did, what succeeded, what failed, and so on. Many times, when you deal with logs, you'll want to see what happened in the last few seconds, or minutes. You'll find those log lines at the end, or tail of the file. You can view the last lines with a command like:

tail /var/log/apt/term.log

By default, the tail command will show you the last 10 lines. But you can tell it how many lines you want to see with the -n option:

tail -n 20 /var/log/apt/term.log

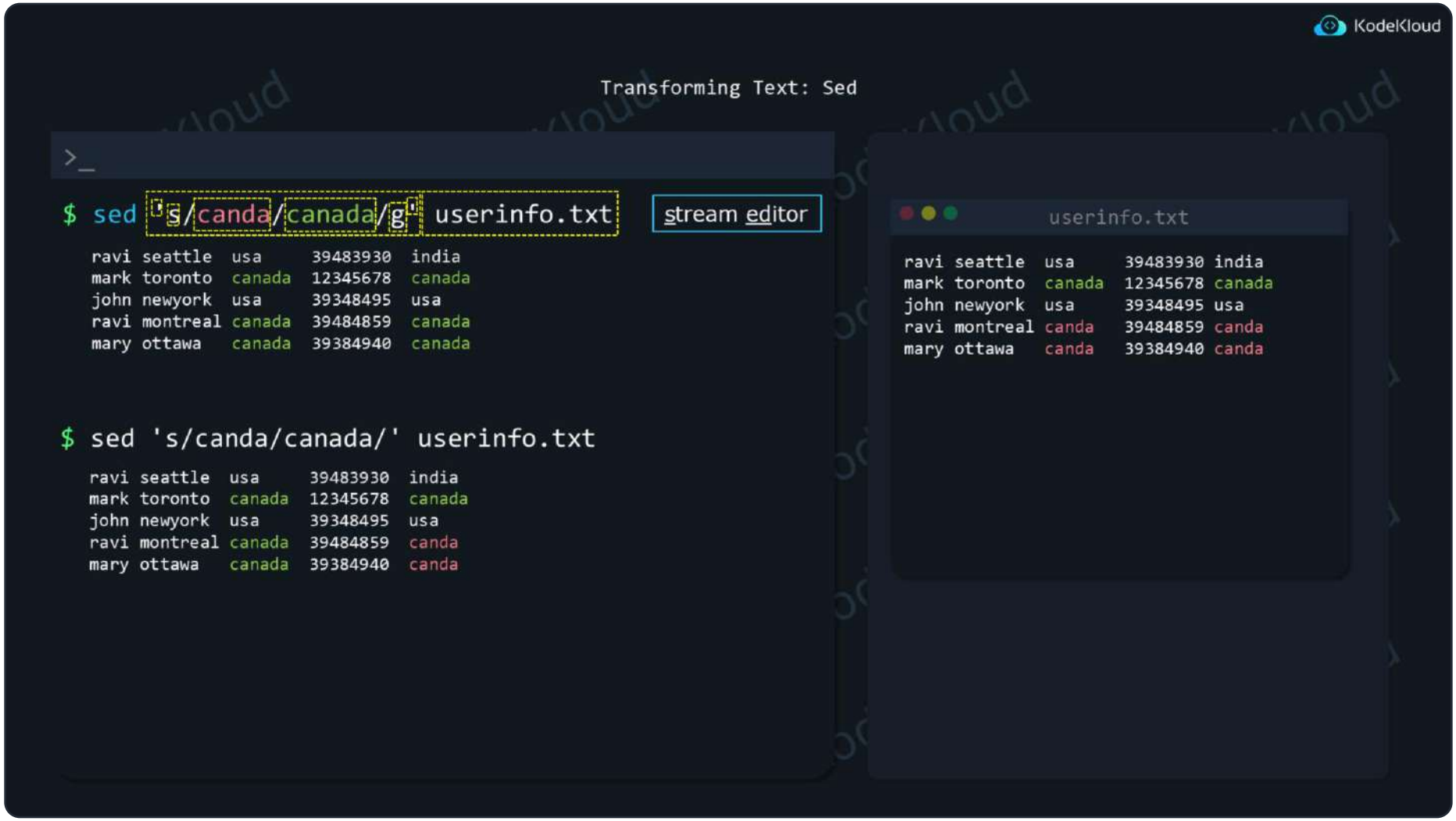
The opposite of this command, to view the first few lines instead of the last ones is the head command:

head /var/log/apt/term.log

We have 9 lines of text here, but the first one in the log is an empty line, bringing the total to 10.

To specify how many lines to return from the start of the file, we use the same –n option.

head -n 20 /var/log/apt/term.log



Sometimes, we'll have to deal with large files that require many changes in many places. Manually editing and changing the same thing in many lines of text can be tedious. But we have tools that can automate such tasks.

Search and Replace

Imagine you have a file with this content. A list of people, their addresses, phone numbers, and citizenships.

We see that canada spelled incorrectly in some places. Showing up as "canda" instead, missing an a in the middle. We would like to replace all occurrences of canda to canada.

There is a utility called sed that can search and replace all the required text.

The name sed comes from stream editor (stream of text enters utility, it gets transformed in some way, then it outputs the modified stream)

With this command we can preview what sed would do, without actually changing the file. It's recommended that you first preview what would happen, to make sure there would be no mistakes.

Let's break down how our command works:

sed 's/canda/canada/g' userinfo.txt

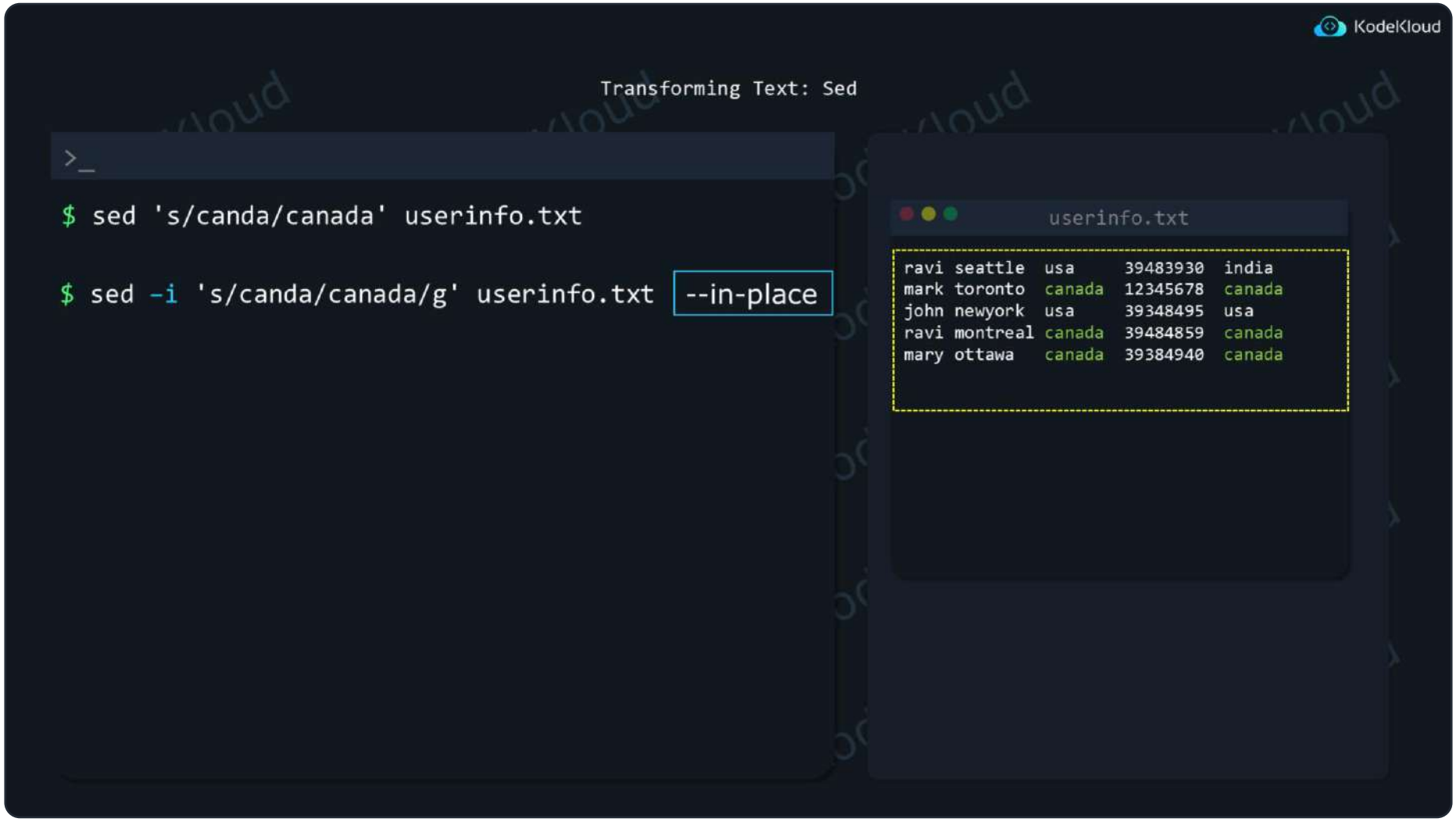
The easiest part we see here: at the end, we pass the name of our file or the path to that file (if it's not in our current directory).

's/canda/canada/g' is obviously where the magic happens. We must wrap this between ' ' single quotes to make sure our command interpreter does not interfere with the content here. We can have special characters like * an asterisk, and if we don't wrap it between quotes, bash, our command interpreter might transform that into something else (because it interprets we want that * to do something special). By wrapping our content between ' ' single quotes bash knows it should not interpret anything between those quotes.

The first s here tells sed this is a substitute command (search and replace)
then we tell sed to look for this exact text: "canda"
then we tell it to replace it with "canada"
Finally, the last g is for global search and replace. Normally, sed would only replace the first thing it finds on each line. By passing global here, we tell it to search and replace all occurrences, even if there are multiple per line.

So, in this command, for example, without the global option we see that only the first occurrence of canda is changed to canada. The remaining occurrences of canda on

each line are left as they are. That's what the global option helps with, to change all occurences on each line, not just the first one.



Up to this point, sed just showed us what it would do but it did not edit the file. If we look at the contents of the file after running the sed command we see its still the same.

When we're happy with the preview it shows us, we can tell it to actually edit the file by passing the -i command line option.

-i is short for --in-place to edit in place, directly in the file.

```
sed -i 's/canda/canada/g' userinfo.txt
```

We won’t see any change on the command line, but if we were to run cat on userinfo, we can see that the changes were made to the file.

Now let's say we would like to extract the names alone from this file. The first column in the file. For this we could use the cut command.

So "cut" is an utility that can extract the parts we need from a file.

For example, if we want only the first column that appears on each line:

cut -d ' ' -f 1 userinfo.txt

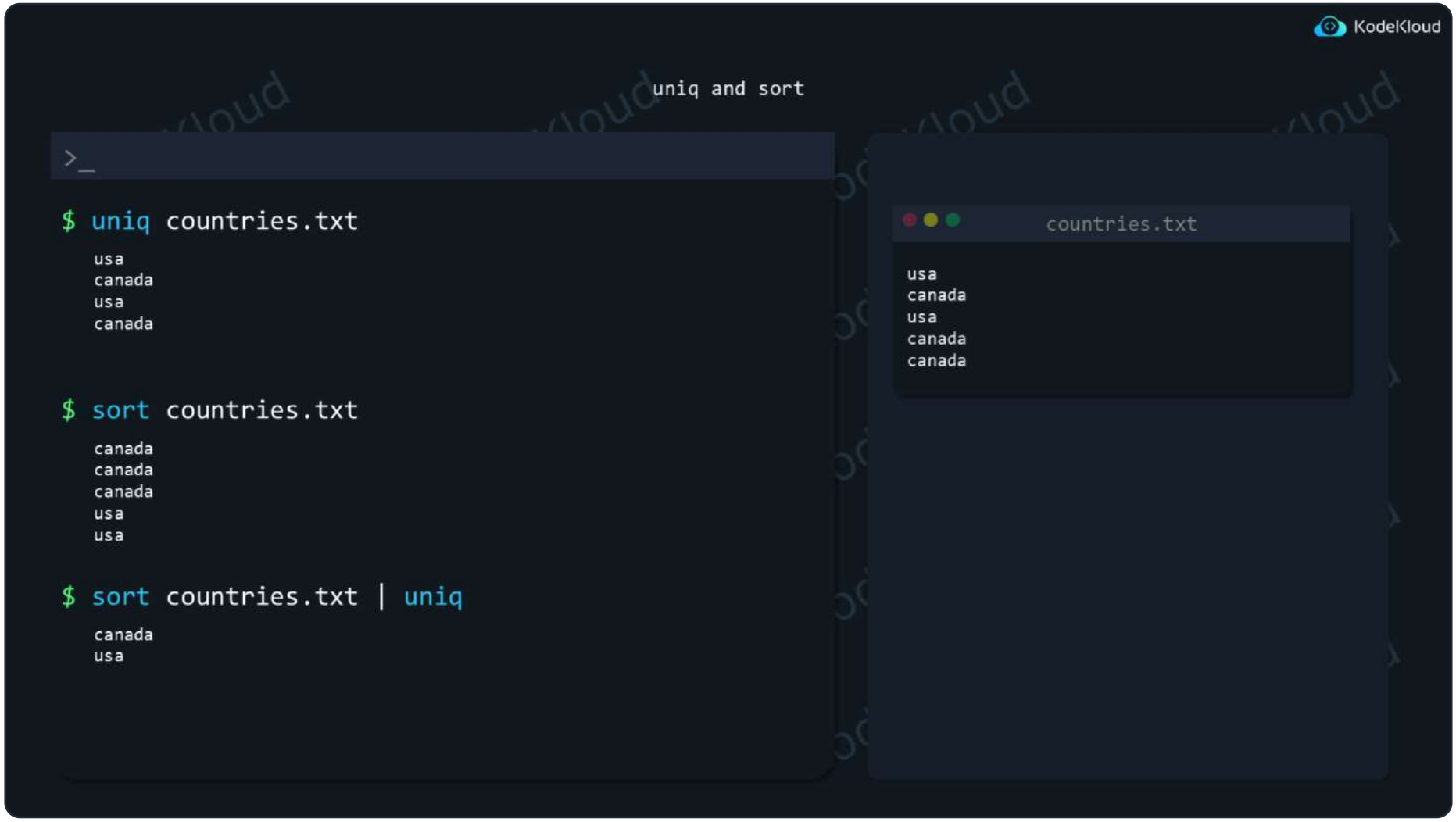
With -d we specify the delimiter. Words are separated by a space so we add a space wrapped between single quotes. This tells it that the space character is the one separating the parts we need. We can use anything as a delimiter, for example if we'd have numbers such as "10.22" "12.34" and so on, and we'd want to extract only the integer part, we'd use '.' as a delimiter and we'd be able to extract "10" and "12" from those numbers.

With -f we specify the field(s) we want to extract. In this case, each word is a field, and we want to extract field 1, the first word on each line.

Let's look at another example. Let's say that this other file separates different entries with commas. And we would like to extract all the countries from this content. Countries are the 3rd field in this case. Also, after we extract our country names, we will tell our command to save this output in a new file called countries.txt. We'll do that with a simple redirection. We'll learn more about redirections in a later lesson. For now, all you need to understand is that we can redirect with a "greater than" sign, and that will save the output to a file.

For this purpose we run the command cut –d. Then we follow up with the delimiter, a comma. And with –f 3 we extract the third field. We pass the name of our file "userinfo.txt" And we redirect the output to countries.txt.

This creates a new file with this content:



In the countries.txt file we have a list of countries, but there are duplicates. We'd like to get the list of unique countries from this file, so we know the total number of different countries the users originate from.

For this purpose we can use the uniq command and provide the filename to get unique entries from. But check that out. When we look at the output that our command generated, we still see some duplicates. But why?

All that the uniq command does is remove repeating lines that are next to each other. In this case, it removed the two lines that mention canada at the end. But it didn't remove the duplicates that aren't next to each other.

So how do we get all similar lines to be adjacent?

With the sort utility. This sorts the entries in the file alphanumerically. Which means countries with the same name will be next to each other.

Next, we can pipe the output of the sort command to the uniq utility. We'll learn more about piping output in a later lesson. Piping simply passes the output of one command to another command.

This command will finally give us the result we were looking for:

```
sort countries.txt | uniq
```

Now imagine that an upgrade changed an old configuration file. The old config file had 250 lines of text. The new config file has 268 lines. We could open them both and analyze line by line, to see what changed. But it would be hard to do this. That's why we have tools that can analyze files for us and quickly show us the differences.

Imagine the files on the right. Lines 1 and 4 differ. But lines 2 and 3 are identical in both files.

With the diff tool we can see differences between these files with a command like:

diff file1 file2

We'd see output like this:

Since its focus is on differences, the identical lines are not shown here. But we see some useful output. 1c1 tells us that line 1 from file1 is changed in line 1 from file2. Otherwise said, line1 in file1 is different from line1 in file2. The < sign tells us this content exists in the first file (file at left). The > sign tells us this content exists in the second file (file at right).

In large files we might have no idea where these lines are situated. So, we can tell diff to give us a little bit of context. This shows us what other text is around these areas where lines are different. To show context, we use the -c option.

diff -c file1 file2

We'll see lines that differ marked with !, + or -. The identical lines won't be marked in any way. ! signals a difference between lines while + or - signal content that was added or removed.

And probably the easiest way to spot differences is doing a side-by-side comparison with the -y option:

diff -y file1 file2

An equivalent command that is easier to remember is:

sdiff file1 file2

Think of the expression "side-by-side diff", and you'll remember the sdiff command.

We'll see differences marked with |.

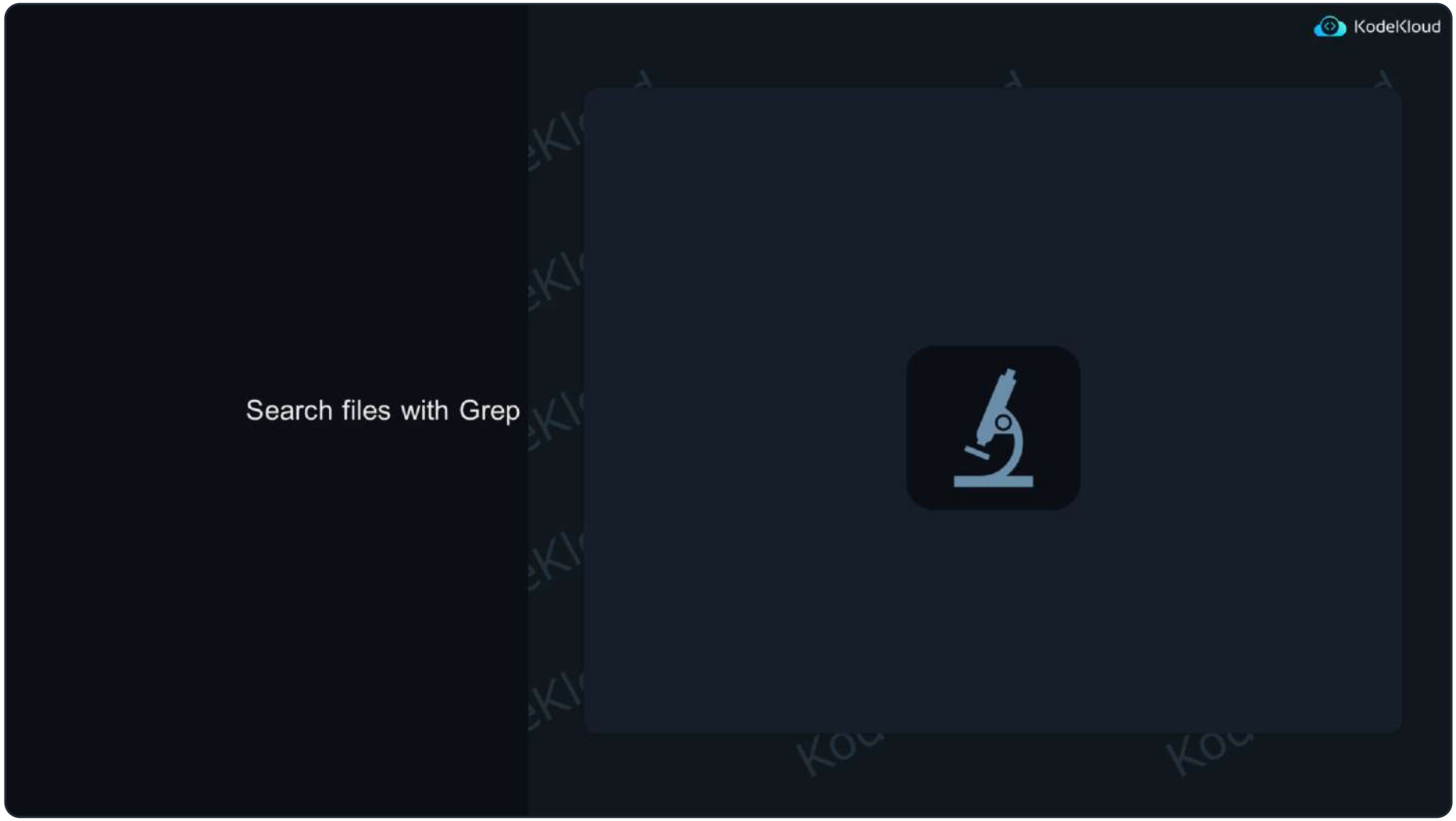
KodeKloud

Visit www.kodekloud.com to discover more.

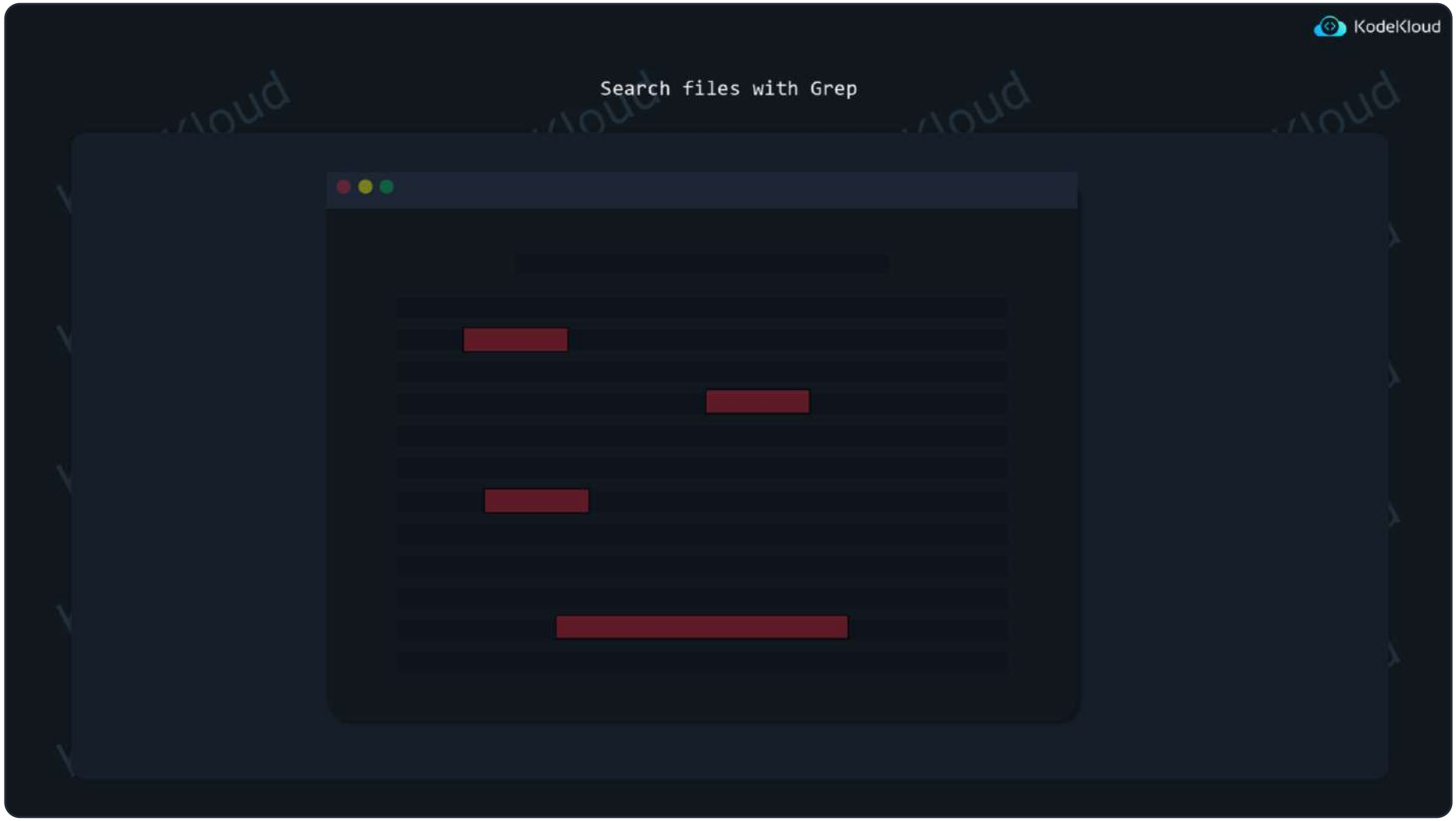
KodeKloud

Demo

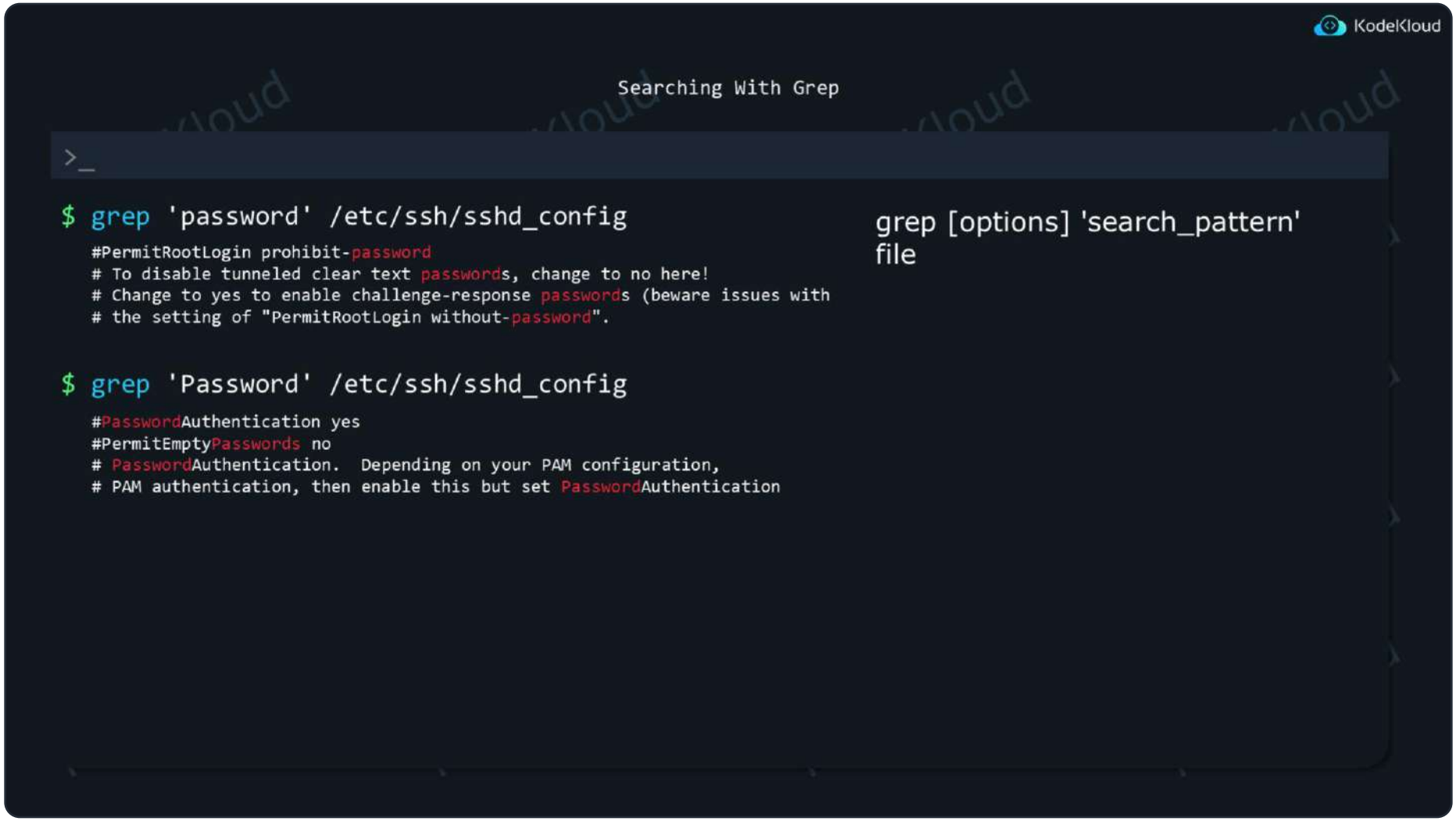
Pagers and VI



In this lesson we'll learn how to search for specific text with the grep utility.



Sometimes we have to deal with very large files that contain thousands of lines of text. And we need a way to filter out and find the information we need.



One important Linux command that lets you search through text files is grep. Let's see how it works.

The general syntax we will use with grep is:

grep [options] 'search_pattern' file

It's useful to make it a habit to wrap the search pattern between single quotes to prevent bash from interpreting special characters. Because in complex searches we might have characters like the * asterisk, and when we'd execute those commands we'd run into unexpected behavior if we forget to wrap between single quotes.

We can omit command line options, but we must specify a search pattern and a file we want to search through.

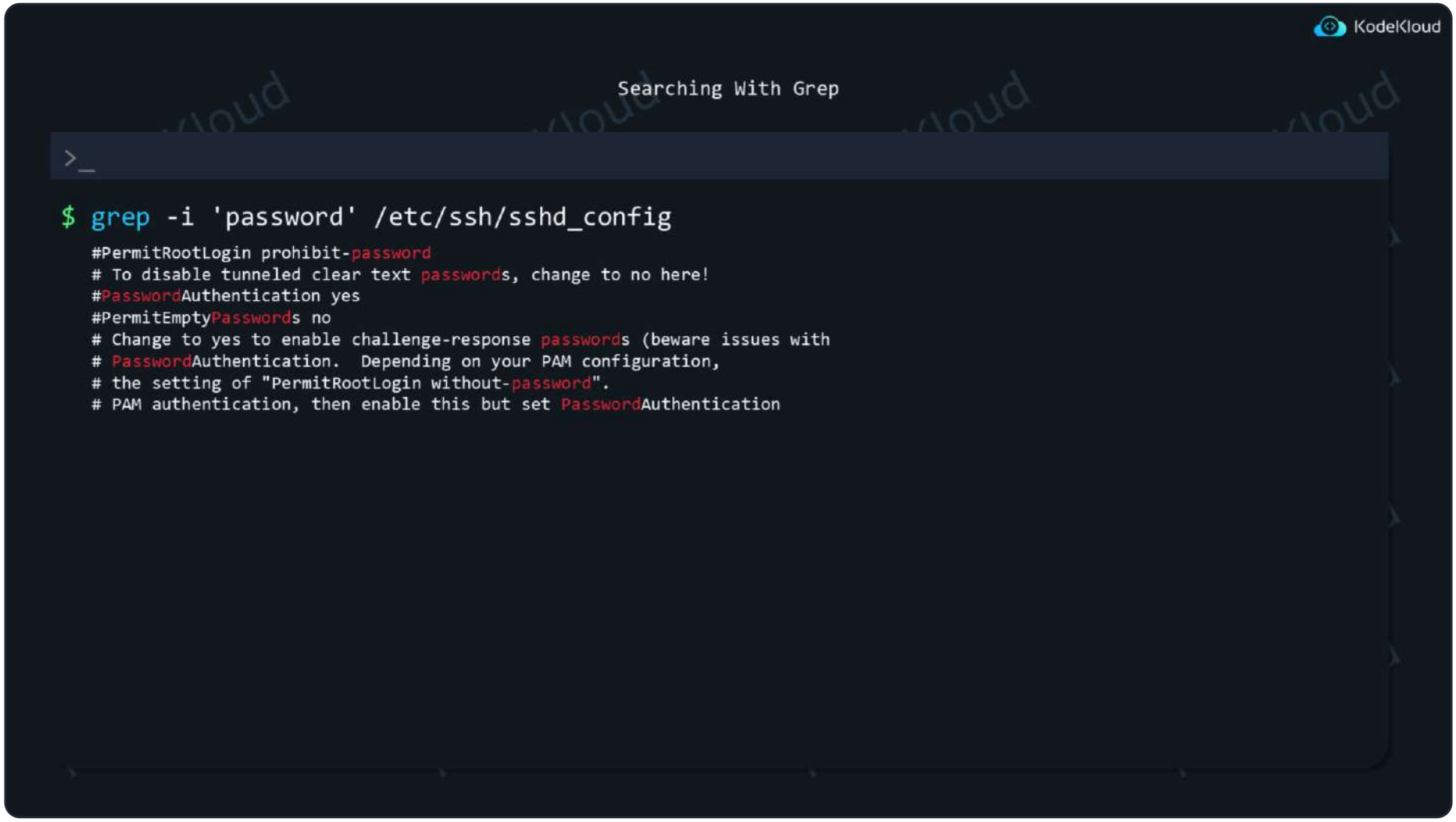
For example, to search for any line mentioning the word "password" in the configuration file of the SSH daemon we can run:

```
grep 'password' /etc/ssh/sshd_config
```

Search is case-sensitive so we'll only find occurences of the word "password" with all lowercase letters. To find lines that contain the word "Password" with a capital P, we can run this instead:

```
grep 'Password' /etc/ssh/sshd_config
```

And now we'll see different output since different lines match our search.



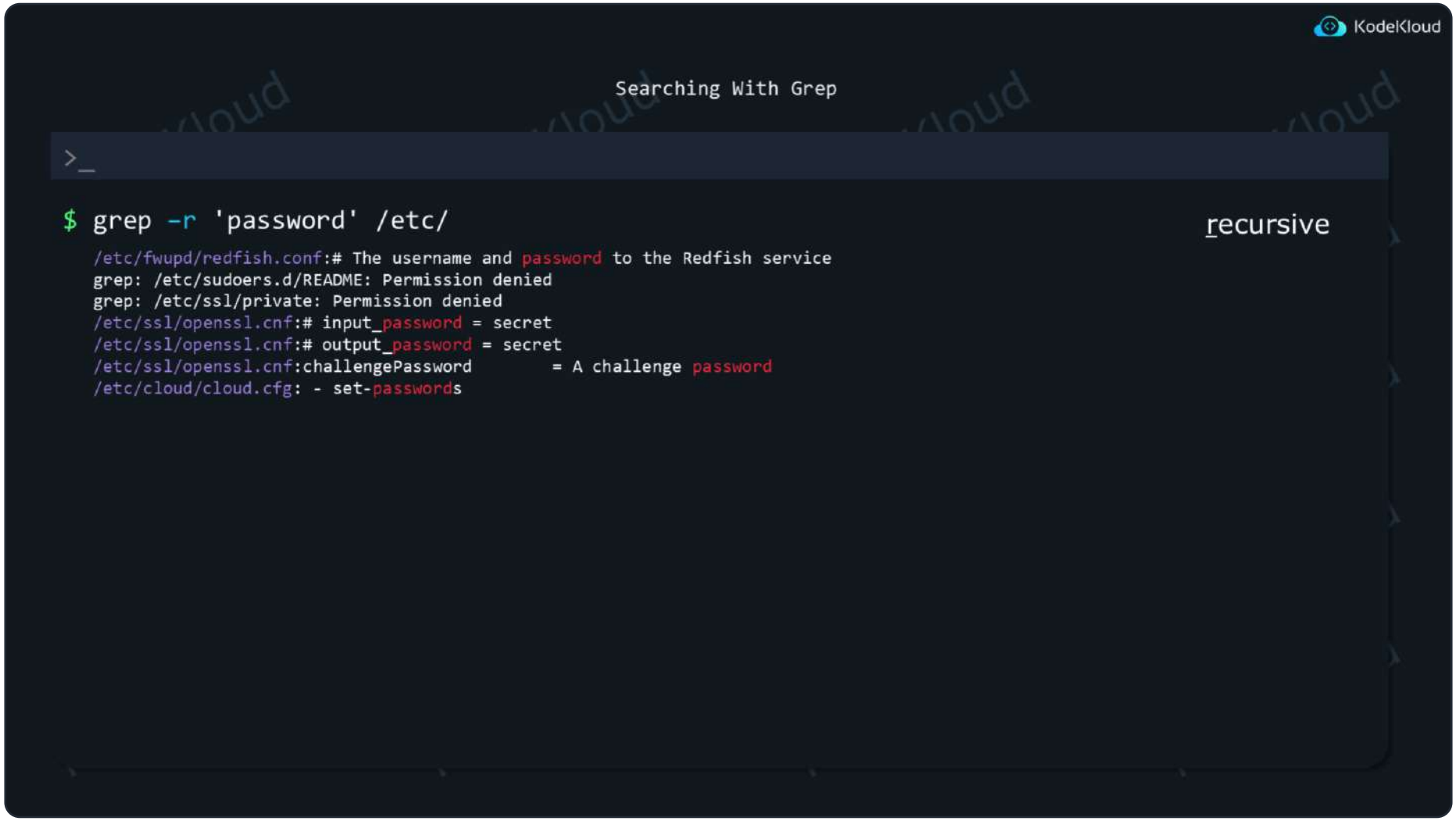
But in many situations we'll see case-sensitive searches are counter-productive. Because they give us partial results.

For example, imagine we want to find any line that mentions anything about password configuration. We don't care if the word appears in the text file with a capital P or a lowercase p letter.

So grep has an useful option that tells it to ignore if the text has lowercase or uppercase letters. It will match every line that contains that word, no matter the case of the letters.

