Linux, Shell Scripting & Git

Complete Study Guide for DevOps Engineers

Beginner to Advanced Level

Includes: Concepts | Real Examples | Interview Q&A | Practice Assignments | Projects

Manual Tester → DevOps Transition Path

TABLE OF CONTENTS

PART 1
LINUX FUNDAMENTALS

1. Linux File System Structure

1.1 Understanding the Linux File System

Linux follows a hierarchical directory structure starting from the root directory (/). Unlike Windows which has multiple drives (C:, D:, etc.), Linux has a single tree structure where everything is a file — including directories, devices, and processes.

Core Directories You Must Know

Directory	Purpose
/	Root directory — the top of the entire file system hierarchy
/home	Home directories for regular users (e.g., /home/john)
/root	Home directory for the root user (superuser)
/etc	System configuration files — all .conf files live here
/var	Variable data — logs, databases, email queues, website content
/tmp	Temporary files — cleared on reboot, any user can write here
/bin	Essential command binaries (ls, cp, mv, cat) — used by all users
/sbin	System binaries for system administration (used by root)
/usr	User programs and data — largest directory
/usr/bin	User command binaries
/usr/local	Locally installed software not managed by package manager
/lib	Shared libraries needed by programs in /bin and /sbin
/opt	Optional/third-party application software packages
/proc	Virtual filesystem containing process and system information
/dev	Device files (hard drives, USB devices, terminals)
/mnt	Temporary mount point for mounting file systems
/media	Mount points for removable media (USB drives, CD-ROMs)
/boot	Boot loader files (kernel, initramfs)

Real-Time Example

When you install nginx web server:

Binary goes to: /usr/sbin/nginx

Configuration goes to: /etc/nginx/nginx.conf

Logs go to: /var/log/nginx/access.log and error.log

Website content goes to: /var/www/html/

Process info visible at: /proc/<pid>/

📌 **NOTE:** In DevOps, 90% of your troubleshooting involves /var/log (logs), /etc (config), and /proc (process info).

1.2 Navigation & File Operations

Basic Navigation Commands

```
pwd                     # Print Working Directory — shows where you
are

ls                      # List files in current directory

ls -l                   # Long format with permissions, owner,
size, date

ls -la                  # Include hidden files (starting with .)

ls -lh                  # Human-readable file sizes (KB, MB, GB)

ls -ltr                 # Sort by time, newest at bottom (very
useful!)

cd /var/log             # Change to /var/log directory

cd ..                   # Go one level up

cd ~                    # Go to your home directory

cd -                    # Go to previous directory
```

File Operations

```
touch file.txt              # Create empty file

mkdir mydir                 # Create directory

mkdir -p /app/config/prod   # Create nested directories

cp file.txt /tmp/           # Copy file to /tmp

cp -r mydir /backup/        # Copy directory recursively

mv old.txt new.txt          # Rename file

mv file.txt /tmp/           # Move file to another location
```

```
rm file.txt                    # Delete file

rm -r mydir                    # Delete directory recursively

rm -rf /tmp/test               # Force delete without confirmation
(DANGEROUS!)
```

⚠ **WARNING:** rm -rf / will delete your ENTIRE system. Always double-check paths before using rm -rf.

Viewing File Contents

```
cat file.txt                   # Print entire file to terminal

less file.txt                  # View file page by page (q to quit,
/ to search)

head -20 file.txt              # Show first 20 lines

tail -20 file.txt              # Show last 20 lines

tail -f /var/log/syslog        # Follow log file in real-time
(Ctrl+C to stop)

wc -l file.txt                 # Count number of lines
```

Real-Time DevOps Use Case

Checking live application logs during deployment:

```
tail -f /var/log/myapp/application.log | grep ERROR
```

This shows only ERROR lines from the live log, helping you spot issues immediately.

1.3 File Permissions (CRITICAL for DevOps)

Every file in Linux has three types of permissions for three categories of users.

Permission Types

Permission	Numeric Value	Meaning
r (read)	4	View file contents or list directory contents
w (write)	2	Modify file or create/delete files in directory
x (execute)	1	Run file as program or enter directory

User Categories

Category	Description
Owner (u)	The user who owns the file

Group (g)	Users belonging to the file's group
Others (o)	Everyone else on the system

Reading Permissions

```
ls -l myfile.txt

Output: -rwxr-xr--  1 john developers 4096 Jan 15 10:30 myfile.txt
```

Breaking it down:

-rwxr-xr-- → File type and permissions

- = regular file (d = directory, l = symbolic link)

rwx = owner can read, write, execute

r-x = group can read and execute (no write)

r-- = others can only read

1 = number of hard links

john = owner

developers = group

4096 = file size in bytes

Jan 15 10:30 = last modification time

Changing Permissions

```
chmod 755 script.sh          # rwxr-xr-x  (owner: all, others:
read+exec)

chmod 644 config.txt         # rw-r--r--  (owner: read+write,
others: read)

chmod 600 secret.key         # rw-------  (only owner can
read/write)

chmod +x script.sh           # Add execute permission for all

chmod u+x,g-w file.txt       # Add execute for owner, remove write
for group
```

Changing Ownership

```
chown john file.txt          # Change owner to 'john'

chown john:developers file.txt  # Change owner and group

chown -R www-data:www-data /var/www/html  # Recursively change
ownership
```

Common Permission Patterns in DevOps

Permission	Use Case
755	Scripts and executables that should be runnable by everyone
644	Configuration files — readable by all, writable only by owner
600	SSH private keys, secrets — only owner can read/write
777	NEVER use this! Allows anyone to do anything (security risk)
700	Directories with sensitive data — only owner access

Real-Time Example

When deploying an application:

```
# Deploy script - must be executable
chmod 755 /opt/app/deploy.sh


# Config file - readable by app user, not writable by others
chmod 640 /etc/myapp/config.yml
chown appuser:appgroup /etc/myapp/config.yml


# Log directory - app must write logs
chmod 755 /var/log/myapp
chown appuser:appgroup /var/log/myapp
```

📌 **NOTE:** Wrong permissions are the #1 cause of 'Permission Denied' errors in DevOps deployments.

1.4 Process Management

A process is a running instance of a program. As a DevOps engineer, you will constantly check which processes are running, kill hung processes, and restart services.

Viewing Processes

```
ps                      # Show processes for current user in
current terminal
ps aux                  # Show ALL processes on the system
ps aux | grep nginx     # Find all nginx processes
```

```
top                     # Live view of processes sorted by CPU
(press q to quit)

htop                    # Better interactive process viewer (if
installed)

pgrep nginx             # Get process IDs of nginx

pidof nginx             # Another way to get PID
```

Understanding ps aux Output

```
USER  PID %CPU %MEM    VSZ   RSS TTY   STAT START   TIME COMMAND

root  1234  2.5  1.3 180000 52000 ?   Ss   10:30   0:05
/usr/bin/java -jar app.jar
```

USER = who owns the process

PID = Process ID (unique number)

%CPU = CPU usage percentage

%MEM = Memory usage percentage

VSZ = Virtual memory size

RSS = Resident Set Size (actual RAM used)

STAT = Process state (R=running, S=sleeping, Z=zombie, D=uninterruptible)

Killing Processes

```
kill 1234               # Gracefully terminate process (sends
SIGTERM)

kill -9 1234            # Force kill (sends SIGKILL) — use when
normal kill fails

killall nginx           # Kill all processes named nginx

pkill -f 'java.*myapp'  # Kill processes matching pattern
```

⚠ **WARNING:** kill -9 should be last resort. It doesn't give the process time to clean up properly.