We will use the "-i" ignore case option:

```
grep -i 'password' /etc/ssh/sshd_config
```

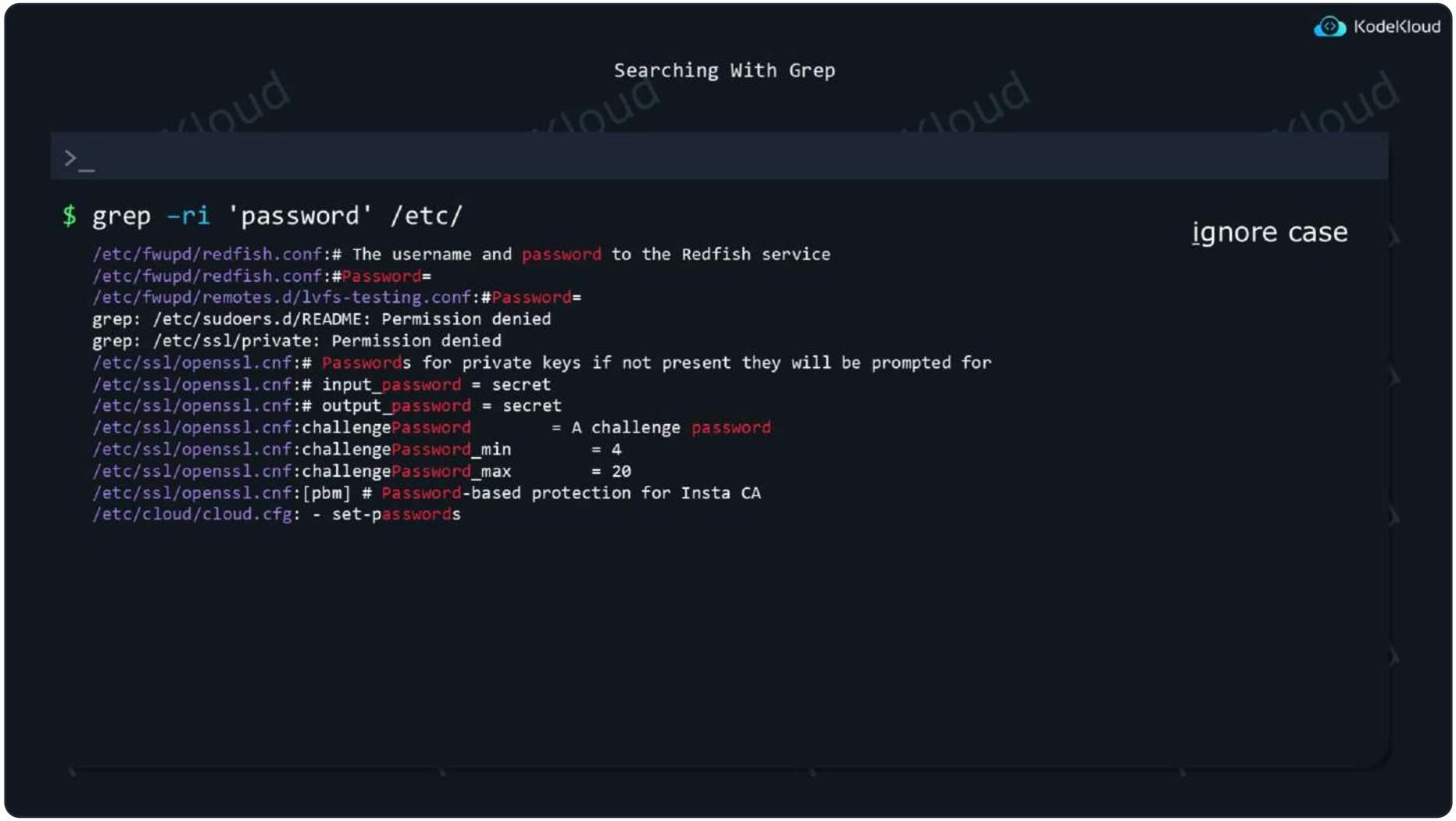


Instead of searching through a specific file we can also search through all files that exist under a directory, and its subdirectories. To do this, we add the -r (--recursive) option, and instead of specifying the path to a file, we specify the path to a directory:

grep -r 'password' /etc/

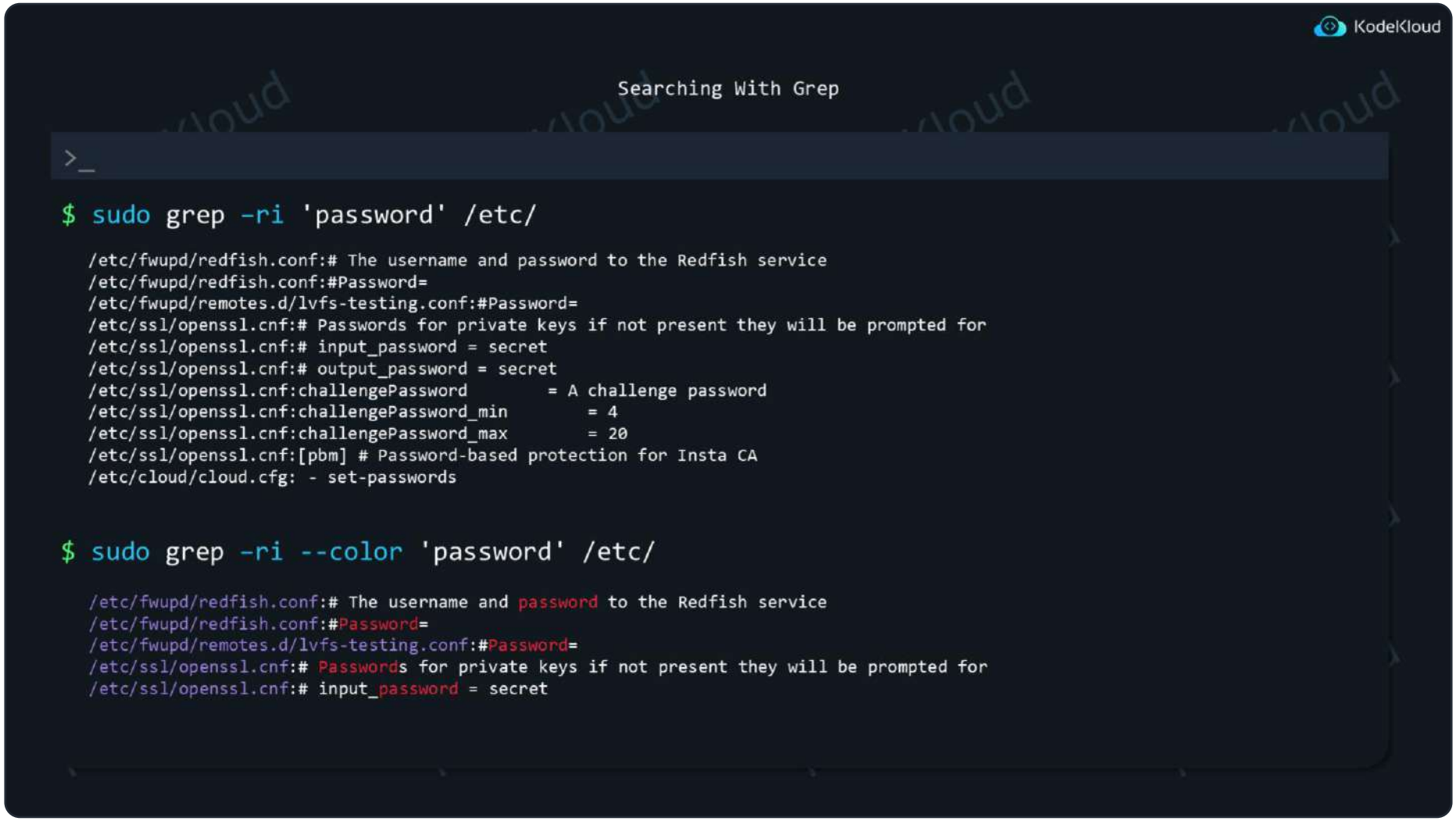
By default, grep uses different colors to highlight important stuff and make the output easier to scan through. In this case, the matched files are also highlighted so we can see at a glance

where this text was found. Also, note the lines that mention "permission denied". We'll get to those soon.



We can group the recursive search option, with the -i ignore case option:

grep -ri 'password' /etc/



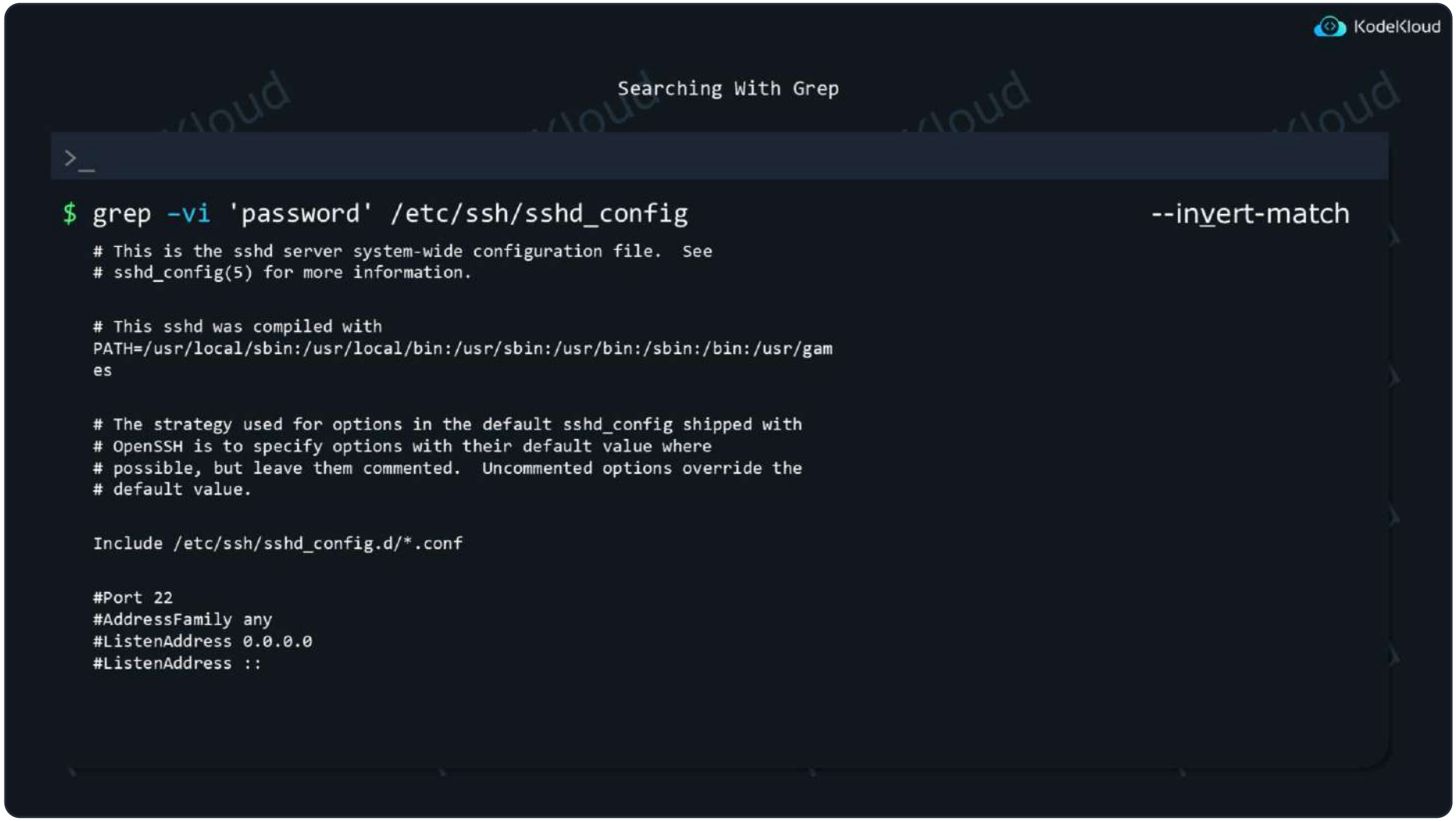
Our regular user cannot read some of these files, so we get permission denied errors. If we want to search through system files, that only the administrator account, called root, can access, we can add "sudo" in front of our grep command:

sudo grep -ri 'password' /etc/

But right away we'll notice a difference. The output is not color-coded anymore, making it harder to read and spot what we're looking for. To force grep to color code output, we can add the --

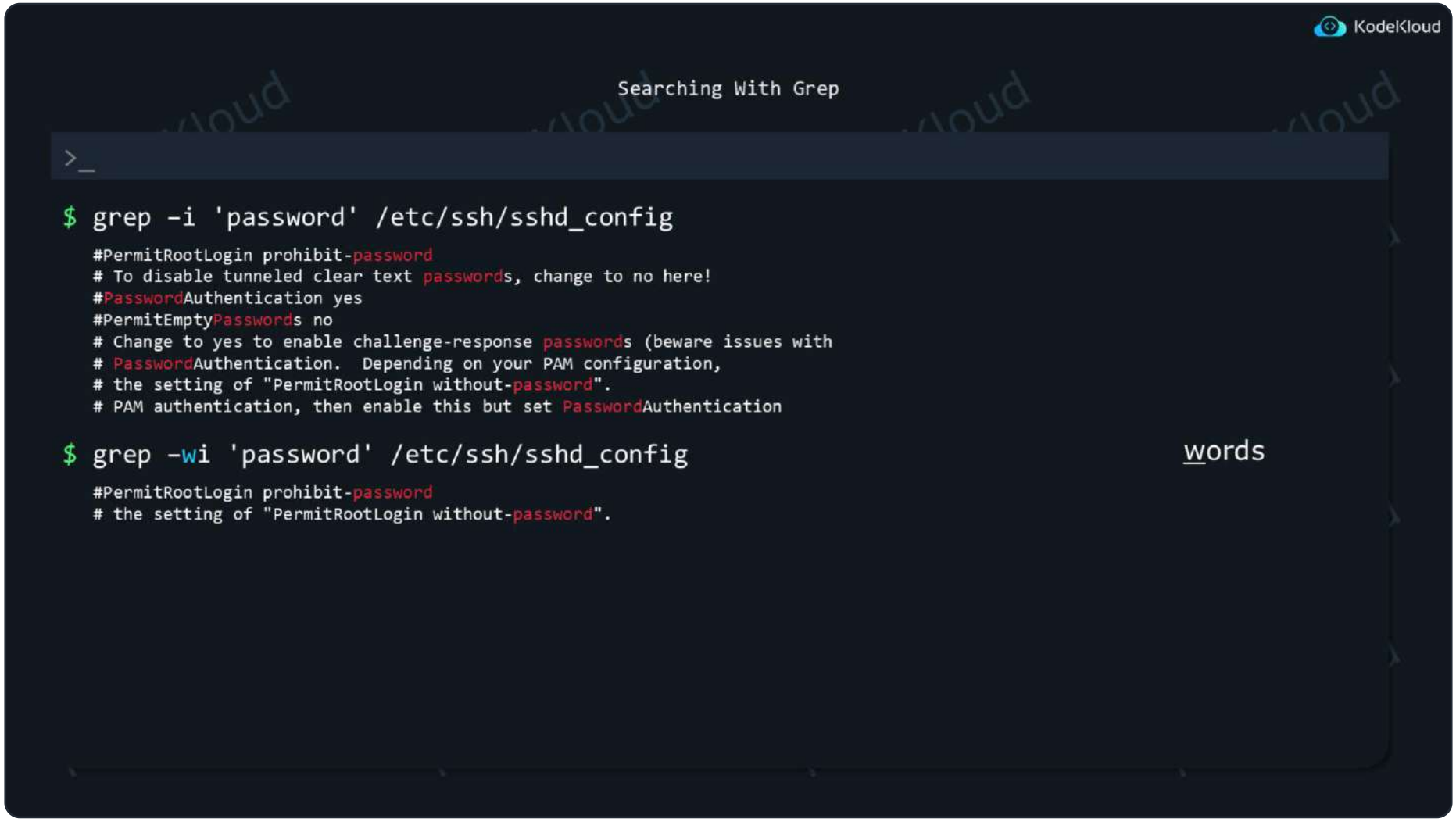
color option:

sudo grep -ri --color 'password' /etc/



We can also "invert" search results. For example, we can search for lines that don't contain the text "password" by adding the -v (--invert-match) option:

grep -vi 'password' /etc/ssh/sshd_config



Now consider this command:

grep -i 'password' /etc/ssh/sshd_config

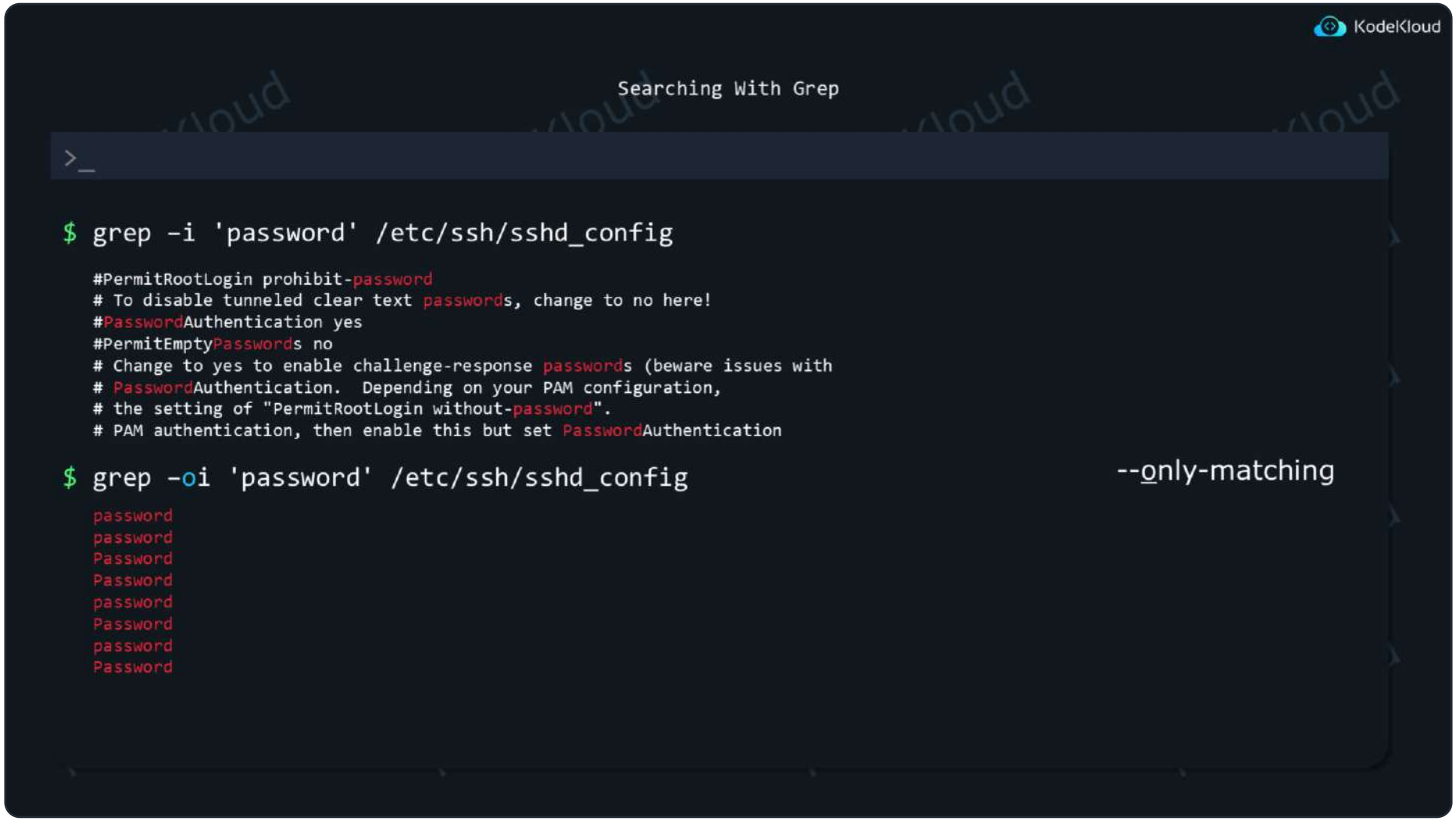
This matches two words: "password" singular, but also "passwords" plural. Simple because the plural form contains the same word "inside", so to speak, but just has an added "s" letter at the end. But what if we're only interested in matching the word "password" specifically? We can pass the –w word option to grep.

grep -wi 'password' /etc/ssh/sshd_config

Now we'll notice an unexpected thing. We have just two lines that match this search expression. Why is that? For example, why does the line

#PasswordAuthentication yes

not appear in the results here? Because "PasswordAuthentication", as far as grep is concerned, is a different word. Just like the word "book" does not mean the same thing for us as the word "booking". So when we use the –w word option, grep will only match this exact word, without any extra letters before it or after it. Although, as we can see here, the word can have a minus sign before it, or a period, which ends a sentence, any punctuation sign basically, but not a letter.



Throughout these examples, we can notice how grep displays the entire line that was matched:

grep -i 'password' /etc/ssh/sshd_config

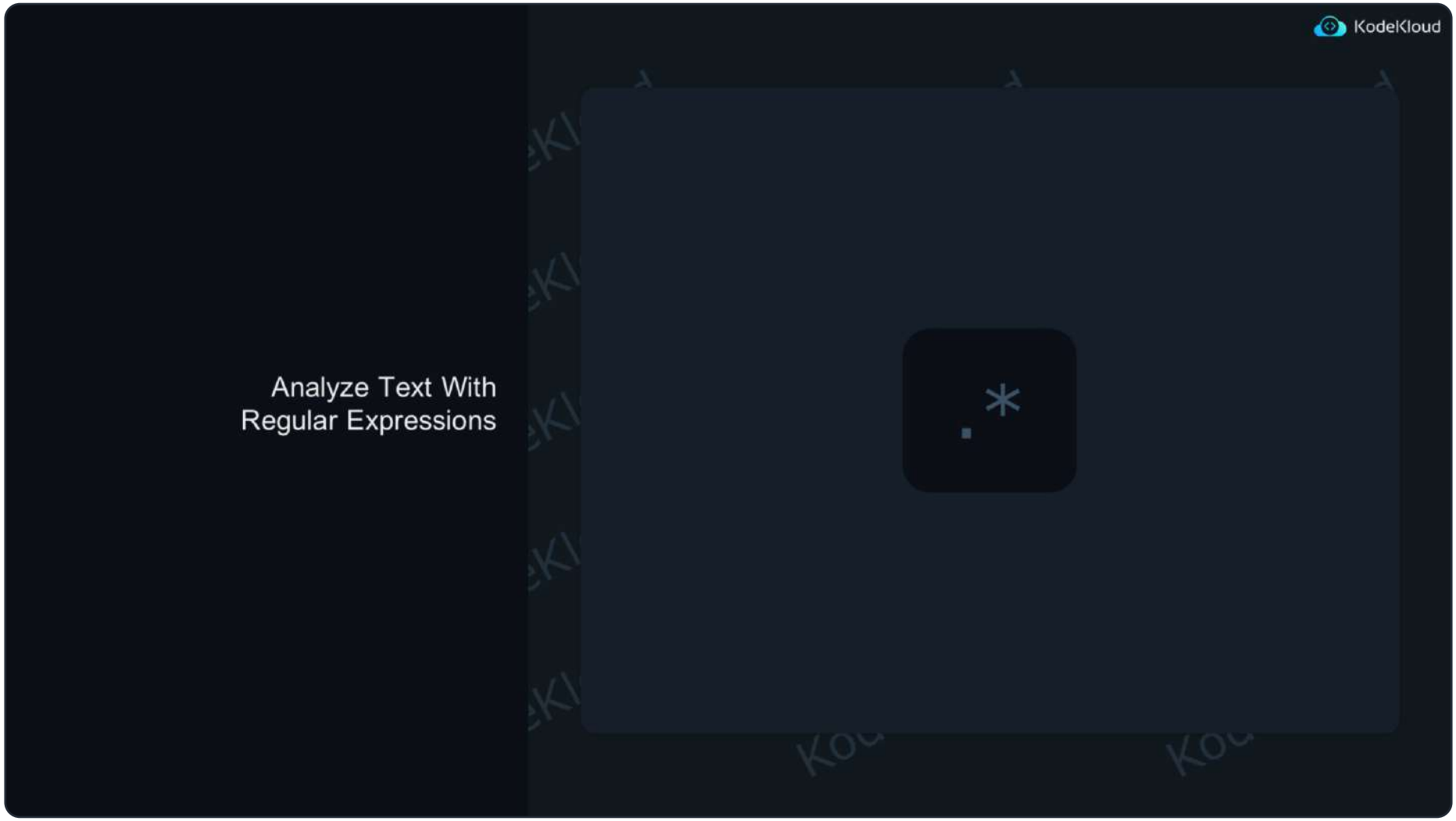
This is useful as it displays the entire context, showing us where these matched characters or words have been found. But sometimes, we will only want to extract the results themselves, not caring about the rest of the line. We can tell grep to do this by passing the -o (--only-matching) option.

```
grep -oi 'password' /etc/ssh/sshd_config
```

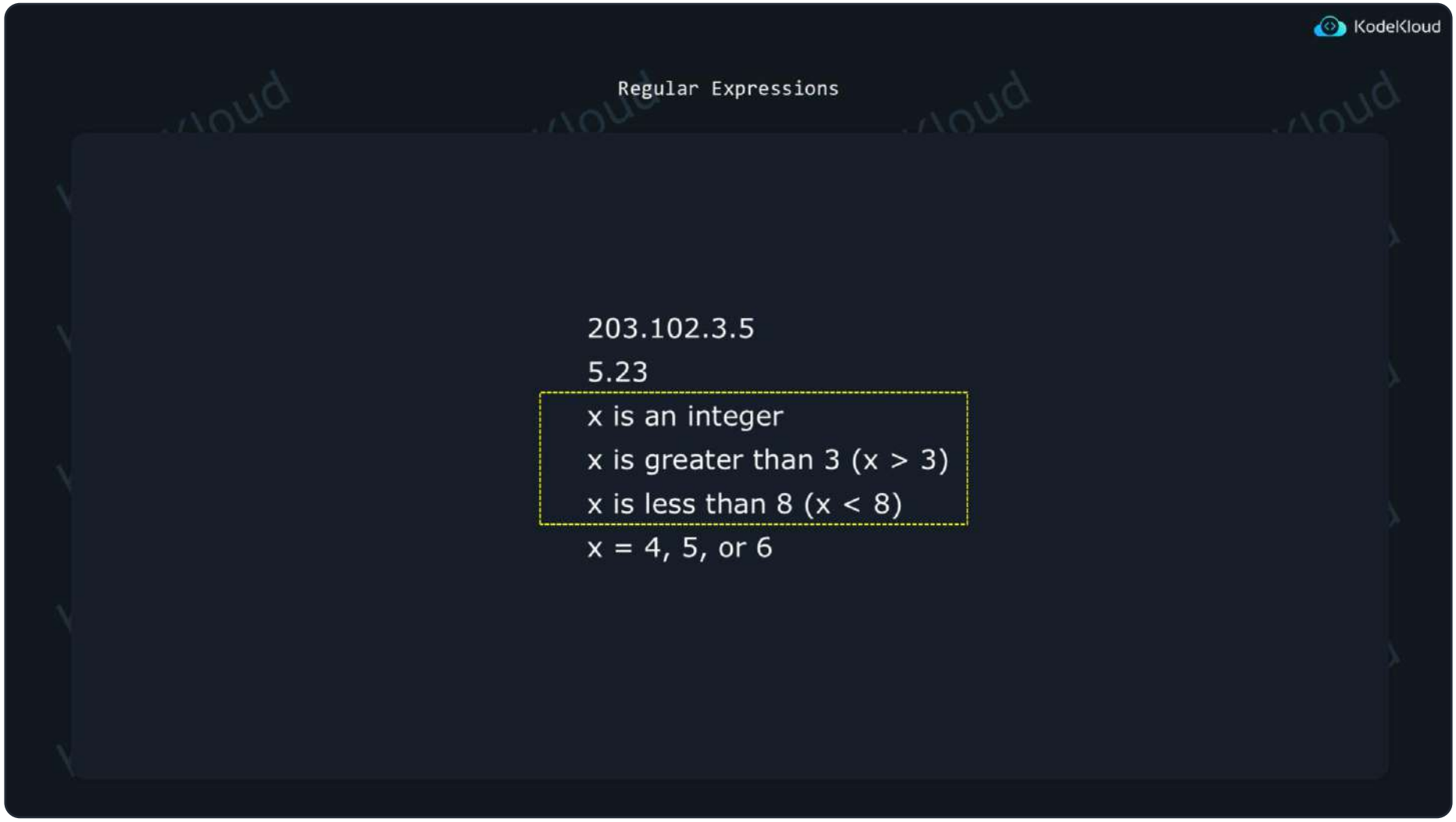
We can see that as output to simply look at, this isn't very useful to us as humans. But it can be useful in tasks that automatically extract and process data. Also, when we use this with more complex search patterns, it becomes more useful, as we won't see the same word repeated in the results list. But, rather, different results. For example, we could have a search pattern that shows us what settings are enabled for an application. We'd see the setting names that are enabled, but without the rest of the content on the line, since that would be of no importance in that case.

KodeKloud

Visit www.kodekloud.com to discover more.



In this lesson we'll explore regular expressions, also called RegEx for short.



In our previous commands, we used simple search patterns, looking for some specific pieces of text, like "password". But what if we need more complex search conditions?

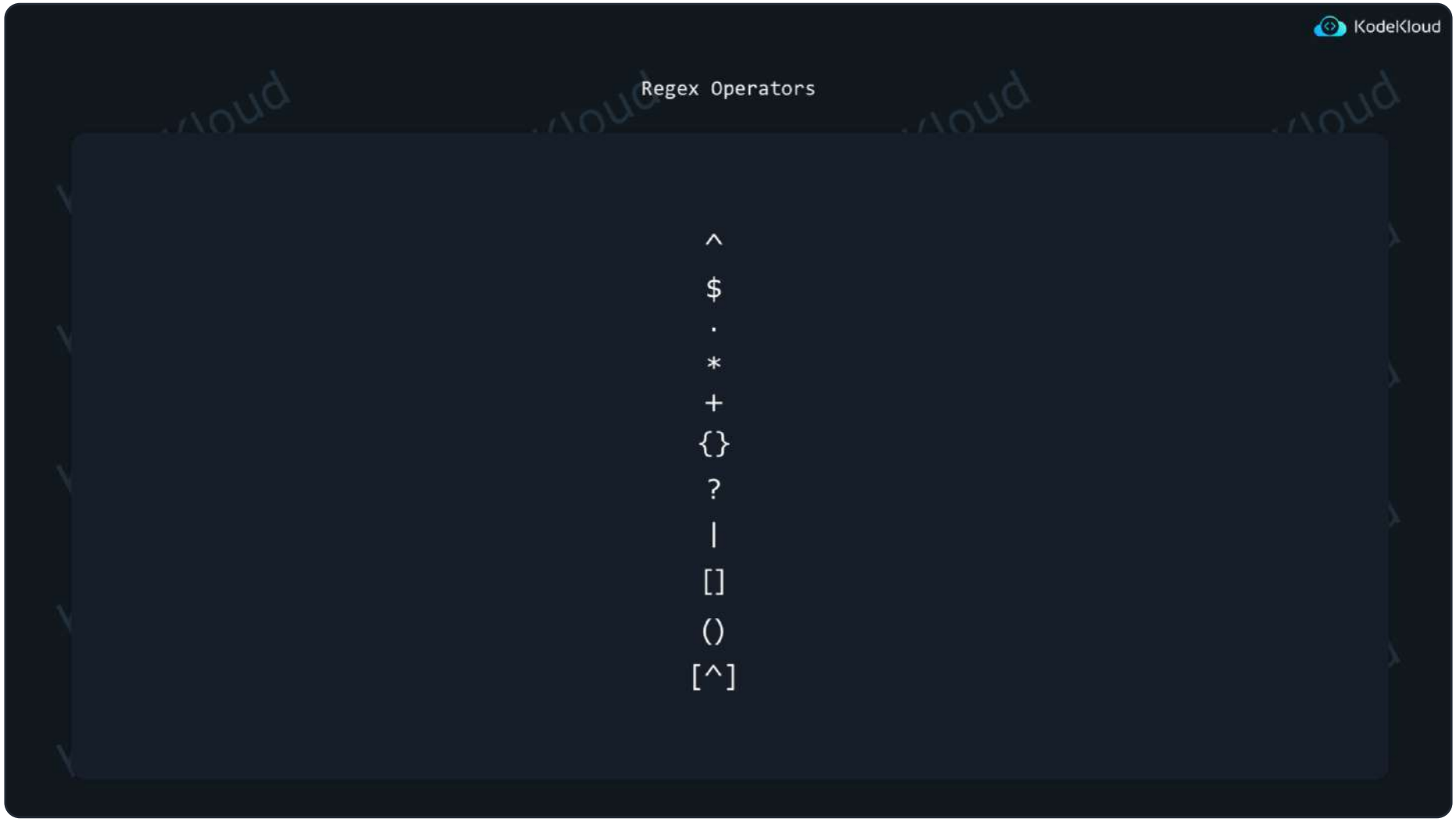
Imagine we have some application code scattered in hundreds of files. And we need to extract all IP addresses used in this app. That would require more advanced search instructions. An IP has a form like 203.102.3.5. But we can't just make a search pattern look for numbers with a . between them, as this would also match numbers like "5.23", which are not IP addresses.

In math, we can say something like:

x is an integer
x is bigger than 3, x>3
x is smaller than 8, x<8

And this would mean x is either 4, 5, 6 or 7. Regular expressions work in a similar way. We specify some conditions, tie all of them together, and our search pattern only matches what perfectly fits within those conditions.

Let's start out with some super simple examples and then build up to slightly more advanced expressions.



All regular expressions are built with the help of operators like:

^ (caret)

$ (dollar sign)

. (period)

* (asterisk)

+ (plus sign)

{ } (braces)

question mark

vertical pipe

brackets

parentheses

square brackets with caret

Let's see what each of them does.

In Linux, configuration files can have lines that begin with a # sign. These are called "commented lines". They are inactive. The program looking for settings in such a file will ignore all lines that begin with a #. But comments are useful for humans, as they let us see examples of config settings in that file, and descriptions for what they do, without interfering with the program that reads them.

This means that we can search for commented lines, specifically, by creating a regular expression that looks for all lines that begin with a "#".

The regular expression would be:

^#

And we can use it in grep like this:

grep '^#' /etc/login.defs

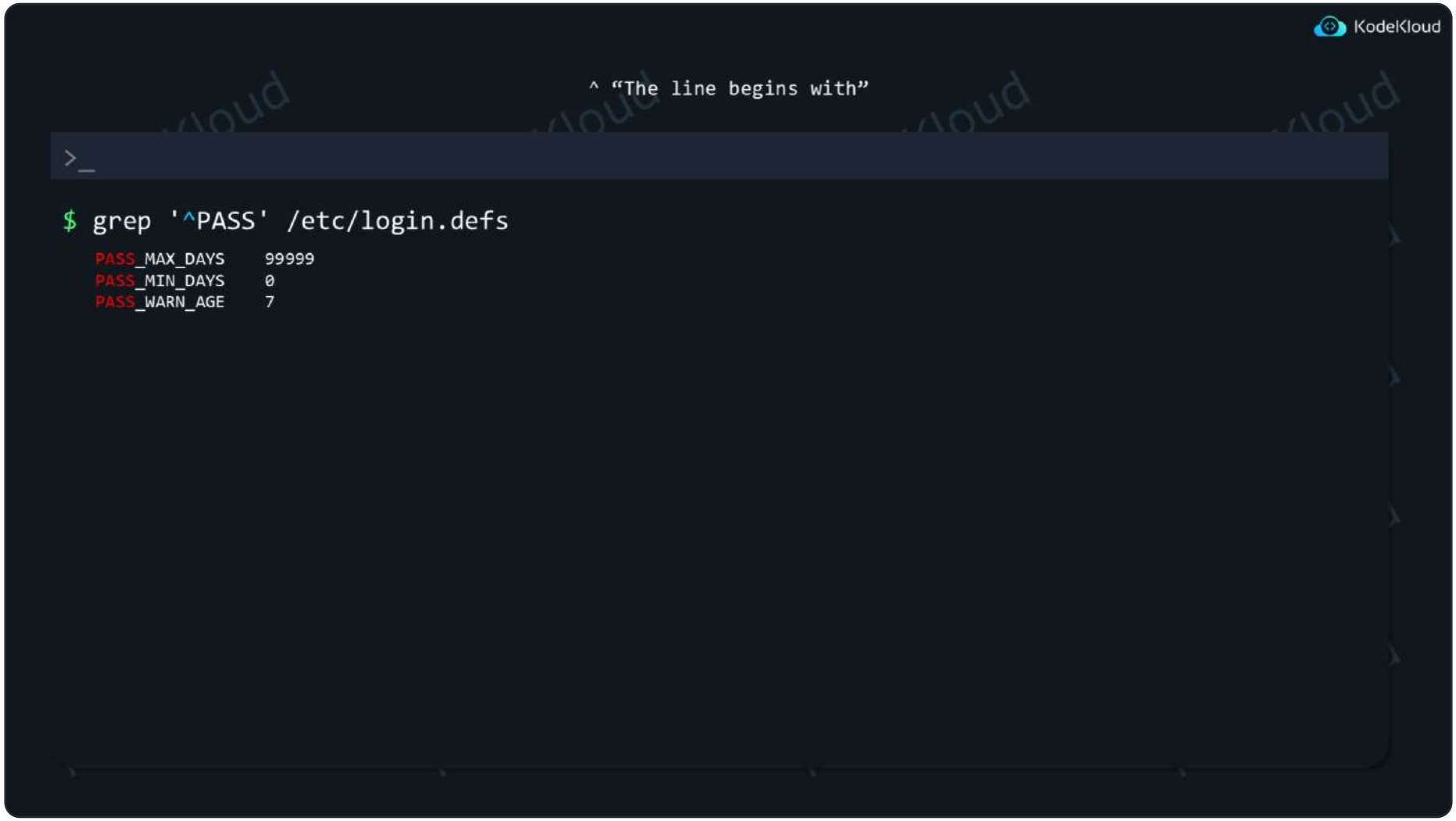
But this doesn't seem to be terribly useful. We'd just see the commented lines we see above. However, combined with grep's option to invert results, it becomes useful.

By inverting we tell grep to show us lines that don't begin with a # sign.

grep -v '^#' /etc/login.defs

And now we can see exactly what we wanted: settings actively used.

Imagine how useful this would be in a very long file with hundreds of comments that make it hard to spot what we're looking for.



And just to show another example, we could look for lines that start exactly with these four letters: PASS. To do that, we precede our search pattern with the ^ sign.

grep '^PASS' /etc/login.defs

Now let's imagine a different scenario. Let's say we need to change a setting that is currently set to "7" days. Easy enough, we could look for a 7, right?

grep '7' /etc/login.defs

But this shows us some stuff we don't need.

However, we know that our file uses this syntax: a VARIABLE NAME followed by a [space] then a VARIABLE VALUE. The variable value is last. Which means that if some variable is set to have a value of 7, this number will be the last character on the line.

We can use a regex pattern that looks for a line that ends with "7", with this expression:

7$

In grep, we'd use this command with our RegEx pattern:

grep -w '7$' /etc/login.defs

We also added the -w match word option to grep. Otherwise another line would also match. One that has multiple sevens at the end. But with -w we essentially told grep to match only what contains a single 7 digit at the end.

To look for all lines that end with the text "mail":

grep 'mail$' /etc/login.defs

Please take note how these operators are placed differently

mail$
^PASS

If you mix up their location you won't get any results, which can lead to confusion why your regex is not working. To easily remember their locations, think like this:

The "line begins with" operator, ^, should be placed at the beginning of my search pattern.
The "line ends with" operator, $, goes at the end of my pattern.

Anywhere you add a . in your expression, it will match any character in that spot. For example:

c.t will match cat, cut, crt, and even c1t or c#t. But it won't match ct. There must be exactly one random character between c and t. With c..t it will only match what has two characters in the middle.

Example grep command:

```
grep -r 'c.t' /etc/
```

We can see that even "execute" is a match because that sequence fits inside that word

If we'd only want to match whole words with this RegEx, and not parts of bigger words, we can use grep's -w option:

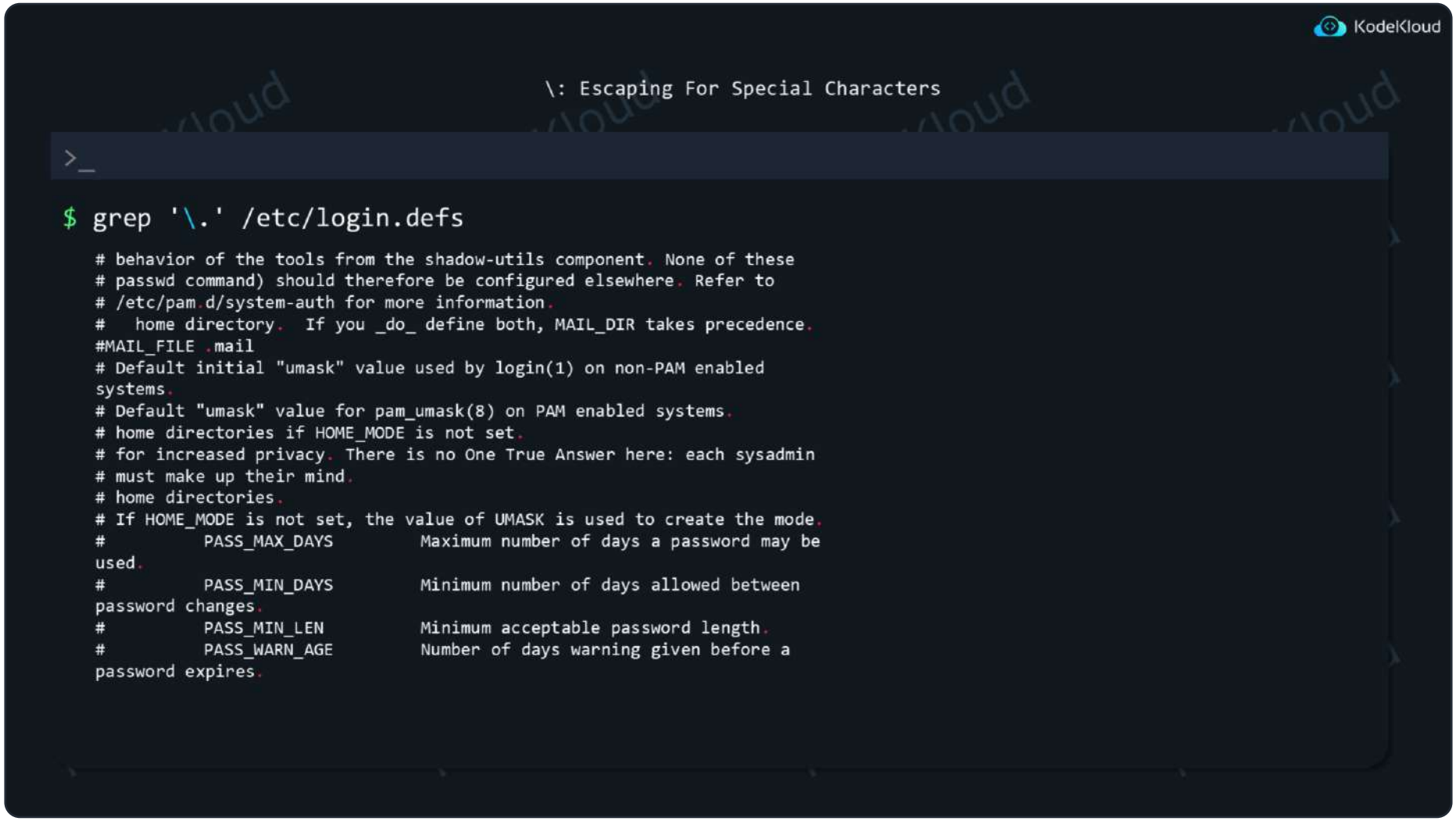
grep -wr -r 'c.t' /etc/

And this brings us to an interesting problem. This . has a special meaning in RegEx. But what if we need to search for an actual . in our text?

This won't work:

grep '.' /etc/login.defs

as this RegEx will basically match each character, one by one.



The solution, however, is simple. We look for a regular . by escaping this. Escaping is how we tell our grep command: "Hey, don't consider this . a match any one character operator. Instead, interpret it as a regular .".

To escape some special character we just add a backslash \ before it. Instead of

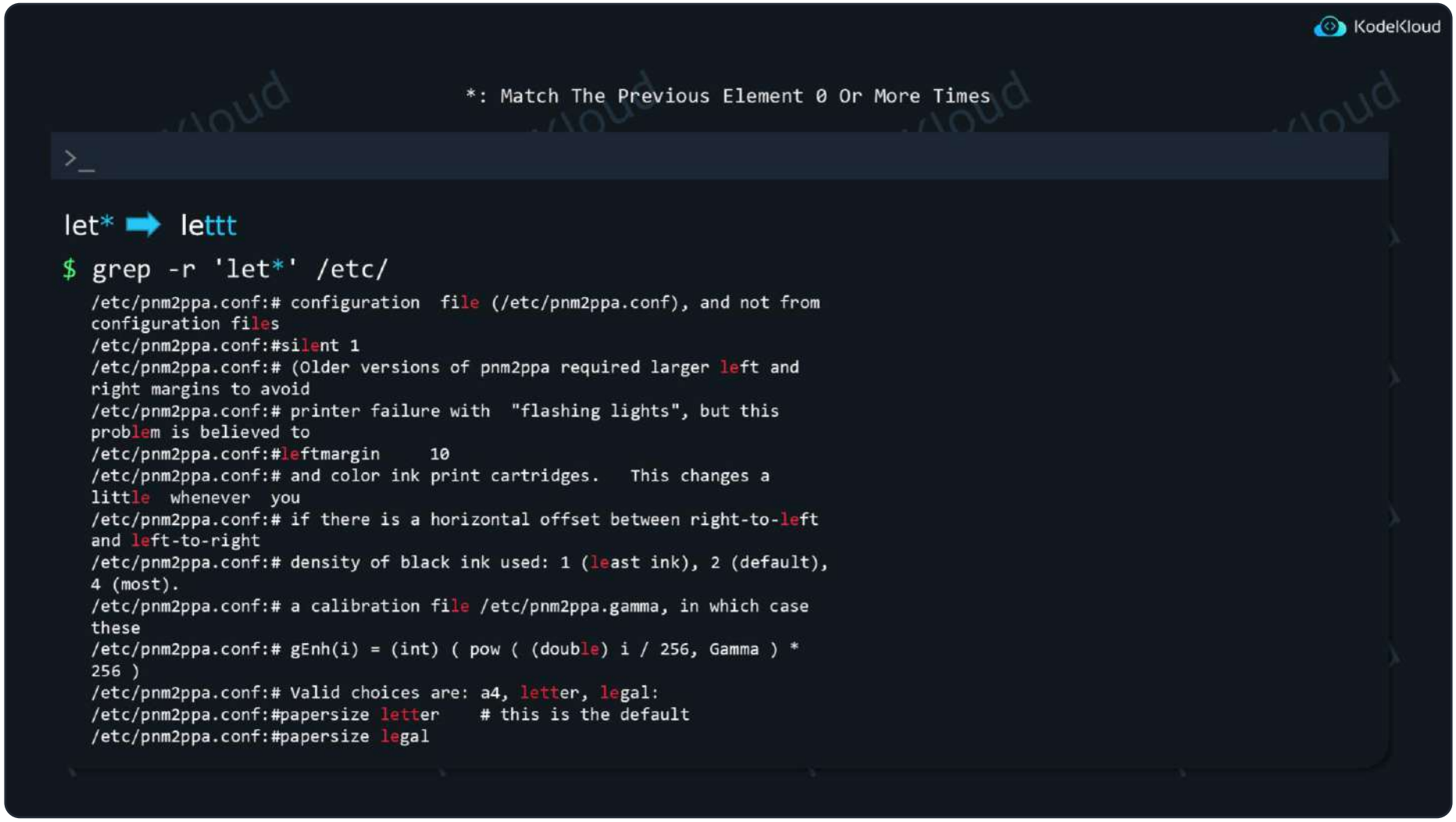
.

we write

\.

So our grep command becomes:

grep '\.' /etc/login.defs



An expression like:

let*

will match le, let, lett, lettt, and so on, no matter how many "t" letters at the end. Another way of saying this is that the * allows the previous element to:

be omitted entirely

appear once

appear two or more times

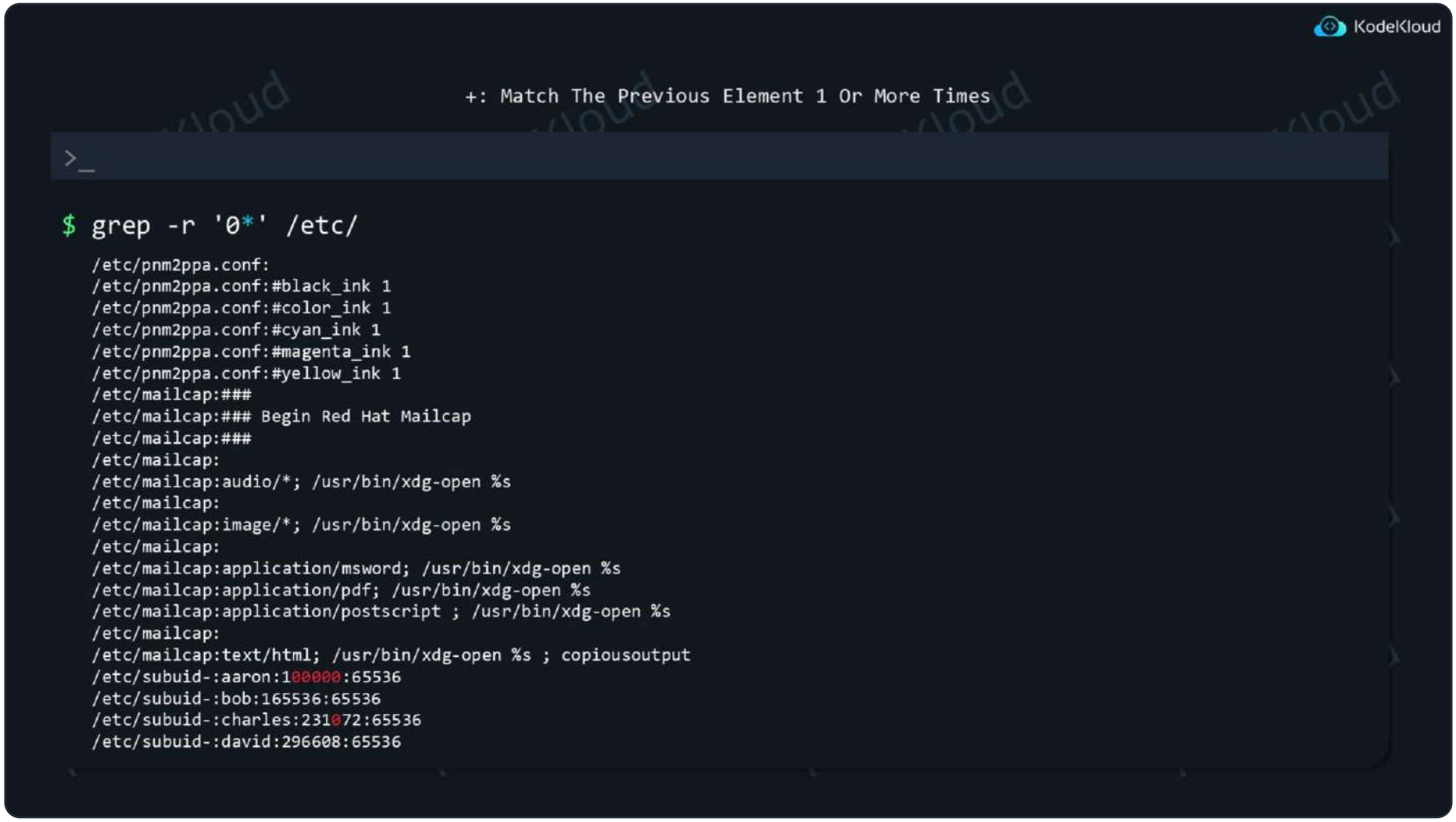
In a grep command, we'd use it like this:

```
grep -r 'let*' /etc/
```

The * operator can be paired up with other operators. For example, to look for something for sequences that begin with a /, have 0 or more characters and between, and end with another /, we could use:

/.*/

Since . matches any ONE character and the * says "previous element can exist 0, 1, 2 or many more times" we basically allow any sequence of characters to exist between / and /.



Now let's say we want to find all sequences of characters where 0 appears one or more times. We might be tempted to use:

grep -r '0*' /etc/

But this also matches lines that contain no zeroes at all. Why is that? Because * lets the previous character exist one or more times, but also ZERO times. It basically allows that element to be optional in our search. So, we need another operator that forces the element to exist at least one time, or many more. + does this:

0+

would find strings like:

0
00
000
0000

and so on

We might think we can use this regular expression in grep like this:

grep -r '0+' /etc/

But this doesn't look like the result we want. Our + works like a literal + instead of an operator. Why is this? By default, grep uses "basic regular expressions".

Its manual page has this to say: "In basic regular expressions the meta-characters ?, +, {, |, (, and) lose their special meaning; instead use the backslashed versions \?, \+, \{, \|, \(, and \)"

That means, to use "+" as an operator here, we have to add a \ before it, make it "\+". Our command becomes:

grep -r '0\+' /etc/

But this can become confusing really fast. We saw we already use something like \. to turn the . operator into a regular . Now we use \ to turn a regular + into the + operator. It will be hard to keep track of what to prepend with a backslash and what not to. So we can go the easier route, use "extended regex" instead, which doesn't require us to backslash ?, +, {, |, (, and).

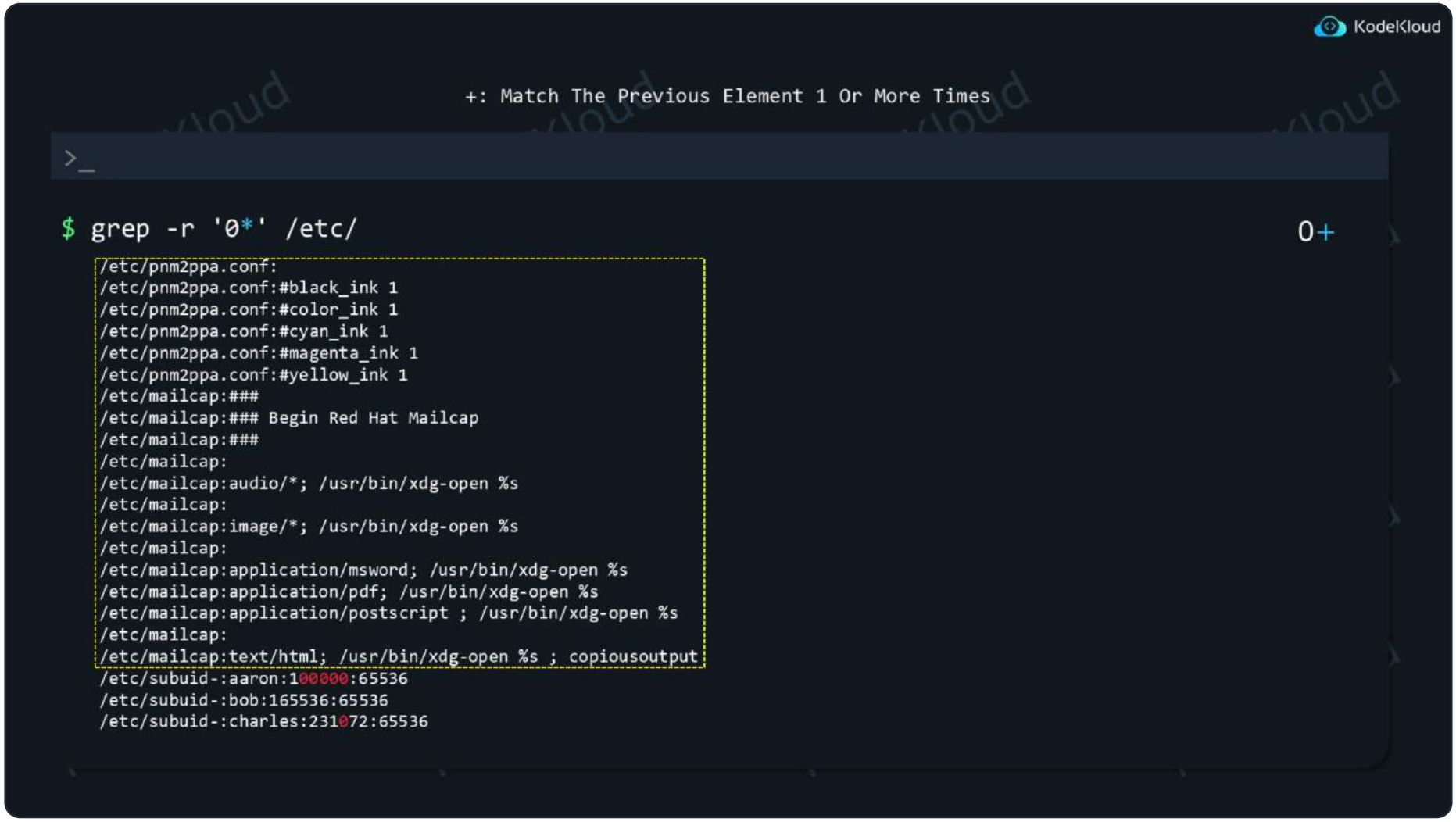
We use extended regex by adding the -E option to grep

grep -E -r '0+' /etc/

Or even easier, we use the equivalent egrep command. Using "egrep" is the same as typing "grep -E".

egrep -r '0+' /etc/

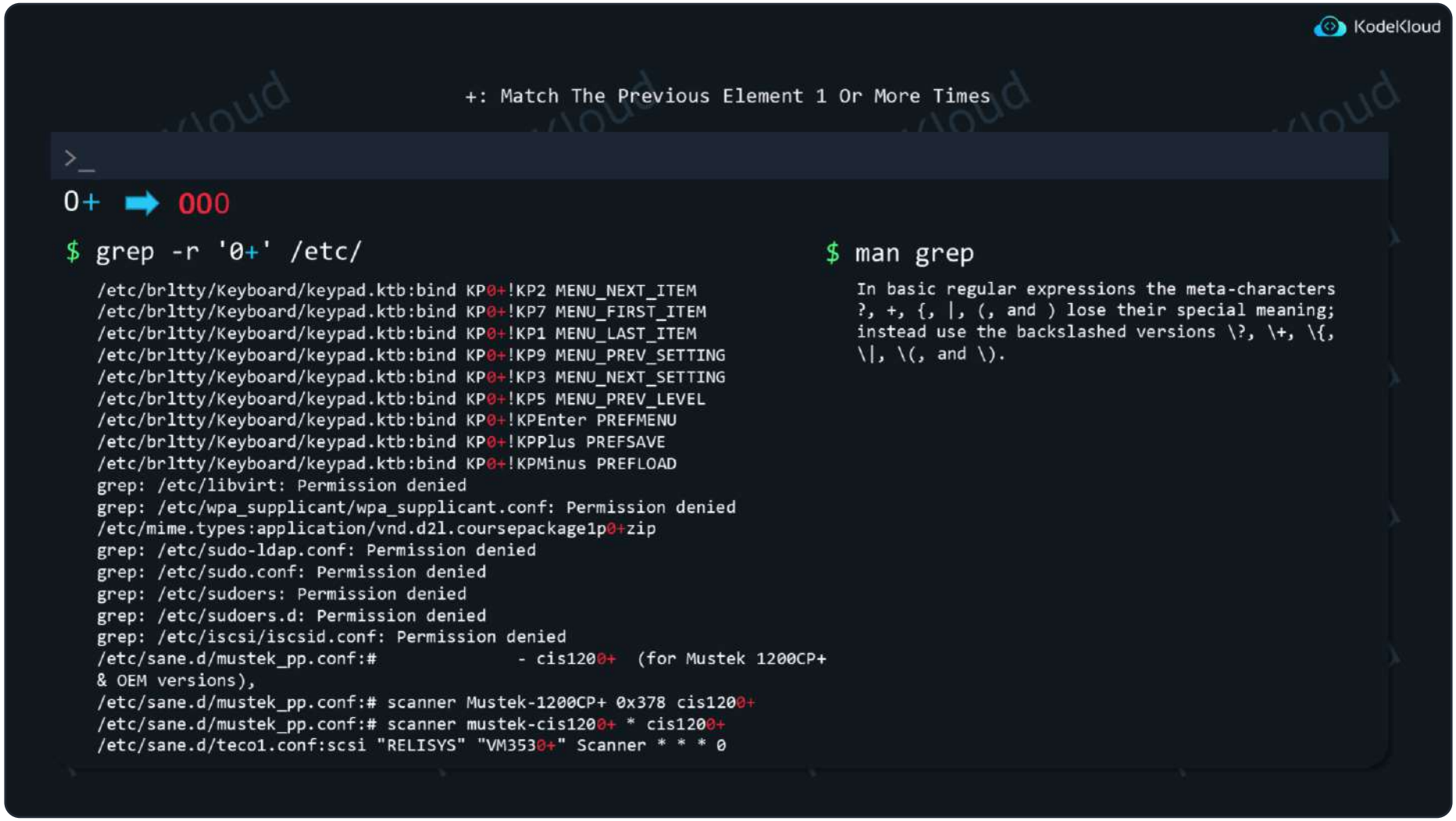
So you can make it a habit to always use egrep instead of grep, to avoid mistakes where you forgot to backslash one of the regex operators.



Let's say we want to find all sequences of characters where 0 appears one or more times. We might be tempted to use:

grep -r '0*' /etc/

But this also matches lines that contain no zeroes at all. Why is that? Because * lets the previous character exist one or more times, but also ZERO times. It basically allows that element to be optional in our search. So, we need another operator that forces the element to exist at least one time, or many more. + does this:



0+

would find strings like:

0
00

000

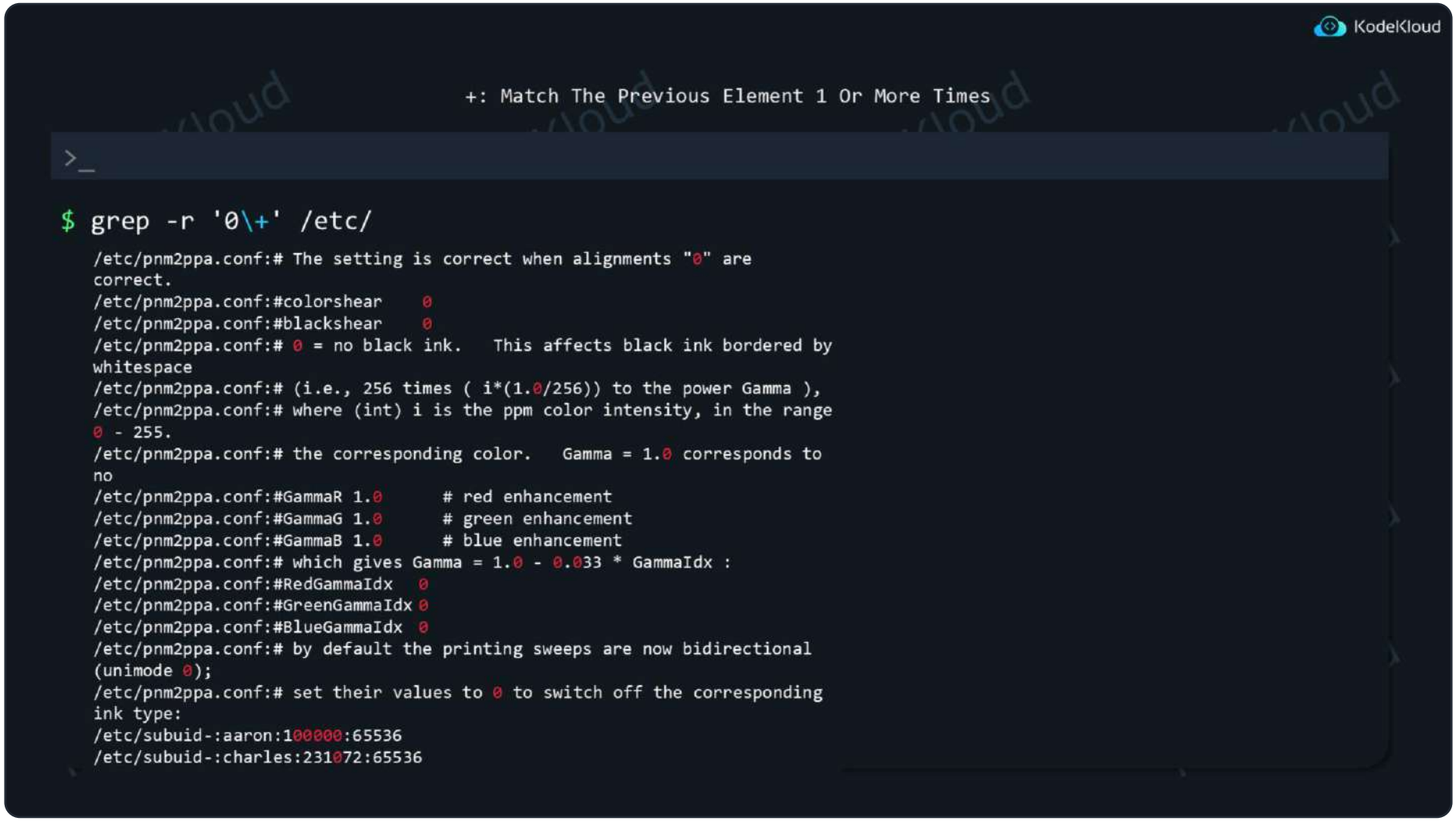
and so on.

We might think we can use this regular expression in grep like this:

grep -r '0+' /etc/

But this doesn't look like the result we want. Our + works like a literal + instead of an operator. Why is that? Because, by default, grep uses "basic regular expressions".

Its manual page has this to say: "In basic regular expressions the meta-characters ?, +, {, |, (, and) lose their special meaning; instead use the backslashed versions \?, \+, \{, \|, \(, and \)"



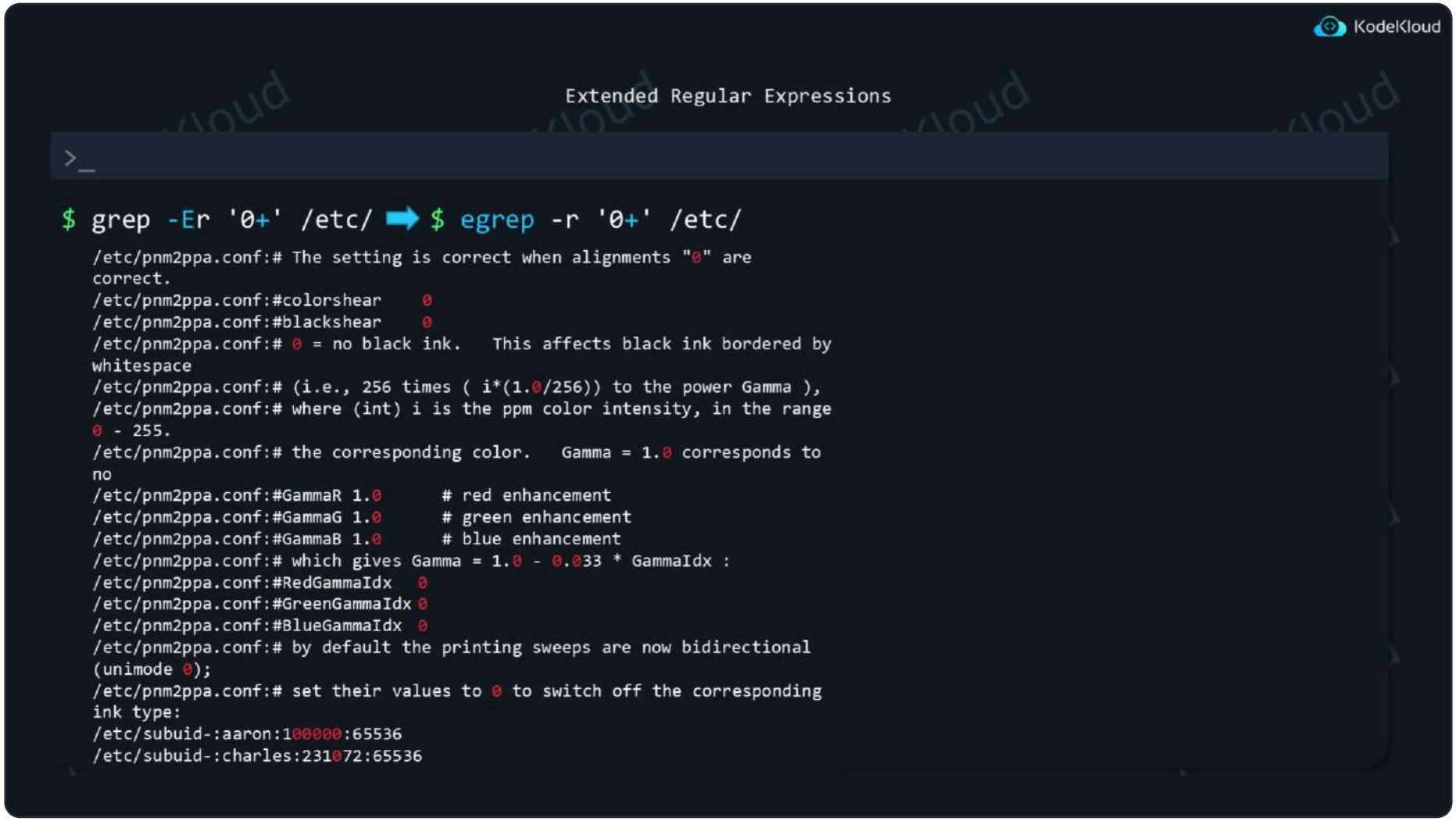
That means, to use "+" as an operator here, we have to add a \ before it, make it "\+". Our command becomes:

grep -r '0\+' /etc/

But this can become confusing fast. We saw we already use something like \. to turn the . operator into a regular . Now we use \ to turn a regular + into the + operator. It will be hard to keep track of what to backslash and what not to. So, we can pick an easier route, use "extended regex" instead, which doesn't require us to backslash ?, +, {, |, (, and).

We'll explore this in our next lesson.

In this lesson we'll explore extended regular expressions.



In a previous lesson we mentioned how we can escape special characters in grep. But sometimes escaping turns a . operator into the regular . character. Other times, it works in reverse. Escaping a + sign turns it from a regular + into the + operator.

So to avoid confusion we can simply use extended regular expressions. Which won't require us to remember what characters need to be escaped for grep to consider them RegEx operators. Now we'll only need to escape characters when we want to turn them from operators into regular characters we want to look for, like a + sign, or a period.

We use extended regex by adding the -E option to grep. That's E with a capital letter.

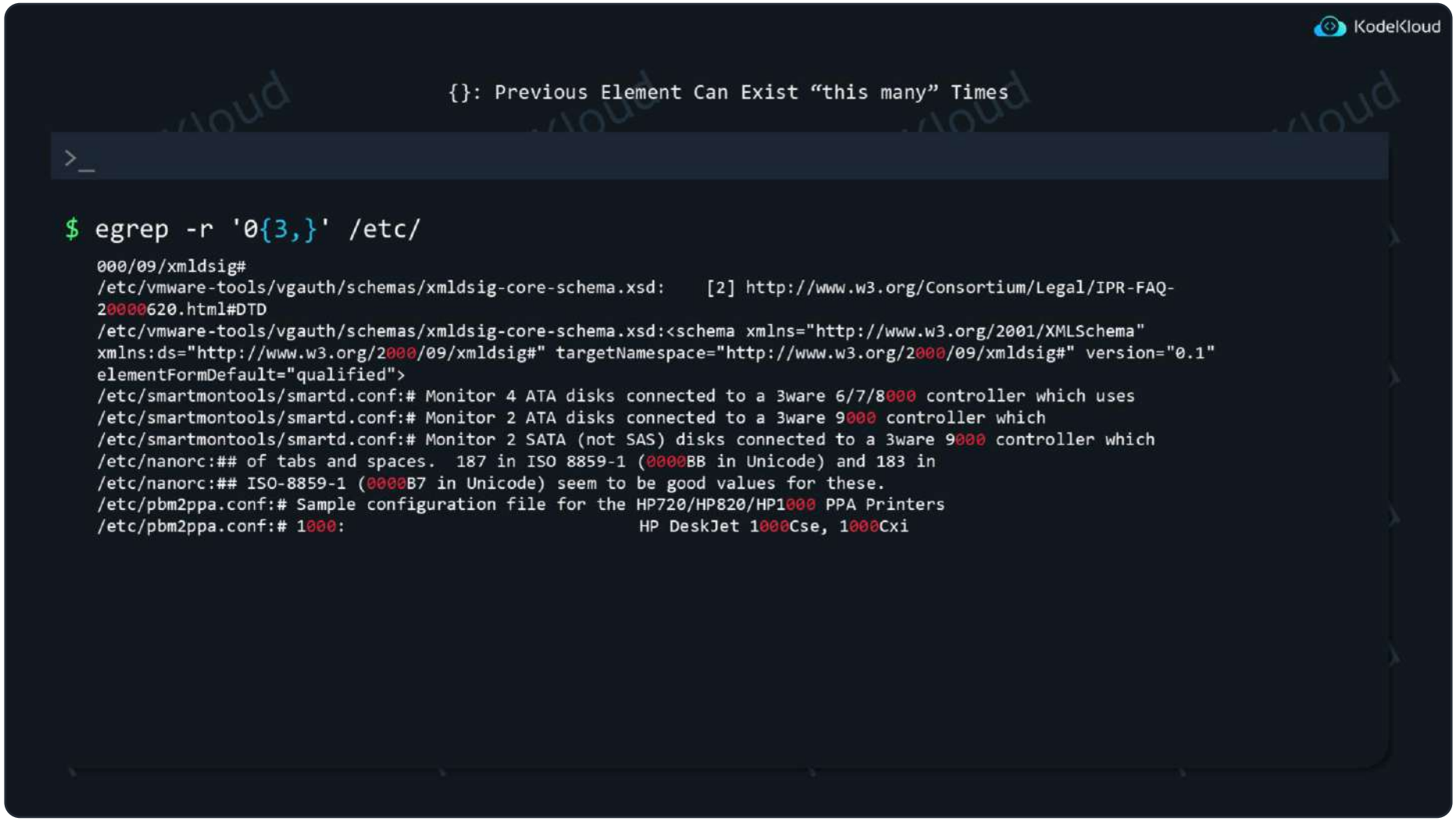
To find all lines that contain the digit 0, one or more times we can use:

grep -E -r '0+' /etc/

Or even easier, we can use the equivalent egrep command. Using "egrep" is the same as typing "grep -E".

egrep -r '0+' /etc/

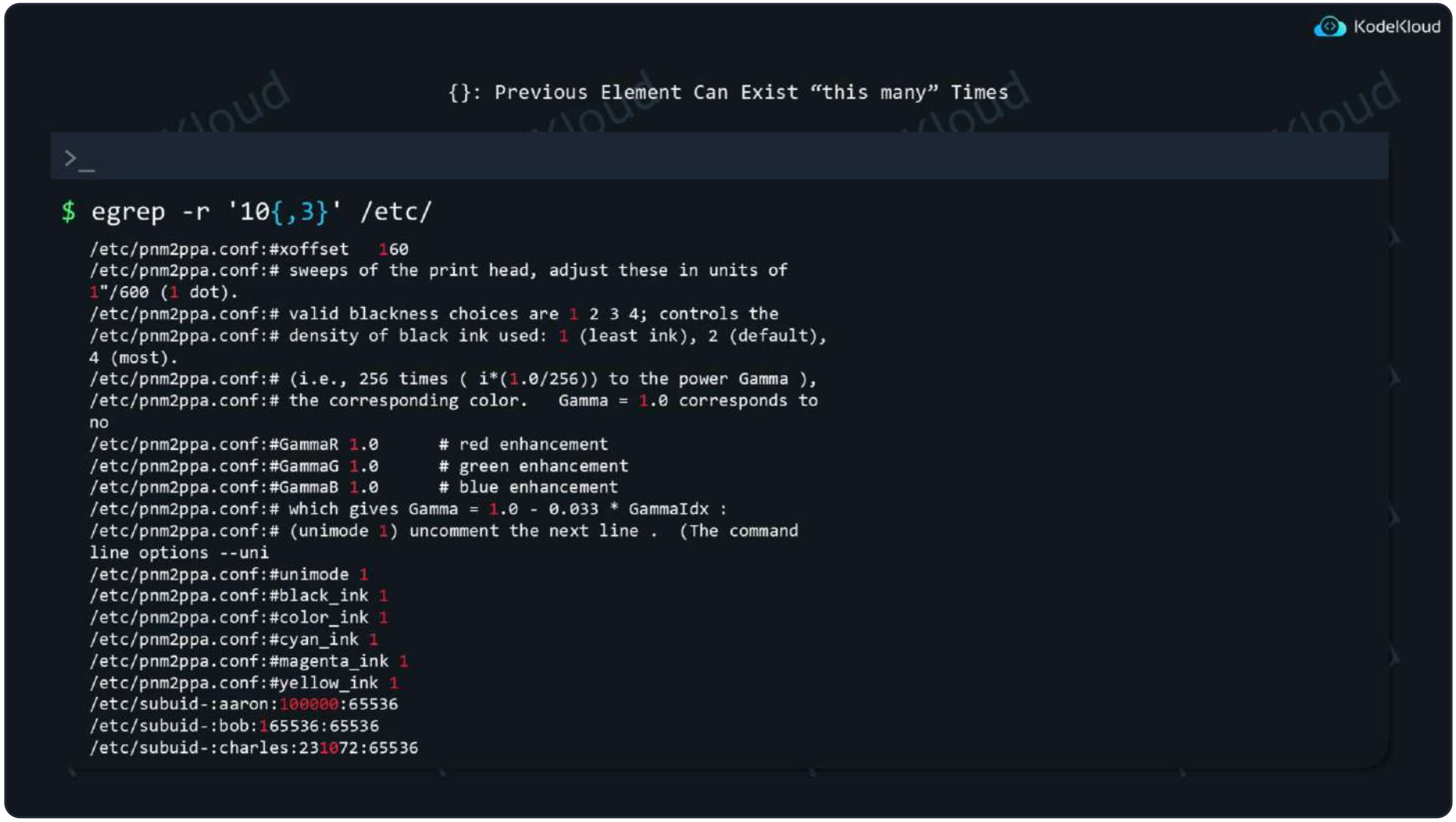
So, you can make it a habit to always use egrep instead of grep, to avoid mistakes where you might forget to escape a RegEx operator with a backslash.



To find all strings that contain at least 3 zeroes, we can use the { } operator. Within, we specify two values separated by a comma. The first value specifies the minimum amount of repetitions, for the previous element. And the second value specifies the maximum amount of repetitions.

In our case we'd need to specify the minimum amount of repetitions, 3. But since we don't want to place a limit on the maximum amount of repetitions, we can omit the second value. So our command becomes:

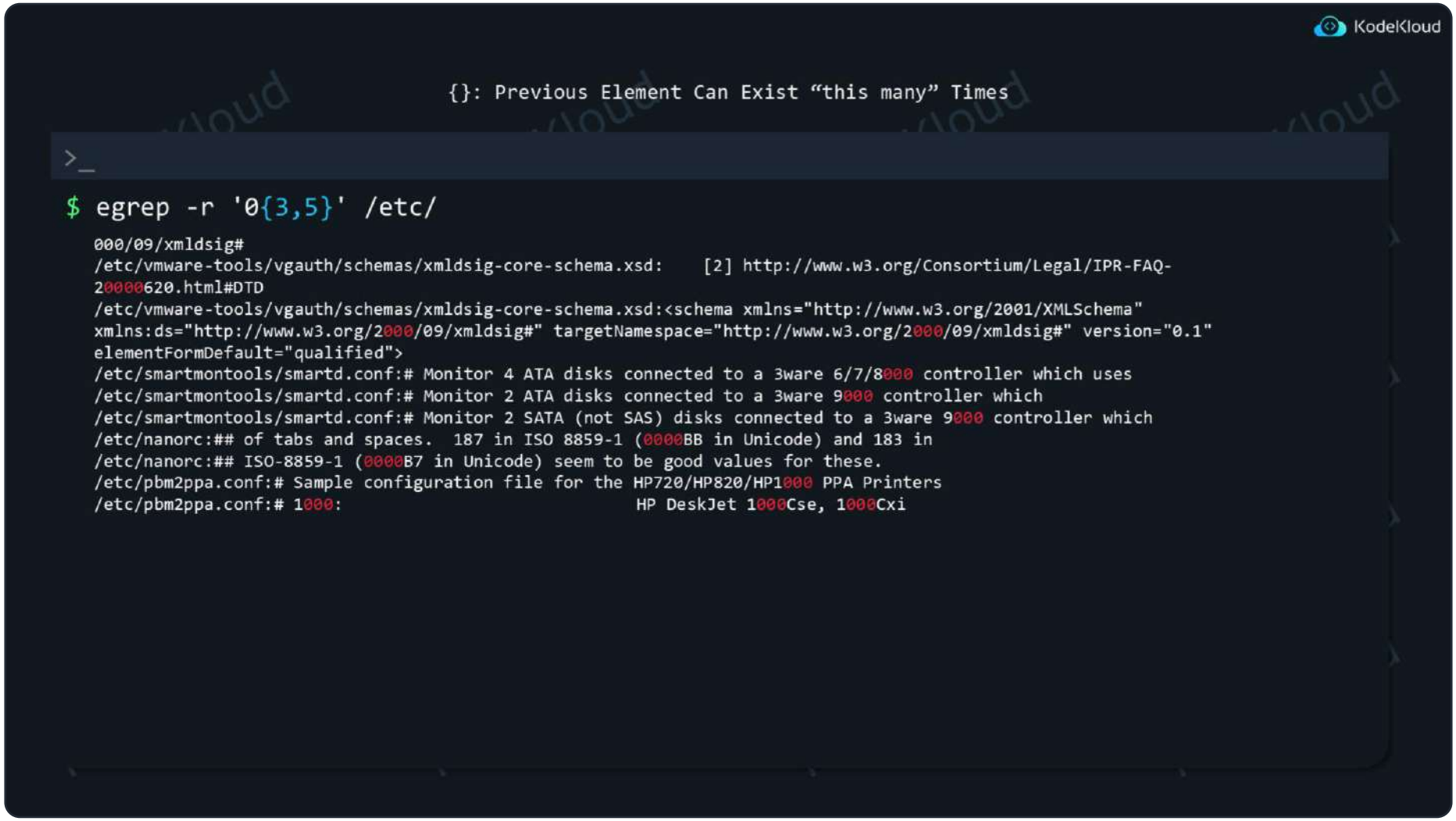
```
egrep -r '0{3,}' /etc/
```



To find all strings that contain "1" followed by at most 3 zeroes, first of all, we add the first 1 in our search pattern. Then we follow that up with the 0 digit, and add our regular expression. This time, in reverse. We omit to specify a minimum in the first field, but we specify our maximum number of repetitions in the second field:

egrep -r '10{,3}' /etc/

Note: This will also match ones followed by no zeroes. Since we didn't choose any specific minimum amount of repetitions.