Managing Services (systemd)

Modern Linux uses systemd to manage services. This is what you'll use 90% of the time in DevOps.

```
systemctl start nginx      # Start service

systemctl stop nginx       # Stop service
```

```
systemctl restart nginx    # Restart service
systemctl reload nginx     # Reload config without stopping
systemctl status nginx     # Check if service is running
systemctl enable nginx     # Auto-start service on boot
systemctl disable nginx    # Don't auto-start on boot
journalctl -u nginx -f     # Follow service logs in real-time
journalctl -u nginx --since '10 min ago'  # Recent logs
```

Real-Time Scenario

Application is slow — let's troubleshoot:

```
# 1. Check which process is using most CPU
top

# 2. Check memory usage
free -h

# 3. Check if app process is running
ps aux | grep myapp

# 4. Check app service status
systemctl status myapp

# 5. Check recent logs for errors
journalctl -u myapp --since '1 hour ago' | grep -i error
```

1.5 Disk & Memory Commands

Disk Usage

```
df -h                    # Show disk space usage (human-readable)
df -h /var               # Check specific partition
du -sh /var/log          # Size of /var/log directory
du -sh /var/log/*        # Size of each subdirectory
du -sh * | sort -rh      # Sort by size, largest first
```

Finding Large Files

```
# Find files larger than 100MB in /var
find /var -type f -size +100M -exec ls -lh {} \; | awk '{ print $9 ": " $5 }'

# Top 10 largest files in current directory
find . -type f -exec du -h {} \; | sort -rh | head -10
```

Memory Usage

```
free -h                  # Show RAM and swap usage
cat /proc/meminfo        # Detailed memory information
vmstat 1 5               # Virtual memory stats every 1 second, 5 times
```

Real-Time DevOps Example

Disk is 90% full — find and clean up:

```
# 1. Check which partition is full
df -h

# 2. Find large directories
du -sh /* 2>/dev/null | sort -rh | head -10

# 3. Usually it's logs — check log directory
du -sh /var/log/*

# 4. Rotate or delete old logs
```

```
find /var/log -name '*.log' -mtime +30 -delete  # Delete logs
older than 30 days
```

📌 **NOTE:** Always check df -h BEFORE and AFTER cleanup to verify you freed space.

1.6 User & Group Management

User Commands

```
whoami                    # Show current logged-in user

id                        # Show user ID (UID) and group IDs (GID)

who                       # Show all logged-in users

w                         # Show who is logged in and what they're
doing

last                      # Show login history
```

Creating Users

```
sudo useradd -m john      # Create user with home directory

sudo passwd john          # Set password for user

sudo useradd -m -s /bin/bash -G sudo,docker john  # Create user
with bash shell and groups

sudo userdel john         # Delete user

sudo userdel -r john      # Delete user and home directory
```

Managing Groups

```
groups                    # Show groups for current user

groups john               # Show groups for user 'john'

sudo groupadd developers  # Create new group

sudo usermod -aG sudo john  # Add user to sudo group (append,
don't replace)

sudo usermod -aG docker john  # Add user to docker group
```

Switching Users

```
su - john                # Switch to user john (enter their password)
sudo su -                # Switch to root user
sudo -i                  # Another way to become root
sudo -u john command     # Run command as user john
```

Real-Time DevOps Scenario

Setting up a deployment user:

```
# Create user for CI/CD deployments
sudo useradd -m -s /bin/bash -G sudo deploy
sudo passwd deploy

# Generate SSH key for the user
sudo -u deploy ssh-keygen -t rsa -b 4096 -C 'deploy@company.com'

# Copy public key to remote server
sudo -u deploy ssh-copy-id production-server

# Give deploy user permission to restart app service without password
echo 'deploy ALL=(ALL) NOPASSWD: /bin/systemctl restart myapp' | sudo tee /etc/sudoers.d/deploy
```

1.7 SSH (Secure Shell) — Key-Based Authentication

SSH is how you connect to remote servers. In DevOps, you use SSH dozens of times daily. Key-based authentication is more secure than passwords and allows automation.

Password-Based SSH (Basic)

```
ssh username@192.168.1.100     # Connect to remote server
ssh -p 2222 username@server    # Connect on non-standard port
```

```
exit                                # Disconnect
```

Key-Based Authentication (Industry Standard)

- **Step 1: Generate SSH Key Pair (One Time)**

```
ssh-keygen -t rsa -b 4096 -C 'your.email@company.com'

# Press Enter to save in default location: ~/.ssh/id_rsa

# Enter a passphrase (optional but recommended)
```

This creates two files:

~/.ssh/id_rsa — Your PRIVATE key (NEVER share this)

~/.ssh/id_rsa.pub — Your PUBLIC key (safe to share)

- **Step 2: Copy Public Key to Server**

```
ssh-copy-id username@server-ip

# OR manually:

cat ~/.ssh/id_rsa.pub | ssh username@server 'cat >> ~/.ssh/authorized_keys'
```

- **Step 3: Connect Without Password**

```
ssh username@server-ip
```

You'll be logged in without entering password!

Using Different Key Files

```
ssh -i ~/.ssh/my-custom-key.pem ec2-user@aws-server

# Common for AWS EC2 instances
```

SSH Config File (Pro Tip)

Create ~/.ssh/config to save connection details:

```
# ~/.ssh/config

Host prod-server

    HostName 192.168.1.100

    User deploy

    Port 22
```

```
    IdentityFile ~/.ssh/prod-key.pem

Host aws-web
    HostName ec2-54-123-45-67.compute.amazonaws.com
    User ubuntu
    IdentityFile ~/.ssh/aws-key.pem
```

Now you can simply type:

```
ssh prod-server
ssh aws-web
```

SCP — Secure Copy (File Transfer)

```
# Copy local file to remote server
scp file.txt user@server:/tmp/

# Copy remote file to local
scp user@server:/var/log/app.log ./

# Copy directory recursively
scp -r /local/dir user@server:/remote/path/

# Using specific key file
scp -i ~/.ssh/key.pem file.txt user@server:/tmp/
```

Real DevOps Workflow

Deploying application to production:

```
# 1. SSH into production server
ssh prod-server

# 2. Stop application
sudo systemctl stop myapp
```

```
# 3. From local machine, copy new build
scp target/myapp.jar prod-server:/opt/myapp/

# 4. Back on server, start application
ssh prod-server 'sudo systemctl start myapp'

# 5. Check logs
ssh prod-server 'sudo journalctl -u myapp -f'
```

📌 **NOTE:** In real DevOps, this manual process would be automated with CI/CD pipelines, but SSH is still the underlying mechanism.

1.8 Cron Jobs & Scheduling

Cron is the Linux task scheduler. It runs commands automatically at scheduled times. Used for: backups, log rotation, health checks, report generation, certificate renewal.

Cron Syntax

```
* * * * * command-to-run
| | | | |
| | | | └── Day of week (0-7, both 0 and 7 are Sunday)
| | | └──── Month (1-12)
| | └────── Day of month (1-31)
| └──────── Hour (0-23)
└────────── Minute (0-59)
```

Common Cron Patterns

Cron Expression	When it Runs
0 2 * * *	Every day at 2:00 AM
*/5 * * * *	Every 5 minutes
0 */6 * * *	Every 6 hours

0 0 * * 0	Every Sunday at midnight
30 9 1 * *	9:30 AM on 1st of every month
0 9-17 * * 1-5	Every hour from 9 AM to 5 PM, Monday to Friday
*/15 * * * *	Every 15 minutes
0 0 1 1 *	January 1st at midnight (yearly)

Managing Cron Jobs

```
crontab -e              # Edit cron jobs for current user
crontab -l              # List current user's cron jobs
crontab -r              # Remove all cron jobs (be careful!)
sudo crontab -u john -e # Edit cron for user 'john'
```

Example Cron Jobs

```
# Backup database daily at 2 AM
0 2 * * * /scripts/backup-db.sh

# Clean old logs every Sunday at 3 AM
0 3 * * 0 find /var/log -name '*.log' -mtime +30 -delete

# Health check every 5 minutes
*/5 * * * * /scripts/health-check.sh

# Send daily report at 6 PM on weekdays
0 18 * * 1-5 /scripts/send-report.sh
```

Cron Job Best Practices

Always use absolute paths — /usr/bin/python3 /home/user/script.py (not just python3 script.py)

Redirect output to log — Add >> /var/log/cron-job.log 2>&1 to save output

Set PATH in crontab — Add PATH=/usr/local/bin:/usr/bin:/bin at the top

Test script manually first — Make sure it works before adding to cron

Cron with Logging

```
# /var/spool/cron/crontabs/username
```

```
PATH=/usr/local/bin:/usr/bin:/bin


# Database backup daily
0 2 * * * /scripts/backup.sh >> /var/log/backup.log 2>&1
```

TIP: Use crontab.guru website to test and understand cron expressions — great learning tool!

1.9 Environment Variables

Environment variables store configuration values that programs can access. They're essential in DevOps for keeping secrets out of code and configuring applications for different environments.

Viewing Environment Variables

```
printenv                # Show all environment variables
env                     # Same as printenv
echo $HOME              # Print specific variable
echo $PATH              # Shows where system looks for commands
echo $USER              # Current username
```

Common Environment Variables

Variable	Example Value	Purpose
HOME	/home/username	User's home directory
PATH	/usr/bin:/bin:/usr/local/bin	Where to find commands
USER	john	Current username
SHELL	/bin/bash	Current shell
PWD	/var/log	Present working directory
LANG	en_US.UTF-8	System language
EDITOR	vim	Default text editor

Setting Environment Variables

```
# Temporary (current session only)
export MY_VAR='Hello DevOps'
echo $MY_VAR
```

```
# Permanent (for current user)
echo 'export DATABASE_URL=postgresql://localhost/mydb' >> ~/.bashrc
source ~/.bashrc        # Reload to apply

# System-wide (all users)
sudo echo 'export APP_ENV=production' >> /etc/environment
```

Real DevOps Use Cases

1. Application Configuration

```
# Instead of hardcoding database password in code:
export DB_HOST=localhost
export DB_PORT=5432
export DB_NAME=myapp
export DB_USER=appuser
export DB_PASSWORD=secretpassword

# Application reads these at runtime
```

2. Different Environments

```
# Development
export APP_ENV=development
export DEBUG=true

# Production
export APP_ENV=production
export DEBUG=false
```

3. PATH Modification (Add Custom Scripts)

```
export PATH=$PATH:/opt/myapp/bin
```

```
# Now scripts in /opt/myapp/bin are available everywhere
```

Environment Files

Best practice: Store all environment variables in a .env file:

```
# .env
DATABASE_URL=postgresql://user:pass@localhost/mydb
API_KEY=abc123xyz
REDIS_HOST=localhost:6379

# Load them in your script
set -a  # automatically export variables
source .env
set +a
```

⚠ **WARNING:** NEVER commit .env files with secrets to Git! Add .env to .gitignore.

PART 2
SHELL SCRIPTING

2. Shell Scripting Fundamentals

2.1 What is Shell Scripting?

A shell script is a text file containing a sequence of commands for a Unix-based operating system. Instead of typing commands one by one, you write them in a file and execute the file. Shell scripts automate repetitive tasks — exactly what DevOps is about.

Why Learn Shell Scripting for DevOps?

Automation — Deploy apps, backup databases, restart services automatically

CI/CD Pipelines — Most pipeline scripts are written in shell

Server Configuration — Automate server setup and configuration

Log Analysis — Parse and analyze log files

Health Checks — Monitor applications and alert on failures

Your First Shell Script

```
#!/bin/bash

# This is a comment

echo 'Hello, DevOps World!'
```

Save this as hello.sh, then:

```
chmod +x hello.sh      # Make executable

./hello.sh             # Run it
```

📌 **NOTE:** #!/bin/bash is called a shebang. It tells the system to use bash to execute this script.

2.2 Variables

Declaring and Using Variables

```
#!/bin/bash

# Variables (no spaces around =)
NAME='DevOps Engineer'
COUNT=42

# Using variables (use $)
echo "Hello, $NAME"
echo "Count: $COUNT"

# Command substitution
CURRENT_DATE=$(date +%Y-%m-%d)
echo "Today is: $CURRENT_DATE"

# User input
echo 'Enter your name:'
read USER_NAME
echo "Welcome, $USER_NAME!"
```

Special Variables

Variable	Meaning
$0	Name of the script
$1, $2, ...	First, second... command-line arguments
$#	Number of arguments
$@	All arguments as separate strings
$?	Exit status of last command (0 = success)
$$	Process ID of current script
$USER	Current username

$HOME	Home directory
$PWD	Current directory

Example with Arguments

```
#!/bin/bash
# Save as greet.sh

echo "Script name: $0"
echo "First argument: $1"
echo "Second argument: $2"
echo "Total arguments: $#"
echo "All arguments: $@"

# Run it:
./greet.sh John DevOps
# Output:
# Script name: ./greet.sh
# First argument: John
# Second argument: DevOps
# Total arguments: 2
# All arguments: John DevOps
```

2.3 Conditional Statements (if/else)

Basic if Statement

```
#!/bin/bash

AGE=25
```

```
if [ $AGE -ge 18 ]; then
    echo 'Adult'
else
    echo 'Minor'
fi
```

Comparison Operators

Operator	Meaning	Example
-eq	Equal to	if [$a -eq $b]
-ne	Not equal to	if [$a -ne $b]
-gt	Greater than	if [$a -gt $b]
-ge	Greater than or equal	if [$a -ge $b]
-lt	Less than	if [$a -lt $b]
-le	Less than or equal	if [$a -le $b]

String Comparisons

Operator	Meaning	Example
=	Strings are equal	if ["$str1" = "$str2"]
!=	Strings not equal	if ["$str1" != "$str2"]
-z	String is empty	if [-z "$str"]
-n	String is not empty	if [-n "$str"]

File Tests

Operator	Test	Example
-f	File exists and is regular file	if [-f /etc/passwd]
-d	Directory exists	if [-d /var/log]
-e	Path exists	if [-e /tmp/file]
-r	File is readable	if [-r file.txt]
-w	File is writable	if [-w file.txt]
-x	File is executable	if [-x script.sh]

if/elif/else Example

```
#!/bin/bash

DISK_USAGE=$(df / | tail -1 | awk '{print $5}' | tr -d '%')

if [ $DISK_USAGE -gt 90 ]; then
    echo 'CRITICAL: Disk usage is above 90%!'
    # Send alert
```

```
elif [ $DISK_USAGE -gt 70 ]; then
    echo 'WARNING: Disk usage is above 70%'
else
    echo 'Disk usage is normal: '$DISK_USAGE'%'
fi
```

Real DevOps Example: Check if Service is Running

```
#!/bin/bash

SERVICE='nginx'

if systemctl is-active --quiet $SERVICE; then
    echo "$SERVICE is running"
else
    echo "$SERVICE is DOWN! Attempting restart..."
    systemctl restart $SERVICE

    if systemctl is-active --quiet $SERVICE; then
        echo "$SERVICE restarted successfully"
    else
        echo "FAILED to restart $SERVICE"
        exit 1
    fi
fi
```

2.4 Loops

For Loop

```
#!/bin/bash

# Loop through list
for NAME in Alice Bob Charlie; do
    echo "Hello, $NAME"
done

# Loop through numbers
for i in {1..5}; do
    echo "Count: $i"
done

# C-style for loop
for ((i=1; i<=5; i++)); do
    echo "Number: $i"
done
```

While Loop

```
#!/bin/bash

COUNT=0
while [ $COUNT -lt 5 ]; do
    echo "Count is: $COUNT"
    COUNT=$((COUNT + 1))
done
```

Reading File Line by Line

```
#!/bin/bash

while IFS= read -r line; do
```

```
    echo "Line: $line"
done < /etc/passwd
```

Real DevOps Example: Check Multiple Servers

```
#!/bin/bash

SERVERS='web1 web2 web3 db1'

for SERVER in $SERVERS; do
    echo "Checking $SERVER..."

    if ping -c 1 $SERVER &> /dev/null; then
        echo "$SERVER is UP"
    else
        echo "$SERVER is DOWN!"
    fi
done
```

2.5 Functions

Functions make code reusable and organized. Essential for larger scripts.