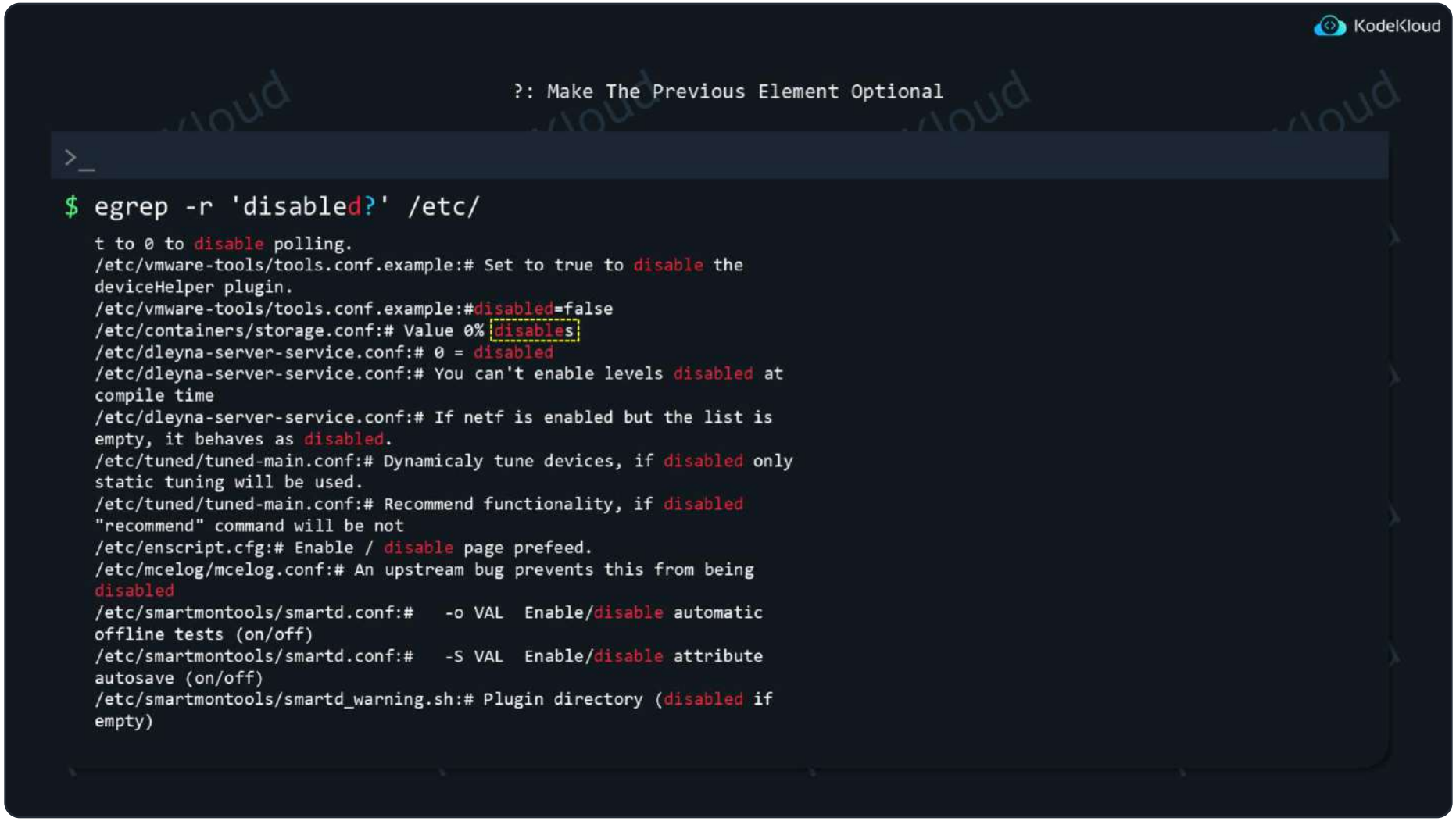


To find all strings that contain 3, 4 or 5 zeroes:

egrep -r '0{3,5}' /etc/

And to find all strings that contain exactly three zeroes, we use the operator in a different way. Instead of two fields with a minimum, and a maximum, we just place one value between our curly brackets:

egrep -r '0{3}' /etc/



? will let the previous element exist precisely 0 or 1 times. This basically makes it optional: it can exist once, or not at all.

Let's say we're trying to find all text that says "disabled" or "disable". This means the last "d" is optional, so we can write an expression like this in grep:

egrep -r 'disabled?' /etc/

Note that this also matches the word “disables.” This is a case where the letter “d” did not come at the end, and “disable” still matches, because it's still part of the word.

But just like before, we can add the -w word option to grep to get rid of that, so that only "disable" or "disabled" specifically match.

KodeKloud

|: Match One Thing Or The Other

```
$ egrep -r 'enabled|disabled' /etc/
/etc/vmware-tools/tools.conf.example:#                disabled.
/etc/vmware-tools/tools.conf.example:#disabled=false
/etc/dleyna-server-service.conf:# 0 = disabled
/etc/dleyna-server-service.conf:# You can't enable levels disabled at
compile time
/etc/dleyna-server-service.conf:netf-enabled=false
/etc/dleyna-server-service.conf:# If netf is enabled but the list is
empty, it behaves as disabled.
/etc/tuned/tuned-main.conf:# Dynamicaly tune devices, if disabled only
static tuning will be used.
/etc/tuned/tuned-main.conf:# Recommend functionality, if disabled
"recommend" command will be not
/etc/tuned/tuned-main.conf:# /etc/sysctl.conf.  If enabled, these sysctls
will be re-appliead
/etc/mcelog/mcelog.conf:# An upstream bug prevents this from being
disabled
/etc/mcelog/mcelog.conf:dimm-tracking-enabled = yes
/etc/mcelog/mcelog.conf:socket-tracing-enabled = yes
/etc/smartmontools/smartd_warning.sh:# Plugin directory (disabled if
empty)
/etc/nanorc:## To make sure an option is disabled, use "unset <option>".
```

However, to match if enabled or disabled, we could use the | operator.

'enabled|disabled'

So this basically matches what it finds on its left side or its right side.

egrep -r 'enabled|disabled' /etc/

And we could combine this with our previous trick (make last "d" letter optional), to also find variations like enable/enabled, disable/disabled. We'll also make the search case insensitive to also match variations where the words have uppercase letters in some spots.

egrep -ir 'enabled?|disabled?' /etc/

Now it's time to see how we can put all this knowledge to use and combine multiple regex operators to fine-tune our searches.

But first, let's learn about ranges and sets. A range is specified in the form of:

[a-z] - this will match any one lowercase letter, from a,b,c,d,e… to z

[0-9] - will match any one digit from 0,1,2… to 9

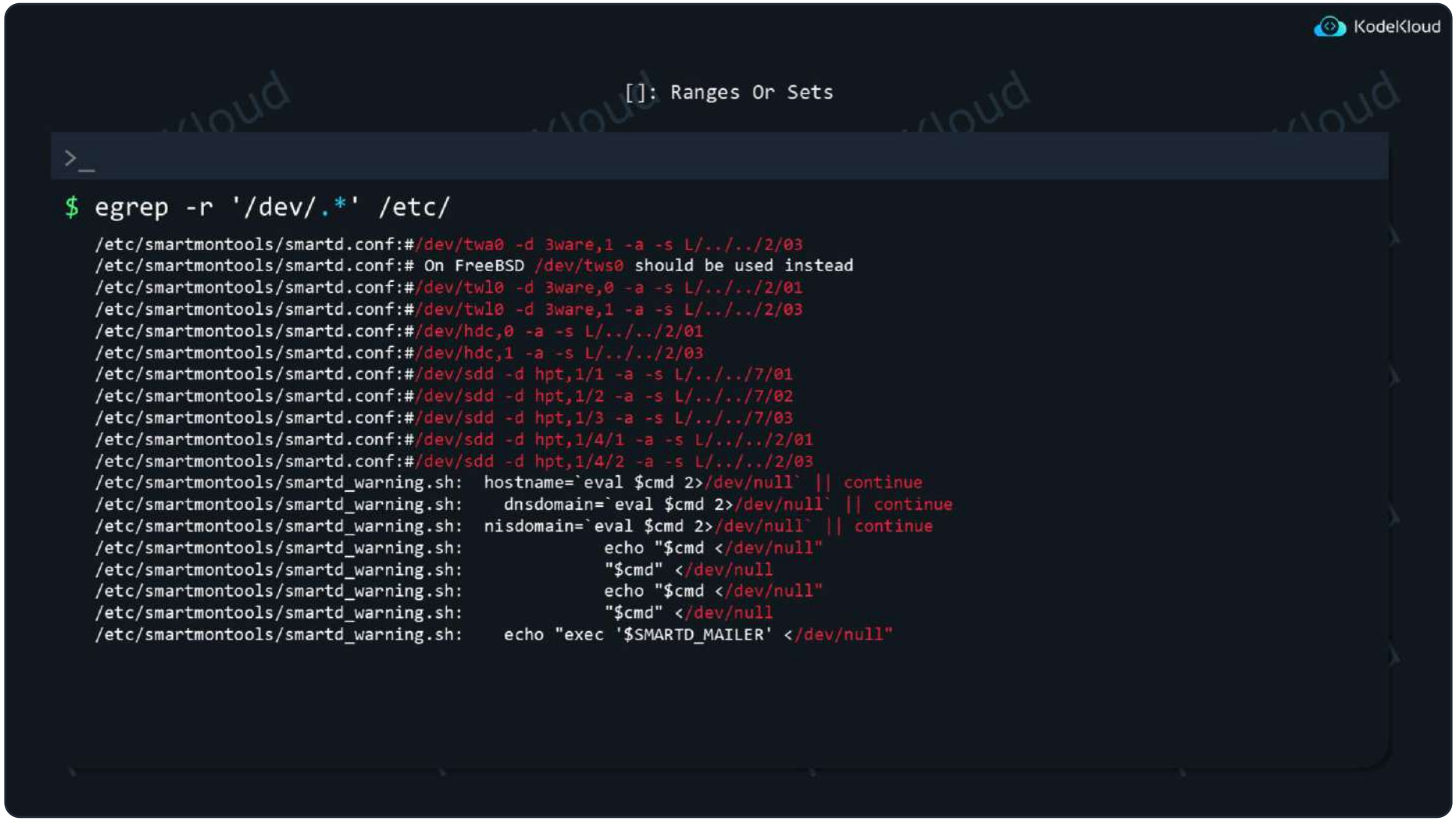
A set is specified in this form:

[abz954] will match any one character specified within, a, b, z, 9, 5 or 4

So,to find all strings that contain the text cat or cut, we'd use:

c[au]t

egrep -r 'c[au]t' /etc/'



With ranges and sets we can make our searches both wide, and specific, even at the same time. For example, let's ask ourselves: how would we find all config files which mention special device files that have names like /dev/sda1, or similar? We could think like this: find all strings that contain "/dev/" followed by any random characters:

/dev/.*

egrep -r '/dev/.*' /etc

But this matches weird stuff. .* is "greedy" matching way too many things after it captures the /dev/ part we're looking for. So we need to make our search wide enough to catch all /dev device names, but specific enough to only capture the parts we need.

We can do this with ranges.

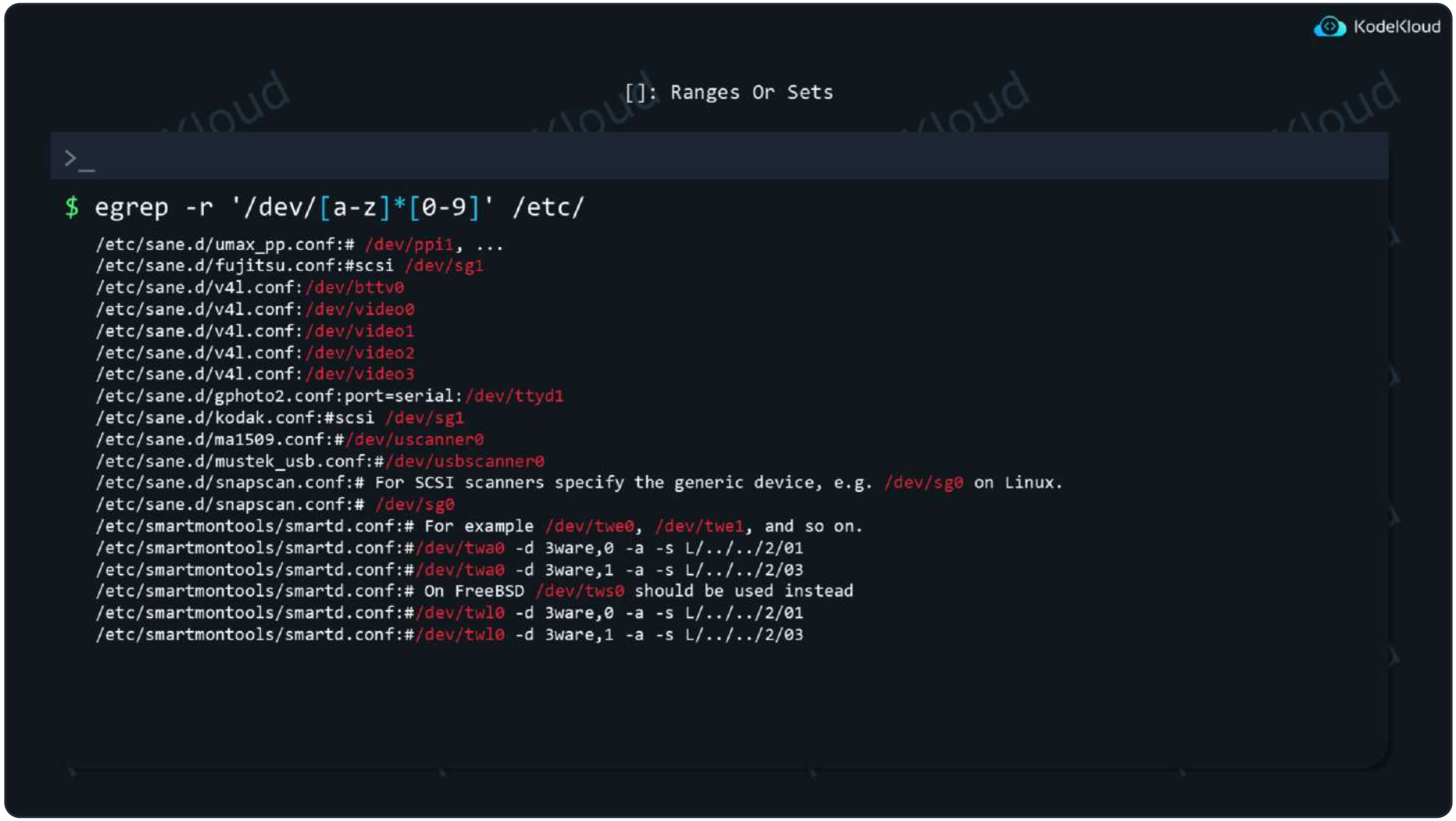
We can say: "after /dev/ match any number of lowercase letters, from a to z". And as we learned, we can do that with the asterisk operator:

/dev/[a-z]*

```
egrep -r '/dev/[a-z]*' /etc/
```

In this case, the * operator will apply to the set specified before it, [a-z].

Now the output looks a little bit better, but we see some things are still missed. /dev/twa is matched instead of the entire /dev/twa0. How can we catch the digits at the end too?



Easy, we specify that a digit from 0 to 9 should exist there

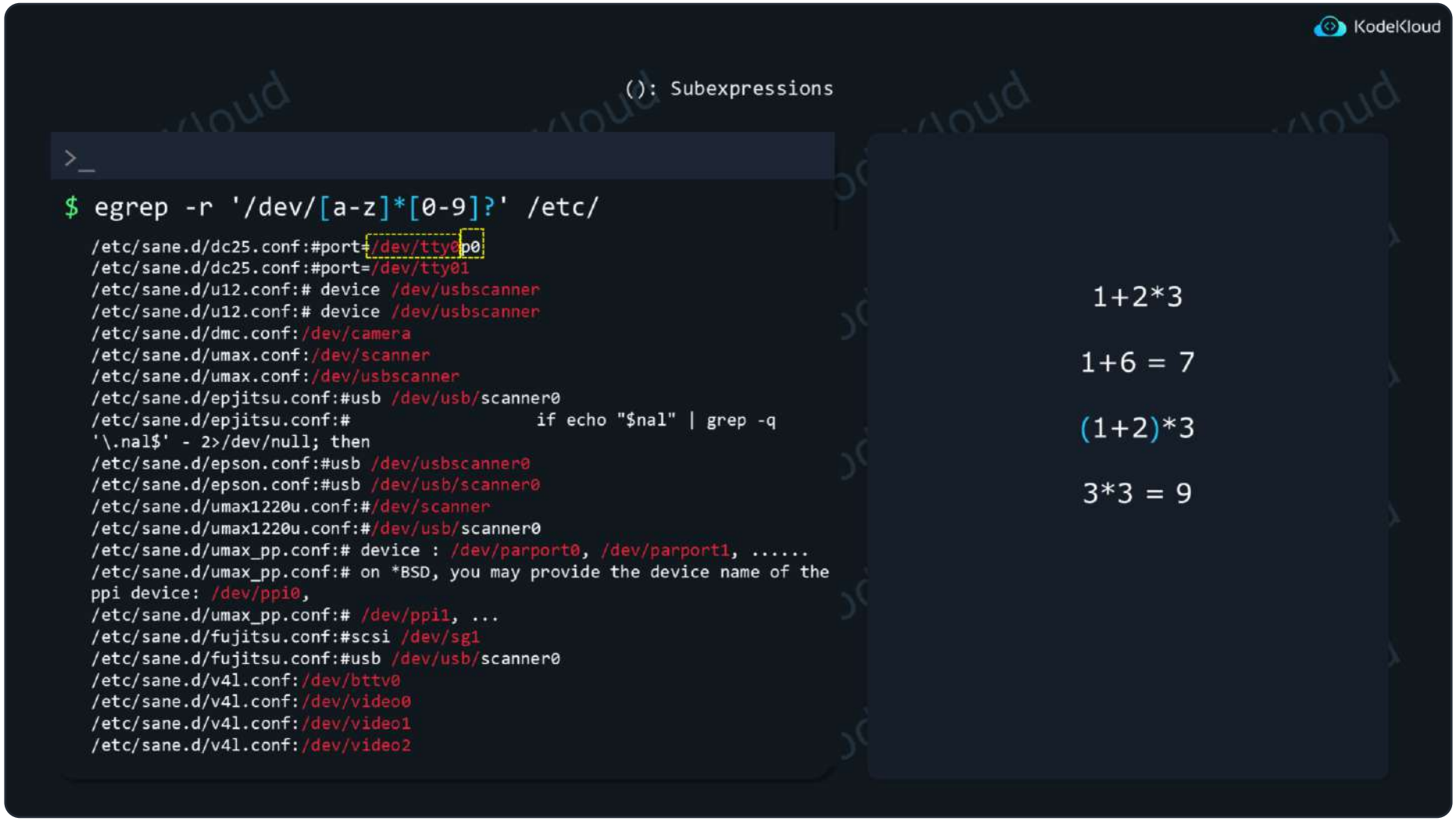
egrep -r '/dev/[a-z]*[0-9]' /etc/

But now we run into another problem. Only things that have a digit at the end are matched with this new regex. We'll only find /dev/sda1 but not /dev/sda.

This is an easy fix once again. We just make the digit at the end optional with the ? operator.

egrep -r '/dev/[a-z]*[0-9]?' /etc/

Looks much better now.



Now let's talk about subexpressions.

In math we can see this:

1+2*3

This is 1+6=7. That's because, first, multiplication will be done between 2 and 3, and only after that, the addition. But what if we first want to add 1+2 and then multiply by 3? We write:

(1+2)*3

Now, first the addition between 1 and 2 will be done, and only afterwards, the multiplication. This will be 3*3=9.

In regex we can do a very similar thing.

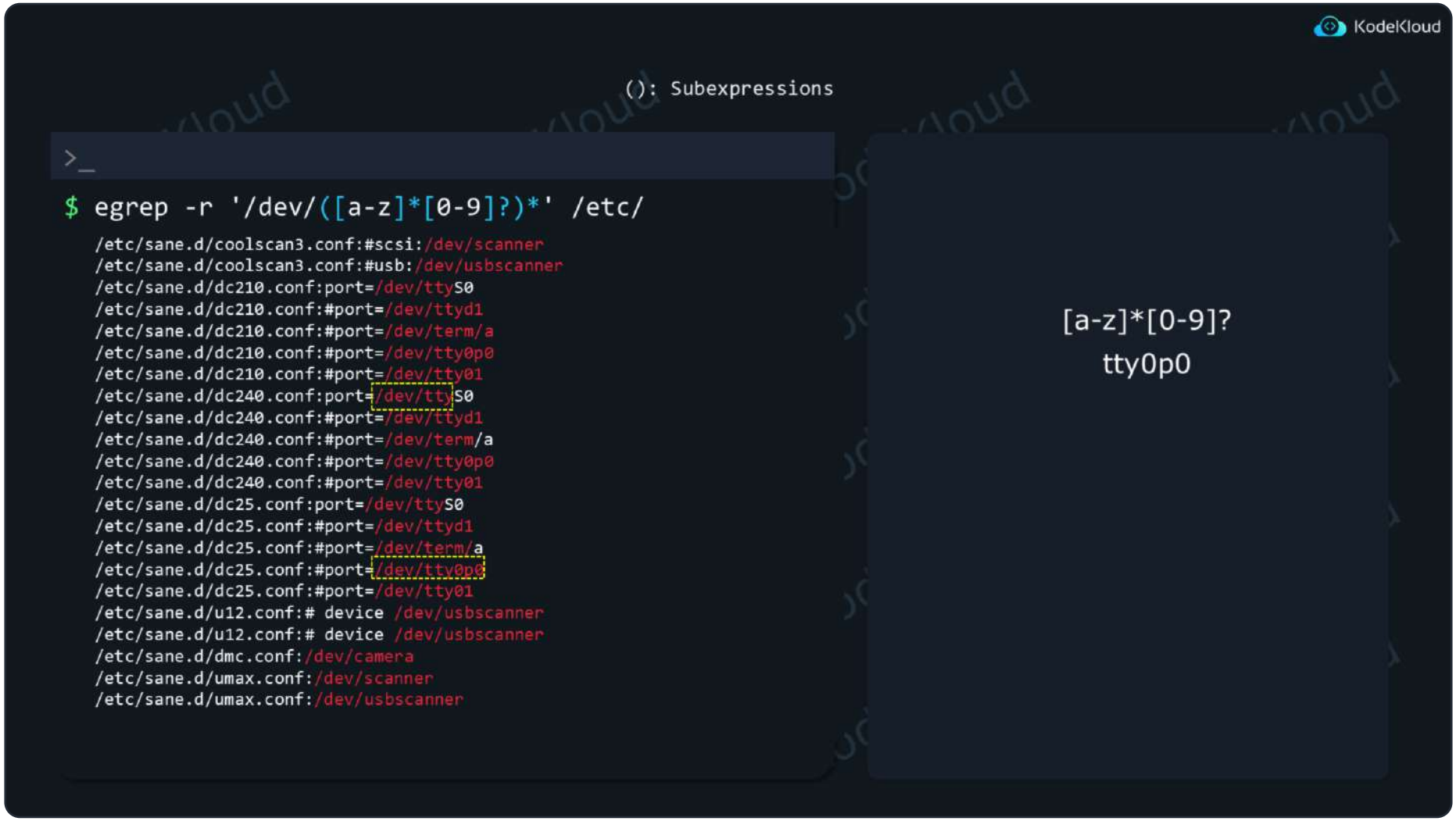
Let's take a look at our last expression:

egrep -r '/dev/[a-z]*[0-9]?' /etc/

If we scroll up in our output, we'll see we still don't match everything we need perfectly. In a line like:

/dev/tty0p0

p0 is left out. Why is that? Because our expression, after it finds /dev/ matches any number of a to z characters, then a digit at the end. And that's it, that's where the match ends. So, in /dev/tty0p0 after that first 0 is hit our regex is happy with the partial result. How could we correct this?



We could tell it that after /dev/ we have some letters, and a digit at the end, but after that, the same thing can repeat 0,1,2,3 or more times. There can be other sequences of letters followed by a digit. This way, /dev/tty0 would match first, then p0 will be added to this match by that repetition.

So we would basically want to say that this part of the regex:

[a-z]*[0-9]?

should look for this pattern existing 0, 1, 2, 3 or many more times, so it can match things like tty0p0. What makes regex look for something that exists 0 or more times? The * operator. But if we add it at the end, we get

```
[a-z]*[0-9]?*
```

This isn't good, as the * would apply to the previous element only. And our asterisk would be placed right after the question mark. However, we want to apply our asterisk operator to our whole construct here. Again, easy solution. We just wrap our construct in ( ) and this way, * will apply to our entire subexpression wrapped in parentheses, instead of the last element only.

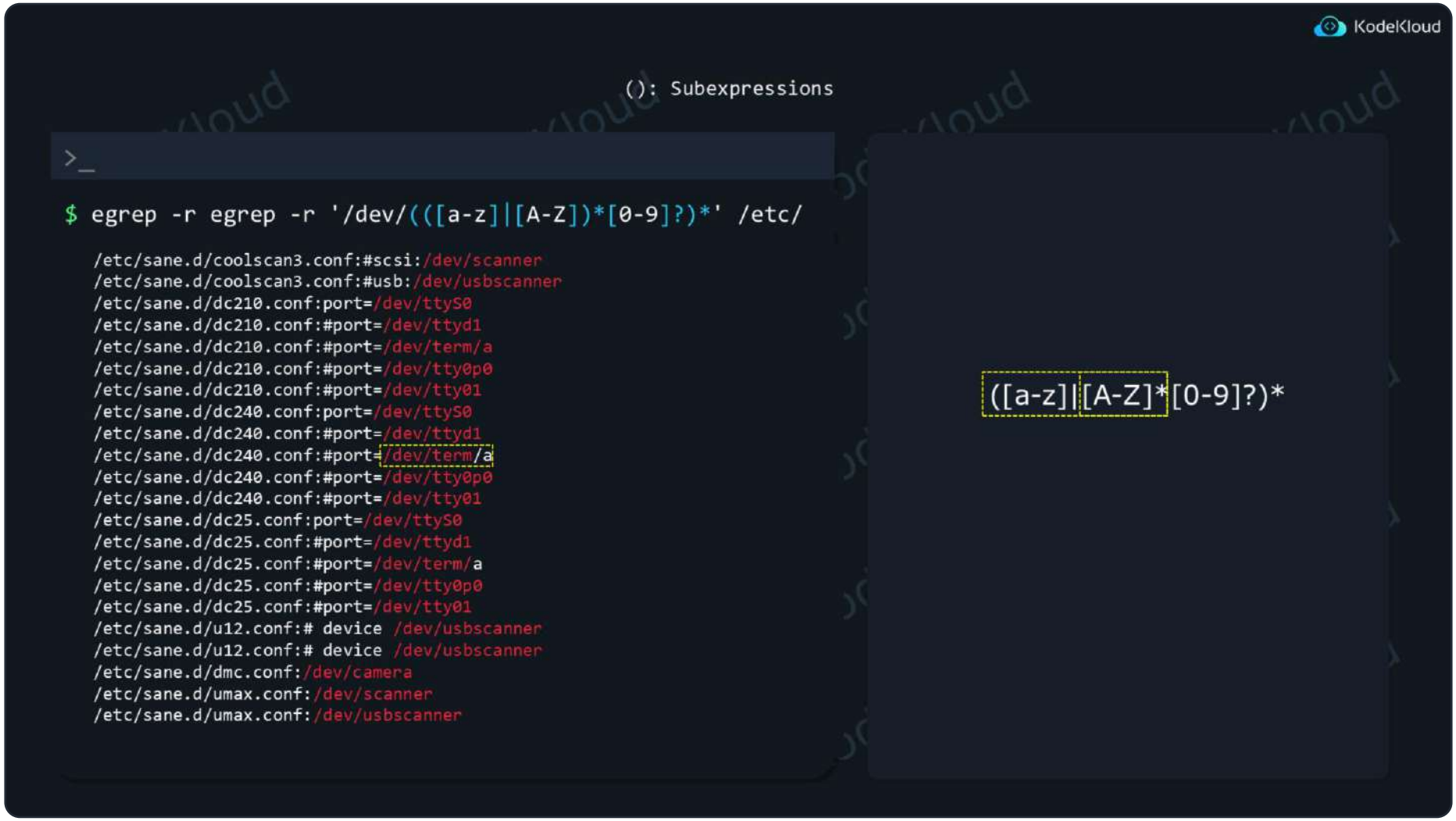
```
([a-z]*[0-9]?)*
```

So we'd end up using this command:

```
egrep -r '/dev/([a-z]*[0-9]?)*' /etc/
```

And now we get a full match for strings like /dev/tty0p0.

But if we scroll up in our result list, we'll still find some things that don't quite work. Like /dev/ttyS0 with the S0 not matching because we didn't include uppercase letters in our regex.



We could tell our expression to look for "lowercase letters OR uppercase" with the | operator.

But writing it like this would be a mistake:

([a-z]|[A-Z]*[0-9]?)*

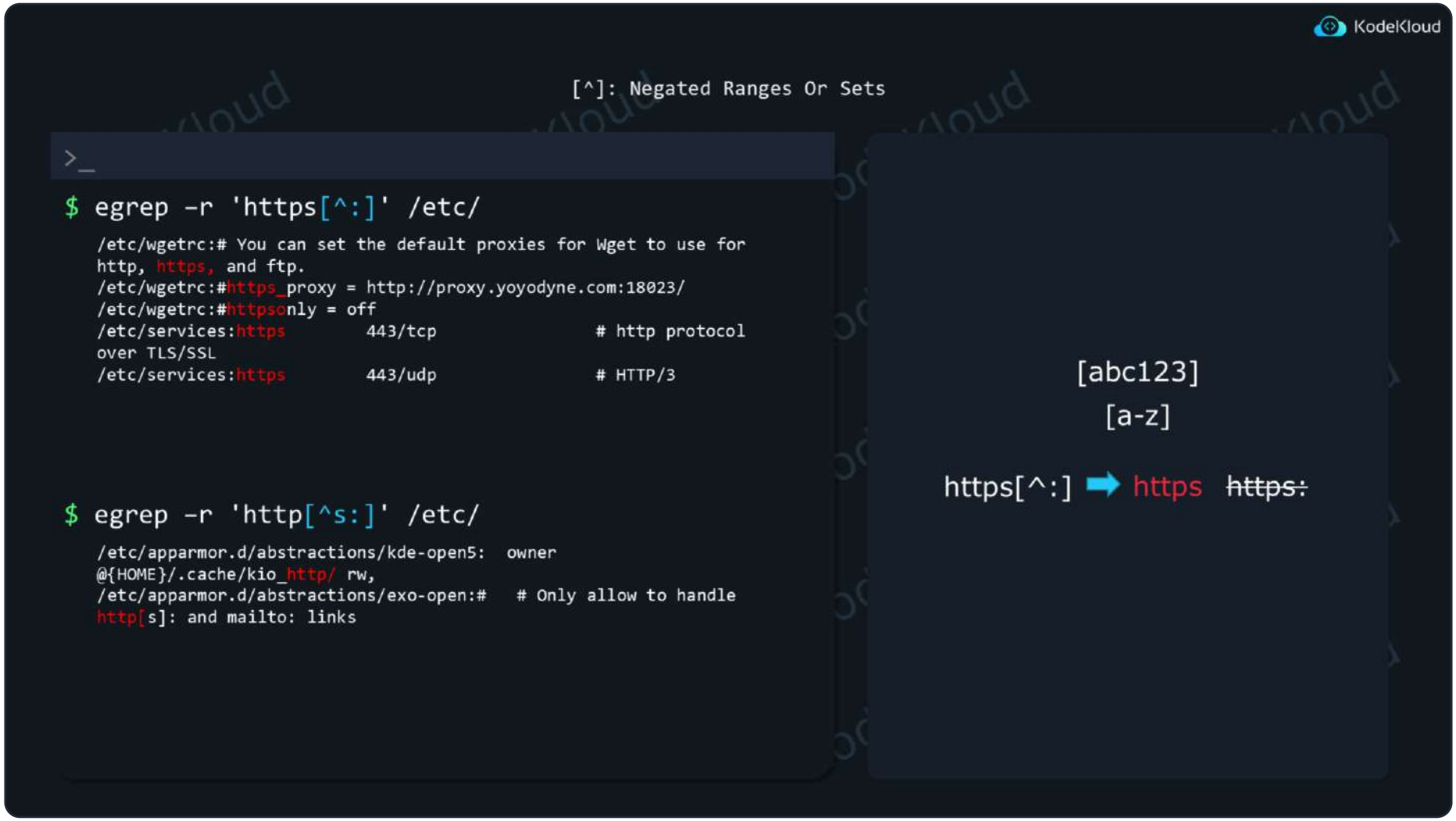
Because now the * asterisk operator would only apply to [A-Z] and we need to apply it to our entire [a-z]|[A-Z] construct. Once again, we can wrap in parentheses to make it a subexpression and fix this.

```
(([a-z]|[A-Z])*[0-9]?)*
```

We end up with this command:

```
egrep -r '/dev/(([a-z]|[A-Z])*[0-9]?)*' /etc/
```

Now ttyS0 matches. And if we would go on, we could fix things like /dev/term/a not matching, because our regex stops when it encounters the next /, and so on. This is the kind of logic and fine-tuning we would go through when fixing our regular expressions or making them laser-focused on what we need to find.



We saw sets are in the form of [abc123] and ranges [a-z]. If we add a ^ operator in there, we can negate them. That tells regex that "the elements in this set or range should not exist at this position"

Now imagine we want to search for the text "https". But we don't want to find links to websites. Since a link is usually in the form of "https" followed by a : sign, we can build a RegEx like this:

https[^:]

Basically saying: "Look for the https string, but make sure it's not followed by a : sign". And we'd end up with this grep command:

```
egrep –r 'https[^:]' /etc/
```

To find the string "http" not followed by the "s" letter, or the ":" character, we just negate the set. Otherwise said, we simply enumerate what we want to negate.

```
egrep -r 'http[^s:]' /etc
```

Up to this point, we used a set with only one or two characters. But as mentioned, we can negate ranges as well.

For example, we could tell our pattern: "After a /, there should not be any lowercase letter":

egrep -r '/[^a-z]' /etc

Keep in mind that for any pattern you're trying to match, there are multiple regex solutions you may find. To get this right, you should practice until you feel comfortable with regular expressions.

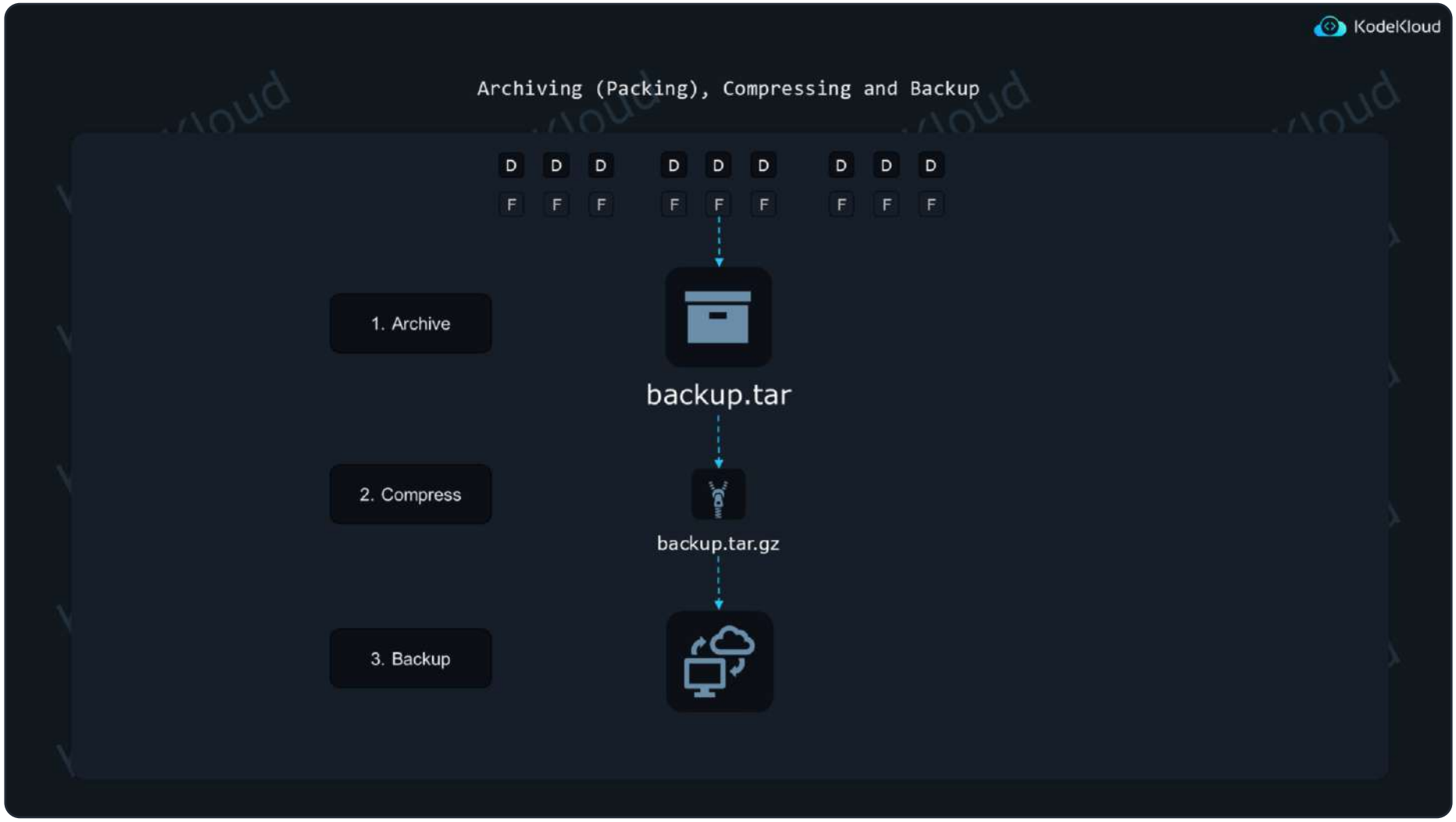
It's also worth noting that regex is not limited to grep. You can use regular expressions in a lot of programs that deal with search patterns. For example, the sed utility also supports regular expressions.

If you want to try out different regular expressions and see their effects, websites like regexr.com can be helpful.

KodeKloud

Visit www.kodekloud.com to discover more.

We'll now look at how to archive files in Linux.



Imagine you have a website, with thousands of files and directories. You don't want to lose anything, so you decide to back up all this data. If you do that manually, you'll usually go through steps like this:

You pack all those files and directories in a single file like backup.tar. This action is called "archiving".