Defining and Calling Functions

```
#!/bin/bash

# Define function
greet() {
    NAME=$1
    echo "Hello, $NAME!"
```

```
}

# Call function
greet 'DevOps'
greet 'Engineer'
```

Function with Return Value

```
#!/bin/bash

add_numbers() {
    local NUM1=$1
    local NUM2=$2
    local SUM=$((NUM1 + NUM2))
    echo $SUM
}

RESULT=$(add_numbers 5 3)
echo "5 + 3 = $RESULT"
```

Real DevOps Function: Service Health Check

```
#!/bin/bash

check_service() {
    SERVICE=$1

    if systemctl is-active --quiet $SERVICE; then
        echo "✓ $SERVICE is running"
        return 0
    else
        echo "✗ $SERVICE is DOWN"
```

```
        return 1
    fi
}

# Check multiple services
check_service nginx
check_service mysql
check_service redis
```

2.6 Text Processing (awk, sed, cut, grep)

DevOps engineers spend a lot of time parsing logs and text files. These tools are essential.

grep — Search for Patterns

```
# Basic search
grep 'error' /var/log/app.log

# Case-insensitive
grep -i 'error' /var/log/app.log

# Count matches
grep -c 'ERROR' /var/log/app.log

# Show line numbers
grep -n 'failed' /var/log/app.log

# Recursive search in directory
grep -r 'TODO' /home/user/project/
```

```
# Invert match (show lines NOT containing pattern)
grep -v 'DEBUG' /var/log/app.log

# Multiple patterns
grep -E 'error|ERROR|Error' /var/log/app.log
```

awk — Text Processing and Data Extraction

```
# Print specific column
ps aux | awk '{print $1, $11}'  # Print user and command

# Print lines where column value meets condition
ps aux | awk '$3 > 5.0'  # CPU usage > 5%

# Sum a column
cat sales.txt | awk '{sum += $2} END {print sum}'

# Print with custom delimiter
awk -F':' '{print $1, $6}' /etc/passwd  # Username and home dir
```

sed — Stream Editor (Find and Replace)

```
# Replace first occurrence
sed 's/old/new/' file.txt

# Replace all occurrences (global)
sed 's/old/new/g' file.txt

# Replace in file (in-place)
sed -i 's/DEBUG/INFO/g' config.txt
```

```
# Delete lines containing pattern
sed '/DEBUG/d' app.log

# Print specific line range
sed -n '10,20p' file.txt  # Lines 10-20
```

cut — Extract Columns

```
# Extract by delimiter
cut -d':' -f1 /etc/passwd  # First field (username)

# Extract specific columns from CSV
cut -d',' -f2,4 data.csv  # 2nd and 4th columns

# Extract characters
cut -c1-10 file.txt  # First 10 characters of each line
```

Real DevOps Example: Log Analysis

```
#!/bin/bash
# Count error types in application log

LOG_FILE='/var/log/myapp/app.log'

echo 'Top 10 error messages:'
grep -i 'error' $LOG_FILE \
  | awk -F'ERROR:' '{print $2}' \
  | sort | uniq -c | sort -rn | head -10

echo ''
echo 'Errors per hour today:'
grep $(date +%Y-%m-%d) $LOG_FILE \
```

```
  | grep -i 'error' \
  | awk '{print $2}' \
  | cut -d':' -f1 \
  | sort | uniq -c
```

2.7 Exit Codes

Exit codes tell you if a command succeeded or failed. 0 means success, anything else means failure. This is CRITICAL for automation and error handling.

Understanding Exit Codes

```
ls /tmp
echo $?      # Prints: 0 (success)

ls /nonexistent
echo $?      # Prints: 2 (error)
```

Using Exit Codes in Scripts

```
#!/bin/bash

# Good practice: exit with appropriate code
if [ ! -f /etc/app/config.yml ]; then
    echo 'ERROR: Config file not found'
    exit 1  # Exit with error
fi

# Do work
echo 'Processing...'

# Success
```

```
exit 0
```

set -e — Exit on Any Error

```
#!/bin/bash
set -e  # Script will exit immediately if any command fails

mkdir /opt/app
cp config.yml /opt/app/
systemctl restart myapp

echo 'Deployment successful'  # Only reached if all above succeed
```

⚠ **WARNING:** Always use set -e in production deployment scripts. If any step fails, the script stops instead of continuing with broken state.

Checking Command Success

```
#!/bin/bash

if cp important.txt /backup/; then
    echo 'Backup successful'
else
    echo 'Backup FAILED!'
    exit 1
fi
```

2.8 Real Deployment Script Example

Putting it all together — a real-world deployment script you might use in production.

```
#!/bin/bash
set -e  # Exit on any error
```

```
# Configuration
APP_NAME='myapp'
APP_DIR='/opt/myapp'
BACKUP_DIR='/backup'
BUILD_FILE='target/myapp.jar'

# Colors for output
GREEN='\033[0;32m'
RED='\033[0;31m'
NC='\033[0m'  # No Color

log_info() {
    echo -e "${GREEN}[INFO]${NC} $1"
}

log_error() {
    echo -e "${RED}[ERROR]${NC} $1"
}

# 1. Pre-deployment checks
log_info 'Running pre-deployment checks...'

if [ ! -f "$BUILD_FILE" ]; then
    log_error 'Build file not found!'
    exit 1
fi

if ! systemctl is-active --quiet $APP_NAME; then
    log_error 'Application is not running — cannot deploy'
```

```
    exit 1
fi

# 2. Backup current version
log_info 'Creating backup...'
TIMESTAMP=$(date +%Y%m%d_%H%M%S)
cp $APP_DIR/myapp.jar $BACKUP_DIR/myapp_$TIMESTAMP.jar
log_info "Backup created: $BACKUP_DIR/myapp_$TIMESTAMP.jar"

# 3. Stop application
log_info 'Stopping application...'
systemctl stop $APP_NAME
sleep 2

# 4. Deploy new version
log_info 'Deploying new version...'
cp $BUILD_FILE $APP_DIR/myapp.jar
chown appuser:appgroup $APP_DIR/myapp.jar
chmod 644 $APP_DIR/myapp.jar

# 5. Start application
log_info 'Starting application...'
systemctl start $APP_NAME
sleep 5

# 6. Health check
log_info 'Running health check...'
MAX_ATTEMPTS=10
ATTEMPT=0
```

```
while [ $ATTEMPT -lt $MAX_ATTEMPTS ]; do
    HTTP_CODE=$(curl -s -o /dev/null -w '%{http_code}' http://localhost:8080/health)

    if [ "$HTTP_CODE" = '200' ]; then
        log_info 'Application is healthy!'
        log_info 'Deployment SUCCESSFUL'
        exit 0
    fi

    ATTEMPT=$((ATTEMPT + 1))
    log_info "Health check attempt $ATTEMPT/$MAX_ATTEMPTS..."
    sleep 5
done

# 7. Rollback on failure
log_error 'Health check FAILED — rolling back...'
systemctl stop $APP_NAME
cp $BACKUP_DIR/myapp_$TIMESTAMP.jar $APP_DIR/myapp.jar
systemctl start $APP_NAME
log_error 'Deployment FAILED — rolled back to previous version'
exit 1
```

📌 **NOTE:** This script demonstrates real DevOps patterns: pre-checks, backup, graceful stop/start, health check, and automatic rollback.

PART 3
GIT & VERSION CONTROL

3. Git & Version Control

3.1 What is Version Control?

Version control tracks changes to files over time. It allows multiple people to work on the same codebase, keep history of all changes, revert to previous versions, and understand who changed what and when.

Why Git Matters for DevOps

Source Control — All code lives in Git

CI/CD Trigger — Git commits trigger automated pipelines

Infrastructure as Code — Terraform, Ansible configs are version-controlled in Git

Collaboration — Teams work on same codebase via branching and PRs

Audit Trail — Complete history of who changed what, when, and why

3.2 Core Git Workflow

Repository Basics

```
# Initialize new repository
git init

# Clone existing repository
```

```
git clone https://github.com/company/project.git
git clone https://dev.azure.com/org/project/ git/repo
```

The Basic Cycle (Learn This First)

```
# 1. Check status
git status

# 2. Add files to staging area
git add file.txt            # Add specific file
git add .                   # Add all changed files
git add src/               # Add all files in directory

# 3. Commit changes
git commit -m 'Add user authentication feature'

# 4. Push to remote
git push origin main

# 5. Get latest changes from remote
git pull origin main
```

Checking What Changed

```
git status                  # See what files changed
git diff                    # See exact changes (not staged)
git diff --staged           # See changes that are staged
git log                     # View commit history
git log --oneline           # Compact history
git log --oneline --graph   # Visual branch graph
```

Real DevOps Example: Fix Bug Workflow

```
# 1. You're on main branch, working on app
```

```
git status  # Everything clean

# 2. Production bug reported — make quick fix
vim src/app.py  # Fix the bug

# 3. Check what you changed
git diff

# 4. Stage and commit
git add src/app.py
git commit -m 'Fix: Resolve null pointer in user login'

# 5. Push to trigger CI/CD pipeline
git push origin main
```

3.3 Branching Strategies

Branches let you work on features/fixes without affecting the main code. In DevOps, proper branching is crucial for safe deployments.

Basic Branch Commands

```
git branch                      # List local branches
git branch -a                   # List all branches (local + remote)
git branch feature/new-api      # Create new branch
git checkout feature/new-api    # Switch to branch
git checkout -b bugfix/login    # Create and switch in one command
git branch -d feature/old       # Delete merged branch
git branch -D feature/old       # Force delete unmerged branch
```

GitFlow — Traditional Approach

GitFlow uses multiple long-lived branches:

Branch	Purpose
main (or master)	Production code — always deployable
develop	Integration branch — latest development changes
feature/*	New features — branch from develop, merge back via PR
hotfix/*	Urgent production fixes — branch from main
release/*	Release preparation — branch from develop when ready

```
# GitFlow Example

# Start new feature

git checkout develop

git checkout -b feature/user-dashboard

# ... work on feature ...

git add .

git commit -m 'Add user dashboard'

git push origin feature/user-dashboard

# Create Pull Request: feature/user-dashboard → develop
```

Trunk-Based Development — Modern DevOps Approach

All developers commit to main (trunk) frequently — at least once per day. Short-lived feature branches (1-2 days max). Used by high-performing DevOps teams (Google, Netflix, etc.).

```
# Trunk-Based Example

git checkout main

git pull origin main

git checkout -b add-logging

# ... make changes ...

git add .

git commit -m 'Add structured logging to API'

git push origin add-logging

# Create PR immediately, get reviewed, merge same day
```

Aspect	GitFlow	Trunk-Based
Long-lived branches	main, develop, release branches	Only main

Feature branch lifetime	Days to weeks	Hours to 1-2 days
Merge frequency	Weekly or less	Multiple times daily
Best for	Teams new to Git, regulated industries	Fast-moving DevOps teams, CI/CD
Release process	Merge develop to main when ready	Main is always production-ready

📌 **NOTE:** In interviews, know BOTH approaches. Many companies are transitioning from GitFlow to Trunk-Based.

3.4 Merging & Pull Requests

Merging Branches

```
# Merge feature branch into main
git checkout main
git pull origin main         # Get latest
git merge feature/new-api    # Merge feature into main
git push origin main         # Push merged code
```

Merge vs Rebase

Command	What it Does	History	Use Case
git merge	Creates merge commit	Preserves history	Good for feature branches
git rebase	Rewrites history	Linear history	Good for cleaning up before PR

```
# Rebase example (update feature branch with latest main)
git checkout feature/api
git rebase main            # Replay feature commits on top of main
git push origin feature/api --force  # Rewrite remote
```

⚠ **WARNING:** Never rebase commits that have been pushed to shared branches. Only rebase local commits or feature branches.

Pull Requests (PRs) / Merge Requests (MRs)

Pull Requests are how teams review code before merging. They're central to DevOps collaboration.

▸ **PR Workflow**

Create feature branch and make changes

Push branch to remote

Create Pull Request from feature → main

Team reviews code, adds comments

CI/CD pipeline runs tests automatically

Developer addresses feedback

Reviewer approves PR

PR is merged into main

Feature branch is deleted

Azure Repos Branch Policies

In Azure DevOps, you can enforce rules on branches (usually main):

Require minimum reviewers — At least 1-2 people must approve

Require linked work items — Every PR must link to a User Story/Task

Require passing builds — CI pipeline must succeed before merge

Require comment resolution — All review comments must be resolved

Limit merge types — Only squash or rebase merges allowed

```
# Example Azure DevOps PR workflow
# 1. Create branch and push
git checkout -b feature/add-caching
# ... make changes ...
git push origin feature/add-caching

# 2. In Azure DevOps UI: Create Pull Request
# - Set reviewers
# - Link to Work Item #1234
# - CI pipeline runs automatically

# 3. After approval, merge via UI (squash merge)
# 4. Delete feature branch
```

3.5 Git History & Undoing Mistakes

Viewing History

```
git log                             # Full history
git log --oneline                   # Compact view
git log --oneline --graph --all     # Visual tree of all branches
git log --author='John'             # Commits by specific author
git log --since='2 weeks ago'       # Recent commits
git log --grep='bug'                # Search commit messages
git log -- file.txt                 # History of specific file
```

Show Specific Commit

```
git show abc123                     # See changes in commit abc123
git show HEAD                       # See latest commit
git show HEAD~1                     # Previous commit
git show HEAD~3                     # 3 commits ago
```

Undoing Changes — Different Scenarios

▸ **Scenario 1: Undo Unstaged Changes**

```
# Modified file but didn't add to staging yet
git restore file.txt                # Discard changes to file
git restore .                       # Discard all changes
```

▸ **Scenario 2: Undo Staged Changes**

```
# Already did git add, want to unstage
git restore --staged file.txt       # Unstage file
# File is still modified, just not staged anymore
```

▸ **Scenario 3: Undo Last Commit (Not Pushed)**

```
# Committed but didn't push yet, want to undo

git reset --soft HEAD~1           # Undo commit, keep changes
staged

git reset HEAD~1                  # Undo commit, keep changes
unstaged

git reset --hard HEAD~1           # Undo commit, delete changes
(DANGEROUS!)
```

- **Scenario 4: Undo Pushed Commit (Safe Way)**

```
# Already pushed to remote — use revert (doesn't rewrite history)

git revert abc123                 # Create new commit that undoes
abc123

git push origin main              # Push the revert commit
```

Real DevOps Scenario: Production Hotfix

```
# Oh no! Bad commit was pushed to production


# 1. Check recent commits

git log --oneline -5

# Output:

# abc123 (HEAD -> main) Add new feature

# def456 Fix authentication

# ghi789 Update dependencies  ← This broke production


# 2. Revert the bad commit

git revert ghi789

git push origin main


# Production is fixed! Bad commit is undone without rewriting
history
```

⚠ **WARNING:** In production environments, NEVER use git reset --hard or git push --force. Always use git revert.

3.6 Git Best Practices for DevOps

Commit Message Guidelines

Be descriptive — Not 'fix bug' but 'Fix: Resolve null pointer in payment gateway'

Use imperative mood — 'Add feature' not 'Added feature'

First line under 50 chars — Short summary, detailed explanation in body

Reference issues/tickets — 'Closes #123' or 'Fixes #456'

▸ **Good vs Bad Commit Messages**

Bad Example	Good Example
✗ fix stuff	✓ Fix: Resolve database connection timeout
✗ update	✓ Update: Increase API rate limit to 1000 req/min
✗ wip	✓ WIP: Implement user authentication (partial)
✗ bug fix	✓ Fix: Prevent race condition in order processing

.gitignore — Never Commit These

```
# .gitignore

.env                # Environment variables with secrets

*.log               # Log files

node_modules/       # Dependencies (re-download via package
manager)

target/             # Build artifacts (Java)

dist/               # Build artifacts (JavaScript)

*.pyc               # Python compiled files

.DS_Store           # macOS system files

*.swp               # Vim swap files

.idea/              # IDE settings

*.pem               # SSH private keys
```

⚠ **WARNING:** NEVER commit secrets (passwords, API keys, SSH keys) to Git! Use .gitignore and environment variables.

Daily DevOps Git Workflow Checklist

Start day: git pull origin main

Create feature branch: git checkout -b feature/name

Make changes, test locally

Commit frequently with good messages

Before pushing: git pull --rebase origin main (stay updated)

Push: git push origin feature/name

Create Pull Request

After merge: delete feature branch, switch to main, git pull

PART 4
INTERVIEW QUESTIONS & ANSWERS

4. Interview Questions with Detailed Answers

4.1 Linux Interview Questions

[BASIC] What is the difference between absolute path and relative path?