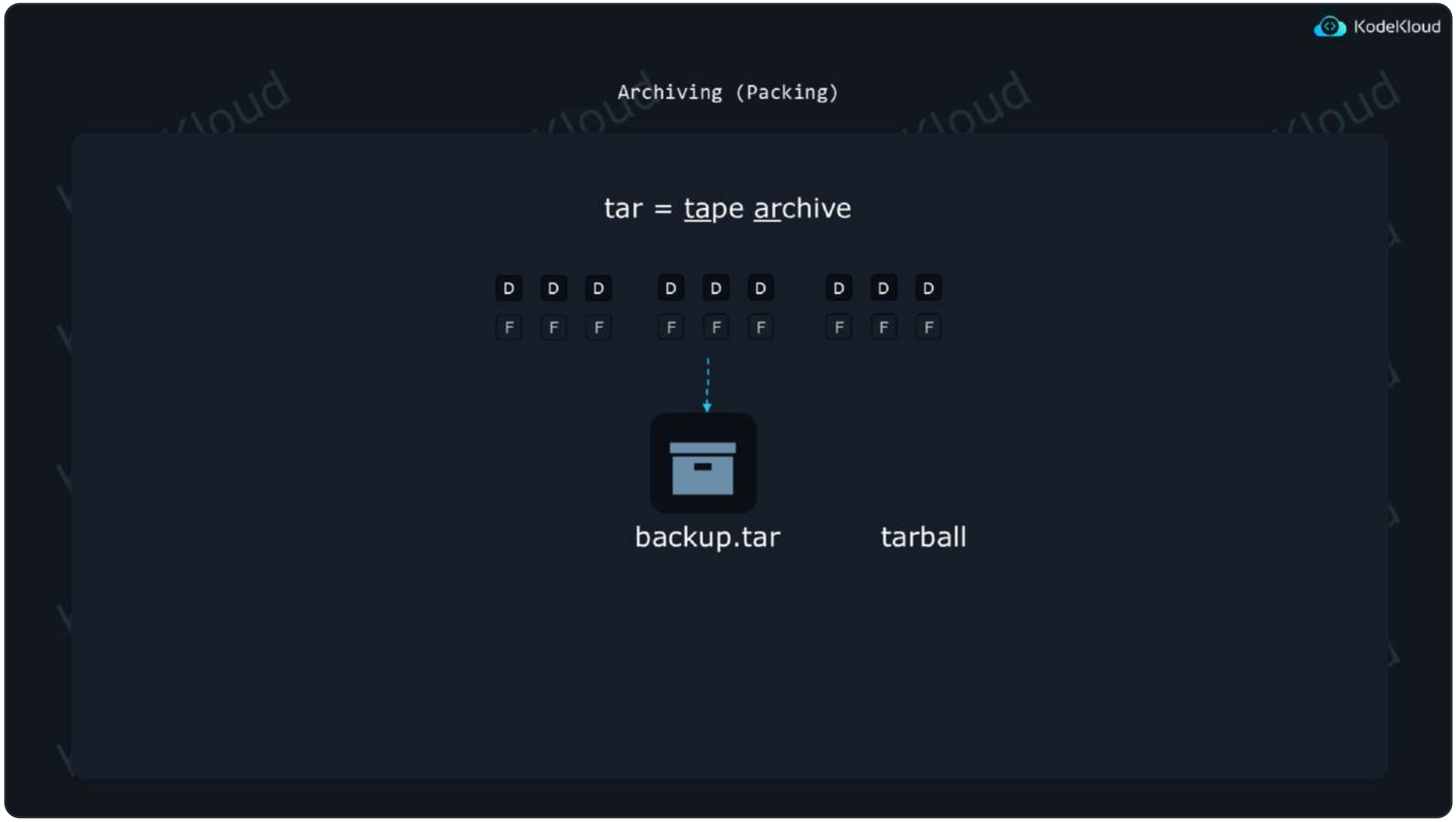
You compress backup.tar so you can store 10GB of data in a smaller, compressed file which might take up only 8 or 7GB of storage space; maybe even less. You'll end up with a file like

backup.tar.gz.

Finally, you copy the compressed file to a remote location. So you have a second copy of your data in another place. That's your backup.

Let's learn how to perform these actions.

In this lecture we will explore archiving and later we will look at compressing, and backing up data to a remote location.



The name tar comes from TApe aRchive. There was a time when keeping backups on magnetic tapes was very popular. tar was used to prepare data that would be sent out to these tapes. But even if this backup format on tapes is used less often these days, tar is still a popular tool because it does its job well. So, what does it do?

tar is basically a packer and unpacker. It can take any number of separate files and directories, and pack them all within a single .tar file (also called a tarball). Since we now have all our files in a single package, this makes it easier to move it around, upload it to a website, let people download our archive, and so on.

Imagine we have an archive.tar file on our system. tar lets us specify command line options in three different ways. For example, to display the contents of that archive, the correct command line option is either --list (long option), or -t (short option), or t (shortest option). So, we can use one of the following forms:

tar --list --file archive.tar

tar -tf archive.tar

tar tf archive.tar

After you memorize what each letter like t or f does, you'll prefer the last method as it's fastest to write. But it might be hard to remember in the beginning what letter is used for listing (t) whereas --list is much easier to memorize.

The most important thing though: when you point the tar utility to the .tar archive, it's a good habit to add the --file, or -f, or f, option at the very end of your list of options. Because tar expects the path to a tar file immediately after the --file option. If you put that option in another spot, for example at the beginning, you might add a second option after it. And tar would get confused thinking your second option is actually a path to a file.

Let's look at some tar commands.

To take file1 and pack it into a file called archive.tar:

tar --create --file archive.tar file1

tar cf archive.tar file1

--create tells tar that we want to create a new tar archive with the name specified after --file.

To add another file to our archive:

tar --append --file archive.tar file2
tar rf archive.tar file2

--append is just a fancy word that means "add to" existing archive.

To add an entire directory (and its contents) to a .tar archive:

tar --create --file archive.tar Pictures/

If you use a relative path here, like Pictures/, the paths stored in the archive will look like this:

But if you'd use an absolute path:

tar --create --file archive.tar /home/aaron/Pictures/

The content would look like this.

Before extracting from a .tar archive you should always use

tar --list --file archive.tar
tar tf archive.tar

and check out these paths. Why? To get an idea of where your extracted files will end up. For example, we could extract with this command:

tar --extract --file archive.tar
tar xf archive.tar

And our files will end up in our current directory + the paths we saw stored in the archive. If we were in /home/aaron/work/,
these files would be extracted at /home/aaron/work/Pictures/.

By default, tar extracts these in our current directory. But what if we want to extract this to another directory? Let's say we're in /home/aaron/. We have archive.tar here, but we want to extract its contents to /tmp/. We can tell tar "Hey, don't extract to my current directory, extract to this other directory here:"

tar --extract --file archive.tar --directory /tmp
tar xf archive.tar -C /tmp/

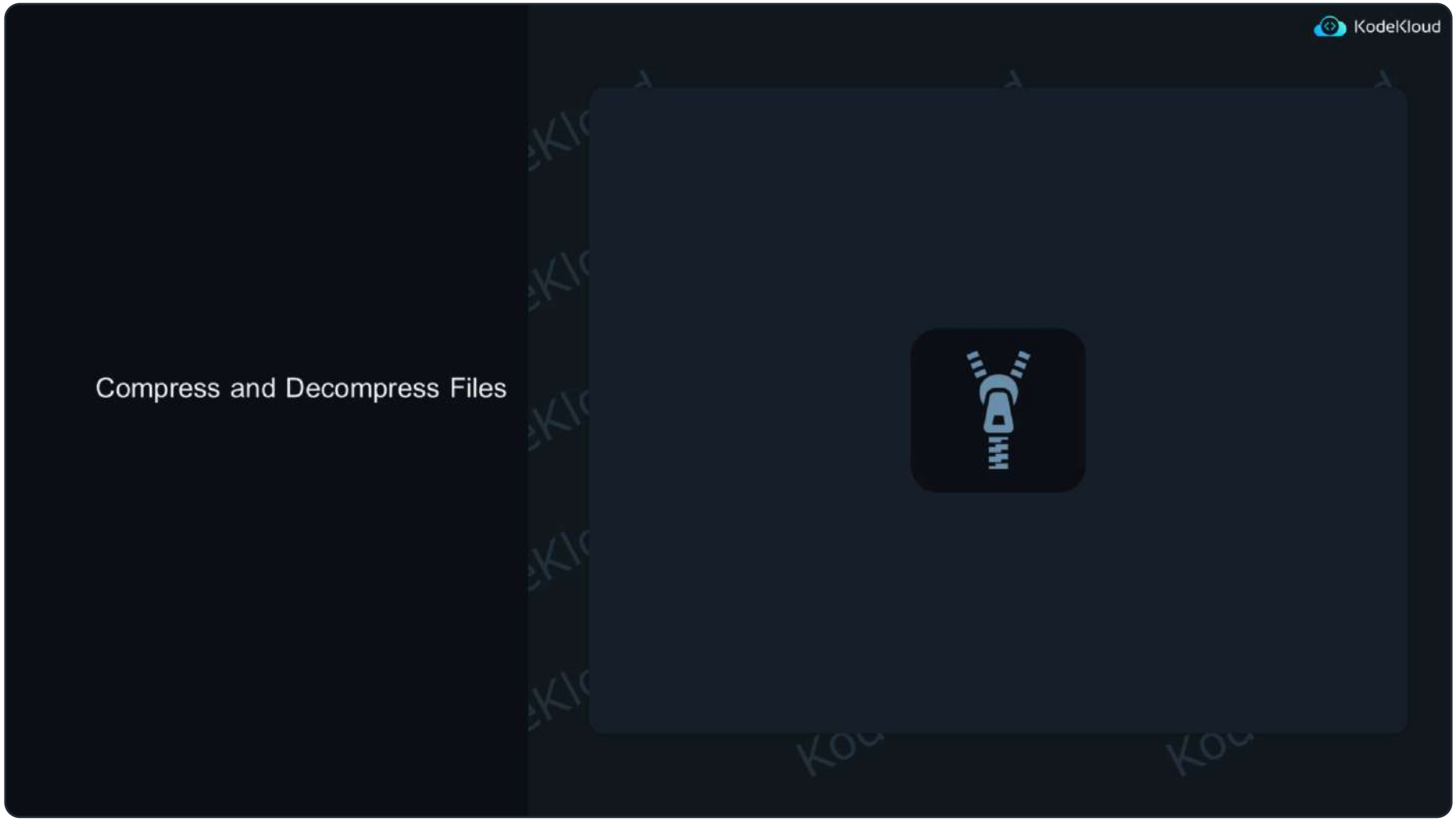
A tar archive also stores permission and ownership information of all files and directories. But if we're logged in as "aaron" and some file in the archive is owned by "jane", aaron does not have necessary permissions to create a file owned by "jane". However, the root user has that privilege. So, if we want to make sure that all ownership and permission information gets restored exactly as it was captured in the archive, we can use:

sudo tar --extract --file archive.tar --directory /tmp

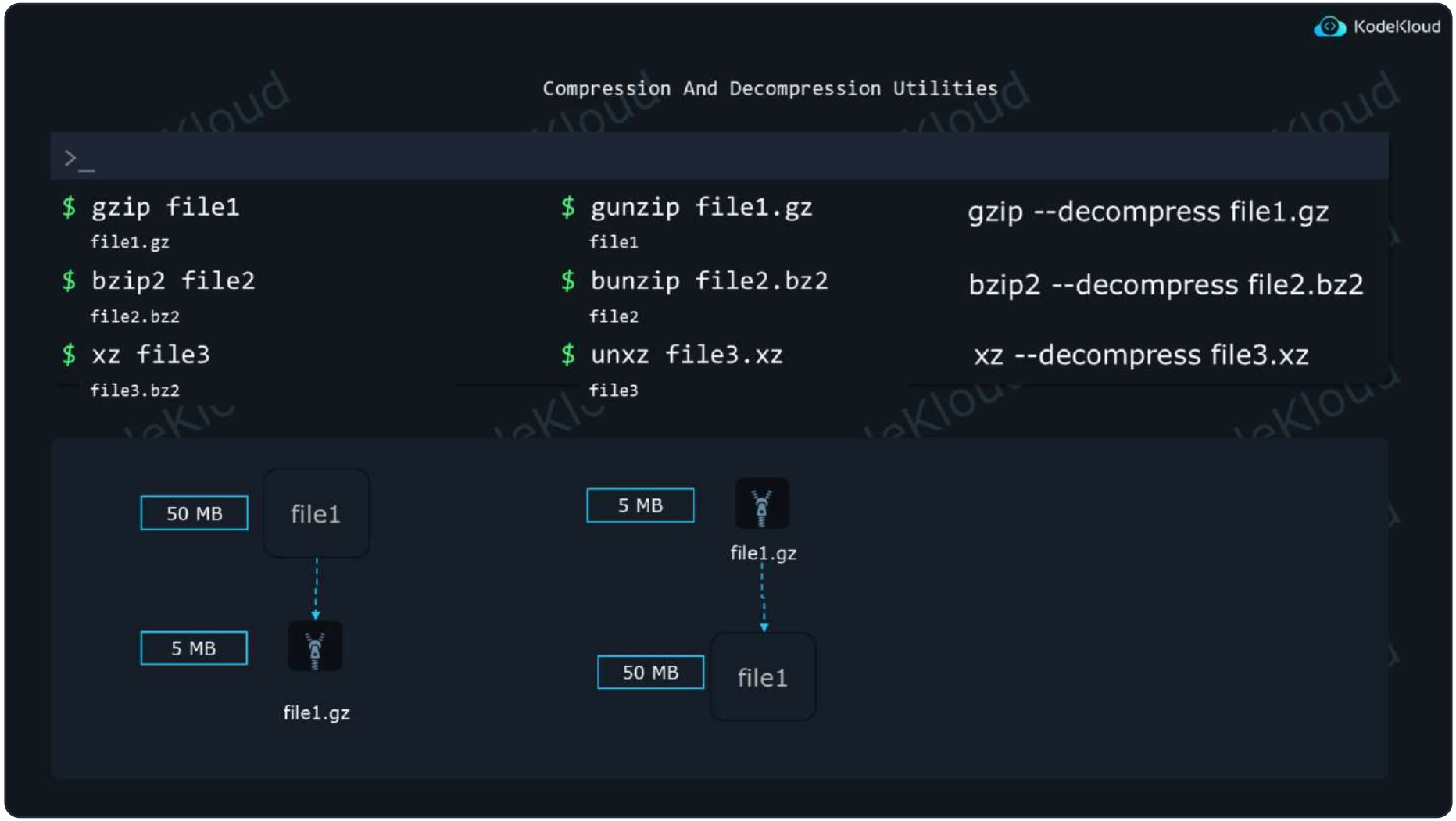
"sudo" is a command that temporarily gives us root privileges. We will learn more about it in future lessons.

KodeKloud

Visit www.kodekloud.com to discover more.



We'll now look at how to compress and uncompress files in Linux.



Reducing file size with compression is not only useful because it requires less storage space. It's also useful because it helps us transfer files, from system to system, much faster.

Most Linux systems will have at least three compression utilities preinstalled:

gzip

bzip2

xz

Compressing a file with these utilities is super easy:

```
gzip file1
bzip2 file2
xz file3
```

Such commands would compress file1, file2, and file3, create compressed versions of these files, and automatically delete the original files after. So, you end up with file1.gz, file2.bz2, or file3.xz, the compressed versions of your original files.

To decompress these:

```
gunzip file1.gz
bunzip2 file2.bz2
unxz file3.xz
```

Equivalent commands:

```
gzip --decompress file1.gz
bzip2 --decompress file2.bz2
xz --decompress file3.xz
```

When file1.gz is decompressed, the opposite thing happens. The original, uncompressed, file1 is recreated and the compressed file1.gz gets deleted.

Sometimes this automatic deletion of the compressed or uncompressed file is undesired. This means that after we gzip file1, maybe we still want file1 to stick around. How could we do that?

Remember, we have an easy way to see extra command line options supported by any program:

gzip --help

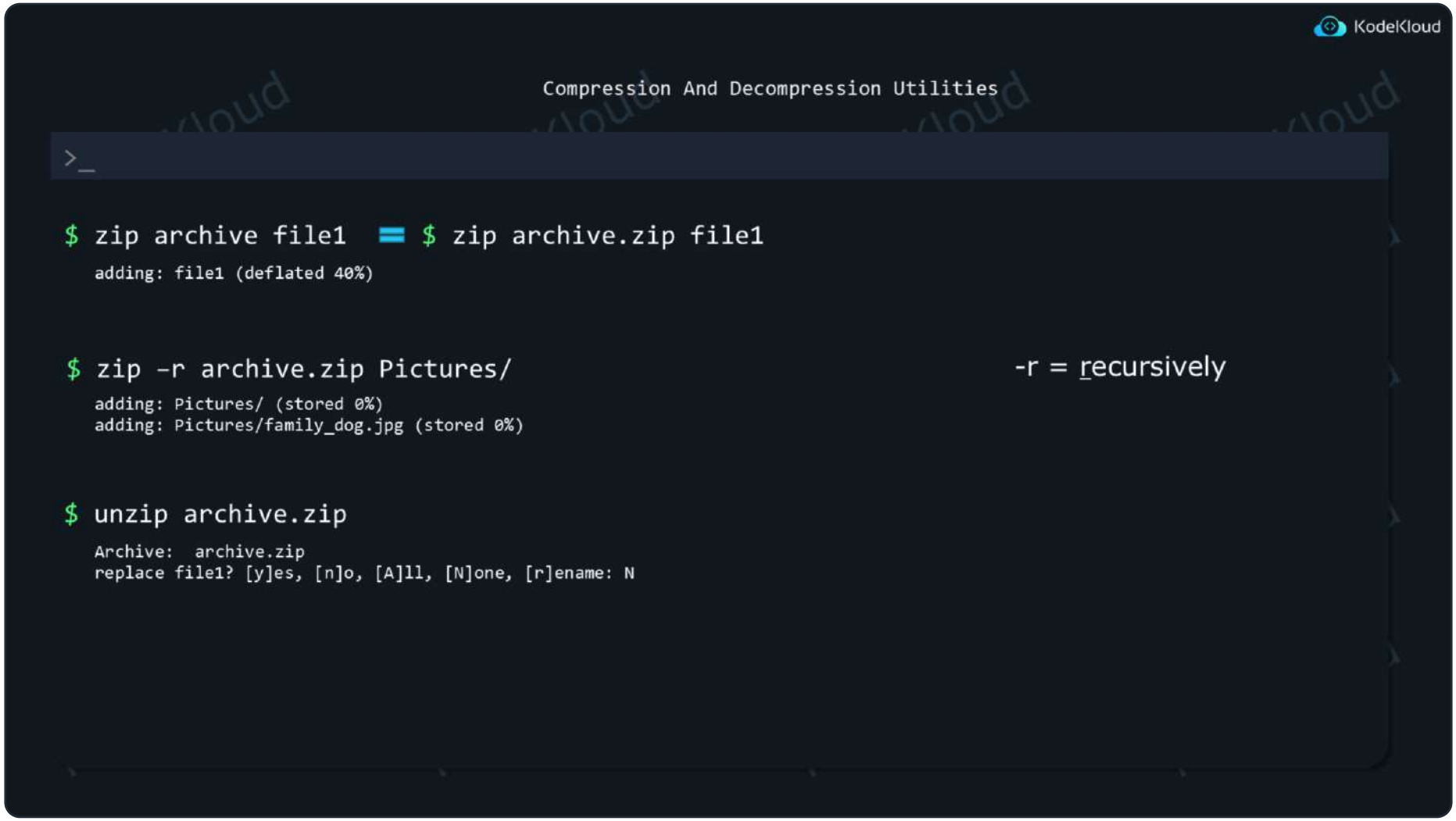
We notice here that the --keep option (or -k for short) is what we're looking for. So to keep our original files around, we can use commands like:

```
gzip --keep file1
bzip2 --keep file2
xz --keep file3
```

And if we ever need to take a look at the contents of a compressed file, some commands support a --list option:

```
gzip --list file1.gz
```

Normally, these three utilities are most often used for compression/decompression. But you might occasionally encounter additional programs like zip. One advantage of the "zip" utility is that it can also pack and compress entire directories, or multiple files, in the same archive. While utilities like gzip, and the others can only create compressed data with a single file inside.



To create an archive called archive.zip and compress file1 within it, we can use a command like:

zip archive file1

or

zip archive.zip file1

To pack and compress the entire Pictures/ directory into archive.zip:

zip -r archive Pictures/

-r = recursively compress every file and subdirectory in the Pictures/ directory.

To unpack and decompress a .zip file:

unzip archive.zip

As we noted previously, gzip has no option to take a directory and jam it in a single archive. It cannot pack multiple files in a single file, it can only compress a single file. zip supports both packing and compression. Since utilities like gzip, bzip2, xz don't do that, they're often used in conjunction with the other utility called tar.

In the previous lecture we talked about packing a file into a tar archive using the tar command like this:

$ tar --create --file archive.tar file1

After we generate our .tar file we can also compress it with any utility we want, with commands like:

gzip archive.tar
gzip --keep archive.tar

But instead of going through two steps: Make the tar archive and then compress it (pack and compress), we can tell tar itself to do both things for us in one step:

tar --create --gzip --file archive.tar.gz file1
tar czf archive.tar.gz file1

tar --create --bzip2 --file archive.tar.bz2 file1
tar cjf archive.tar.bz2 file1

tar --create --xz --file archive.tar.xz file1
tar cJf archive.tar.xz file1

Or, even better, tar has an option (--auto-compress) to automatically figure out what compression utility to use, based on the filename extension we choose for our archive.
.gz will make it compress with gzip
.bz2 will make it compress with bzip2
.xz will make it compress with xz

tar --create --auto-compress --file archive.tar.gz file1
tar caf archive.tar.xz file1

When unpacking and decompressing, though, we don't have to tell tar what decompression utility to use. It can figure that out by itself. So, a command like this would suffice:

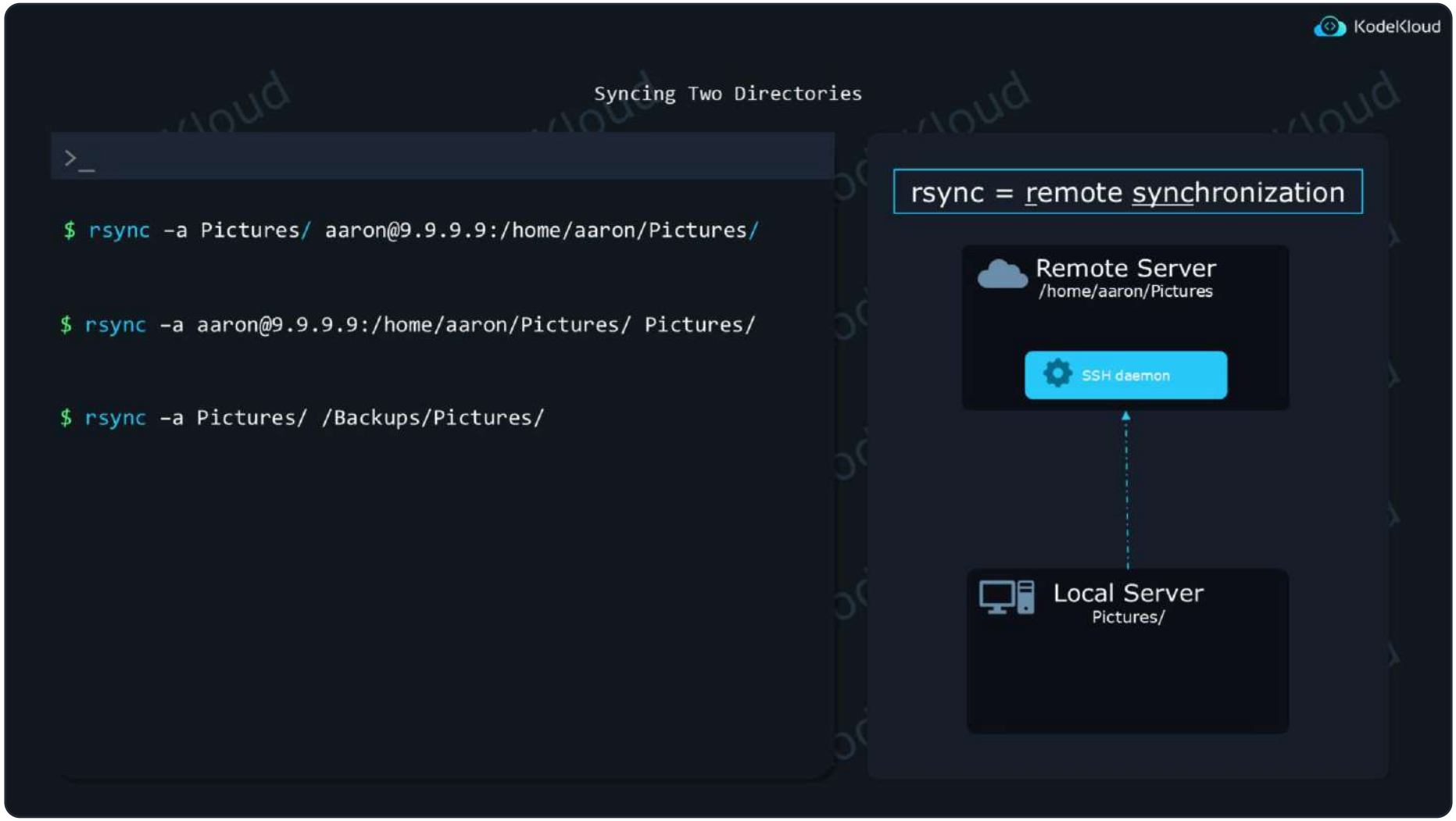
tar --extract --file archive.tar.gz
tar xf archive.tar.gz

KodeKloud

Visit www.kodekloud.com to discover more.

We'll now look at how to back up files in Linux. There are many utilities and sophisticated tools available in the market today to take a backup of a system. However, covering those is not the goal of this chapter.

Here, we will focus on taking a backup in the most basic way: Copying files from one system to another system that is the designated backup location. Our goal here is to understand the native Linux tools that can do that.



A popular tool to backup data is rsync. Its name originates from "remote synchronization". That's because it can keep /some/directory on server1 synchronized with /some/other/directory on server2, by copying data through a network connection. The remote server must have an SSH daemon running on it.

General syntax is:

rsync -a /path/to/local/directory/ username@IP_address:/path/to/remote/directory/

-a is the archive option to make sure rsync also synchronizes subdirectories, file permissions, modification times, and so on.

For example, to sync the local Pictures/ directory to the remote Pictures/ directory, we could use something like:

rsync -a Pictures/ aaron@9.9.9.9:/home/aaron/Pictures/

Here, aaron is the username, 9.9.9.9 is the IP address of the server, and the last part is the destination directory on that remote server.

Make sure to always have a / at the end of your directory names.

The cool thing about this tool: When you run this command the next time, rsync will only copy data that has changed, skipping old data that is still identical at both locations. Which means that future backups will be optimized and take less time to be completed.

In the previous command, your local directory is the source which will get recreated at the destination.

$ rsync –a Pictures/ aaron@9.9.9.9:/home/aaron/Pictures/

But if you reverse these, then the source will be the remote directory and it will get recreated in your local directory.

$ rsync –a aaron@9.9.9.9:/home/aaron/Pictures/ Pictures/

It's also worth noting that you can sync two local directories as well, instead of one local and one remote:

rsync -a Pictures/ /Backups/Pictures/

If instead of backing up files and directories you want to backup an entire disk or partition, you can use the "dd" utility. This takes a sort of "picture" of all data on that disk or partition. An exact bit-by-bit copy, which is why it's also called "imaging". Before saving a disk/partition image, you should unmount that disk/partition, to make sure no data is being changed while you back it up.

Here's an example of a dd command:

sudo dd if=/dev/vda of=diskimage.raw bs=1M status=progress

You need root permissions to be able to access all disk data, so sudo was used.
For if= (input file) you specify the path to your disk/partition device
For of= you specify the path to your output file, where you want to store the image
For bs= blocksize you specify 1M, 1 megabyte, or larger. This speeds up the process. The default blocksize is much smaller if we don't specify it here, leading to ineficient read/write operations.
status=progress tells dd to show us the progress it's making.

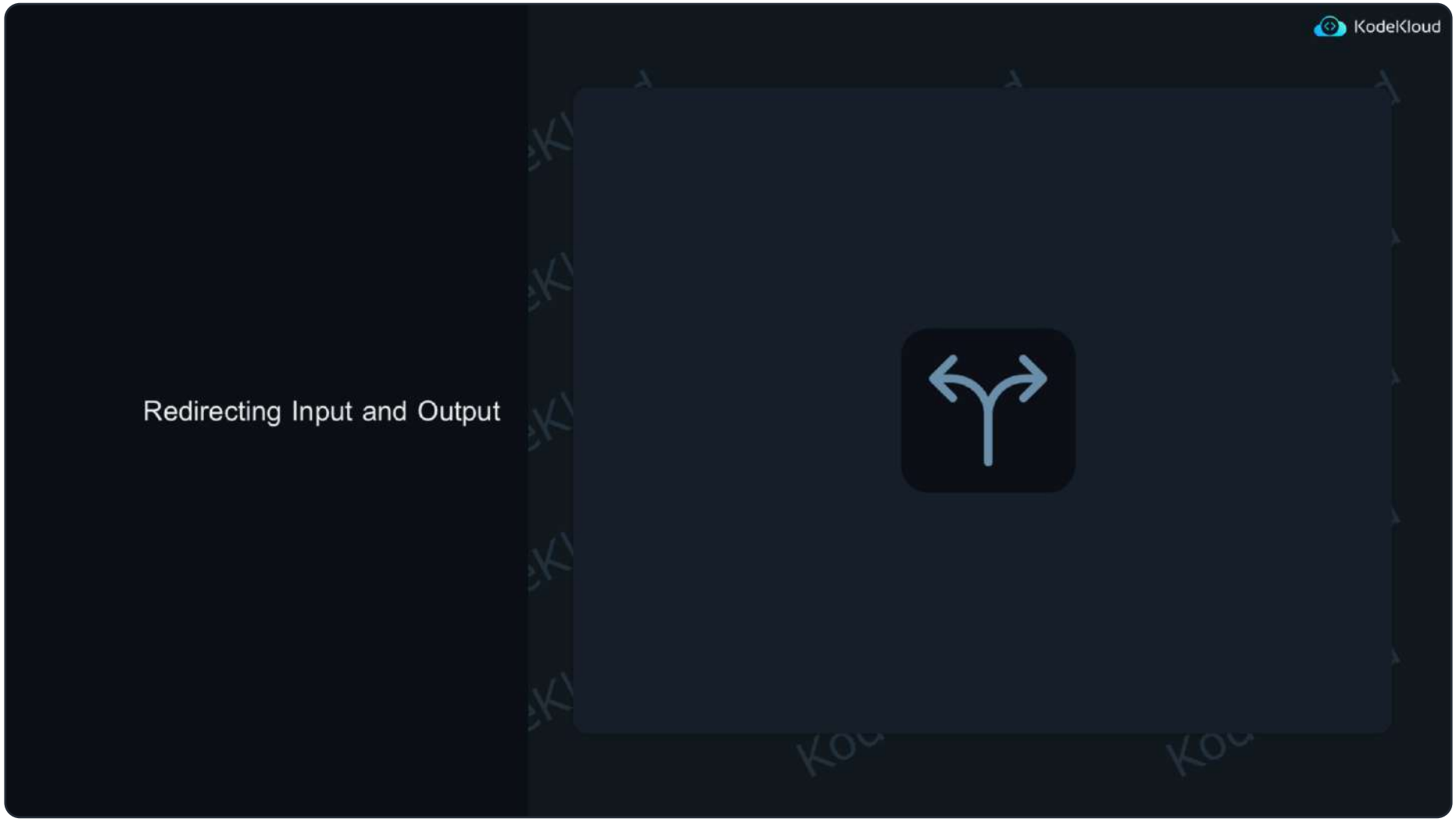
If later you want to restore a disk image from a file -> to disk, you just reverse the if and of options.

Note: don't run this command on your virtual machine, it will overwrite your virtual disk.

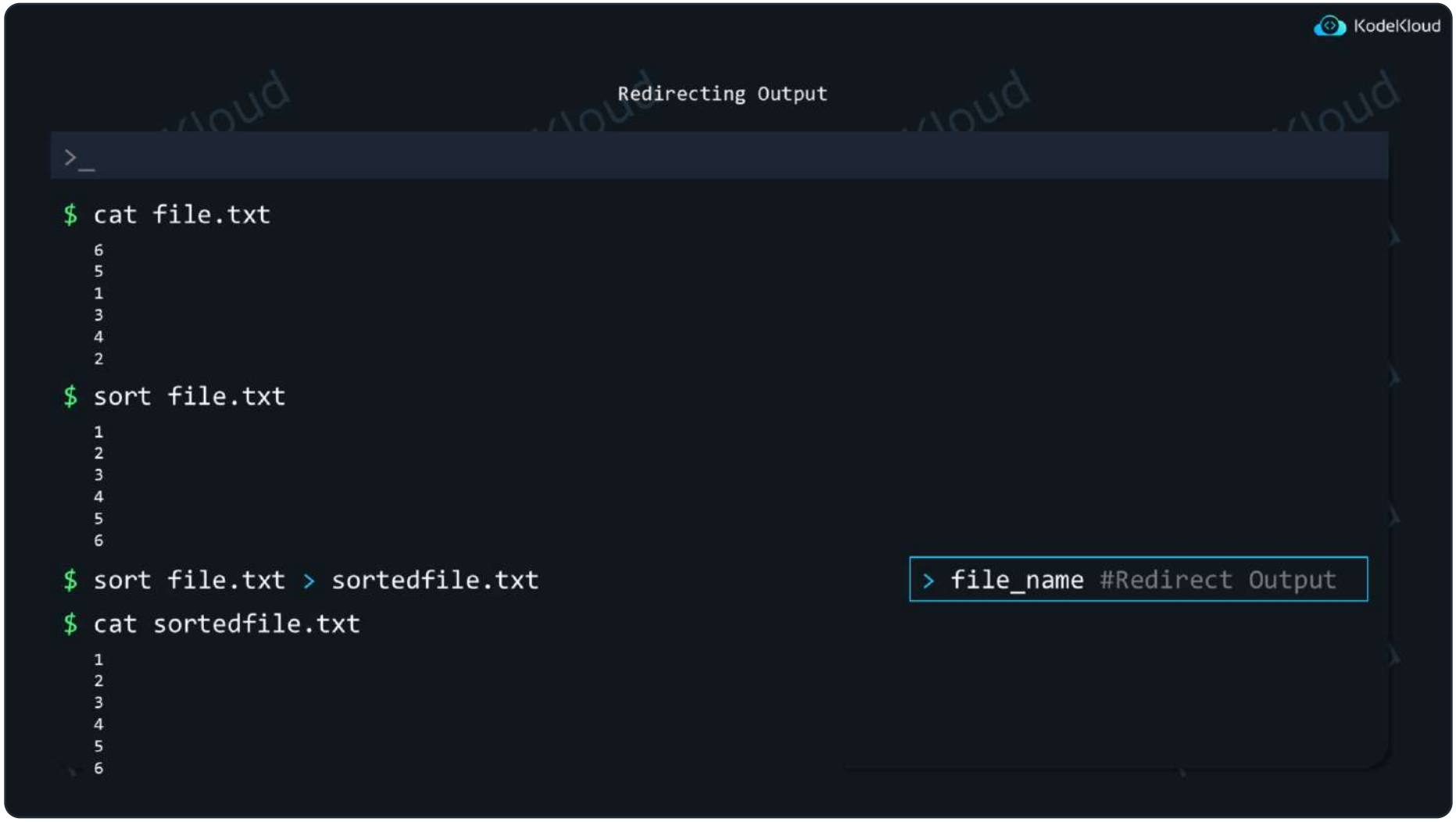
sudo dd if=diskimage.raw of=/dev/vda bs=1M status=progress

KodeKloud

Visit www.kodekloud.com to discover more.



We'll now look at how to redirect input and output in Linux.



To understand input/output redirection let's take a look at a utility like sort. First of all, sort expects to receive some text input. Usually, it will get this input from a file.

Let's assume we have file.txt with the following content:

cat file.txt

If we run a command like

sort file.txt

the following happens:

sort gets input data from file.txt. Then it orders it properly, and generates some text output with the sorted results:

Many Linux utilities (but not all) work this way. So what's with all this input/output redirection stuff? What use does it have?

Well, think about what sort did here. It ordered our numbers correctly, but then, all it did was to display this output on the screen. However, what if we want to save these ordered results? We can do that with output redirection, with a command like this:

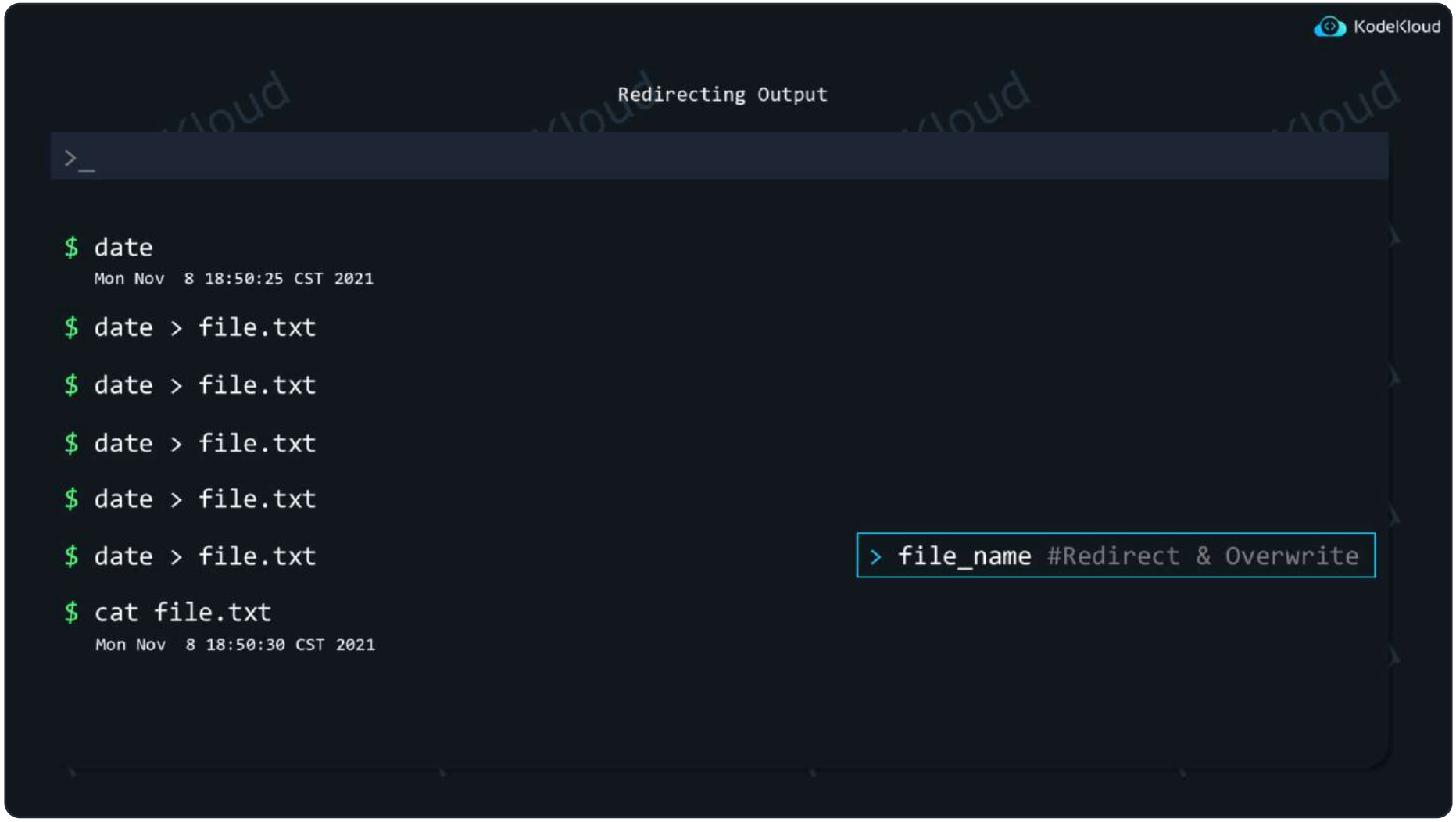
sort file.txt >sortedfile.txt

Now this output, instead of being displayed on screen and then lost, is saved to sortedfile.txt instead. Text output is redirected to that file.

It's not necessary that the file exists beforehand, the redirection can automatically create it.

Why is it called redirection? Because programs have a default location where they'll send output. They'll normally send it to our terminal window, to our screen. Since we change this default output location, we direct it somewhere else, so we call this output redirection.

To redirect output to a file, simply add > at the end of your command and specify the target file where you want to redirect, or save that content.