Answer: Absolute path starts from root directory (/) and specifies complete path (e.g., /home/user/file.txt). Relative path is relative to current directory (e.g., ../docs/file.txt). Use pwd to see where you are, then use relative paths for convenience or absolute paths for scripts to avoid ambiguity.

[BASIC] What is the purpose of chmod command?

Answer: chmod changes file permissions. In Linux, every file has read (r), write (w), and execute (x) permissions for owner, group, and others. Example: chmod 755 script.sh gives owner full permissions (rwx=7), group and others read+execute (r-x=5). This is critical in DevOps for deployment scripts and config files.

[INTERMEDIATE] Explain hard link vs soft link (symbolic link).

Answer: Hard link is another name for the same file — both point to same inode. If original is deleted, hard link still works. Soft link (symlink) is a pointer to the filename/path. If original is deleted, symlink breaks. Create with 'ln target hardlink' vs 'ln -s target symlink'. In DevOps, symlinks are commonly used for pointing to current version of deployed app (e.g., /opt/app/current → /opt/app/v1.2.3).

[INTERMEDIATE] How do you find all files modified in the last 7 days?

Answer: Use find command: 'find /var/log -type f -mtime -7'. -mtime -7 means modified within last 7 days. -mtime +7 would mean older than 7 days. Add -delete to remove them, or -exec to run a command on each. Real use case: cleaning old log files or finding recently changed config files during troubleshooting.

[ADVANCED] A process is using 100% CPU. How do you investigate and fix it?

Answer: 1) Use 'top' or 'htop' to identify which process (PID and command). 2) Check process details: ps aux | grep <PID>. 3) Check what files it's accessing: lsof -

p <PID>. 4) Check if it's stuck: strace -p <PID>. 5) If it's a runaway process: kill -15 <PID> (graceful), then kill -9 <PID> if needed (force). 6) Check application logs to understand why it happened. 7) If it's production service, restart it: systemctl restart service. 8) Investigate root cause to prevent recurrence.

[ADVANCED] What are inodes? What happens when you run out of inodes?

Answer: Inode is a data structure that stores file metadata (permissions, owner, timestamps, location of data blocks). Every file has an inode. You can run out of inodes even with disk space available — happens when you have millions of small files. Check with 'df -i'. When inodes are full, you cannot create new files even if disk has space. Solution: delete old files to free inodes, or increase inode count (requires recreating filesystem).

SCENARIO: Server Suddenly Stopped Responding

You SSH into a production Linux server and it's extremely slow. Users are complaining the application is down. What steps do you take?

Expected Approach: 1) Check system load: 'uptime' or 'top'. If load average is very high (>4 on 4-core machine), system is overloaded. 2) Check CPU: 'top' — identify which process is eating CPU. 3) Check memory: 'free -h' — if swap is heavily used, memory exhausted. 4) Check disk: 'df -h' — if 100% full, that's the culprit. 5) Check I/O: 'iostat 1 5' — see if disk is bottleneck. 6) If application process is hung, restart it: 'systemctl restart myapp'. 7) Check logs immediately: 'tail -100 /var/log/myapp/error.log'. 8) If out of memory, kill non-critical processes or reboot if necessary. 9) Document timeline and root cause for post-mortem.

SCENARIO: Accidental File Deletion

You accidentally deleted an important configuration file with 'rm -rf'. The file is not in any backup. How do you recover it?

Expected Approach: 1) STOP writing to disk immediately — unmount the filesystem if possible to prevent overwriting. 2) Use data recovery tools like 'extundelete' (for ext3/ext4) or 'photorec'. 3) If file was recently deleted and disk not heavily written, recovery chances are good. 4) Prevention is better: always use 'rm -i' (interactive) for critical operations, or create a 'trash' function that moves files to .trash instead of deleting. 5) In production, never work directly on primary configs — copy to backup first. 6) Better solution: keep all configs in Git so you can restore from version control.

4.2 Shell Scripting Interview Questions

[BASIC] What is the purpose of shebang (#!/bin/bash)?

Answer: Shebang tells the operating system which interpreter to use for the script. #!/bin/bash means use bash shell. Without it, the script might run with sh (basic shell) which doesn't support all bash features like arrays and [[ ]]. Always include shebang as the first line of every script.

[BASIC] What is the difference between $1, $@, and $#?

Answer: $1 is the first command-line argument passed to the script. $@ represents all arguments as separate strings. $# is the count of arguments. Example: './script.sh a b c' → $1=a, $@='a b c', $#=3. Use $@ in for loops to process all arguments, use $# to validate minimum arguments required.

[INTERMEDIATE] Write a script to check if a file exists, and create it if it doesn't.

Answer: #!/bin/bash

```
FILE='/opt/app/config.yml'

if [ ! -f "$FILE" ]; then

   echo 'File does not exist, creating...'

   touch "$FILE"

   echo 'File created'

else

   echo 'File already exists'

    fi
```

[INTERMEDIATE] Explain the difference between single quotes and double quotes in bash.

Answer: Single quotes preserve literal value of all characters — variables are NOT expanded. Double quotes allow variable expansion and command substitution. Example: NAME='DevOps'; echo '$NAME' outputs $NAME (literal). echo "$NAME" outputs DevOps (expanded). Use double quotes for variables, single quotes for literal strings.

[ADVANCED] How do you debug a shell script that's failing?

Answer: 1) Add 'set -x' at the top to enable debug mode (prints each command before execution). 2) Use 'set -e' to exit on first error. 3) Add echo statements to print variable values at key points. 4) Run script with 'bash -x script.sh' for debug output. 5) Check exit codes: 'echo $?' after each command. 6) Use shellcheck tool to find

syntax errors and best practice violations before running. 7) Test script in parts — comment out sections and run incrementally.

[ADVANCED] Write a script to monitor disk usage and send alert when it exceeds 80%.

Answer: #!/bin/bash

```
THRESHOLD=80

EMAIL='admin@company.com'

DISK_USAGE=$(df / | tail -1 | awk '{print $5}' | tr -d '%')

if [ $DISK_USAGE -gt $THRESHOLD ]; then

    MESSAGE="WARNING: Disk usage on $(hostname) is ${DISK_USAGE}%"

    echo "$MESSAGE" | mail -s 'Disk Space Alert' $EMAIL

    logger "$MESSAGE"

fi

    # Add this to crontab: */15 * * * * /scripts/disk-monitor.sh
```

SCENARIO: Deployment Script Failed Midway

Your deployment script ran halfway, then failed. The application is now in a broken state — half old code, half new code. How do you handle this?

Expected Approach: This is why we use 'set -e' and implement rollback. Immediate action: 1) Assess state: check which files were changed. 2) If backup exists (which it should!), restore immediately: 'cp /backup/app_20250201_1430/* /opt/app/'. 3) Restart application: 'systemctl restart myapp'. 4) Verify health: 'curl http://localhost:8080/health'. Long-term fix: Improve deployment script with: atomic deployments (deploy to new directory, then symlink switch), pre-deployment backup, health check after deployment, automatic rollback on failure. Never deploy without these safeguards.

4.3 Git Interview Questions

[BASIC] What is the difference between git pull and git fetch?

Answer: git fetch downloads changes from remote but does NOT merge them into your current branch. git pull = git fetch + git merge (downloads and merges automatically). Use git fetch when you want to review changes before merging. Use git pull for quick updates when you trust the remote.

[BASIC] What is a merge conflict and how do you resolve it?

Answer: Merge conflict occurs when two branches modify the same line of code differently. Git cannot auto-merge and asks you to resolve manually. You'll see conflict markers in the file:

<<<<<<< HEAD

your changes

=======

their changes

>>>>>>> branch-name

Resolve by: 1) Edit file to keep correct version. 2) Remove conflict markers. 3) git add file. 4) git commit. In teams, communicate with the other developer to decide which change is correct.

[INTERMEDIATE] Explain the difference between git merge and git rebase.