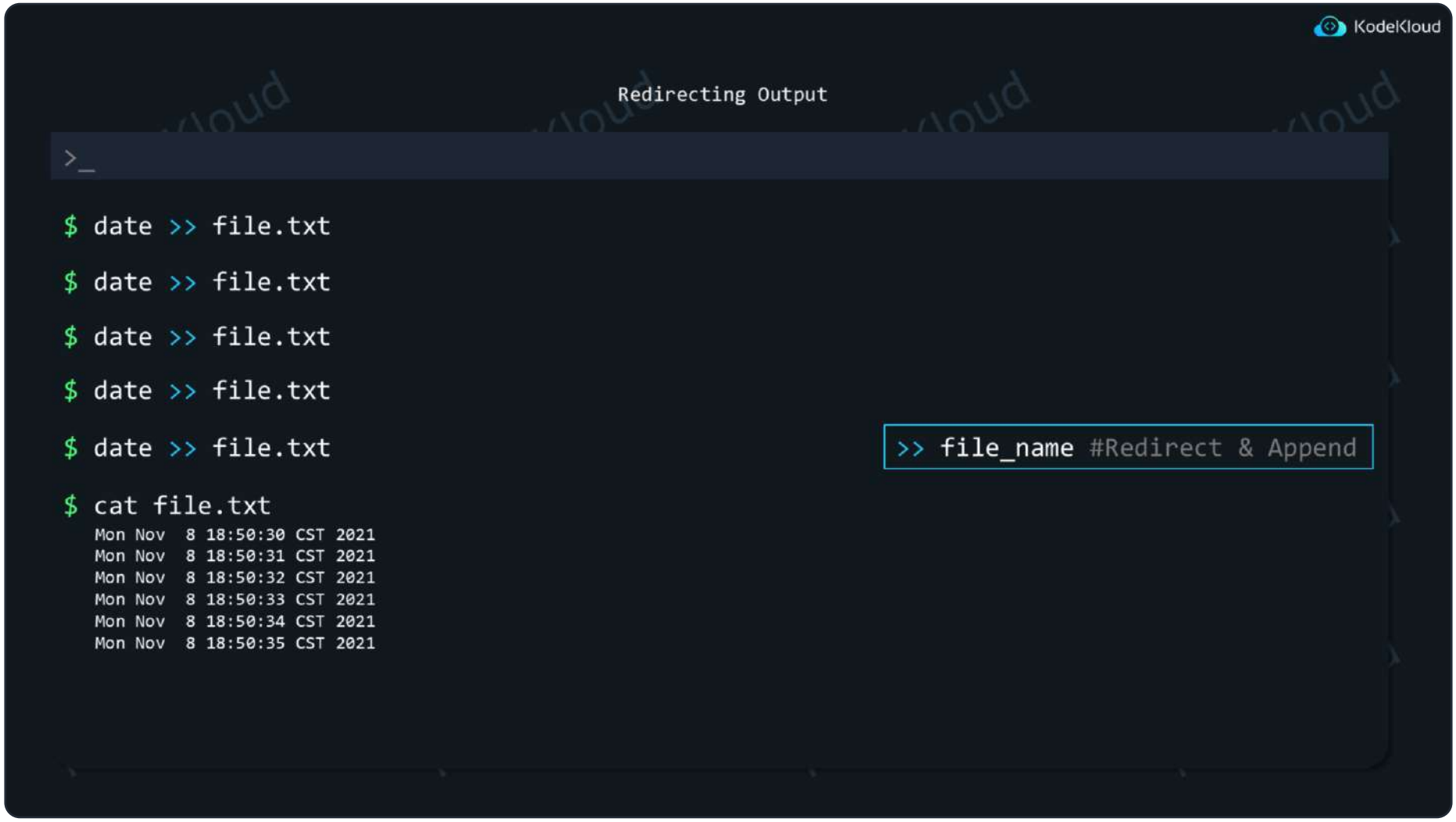


It's important to understand that when you redirect output this way, the file is overwritten. To test this out we can try a command like:

date > file.txt

and run it 5 times in a row. We'll see that only the last generated output remains in our file:

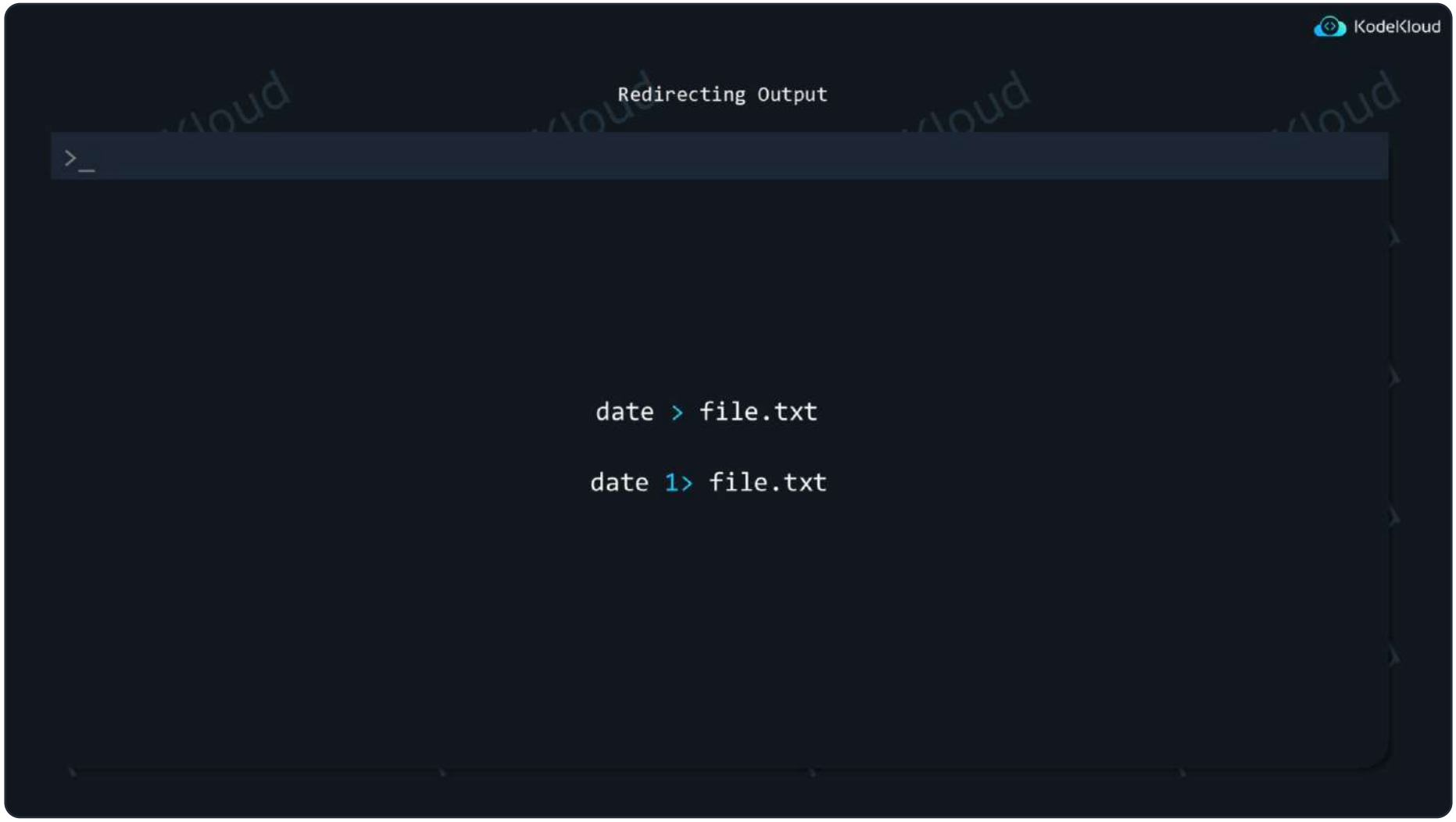
Every time output is redirected with > old content gets lost and new content gets written. Sometimes this is undesirable.



We can change this behavior, and instead of overwriting, add/append new output with >>

date >>file.txt

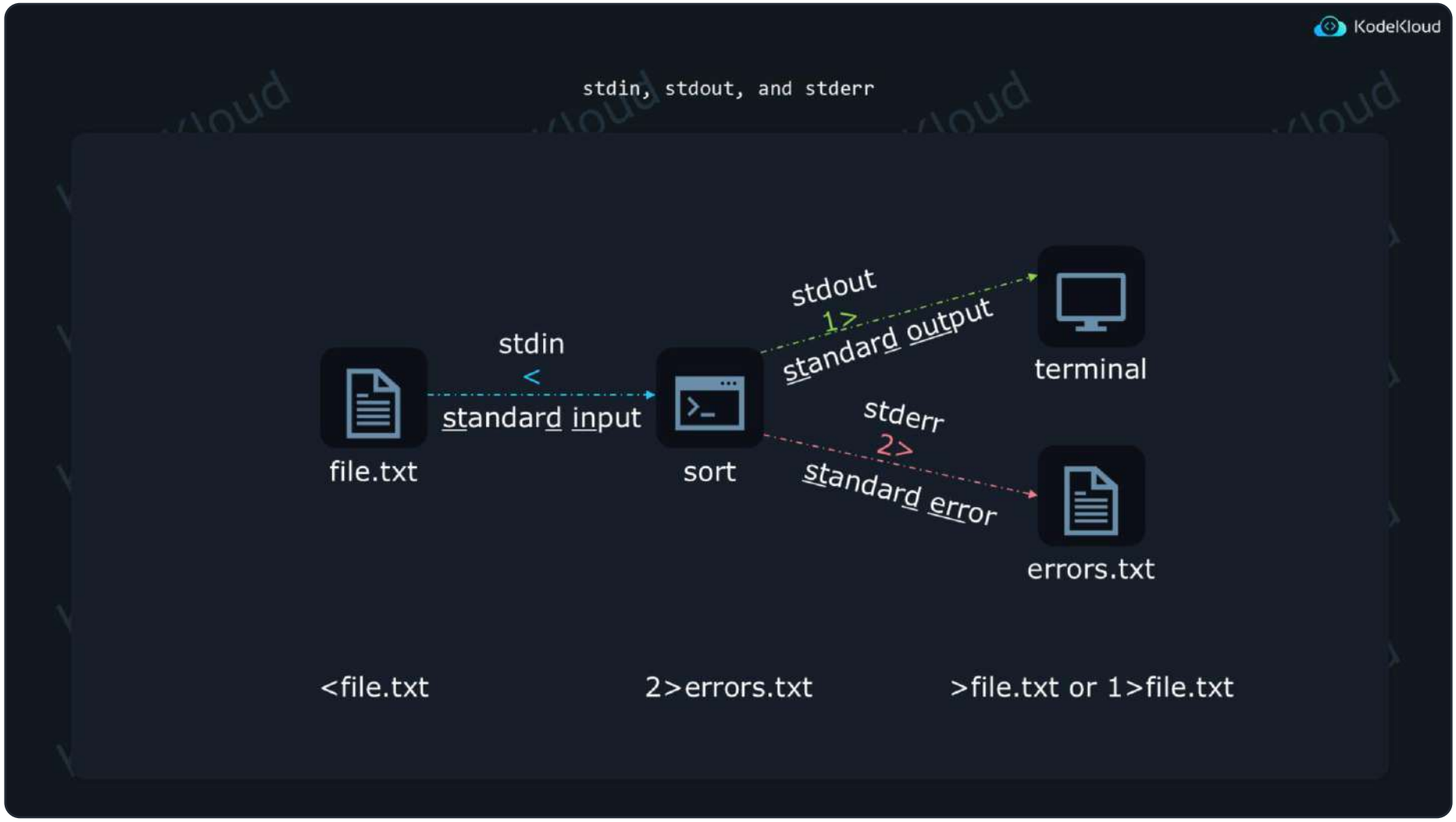
Let's also run this 5 times. Now we'll see each new line of output is added at the end of the file. The top line is the oldest content. The one at the bottom is the newest.



To summarize, the "greater than" sign is used to redirect output to a file. Alternatively this can also be written like this:

1>

with a 1 before the redirection character. This would generate the same result. Let's look at what that means.



A program like sort must know where its input comes from. It also must know where to send the output it generates. But output is of two types. One output is for normal text/data that was successfully processed. The other output is for error messages, warning messages, or anything that signals that something went wrong in some way. That's why we used 1> followed by a file name. To signal we want to redirect normal output in this case.

When a utility tries to figure out where its input comes from and where output should go, it looks at these things:

<c>
stdin - short for standard input
stdout - short for standard output
stderr - short for standard error

The "less than" sign is used to indicate standard input and the greater than sign is used to indicate standard output. Since there are 2 types of output the greater than sign can be prefixed with one of two digits. Prefix 1 for standard output and prefix 2 for standard error.

<c>
So, an overly simplified explanation is this. Program asks, "Where should I send these errors?". Linux says "Take a look at how stderr is set. Send it there." When we use something like 2>errors.txt we basically set stderr to point to this file and the application now knows where to direct its errors.

Now we can recap:

<c>
To redirect input <file.txt
To redirect normal output: >file.txt or 1>file.txt
<c>
To redirect errors/warnings: 2>errors.txt

But why would we want to redirect errors?

If you tested commands from our previous lessons you might have noticed that some find and grep commands displayed a lot of "Permission denied" errors.

grep -r '^The' /etc/

Error messages are normally useful, but in this case, they just get in the way. They make it harder to spot our results. Good news is, with input/output redirection tricks, we can get rid of this mess. For example, we can type a command like this:

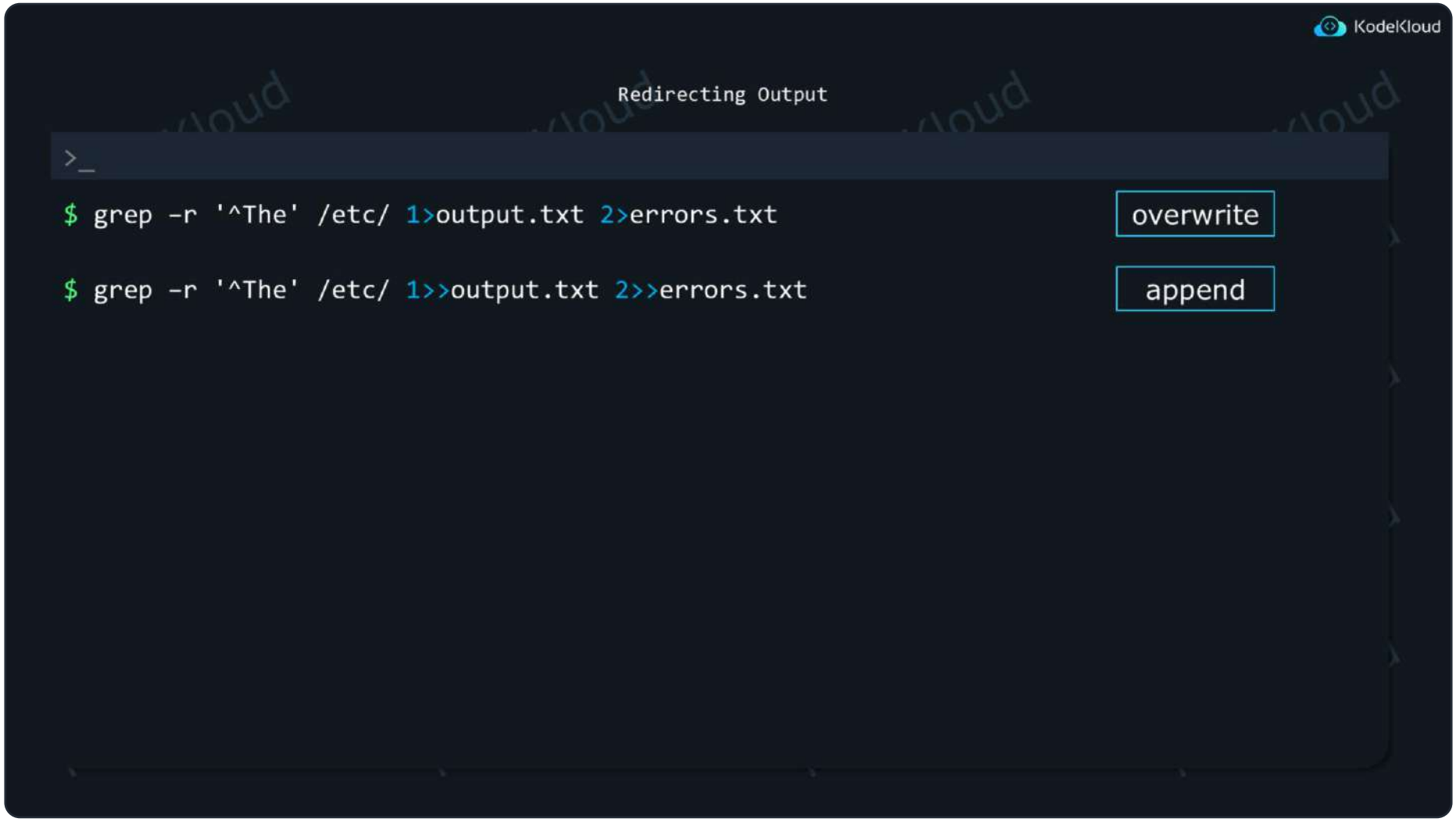
```
grep -r '^The' /etc/ 2>/dev/null
```

Now we have clean output that includes only the results we were looking for, no more error messages. The magic happened at:

```
2>/dev/null
```

2 signals that this refers to stderr, where error messages and warnings should be sent
> indicates we want to redirect this
/dev/null is the location where we chose to redirect stderr

With this, we redirected error messages off-screen, to /dev/null, the "black hole" of Linux. Whatever you send to /dev/null is simply discarded. This special /dev/null file is often used for such purposes, to get rid of unnecessary output.



We can also redirect normal output and error output, both at the same time, to two different files:

To overwrite old file contents:

grep -r '^The' /etc/ 1>output.txt 2>errors.txt

To add/append new contents at the bottom of the file, while keeping old content unchanged:

```
grep -r '^The' /etc/ 1>>output.txt 2>>errors.txt
```

Now if we take another look at the output generated by our grep command:

grep -r '^The' /etc/

we realize that normal messages are intertwined with error messages. If we'd want to save all output, exactly like we see it now, what command would we use?

```
grep -r '^The' /etc/ >all_output.txt 2>&1
```

2>&1 has to be at the end

This will ensure that everything we see on screen, that includes the standard output, and the error messages, is redirected to a file. I'll explain the syntax in a few moments.

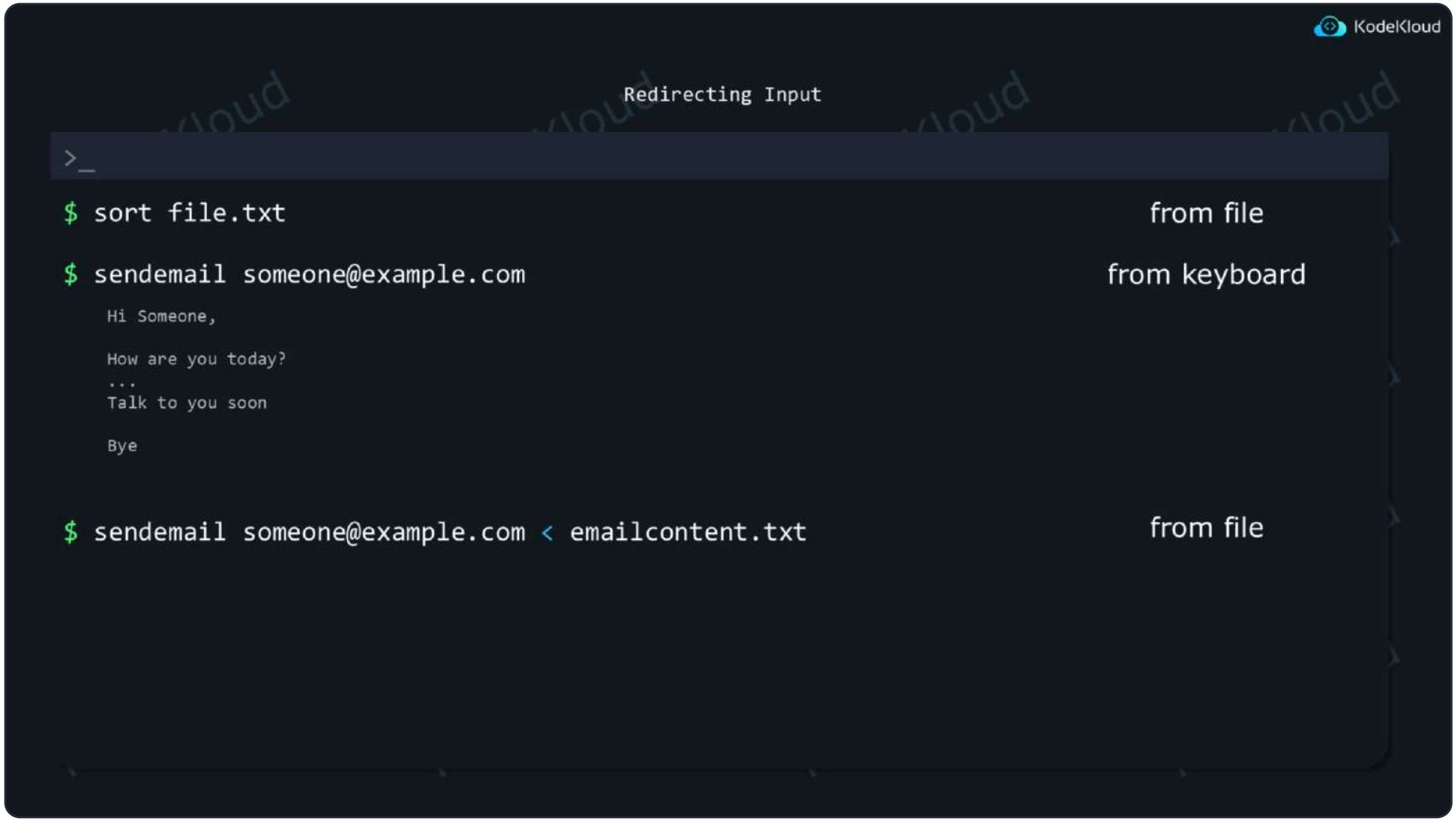
If you're not sure you can remember this order, of where to place these two redirection rules, then you can use an alternate command:

```
grep -r '^The' /etc/ 1>all_output.txt 2>&1
```

This way you see 1> then 2> and you know it's the natural order for these numbers. Which will help you place the redirect rules in their proper spots.

Now let's see. What exactly is happening here? First, with 1>all_output.txt we say "stdout, standard output should go to the all_output.txt file". With the second part, 2>&1 we say "stderr goes to stdout". In this case, &1 instructs our command to redirect to stdout instead of a file (&1 is stdout, &2 is stderr). So, with stderr going into stdout the error messages get mixed with normal messages. And now that we have this mix that we need, all of it will go to all_output.txt.

If you wrongly place 2>&1 in the first spot, the following happens. 2>&1 redirects errors to the current location of stdout. Which is: on screen (stdout at this point still directs to its default location, since it hasn't been redirected yet). So instead of errors ending up in all_output.txt file, they end up on the screen and get lost.



We explored output redirection. But why would we need to redirect input? Commands like

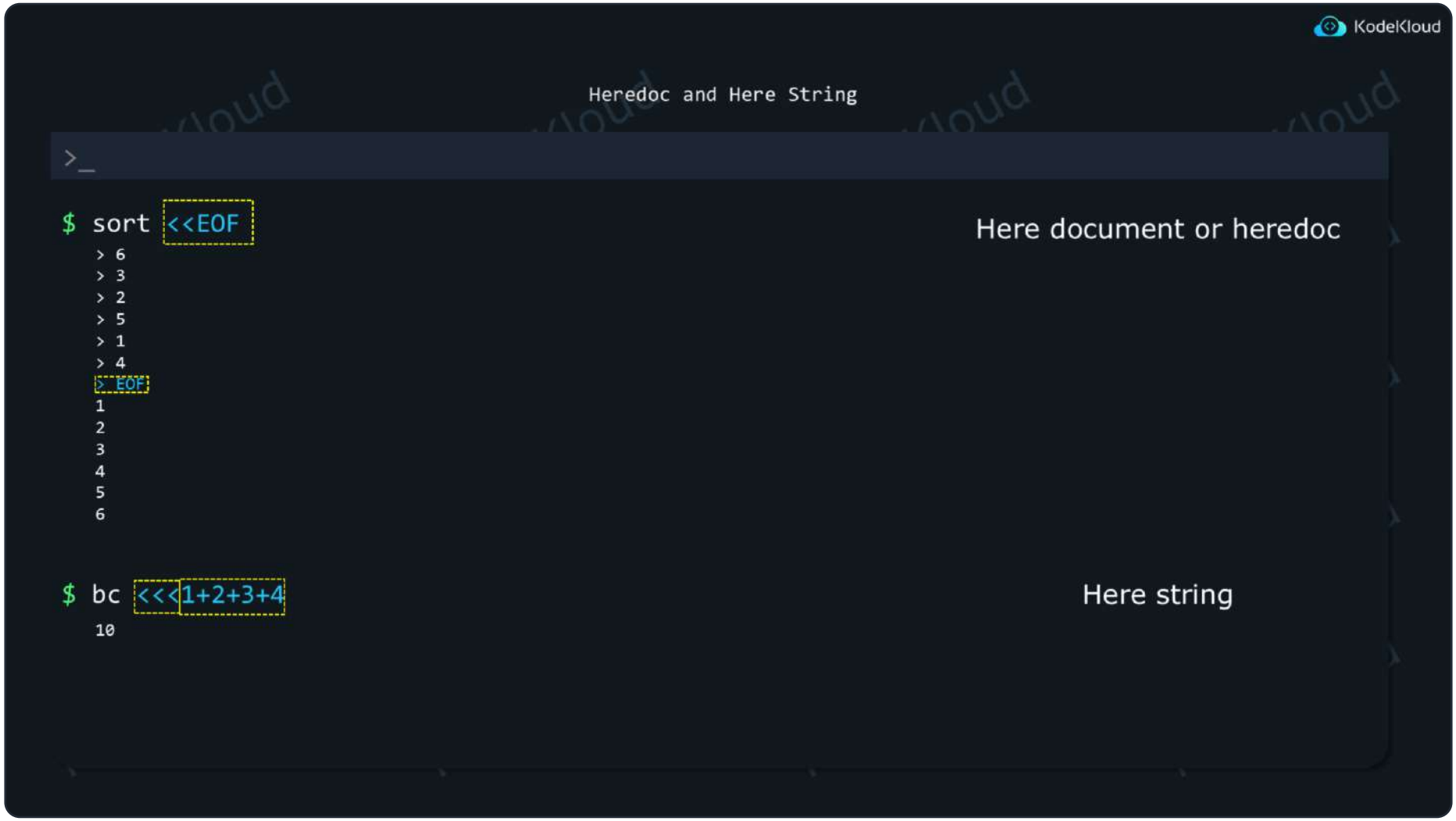
sort file.txt

already know they should take input data from file.txt.

Here's an example of one good use-case for input redirection. Think of an imaginary command "sendemail someone@example.com". By default, it expects input from your keyboard, so you can manually type the content of the email. It does not let you pass a file as input, like sort does. But with input redirection we can "trick it" into thinking we typed something on our keyboard.

```
sendemail someone@example.com <emailcontent.txt
```

With the help of < we redirect input, from file, to program. The effect will be exactly like we typed everything stored in the emailcontent.txt file. You will rarely need to redirect input, but when you find yourself not being able to pass a file to the utility, with regular approaches, you might want to try this.



There's also a more-often encountered use case for input redirection. But this time, not from a file.

This is called a "here document" or "heredoc" for short.

With something like <<EOF we signal that the input we want to pass ends before the last line where we type EOF. Any text can be used here instead of EOF (End Of File). We could type "END" or "DELIMITER", but "EOF" is commonly used.

This way we can type multiple lines of text to be passed as the input data to this command.

The advantage is that this can contain both the command and the input it should work with, grouped in one single block of text that can be easily copy/pasted, shared on the Internet, and so on.

A similar variant, called a "here string" is:

bc <<<1+2+3+4

With this, we can pass a single line (string) as input to our program. Whatever follows <<< is the input. Here, the benefit is easier to observe. Normally, bc is a command-line calculator that you open, enter its environment, and then type mathematical expressions.

This forces you to go through multiple steps. Enter command to open bc. Then type your mathematical expression. Then exit bc. Which might be inconvenient in some cases. With a here string you can pass input without needing to open the utility and use the keyboard to type into bc. Such things might be useful in scripts, where you want to automatically solve some math expression, get the result, but keyboard input cannot be available.

Most Linux utilities are very small and usually deal with a single task type. Let's think of these two programs:

grep is for searching
sort is for sorting text

It might seem limiting that each application can only do a single kind of task. Here's an example. We could search for all uncommented lines here, to find settings that are applied by this file:

```
grep -v '^#' /etc/login.defs
```

Easy enough to read through. But now imagine there are 1000 different settings here. 1000 lines of text. And we keep having to scroll up and down, again and again, to look at different things that interest us. These are entirely disorganized, we wouldn't know where "CREATE_HOME" or "ENCRYPT_METHOD" might be placed, at what line. Now if we could sort all variables here alphabetically, it would make our life much easier. We would scroll to the top when we look for "CREATE_HOME" and scroll a bit down to find "ENCRYPT_METHOD". They should be close to each other since the alphabetic order is ABCDE, so ENCRYPT should be right after anything starting with the D letter.

But if grep can only search, but not sort output, how do we solve this problem? Well, we have the sort utility. But this means we need to somehow send output from grep, directly to sort. And sure enough, we can easily do that with the | sign. This is called piping output, from program A to program B, on the command line.

```
grep -v '^#' /etc/login.defs | sort
```

And just like that, problem solved.

But now notice how columns are misaligned. Some are further to the right, some further to the left. How could we solve this? Pipe to yet another program. We can keep adding as many | signs as we want.

In this case, with grep we find our text. Then we pipe output to the sort utility to alphabetically sort everything. And, finally, we pipe from sort to the column utility which will arrange our columns nicely. Each program gets output from the previous program on the left, works on it, and then sends it to the next program on the right.

```
grep -v '^#' /etc/login.defs | sort | column -t
```

Now it looks perfect!

We notice that piping output can be very powerful. That's because it basically allows us to interconnect many programs and do much more complex things than a single utility can do on its own.

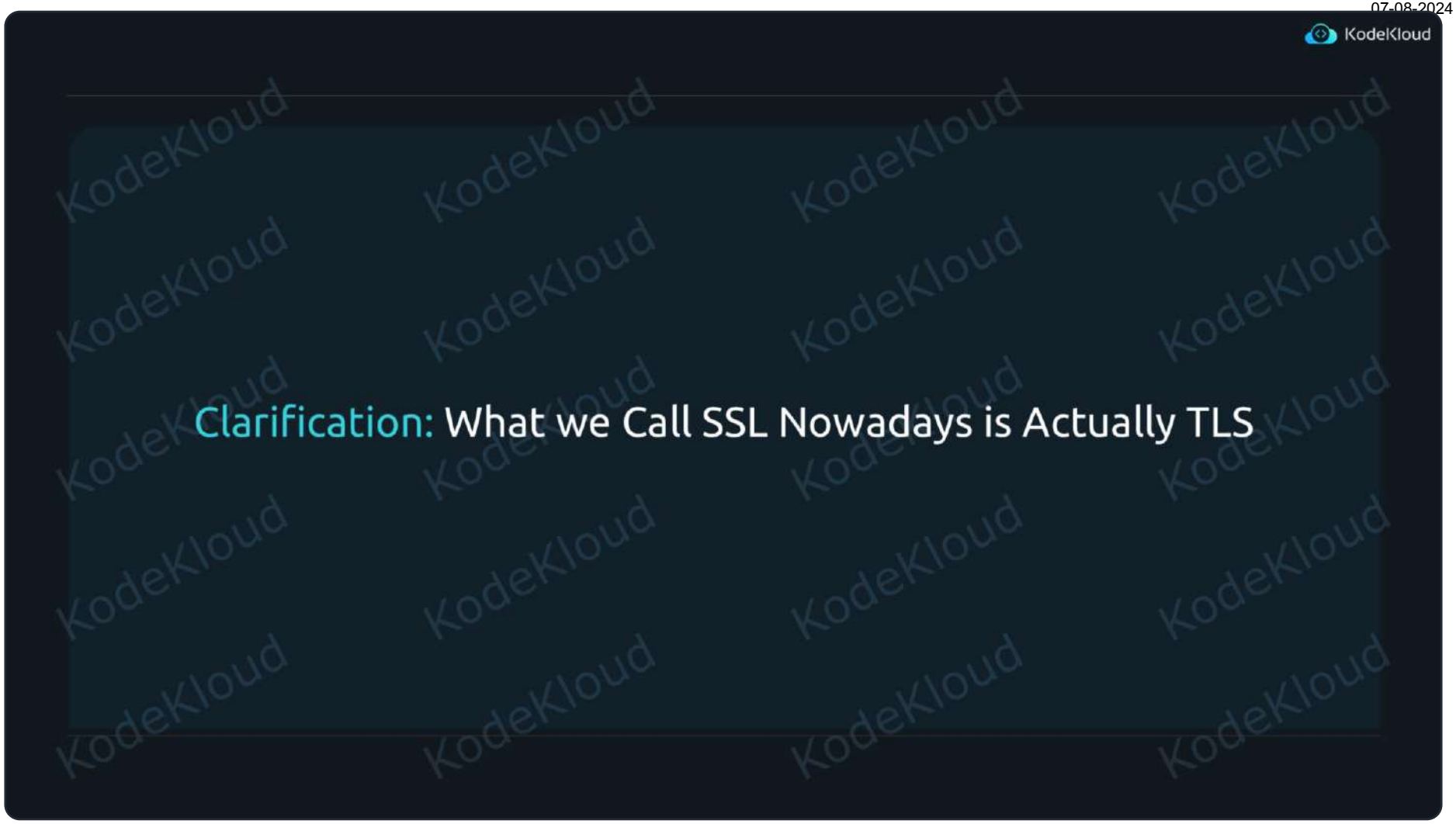
KodeKloud

Visit www.kodekloud.com to discover more.

KodeKloud

Visit www.kodekloud.com to discover more.

In this lesson we'll explore extended regular expressions.



But first, a clarification. What we called SSL in the past is nowadays actually TLS. But they're often still referenced with the old name that does not apply anymore.

Let's explore the terminology. SSL stands for secure sockets layer. TLS is short for transport layer security.

So why do we still use the wrong, old name? Because SSL was used for a very long time. And the name kind of stuck around in various tools and documentation. Even after most certificates migrated to TLS.

We can think of TLS as an upgrade over SSL. SSL had many security issues and TLS closed a lot of those security holes. With this clarification out of the way, let's get to our next topic. What

are these certificates used for?

What Are SSL Certificates?

Well, let's think about what happens when we use a password on a website. Or when we enter credit card details. There are two big problems:

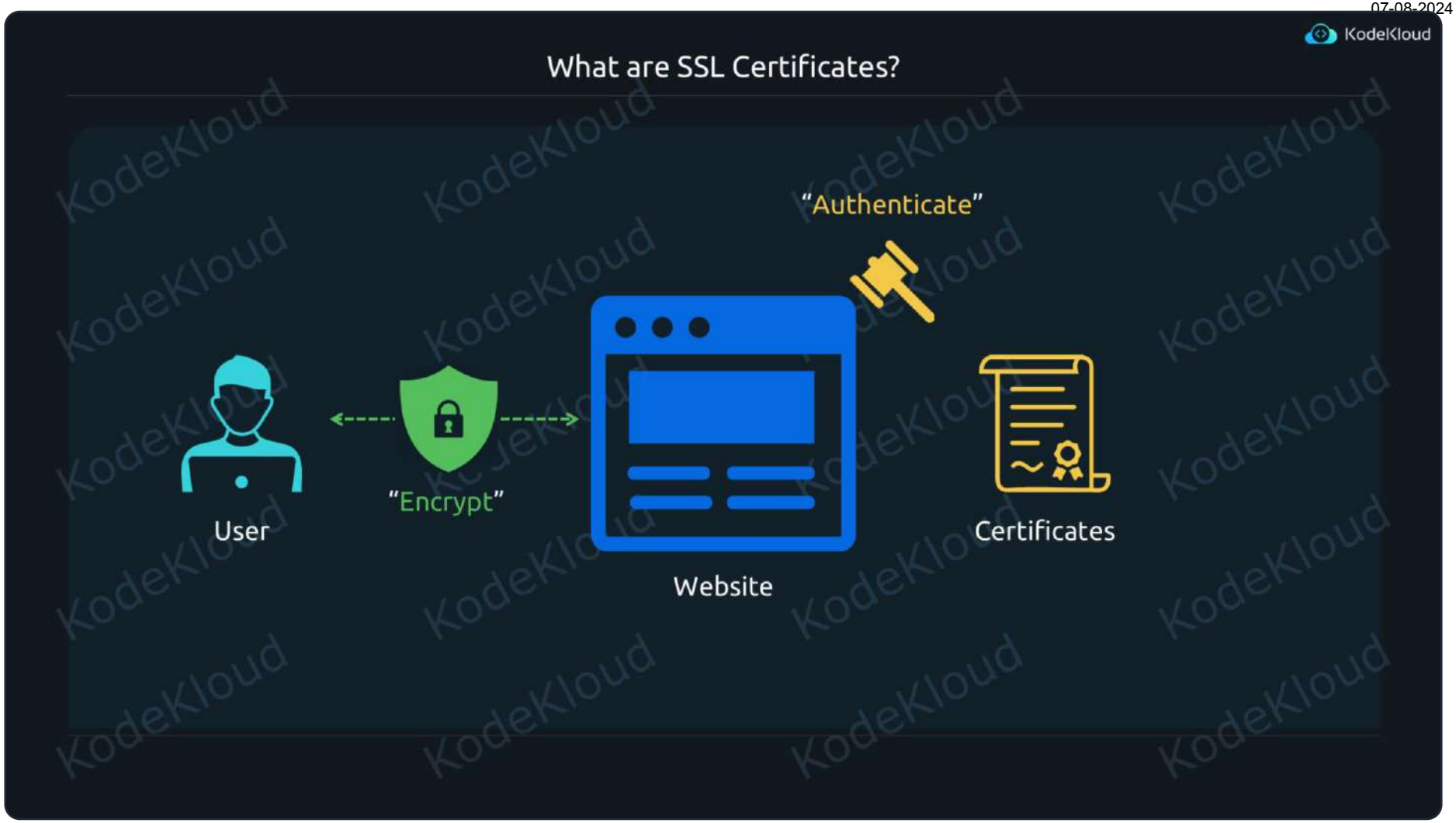
One, how can we be sure that we are sending this data to the legitimate website, and not some clone created by some malicious hacker?

And two, how can we be sure that no one can steal these details as they're being sent through the networks?

Well, certificates solve both of these issues. They authenticate the website, and they encrypt network traffic between user and website.

Authentication means that the website can cryptographically prove to a browser that it is legitimate. It proves that it's actually KodeKloud.com, and not some clone. And the certificate is also used to make the connection private between the user and KodeKloud.com. All data exchanged with the website is encrypted.

Now let's see how we can actually create such certificates.



What Are SSL Certificates?

Well, let's think about what happens when we use a password on a website. Or when we enter credit card details. There are two big problems:

One, how can we be sure that we are sending this data to the legitimate website, and not some clone created by some malicious hacker?

And two, how can we be sure that no one can steal these details as they're being sent through the networks?

Well, certificates solve both of these issues. They authenticate the website, and they encrypt network traffic between user and website.

Authentication means that the website can cryptographically prove to a browser that it is legitimate. It proves that it's actually KodeKloud.com, and not some clone. And the certificate is also used to make the connection private between the user and KodeKloud.com. All data exchanged with the website is encrypted.

Now let's see how we can actually create such certificates.

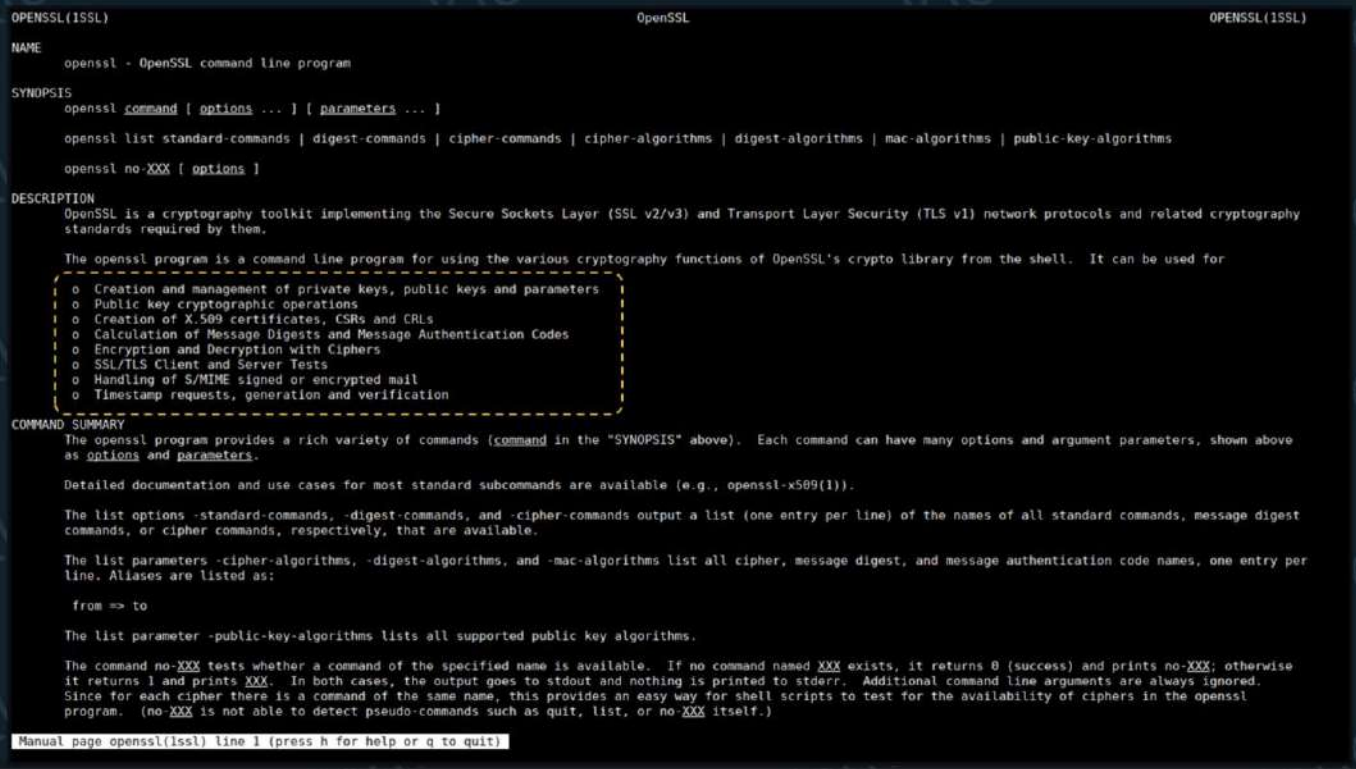
KodeKloud

How to Create TLS/SSL Certificates on Linux

The utility that is normally used on Linux to create and manage TLS certificates is called "**openssl**". And we can already notice an area where the old "SSL" name is still hanging around from the old days. While it's called "open**ssl**", we'll actually create TLS certificates with it.

KodeKloud

How to Create TLS/SSL Certificates on Linux



If we consult the manual of openssl, with the "man openssl" command, we'll see the documentation points out the utility can be used for a lot more than just certificates.

It can deal with many cryptography-related operations. But what interests us in this lesson is the "Creation of X.509 certificates". These are the certificates that we can use on websites to do the authentication and encryption we mentioned before.

Unfortunately, the openssl command supports a very large number of subcommands and options. And it can be hard to remember what we actually need to type for a specific task. But the good thing is, we can use a few tricks to orient ourselves. Let's learn how to do that right on the command line.

Let's quickly explore Certificate Signing Requests.

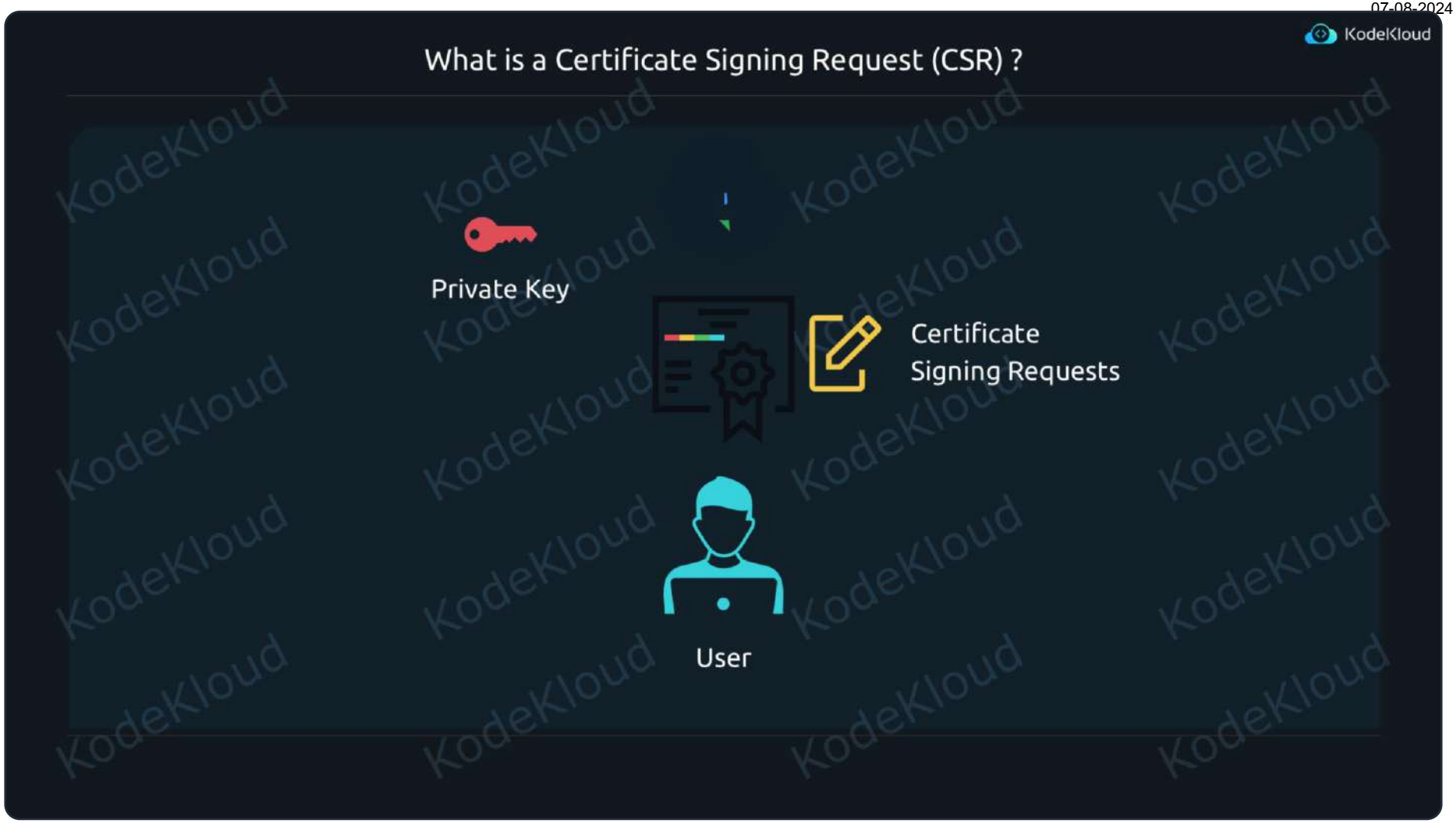
The req subcommand deals with such requests.

The req subcommand deals with certificate signing requests. That's because the digital certificates used to secure website traffic are not enough on their own. "example.com" can use a certificate to secure web traffic between a user and the website. That's no issue. But when a user visits example.com, their browser also needs to trust this certificate. It needs a way to be sure that the certificate is legitimate, the real deal, and not some fake created by some malicious hacker. How can it be sure? By checking if something called a Certificate Authority signed example.com's certificate.

So we can send our certificate that we generate locally, to some company like Google. Then they then use a special private key and sign our certificate. Once that certificate is signed, any

browser can then check this signature and notice that indeed, Google validated it, so it's legitimate.

That's because the digital certificates used to secure website traffic are not enough on their own. "example.com" can use a certificate to secure web traffic between a user and the website. That's no issue. But when a user visits example.com, their browser also needs to trust this certificate. It needs a way to be sure that the certificate is legitimate, the real deal, and not some fake created by some malicious hacker. How can it be sure? By checking if something called a Certificate Authority signed example.com's certificate.



So we can send our certificate that we generate locally, to some company like Google. Then they then use a special private key and sign our certificate. Once that certificate is signed, any browser can then check this signature and notice that indeed, Google validated it, so it's legitimate.

KodeKloud

What is a Certificate Signing Request (CSR) ?

User

KodeKloud

Generating a Key and CSR

Private Key

Certificate
Signing Request

[Just comments for Jeremy. Delete slide after reading. Second, bigger demo starts here]

[At this part from the source document:]

Now let's see how to generate two things:

A private key.

And the Certificate Signing Request.

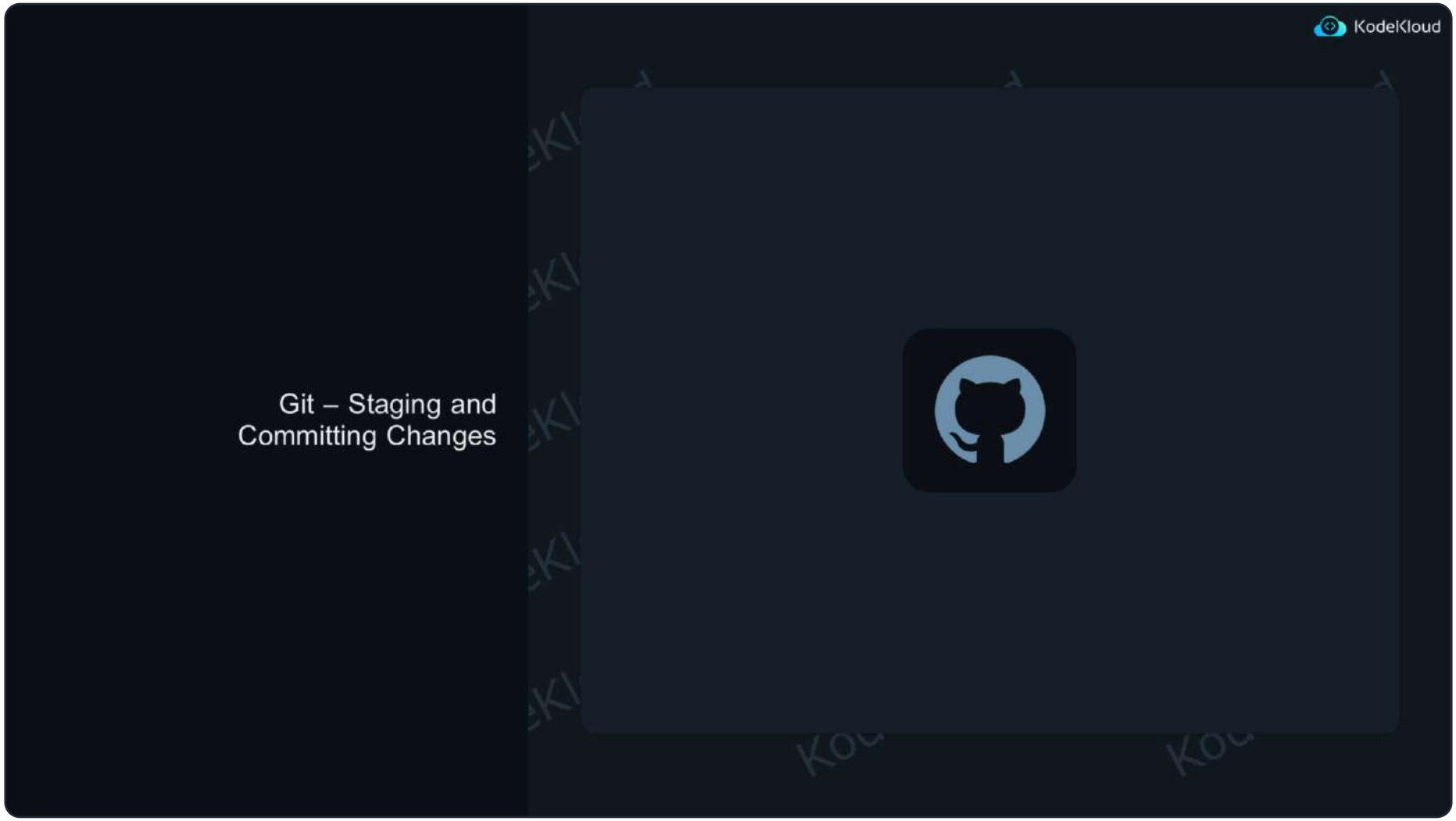
Certificates are normally used in conjunction with such a secret key.

KodeKloud

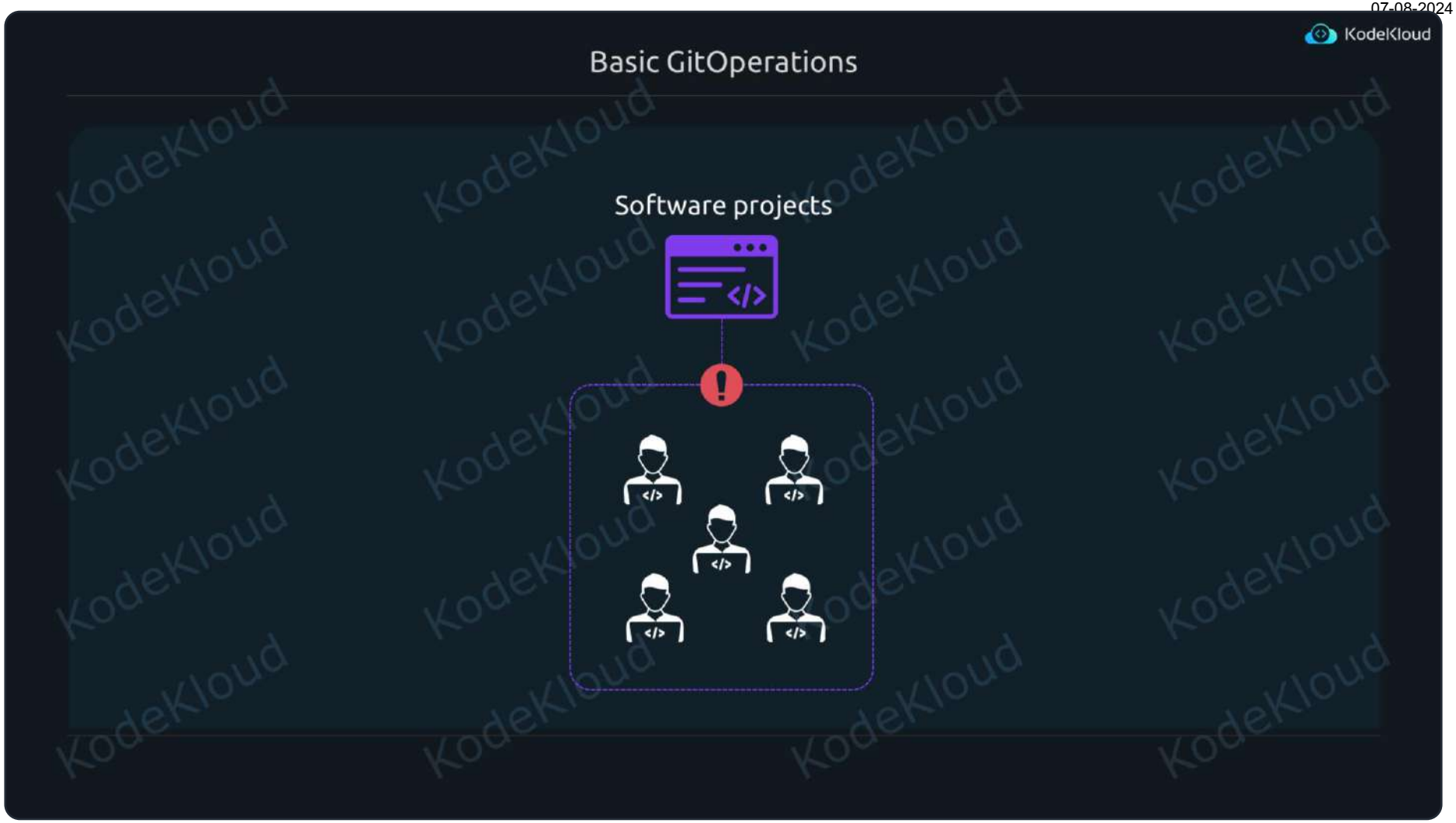
Visit www.kodekloud.com to discover more.

KodeKloud

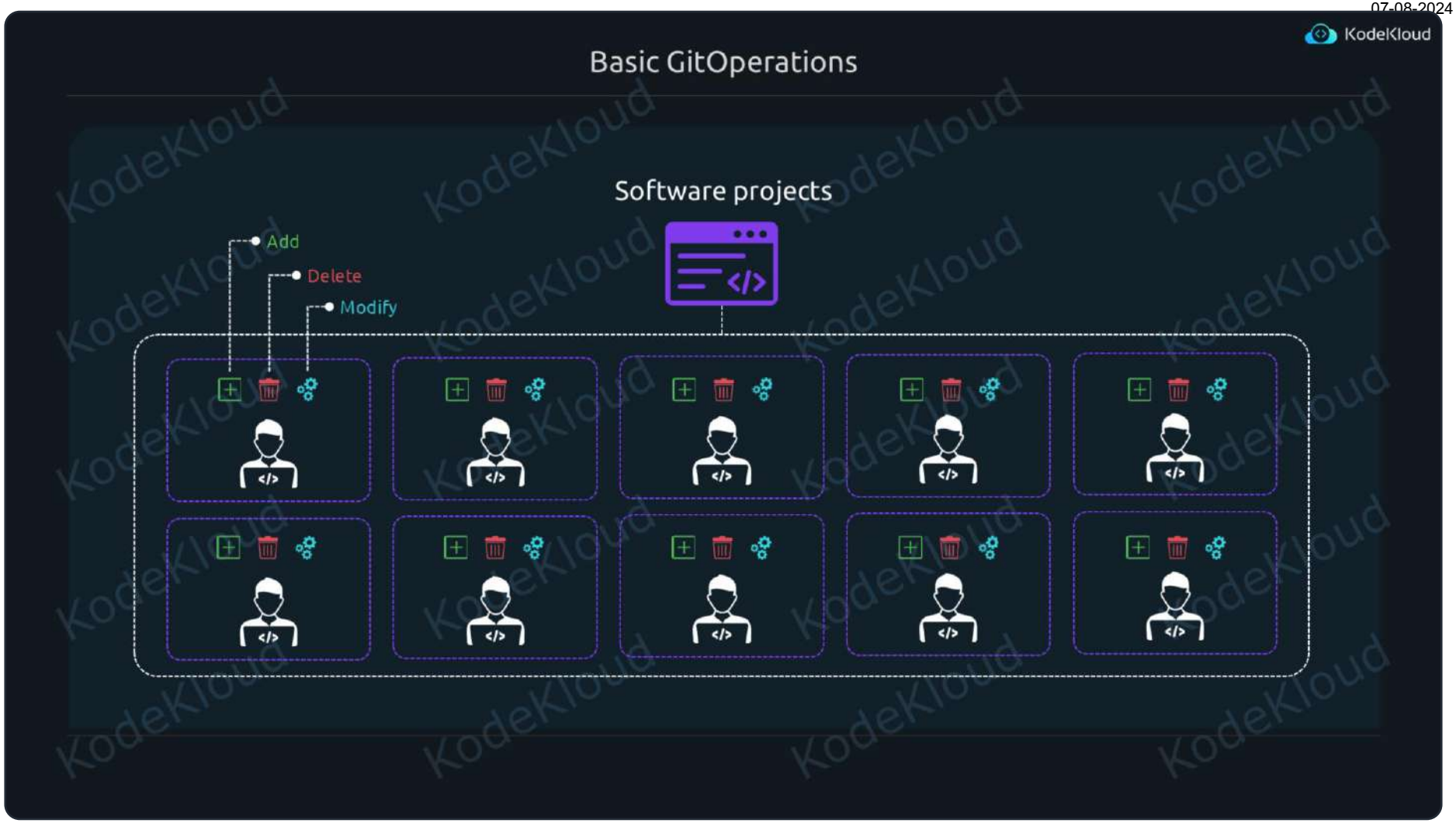
Visit www.kodekloud.com to discover more.



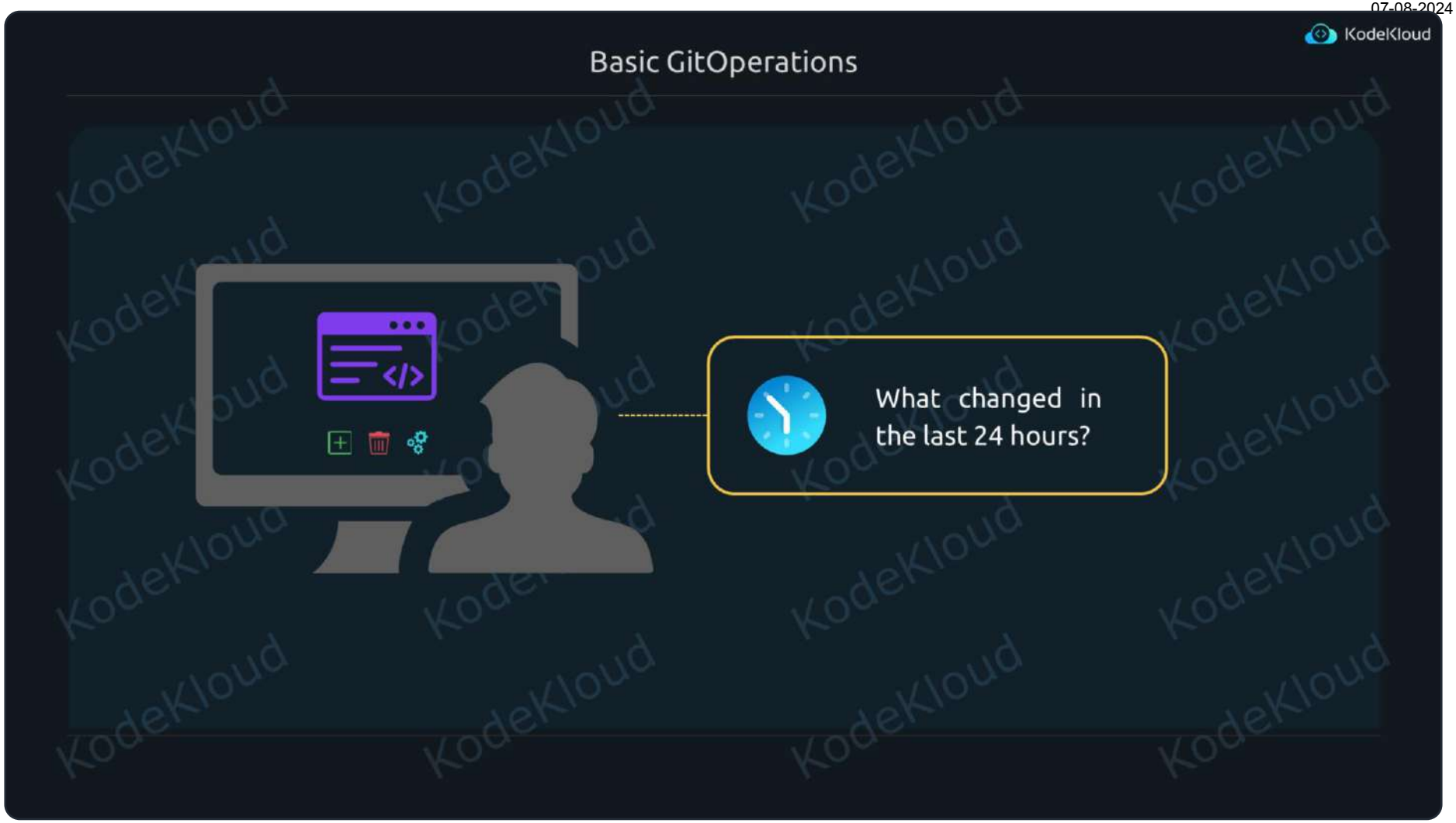
In this lesson we'll explore extended regular expressions.



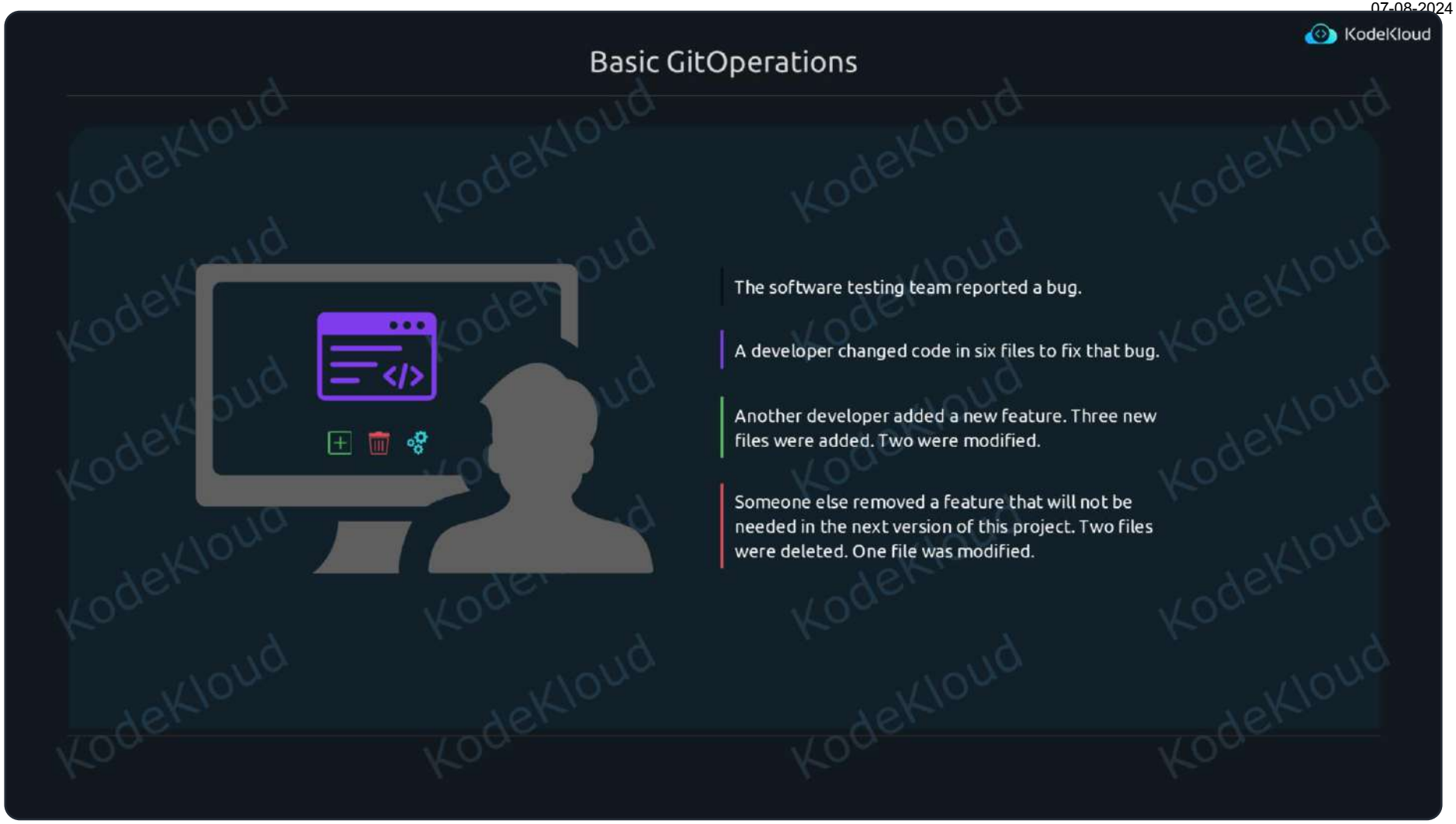
In today's world, software projects are usually developed by large teams. And this comes with an interesting set of problems.



Imagine we have ten developers working on the same project. Each person will add, delete, and modify code in random places.



Now imagine you're part of the same team. You turn on your computer, ready to do some work. How will you know what changed in the last 24 hours?



Here are some examples of activities that might have happened during this time:

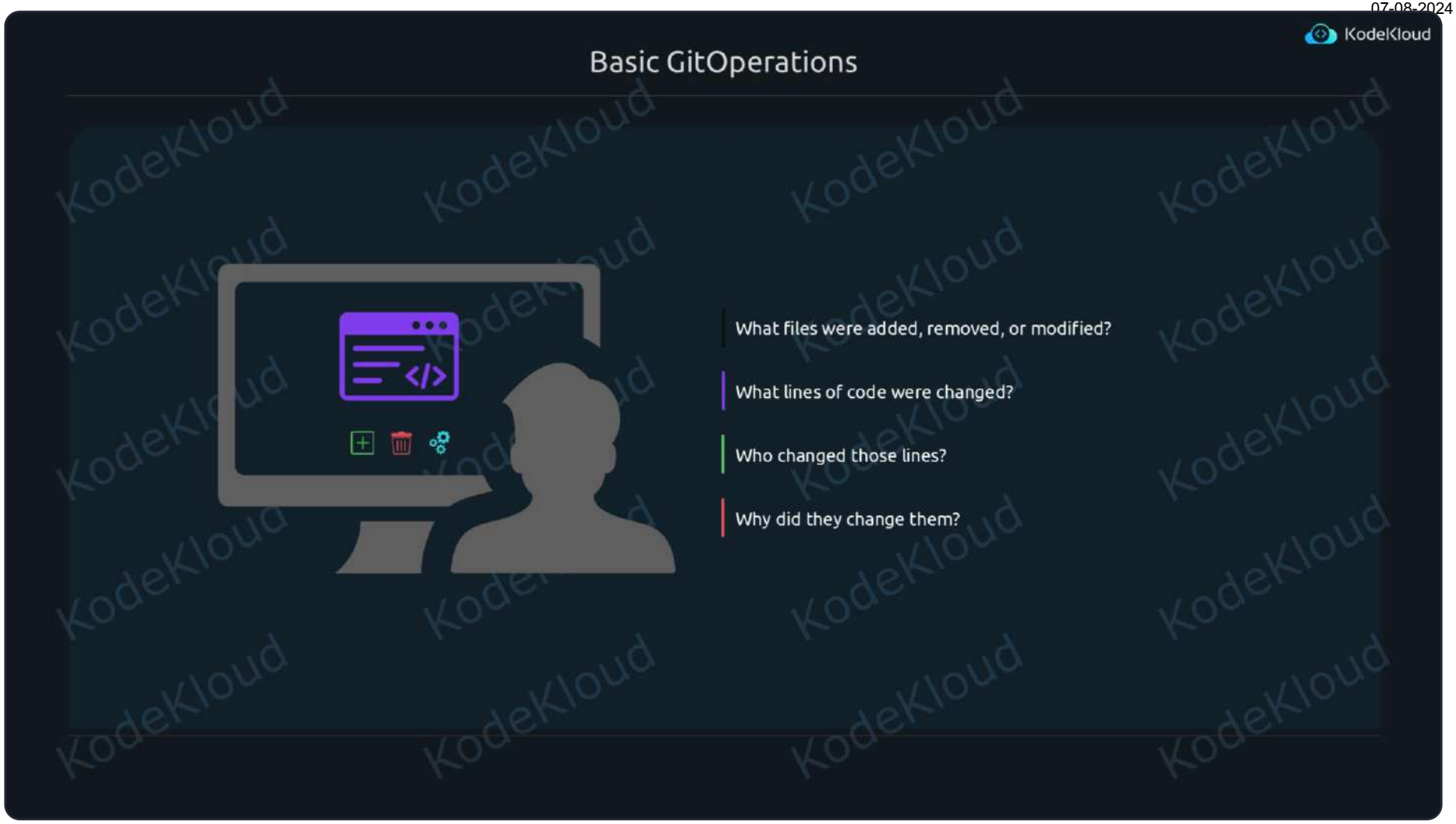
The software testing team reported a bug.

A developer changed code in six files to fix that bug.

Another developer added a new feature. Three new files were added. Two were modified.

Someone else removed a feature that will not be needed in the next version of this project. Two files were deleted. One file was modified.

Quite a large number of changes in a short time. Hard to keep track of everything. Developers could inform each other in chat about what they did. But that would quickly become messy, overwhelming each person with information they need to remember.



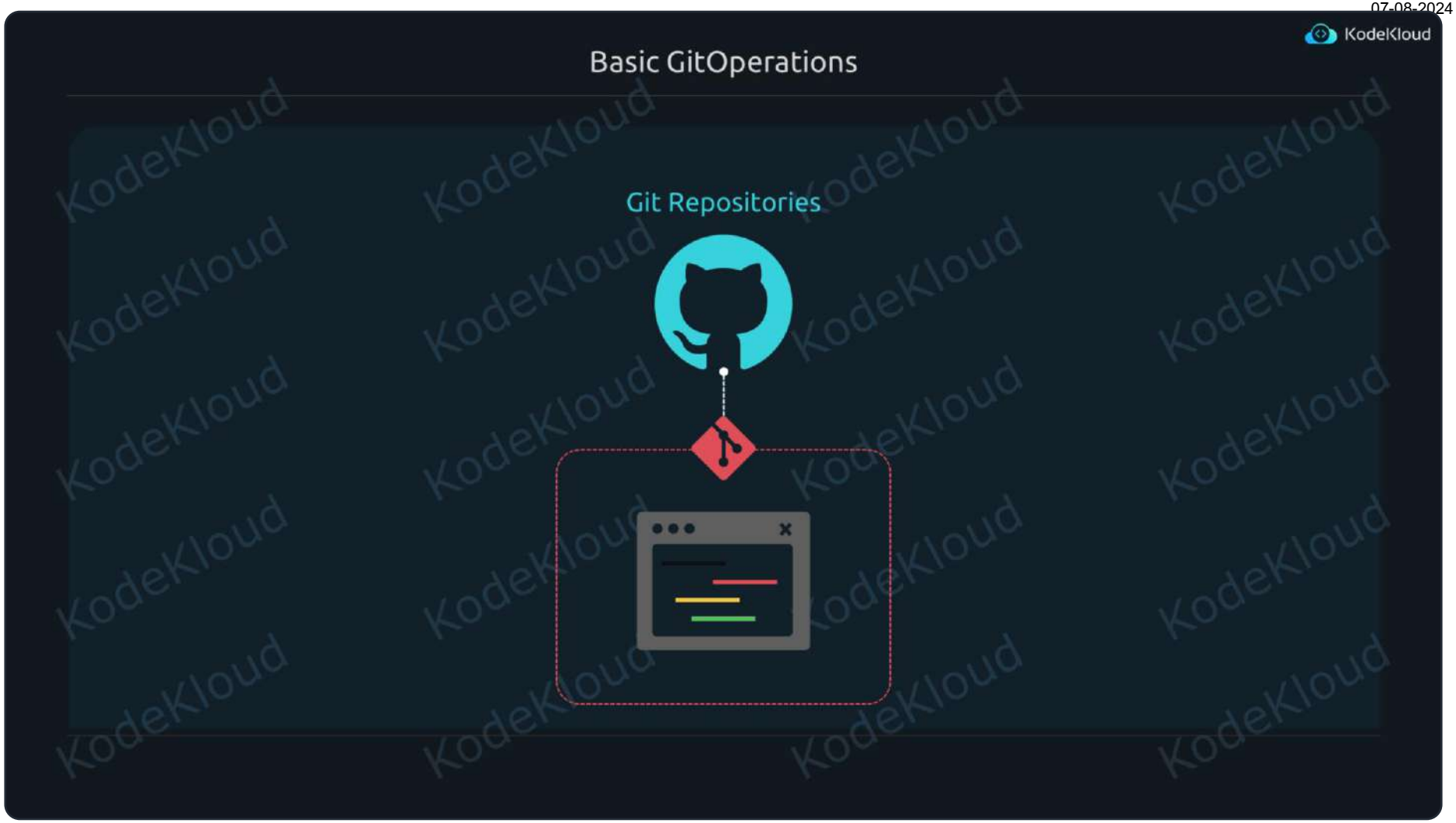
So when you begin work, you need a quick way to get updated on all of these changes. You'll want to know things like this:

What files were added, removed, or modified?
What lines of code were changed?
Who changed those lines?
Why did they change them?

These are just a few examples. In a nutshell, you need to know what the rest of the team did. And when you make your own changes, you need some way to inform the rest of the team about what you changed, and why. With an efficient method to do this, everyone can work on the same code and always know what happened since the last time they logged in.

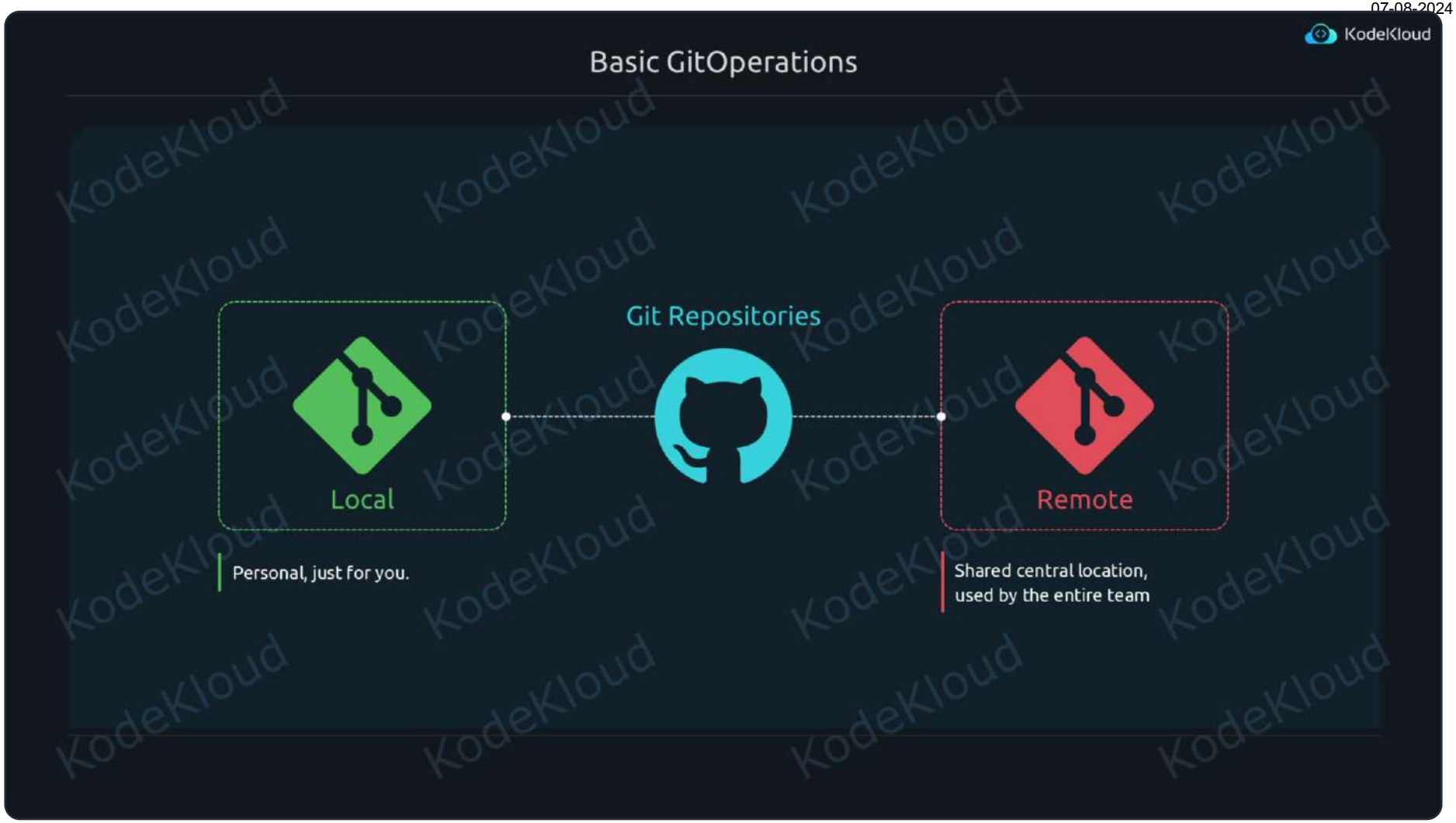
So we need a way to track all of these changes, and keep things organized. The solution is to use a distributed version control system like Git.

At its core, Git's job is to allow a large group of people to work on the same software project.