Answer: git merge combines two branches by creating a merge commit. It preserves full history including branching. git rebase rewrites history by moving your commits to tip of another branch. Result is linear history. Use merge for integrating feature branches (preserves context). Use rebase to clean up local commits before pushing, or to update feature branch with latest main. NEVER rebase commits that are already pushed to shared branches — it rewrites history and breaks others' repos.

[INTERMEDIATE] How do you undo a commit that was already pushed to remote?

Answer: Use git revert (safe way): git revert <commit-hash> creates a NEW commit that undoes the changes. Then git push. This doesn't rewrite history. NEVER use git reset --hard and git push --force on shared branches — this deletes commits and causes major problems for team. Only exception: if you're the only one using the branch and you pushed seconds ago.

[ADVANCED] Explain GitFlow vs Trunk-Based Development. Which is better for

DevOps?

Answer: GitFlow uses multiple long-lived branches (main, develop, feature, hotfix, release). Good for teams new to Git or regulated industries with formal release processes. Trunk-Based Development has one main branch. Developers create short-lived feature branches (hours to 1 day), merge quickly. Requires good CI/CD and feature flags. Better for DevOps because: faster integration, simpler branching model, encourages small frequent commits, enables continuous deployment. Companies like Google, Facebook, Netflix use Trunk-Based.

[ADVANCED] In Azure Repos, how do you enforce code quality before merge?

Answer: Use Branch Policies on main branch: 1) Require minimum reviewers (1-2 people must approve PR). 2) Require successful build (CI pipeline must pass). 3) Require linked work items (every PR links to User Story/Task for traceability). 4) Require comment resolution (all review comments addressed). 5) Limit merge types (squash merge keeps history clean). Set these in Repos → Branches → main → Branch Policies. This ensures NO code reaches main without review and passing tests.

SCENARIO: Production Broken After Merge

Someone merged a Pull Request and production immediately broke. The merge was 30 minutes ago and multiple other commits have been pushed since then. How do you fix it?

Expected Approach: Don't panic. 1) Identify the bad commit: 'git log --oneline -10' and check timestamps against when production broke. 2) Verify it's the cause: 'git show <commit-hash>' to see changes. 3) Revert it: 'git revert <commit-hash>'. 4) Push immediately: 'git push origin main'. 5) CI/CD pipeline will deploy the revert and restore production. 6) Total downtime: <5 minutes if you act fast. 7) After production is restored, create post-mortem: why didn't tests catch this? Why did code review miss it? Update CI/CD and review process to prevent recurrence. DO NOT git reset — that would delete all commits after the bad one.

PART 5
HANDS-ON PRACTICE & PROJECTS

5. Practice Assignments & Projects

5.1 Linux Practice Assignments

EASY Level

Create a directory structure: /tmp/practice/level1/level2/level3 in one command

Create 5 empty files in /tmp with names file1.txt to file5.txt using a loop or range

Find all .log files in /var/log and count how many exist

Create a user named 'testuser' with home directory, then delete it

Change permissions of a file to rw-r--r-- (owner read/write, others read-only)

MEDIUM Level

Write a script that takes a directory path as argument and shows total size of all files

Find all files larger than 10MB in your home directory and save the list to a file

Create a cron job that appends current date and disk usage to a log file every hour

Set up SSH key-based authentication between two Linux machines

Monitor a log file in real-time and print only lines containing 'ERROR'

DIFFICULT Level

Write a script that monitors CPU usage every 5 seconds, and if it exceeds 80% for 3 consecutive checks, send an alert

Create a script that finds all processes using more than 500MB RAM and gives option to kill them

Implement a simple backup script that creates timestamped tar.gz archives of a directory, keeps only last 5 backups, and logs all operations

Set up log rotation for an application: keep last 7 days of logs, compress old logs, and restart application after rotation

Create a user management script that can add users, set passwords, assign to groups, and set up their SSH keys — all from a CSV input file

5.2 Shell Scripting Practice Assignments

EASY Level

Write a script that greets user based on time of day (Good Morning/Afternoon/Evening)

Create a script that accepts filename as argument and displays number of lines, words, and characters

Write a calculator script that takes two numbers and an operator (+, -, *, /) and shows result

Create a script that checks if a given year is a leap year

Write a script that reverses a string entered by the user

MEDIUM Level

Write a script that checks if multiple services (nginx, mysql, redis) are running, and starts any that are down

Create a log analyzer script that counts occurrences of each HTTP status code in Apache/Nginx access.log

Write a script that monitors a directory and alerts when new files are created

Create a backup script with options: full backup, incremental backup, and verify backup integrity

Write a script that parses CSV file and generates HTML report

DIFFICULT Level

Create a deployment script that: validates prerequisites, backs up current version, deploys new version, runs health checks, rolls back on failure, and sends email report

Write a log rotation script that: archives logs older than 7 days, compresses them, uploads to S3/Azure Blob, and deletes local archives older than 30 days

Create a multi-server health check script that: SSHs into list of servers, checks service status, disk usage, memory, CPU, and generates consolidated HTML report

Write a database backup script with encryption: dumps database, encrypts with GPG, uploads to cloud, keeps 7 daily + 4 weekly + 12 monthly backups

Implement a zero-downtime deployment script using blue-green strategy: deploy to inactive slot, run tests, switch traffic, keep old version for rollback

5.3 Git Practice Assignments

EASY Level

Create a Git repository, make 5 commits with meaningful messages, view history with --oneline --graph

Create a .gitignore file for a Python project (ignore .pyc, venv, .env)

Clone a GitHub repository, make a change, and push it back

Create a branch, make changes, merge it back to main

View difference between two commits using git diff

MEDIUM Level

Simulate a merge conflict: create two branches changing the same line, merge them, and resolve the conflict

Rebase a feature branch onto updated main branch

Create a repository with GitFlow branches (main, develop, feature), implement a feature following the workflow

Configure Azure Repos with branch policies: require 2 reviewers, require passing build, require linked work items

Accidentally committed a secret file — remove it from history using git filter-branch or BFG Repo-Cleaner

DIFFICULT Level

Set up a Git workflow: main + staging + multiple feature branches, simulate team of 3 developers working simultaneously, handle merge conflicts, create PRs

Implement pre-commit hooks that: check for secrets, run linters, validate commit message format, prevent commits to main

Create a monorepo with multiple projects, set up branch strategy where each project can release independently

Migrate a project from GitFlow to Trunk-Based Development: document the process, create branching guidelines, set up feature flags

Set up Git LFS (Large File Storage) for a repository with large binary files,

configure .gitattributes correctly

5.4 Mini Project Ideas

Project 1: Complete CI/CD Setup (Linux + Shell + Git)

Goal — Set up automated deployment for a simple web application

Steps:

Create a Git repository with a simple web app (HTML + nginx or Node.js)

Write deployment shell script: stop service → backup → copy new code → start service → health check → rollback on failure

Set up Git hooks: pre-commit runs linter, post-receive triggers deployment

Configure cron job for automatic log cleanup and health monitoring

Document entire process with README

Project 2: Log Aggregation System

Goal — Centralize logs from multiple servers

Steps:

Set up 3 Linux VMs (or containers)

Write script on each server that ships logs to central server via rsync or scp

On central server, write log analyzer: count errors, group by severity, generate daily HTML report

Schedule with cron to run hourly

Create dashboard script showing real-time stats from all servers

Project 3: Infrastructure as Code Practice

Goal — Automate server setup from scratch

Steps:

Write shell script that sets up a complete server: install packages, create users, configure firewall, set up nginx, deploy SSL certificate, configure logging

Make script idempotent (can run multiple times safely)

Store all config files in Git repository

Create separate scripts for different server types (web, database, cache)

Test by destroying and recreating server multiple times

Project 4: Monitoring & Alerting System

Goal — Monitor applications and send alerts

Steps:

Write monitoring script: check CPU, memory, disk, service status, HTTP endpoint health

Implement alert levels: INFO, WARNING, CRITICAL

Send alerts via: log file, email, Slack webhook

Create history tracking: store metrics in CSV/database

Generate daily/weekly reports with graphs (using gnuplot or similar)

Run monitoring as systemd service, not cron

PART 6
COMMON MISTAKES & TROUBLESHOOTING

6. Common Mistakes & How to Avoid Them

6.1 Linux Common Mistakes

Mistake 1: Wrong Permissions

Problem: Script fails with 'Permission denied' or service cannot write to log file.

Solution:

Scripts must be executable: chmod +x script.sh

Check ownership: chown appuser:appgroup /opt/app

Log directories: chmod 755 /var/log/myapp, chown appuser:appgroup

Config files: chmod 640 (readable by app, not world-readable if contains secrets)

💡 **TIP:** Always test permissions as the actual user the service runs as, not as root.

Mistake 2: Using rm -rf Without Checking Path

Problem: Accidentally delete entire system or critical data.

Prevention:

ALWAYS echo the variable before using in rm: echo $DIR then rm -rf $DIR

Use rm -i for interactive confirmation

Create a trash function: alias rm='mv --backup=numbered --target-directory ~/.trash'

In scripts, validate path exists and is correct before deletion

⚠ **WARNING:** If $DIR is empty or undefined, 'rm -rf $DIR/' becomes 'rm -rf /' — disaster!

Mistake 3: Not Using Absolute Paths in Cron

Problem: Cron job script works manually but fails in cron.

Cause: Cron has minimal PATH, doesn't know where to find commands.

Solution:

```
# Bad cron script
python script.py  # Won't work — where is python?

# Good cron script
#!/bin/bash
PATH=/usr/local/bin:/usr/bin:/bin
/usr/bin/python3 /home/user/scripts/script.py
```

Mistake 4: Not Checking Disk Space

Problem: Application fails mysteriously — actually disk is 100% full.

Prevention:

Monitor disk daily: add 'df -h' to daily check script

Set up alerts when usage >80%

Implement log rotation (logrotate)

Before deployment, check: df -h

6.2 Shell Scripting Common Mistakes

Mistake 1: Not Using set -e

Problem: Script continues after commands fail, leaving system in broken state.

Solution: Always add set -e at the top of production scripts.

```
#!/bin/bash
set -e  # Exit immediately if any command fails
```

```
# Now if any command fails, script stops — safe!
```

Mistake 2: Spaces Around = in Variable Assignment

Problem: NAME = 'John' gives syntax error.

Correct: NAME='John' (no spaces)

```
# Wrong
NAME = 'DevOps'  # Error!

# Correct
NAME='DevOps'
```

Mistake 3: Not Quoting Variables

Problem: Variables with spaces break commands.

```
# Problem
FILE='my file.txt'
rm $FILE  # Tries to delete 'my' and 'file.txt' separately!

# Solution — ALWAYS quote variables
rm "$FILE"  # Deletes 'my file.txt' correctly
```

Mistake 4: Using rm -rf with Unvalidated User Input

Problem: User enters malicious input, deletes critical files.

```
# DANGEROUS — Never do this
echo 'Enter directory to delete:'
read DIR
rm -rf $DIR  # What if user enters '/' ?

# Safe version
echo 'Enter directory to delete (inside /tmp):'
read DIR
# Validate input
```

```
if [[ "$DIR" == /tmp/* ]] && [ -d "$DIR" ]; then
    echo "Deleting $DIR"
    rm -rf "$DIR"
else
    echo 'Invalid directory'
    exit 1
fi
```

6.3 Git Common Mistakes

Mistake 1: Committing Secrets to Git

Problem: Passwords, API keys committed to repository.

Prevention:

- Use .gitignore for .env, secrets.yml, *.pem
- Use environment variables instead of hardcoded secrets
- Scan commits before push: use tools like git-secrets or truffleHog

If you accidentally committed secrets:

```
# Remove from history (complex, use BFG Repo-Cleaner)
# Then ROTATE the exposed secret immediately!
```

Mistake 2: Force Pushing to Shared Branches

Problem: git push --force on main rewrites history, breaks everyone's repo.

Rule: NEVER force push to main, develop, or any shared branch.

When force push IS okay: your personal feature branch that only you use.

```
# Wrong
git push --force origin main  # DESTROYS team's work

# Right
git push --force origin feature/my-branch  # OK if only you use it
```

Mistake 3: Huge Commits with Message 'misc changes'

Problem: Impossible to review, impossible to revert specific changes.

Solution: Make small, logical commits with descriptive messages.

```
# Bad: One huge commit
git add .
git commit -m 'misc changes'  # 50 files changed

# Good: Small logical commits
git add src/auth.py
git commit -m 'Add password reset functionality'
git add src/email.py
git commit -m 'Add email notification for password reset'
git add tests/test_auth.py
git commit -m 'Add tests for password reset'
```

Mistake 4: Not Pulling Before Pushing

Problem: Your push fails because remote has new commits.

Solution: Always pull before pushing.

```
# Daily workflow
git pull origin main
# ... make changes ...
git add .
git commit -m 'message'
git pull --rebase origin main  # Get latest changes
git push origin main
```

6.4 Troubleshooting Flowcharts

When a Script Fails

Check exit code: echo $? — if not 0, it failed

Run with debug: bash -x script.sh

Check permissions: ls -l script.sh — is it executable?

Check paths: are all paths absolute? Is PATH set correctly?

Check variables: add echo statements to print variable values

Check logs: stderr output often shows the real error

Test commands individually: run each line of script manually

When Git Command Fails

Read error message carefully — Git is usually helpful

Check current branch: git branch

Check status: git status — often shows the problem

Check remote: git remote -v — is URL correct?

Check network: can you ping the Git server?

Check permissions: do you have SSH key set up? Is it added to agent?

Check disk space: df -h — out of space causes weird Git errors

PART 7
QUICK REVISION GUIDE

7. Quick Revision Cheat Sheet

7.1 Linux Commands Cheat Sheet

Must-Know Commands (Memorize These)

Command	Purpose
ls -la	List all files with permissions
cd /path	Change directory
pwd	Print working directory
chmod 755	Change permissions (rwxr-xr-x)
chown user:group	Change owner
ps aux	List all processes
kill -9 PID	Force kill process
systemctl status service	Check service status
df -h	Disk usage
du -sh dir	Directory size
free -h	Memory usage
tail -f log	Follow log file
grep 'pattern' file	Search in file
find / -name file	Find file
ssh user@server	Connect to server

7.2 Shell Scripting Quick Reference

Syntax	Meaning
#!/bin/bash	Shebang — first line
VAR='value'	Set variable (no spaces!)
$VAR or ${VAR}	Use variable
$1, $2, $@, $#	Script arguments
$?	Exit code of last command
if [cond]; then ... fi	If statement
for i in list; do ... done	For loop
while [cond]; do ... done	While loop
function name() { ... }	Define function
set -e	Exit on any error

set -x	Debug mode

7.3 Git Commands Cheat Sheet

Command	Purpose
git init	Create new repository
git clone URL	Clone repository
git status	Check status
git add file	Stage file
git commit -m 'msg'	Commit changes
git push origin main	Push to remote
git pull origin main	Get latest changes
git branch	List branches
git checkout -b branch	Create & switch branch
git merge branch	Merge branch
git log --oneline	View history
git diff	See changes
git revert hash	Undo commit (safe)

7.4 File Permissions Quick Reference

Number	Permissions	Common Use
777	rwxrwxrwx	Everyone can do everything (NEVER use!)
755	rwxr-xr-x	Scripts and executables
644	rw-r--r--	Config files, readable by all
600	rw-------	SSH keys, secrets, only owner access
700	rwx------	Directories with sensitive data

7.5 Pre-Interview Checklist

Review these the night before your interview:

- Explain Linux file system structure from memory
- Write a shell script with if/else and for loop
- Explain git workflow: clone → branch → commit → push → PR → merge
- Describe how you'd troubleshoot a slow server (CPU, memory, disk, logs)
- Explain chmod 755 vs 644 vs 600
- Write a script to check if a service is running and restart it
- Explain GitFlow vs Trunk-Based Development
- Describe your approach to reviewing a Pull Request
- Explain what happens when you run: git pull origin main
- Walk through a deployment script you would write

END OF STUDY GUIDE

Practice daily. Build projects. You've got this!