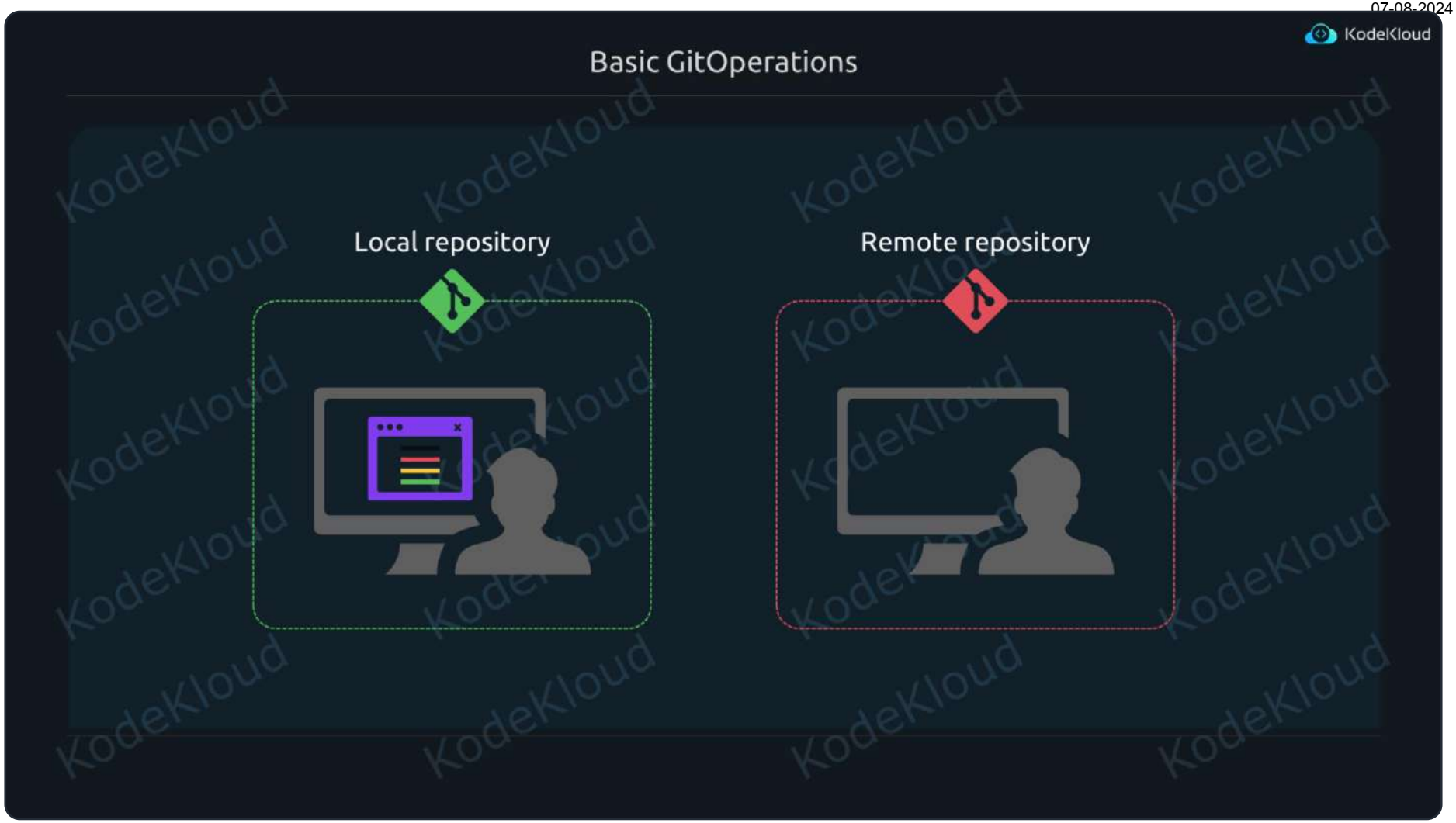


Let's take it step-by-step and discover how it works.

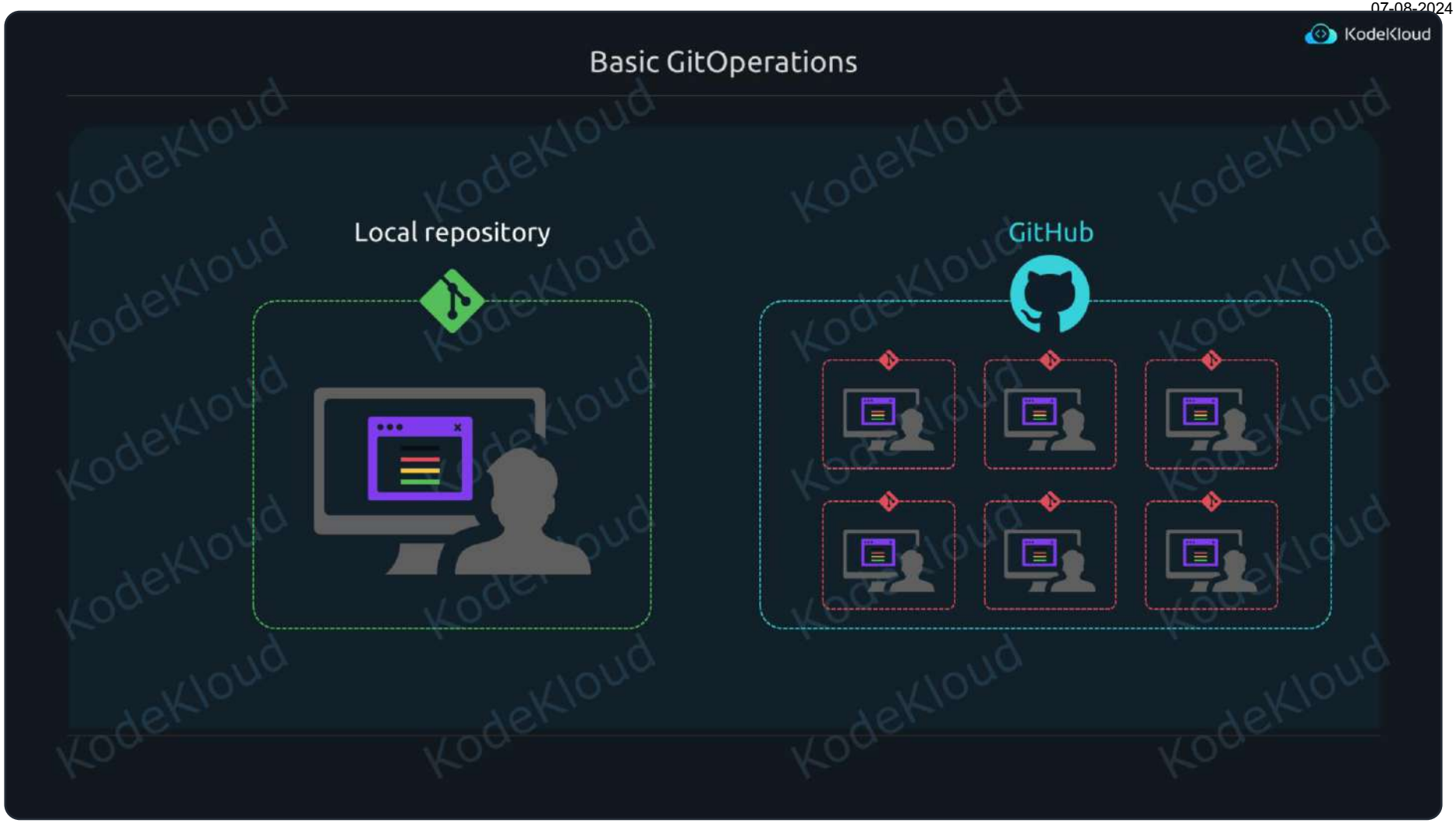
First of all, in Git we have the concept of repositories. These are the places where code is stored, alongside information about each change.



As part of a team, we'll usually have two repositories: a local repository, and a remote repository. The local repository is personal, just for you. And the remote repository is a shared central location, used by the entire team.



For example, as you code on your own computer, you work with your local repository. And when you're happy with the results, you can upload your latest changes to the remote repository.

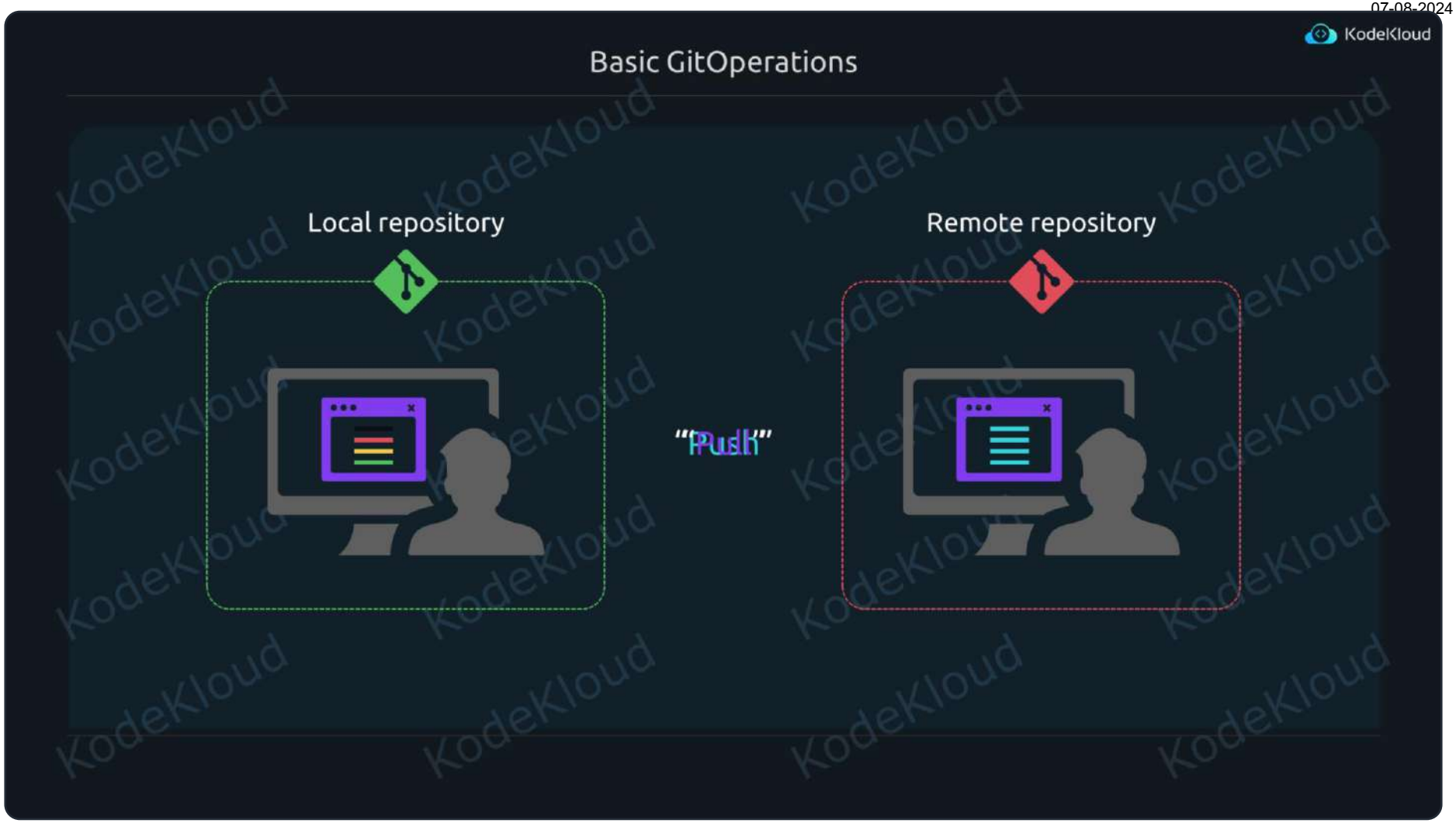


If you ever visited GitHub, you already saw a lot of such remote repositories. Every project you see there is basically a remote Git repository.

By uploading to this remote repository, each team member can update the project at a central location. And by downloading from the remote repository, each team member can see what other people have been working on. This way, they get updated about all the latest changes.

You can think of the remote repository as the place where everyone assembles all the bits and pieces they've been working on. Every small change from local repositories is gathered and

integrated into this remote repository.



When you want to update the remote repository with the latest changes you made, you push the latest code from local to remote. When you want to get access to the latest changes your team members made, you do a pull from the remote repository to your local one. We'll see that the commands for these actions are quite intuitive.

The best way to understand Git better is to just see it in action. So let's go through a few practical exercises.

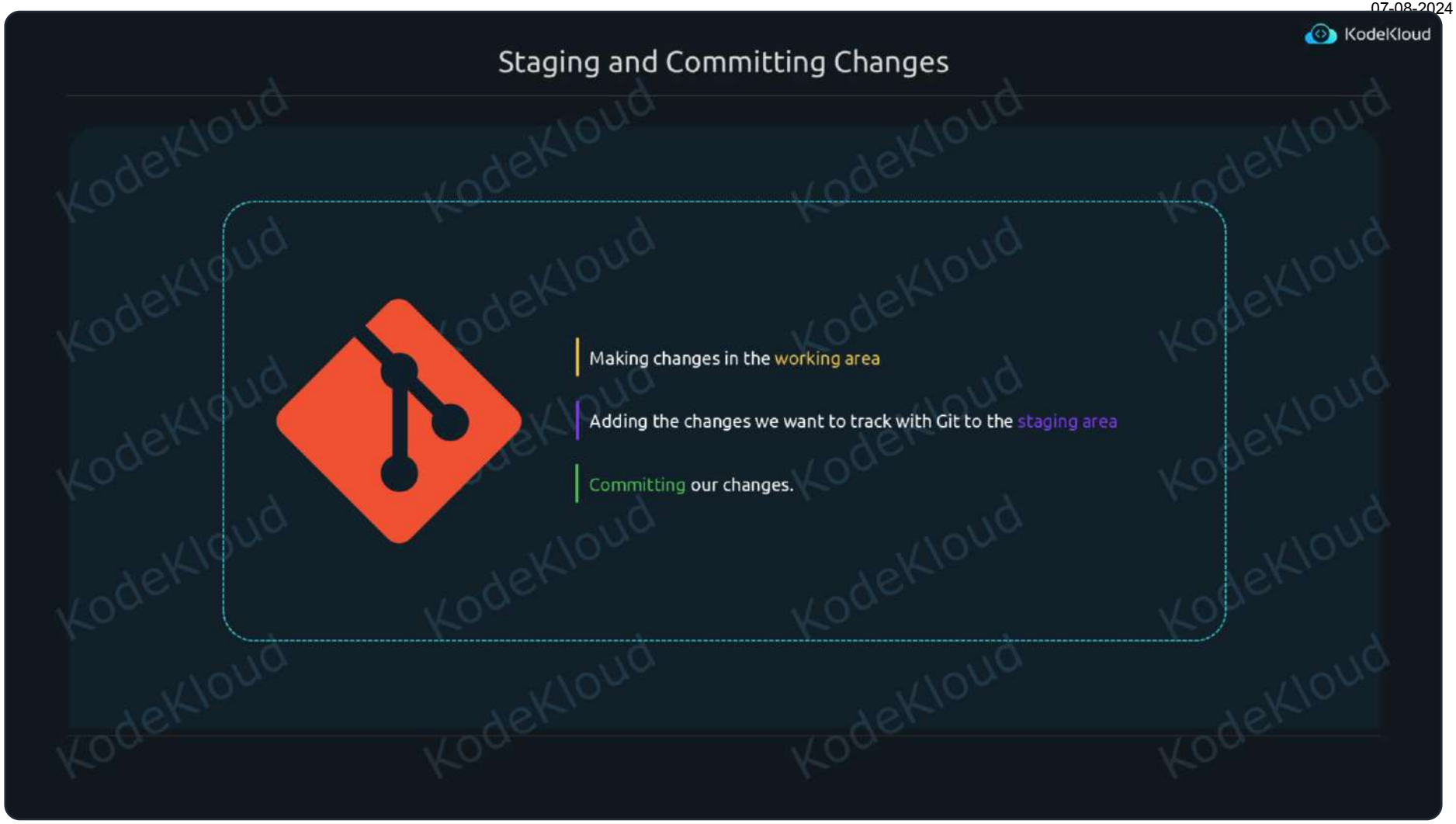
KodeKloud

Visit www.kodekloud.com to discover more.

KodeKloud

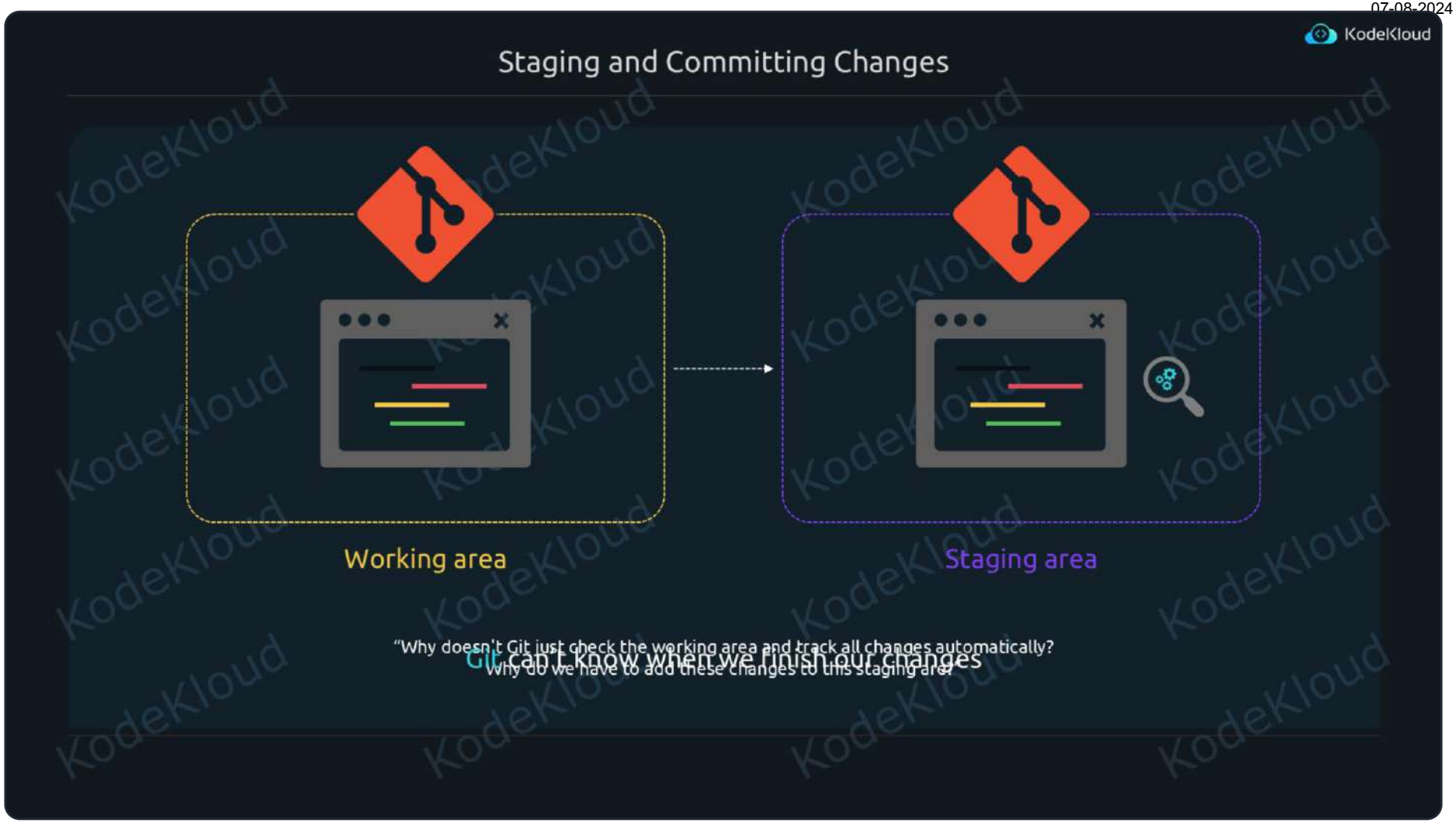
Visit www.kodekloud.com to discover more.

In this lesson we'll look at how to stage and commit changes with Git.



There are three steps to modifying code and then tracking changes with Git:

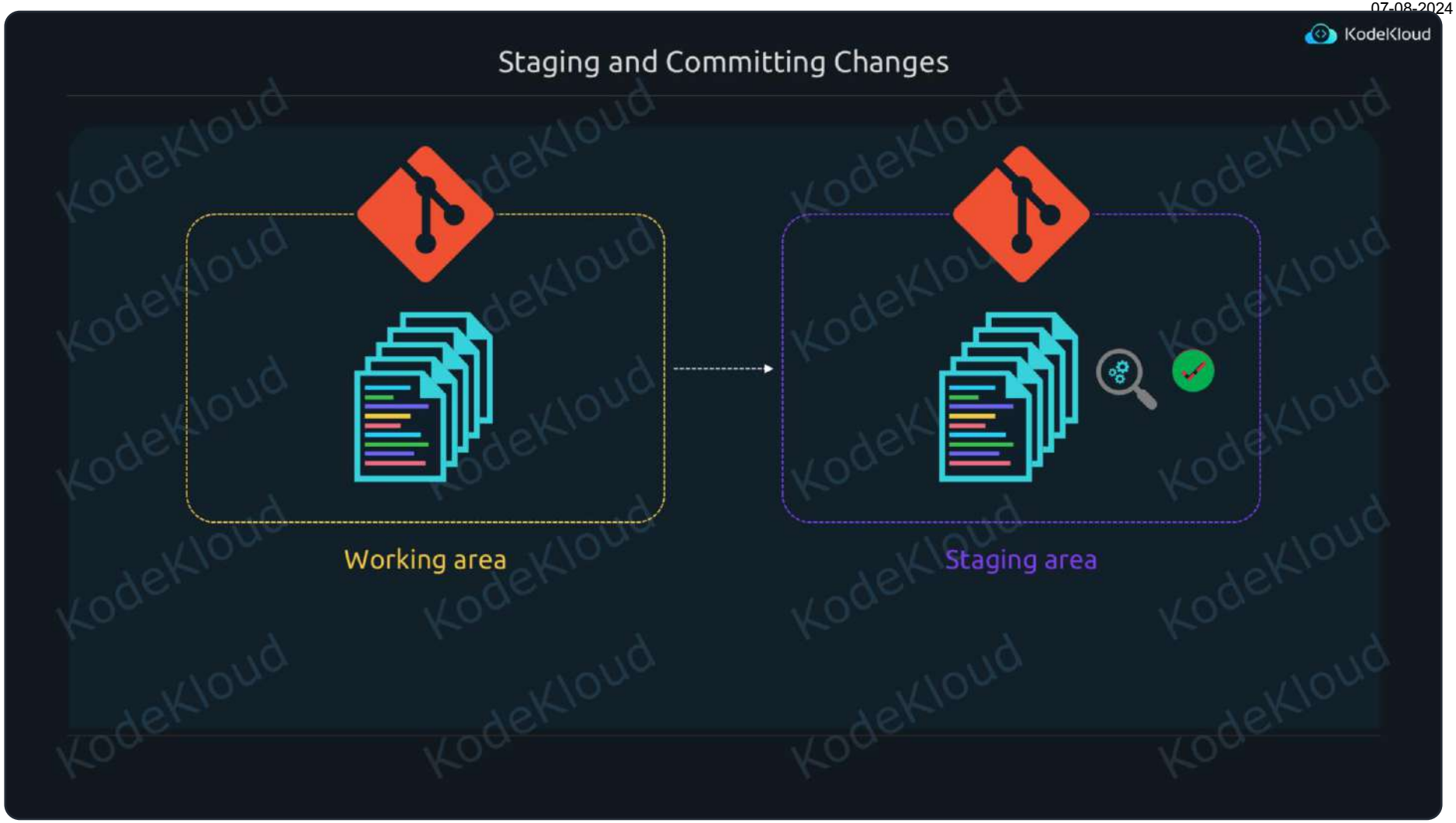
Making changes in the working area.
Adding the changes we want to track with Git to the staging area.
Committing our changes.



At this point, the first step is done. Because the working area is the "project" directory we are currently in. And we already made our changes to the working area when we added our 2 files and the text inside them. So every time we change something in our project's directory, we've modified our working area.

Now we get to phase two, the staging area. This is how we tell Git about the changes we made to our project. Or, more specifically, what changes we want it to track in the next commit. We might wonder: "Why doesn't Git just check the working area and track all changes automatically? Why do we have to add these changes to this staging area?" One answer is that Git can't know when we finish making our changes.

For example, maybe we added 50 lines of code today in some file. But we intend to add another 50 lines tomorrow to wrap things up. So we don't want Git to track this partial change, when we're at 50% progress. We want to wait until we finish adding all our lines of code.



Also, if we modify 10 different files, we don't want it to track 10 different phases, for each file. Instead, we want to track this as a single change. Even if we modified 10 files, maybe this was done just to add a new feature to our software project. So we want to track this as a single modification in our project's lifecycle, not 10 modifications.

Now let's go back to our demo and see how this works.

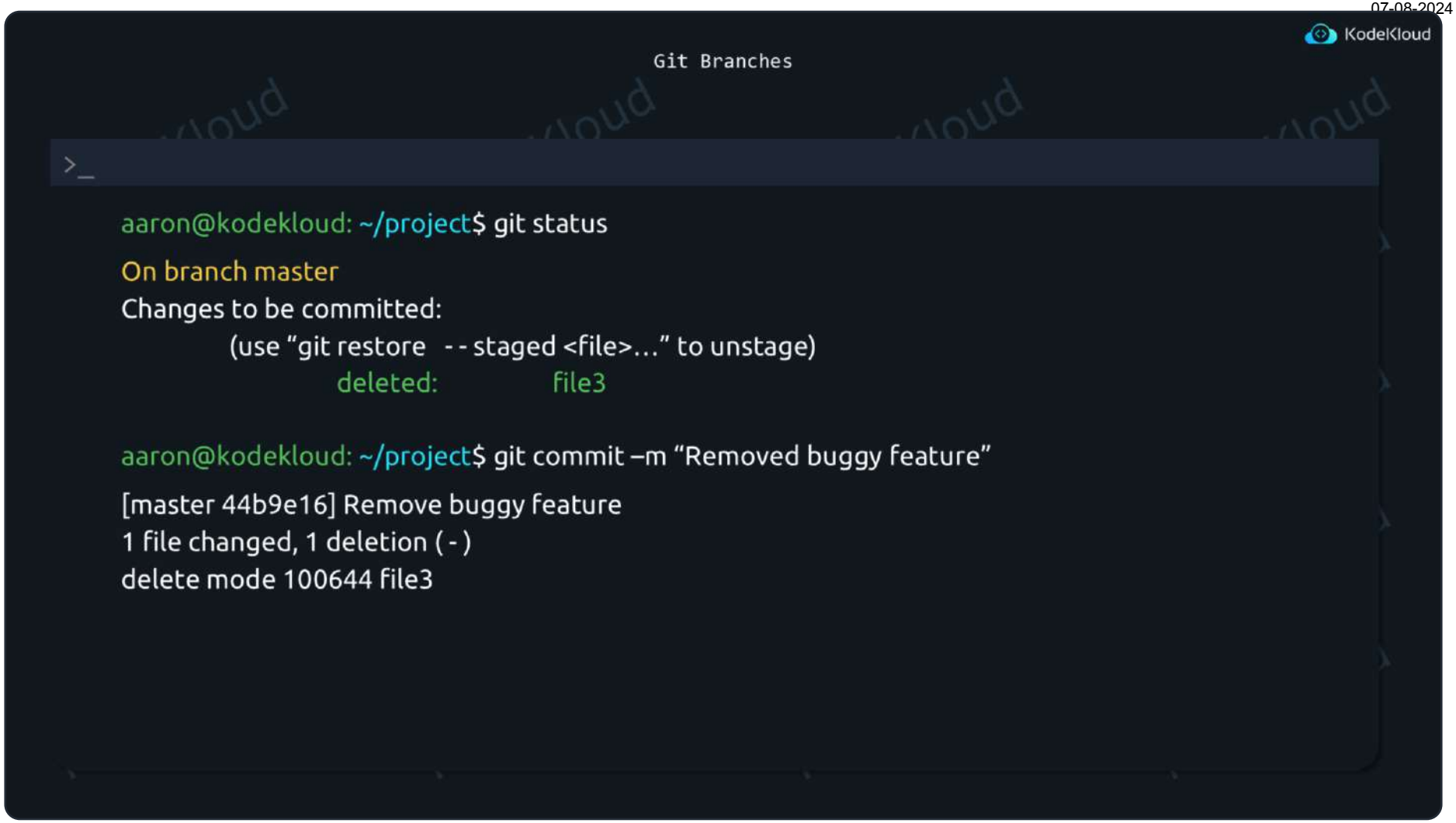
KodeKloud

Visit www.kodekloud.com to discover more.

KodeKloud

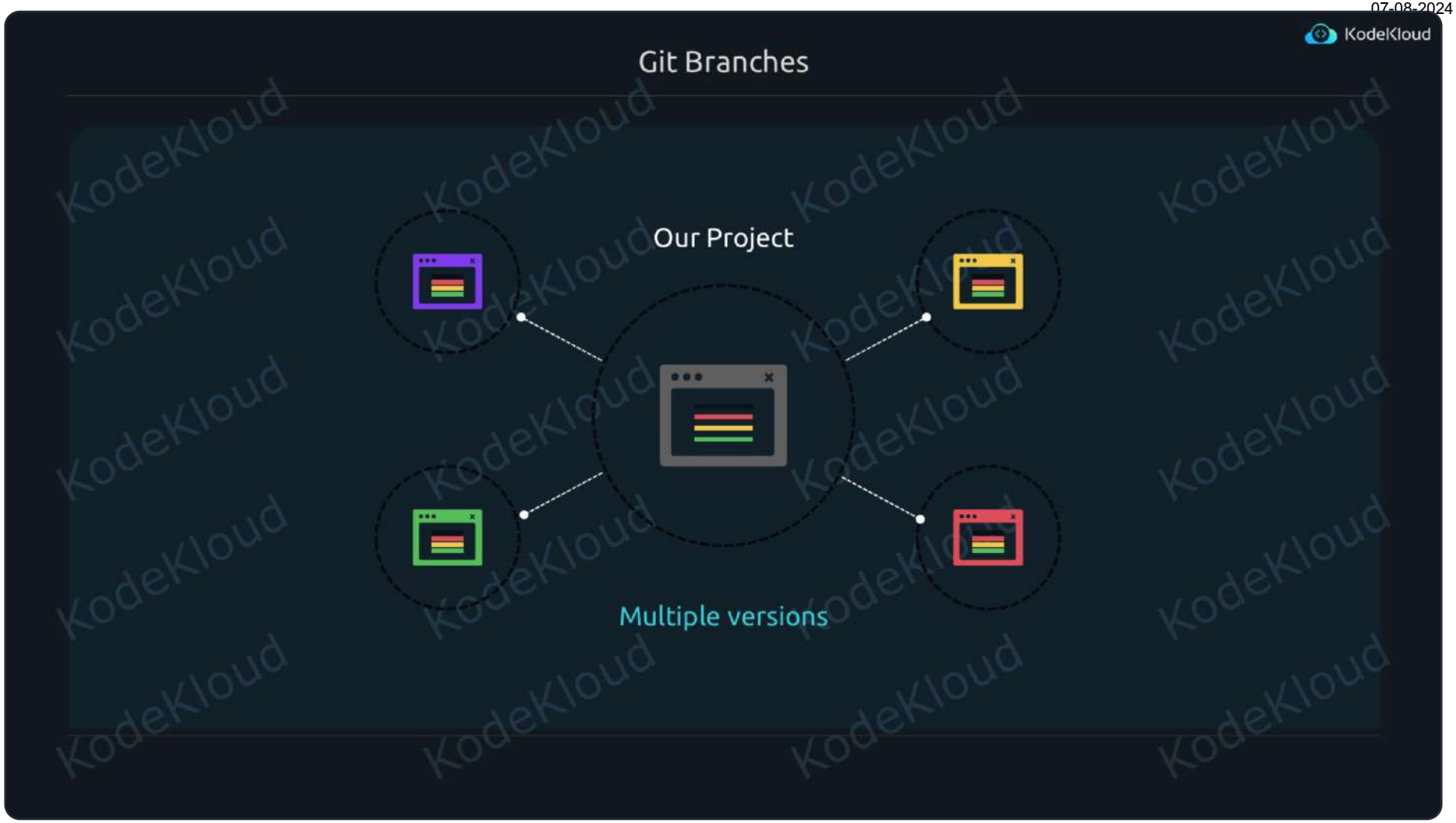
Visit www.kodekloud.com to discover more.

In this lesson we'll talk about Git branches.



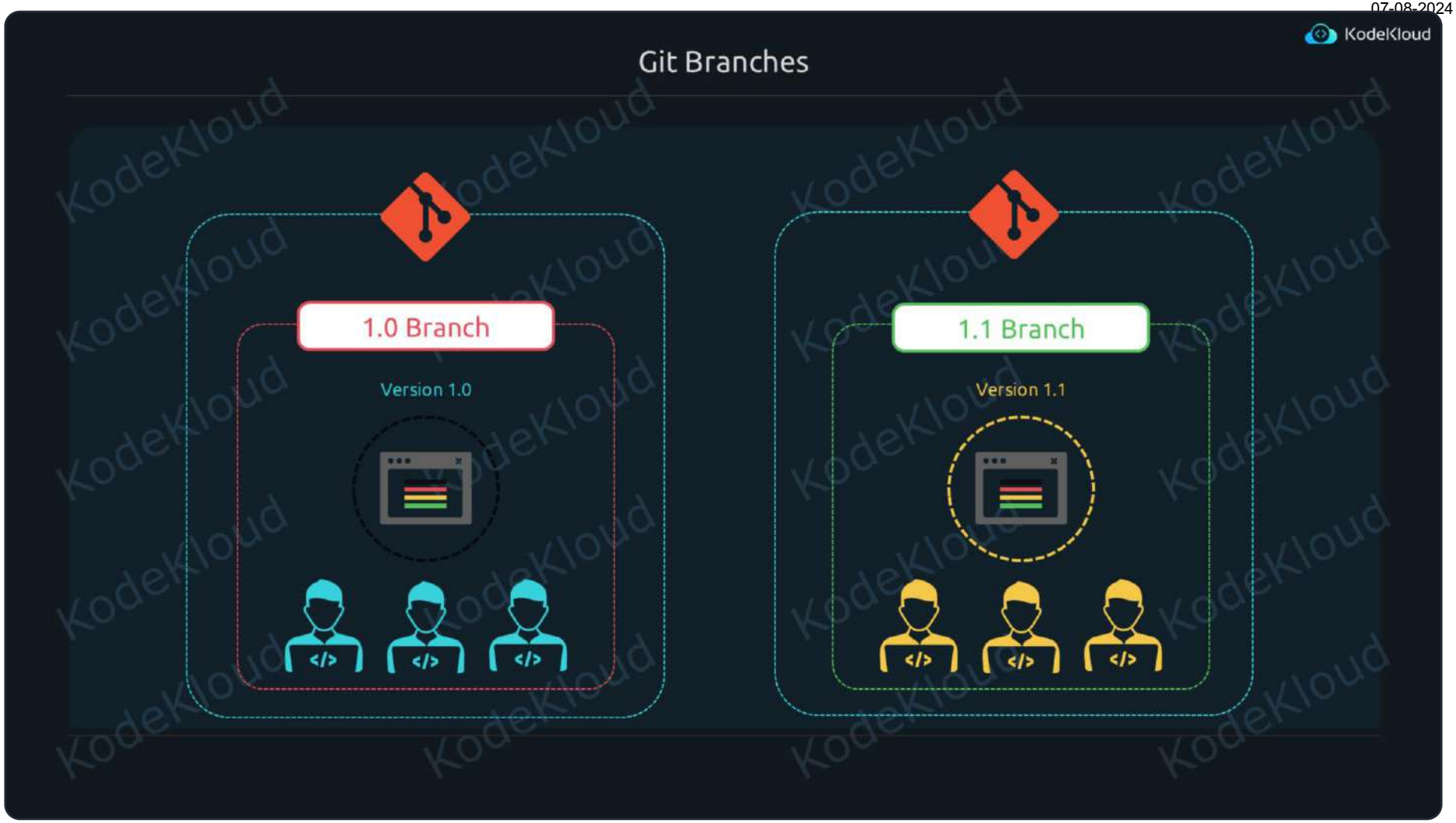
While checking git status and making commits, we can notice a line like this in the output:

"On branch master"



So, what are these branches? Well, here's one example. Our project can have multiple versions.

Maybe we shipped out version 1.0 to our users. And a part of the team is constantly fixing bugs in version 1.0. So they'll work on the 1.0 branch.

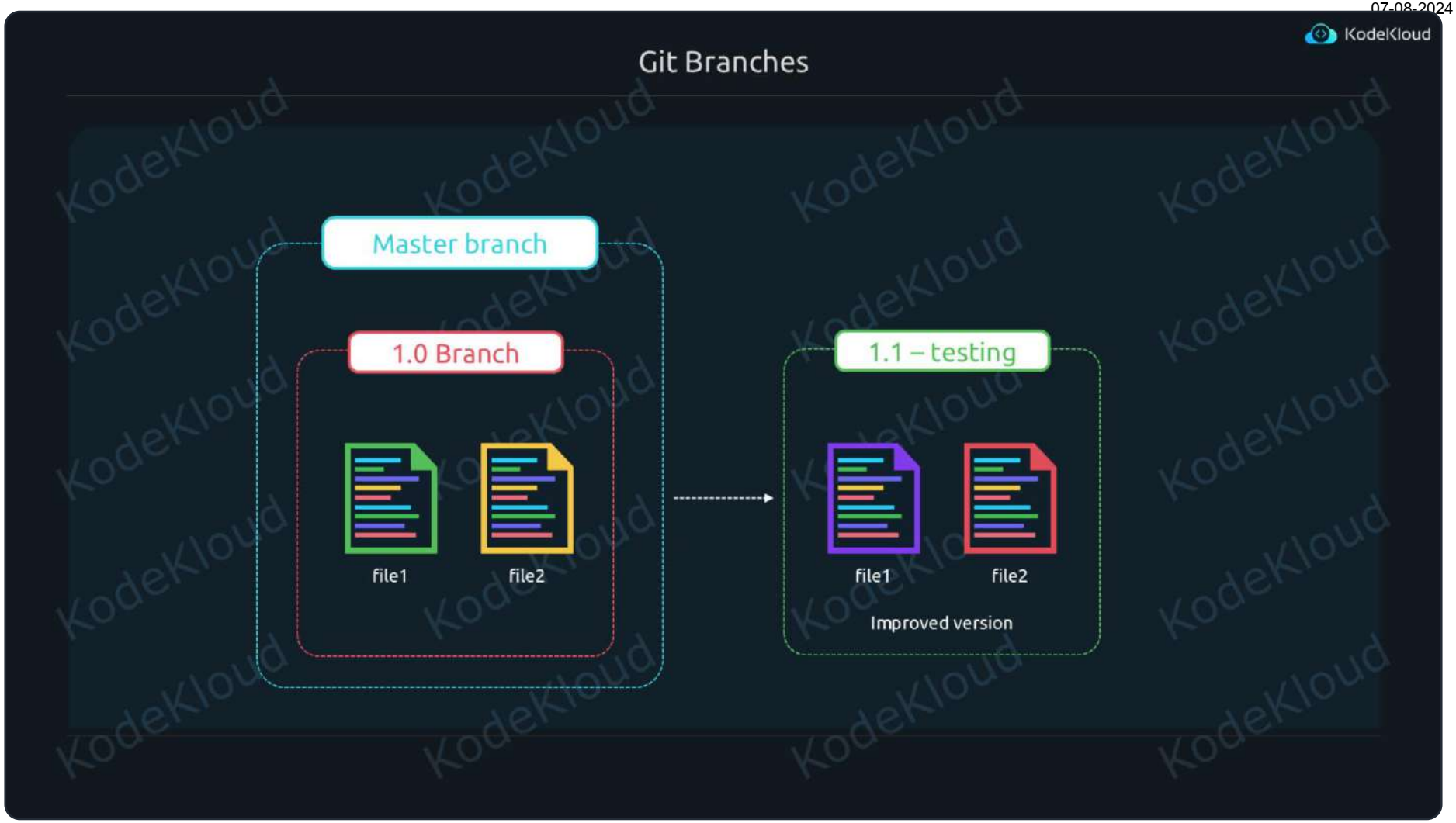


But at the same time, we also intend to build new features in our software. So another part of the team is also working on version 1.1, on an entirely different branch. This way people can work on two versions, two branches, without interfering with each other. Git will track each branch separately.

We can think of branches as different development roads. They can go in similar, but slightly different directions. And from time to time, they can be reunited, which is called merging in Git terminology. We'll see how that works later on.

The "master" branch is the default one Git works with. We might use this to track our stable version of our software project, the one that is actually shipped out to the public. It's up to us what we do with it.

The "master" branch is the default one Git works with. We might use this to track our stable version of our software project, the one that is actually shipped out to the public. It's up to us what we do with it.



Now let's imagine that our master branch tracks the 1.0 version. It contains file1 and file2. Things are looking good. But we want to prepare a new and improved version 1.1. So we start improving our code. We don't want to make changes directly to our master branch. Instead, we want to create a new branch. We won't even call the branch "1.1". Let's call it "1.1-testing". This tells our entire team that this 1.1 branch is not even final yet. We're just modifying lines of code and we intend to test the new software before releasing it to the public.

But seeing this in practice is easier to understand. So let's dive right into how to use Git branches at the command line.

KodeKloud

Visit www.kodekloud.com to discover more.