Firstbyte
Your Path into Tech.

Systems Engineering & DevOps Fundamentals Bootcamp

Student Notes

12 Weeks • 24 Sessions • 72 Hours • Online via Zoom

Schedule	Platform	Registration	Commitment
Fridays & Saturdays, at 7–10 PM CET	**Zoom + Slack**	**Free**	**8–10 hrs/week (class sessions plus self study)**

A Note Before You Begin

The people who finish this program and land jobs are not the ones with the most natural talent. They are the ones who showed up consistently, did the labs even when they were stuck, and asked questions instead of quietly falling behind.

Every lab is a real skill. Every review question is a real interview question.

Good luck.

Welcome to FirstByte Tech

How to Use These Notes

These notes are written to go alongside every session in the bootcamp, not to replace attending class. Think of them as your personal textbook. They explain every concept in plain language, show you real commands and scripts you can run yourself, and give you the context for why things work the way they do.

The course is structured across five phases. Each phase builds on the previous one. Within each phase you will find:

- Concept explanations written for a complete beginner — no assumed knowledge
- Real command examples with line-by-line explanations
- Full code blocks you can copy and run on your own machine
- Callout boxes for important warnings, tips, and things to remember
- Review questions at the end to test yourself honestly

HOW TO STUDY Read each section before the session, not after. You will follow along in class much better if you have already seen the vocabulary. Then come back to the notes after the lab to fill in anything that did not click the first time.

ON CODE EXAMPLES Every command and script in these notes is real. Do not just read them. Open your terminal and run them. You will remember about 20% of what you read and around 80% of what you actually do. The labs are not optional.

Guidelines

1. Attendance is mandatory. Missing two consecutive classes will result in removal from the program. If something comes up, communicate to **FirstByte Admin** before, not after.
2. Speak up. Ask questions, share your thoughts, and contribute to discussions. Your voice matters in this space.
3. Respect everyone, regardless of background, experience level, or where they are in their learning journey. We are all here for the same reason.
4. Confidence is a skill, and this is a safe place to practice it. Do not hold back your questions or ideas out of shyness or fear of looking inexperienced. Nobody in this room knows everything, including the facilitators.
5. If you are stuck or struggling with something, post it in the appropriate Slack channel, not in the facilitator's DM. Chances are someone else has the same question, and the answer helps everyone.
6. Facilitators are human. If there is ever a misstep, an unclear explanation, or any treatment that does not sit right with you, please raise it quickly. We want to know.
7. Do not measure your progress against anyone else's. Go at your own pace, but make sure you are genuinely improving every week.
8. Every session builds on the previous one. Review the last lecture before attending the next one. This is not optional if you want to keep up.
9. Complete all tasks, assignments, and assessments on time. Falling behind on one will make the next one harder.
10. Curiosity is the most important skill in this field. Never stop learning.

PHASE 1

IT Fundamentals & Operations

Weeks 1 to 3 • Sessions 1 to 6

Phase 1 gives you the mental model you need before you ever touch a server. Most people who struggle in technical roles later do so because they skipped this foundation. You need to understand what a computer is, how machines talk to each other over a network, and what an operating system actually does before the hands-on work in later phases makes sense.

Week 1, Session 1: Introduction to IT & Systems Engineering

What Is IT?

IT stands for Information Technology. At a basic level it refers to the use of computers, networks, and software to store, process, and transmit information. In a company context it is much broader than that.

It is the entire infrastructure that keeps a business running: the servers that host applications, the networks that connect employees, the storage systems that hold data, the security systems that protect it, and the people who design, build, and manage all of it.

When you join an IT team, you are not fixing laptops. You are part of the team that decides how systems are built, how they stay up when traffic spikes, how they scale when the business grows, and how they recover when something breaks.

Career Paths in IT

There are many specializations within IT. This course focuses on three of the most in-demand ones.

Systems Engineer / Systems Administrator

A Systems Engineer designs, builds, and maintains the servers and infrastructure that applications run on.

They manage operating systems, configure services, handle user accounts, monitor performance, and troubleshoot failures. This is the core role this bootcamp prepares you for at entry level.

Cloud Engineer

A Cloud Engineer does similar work to a Systems Engineer but on cloud platforms like AWS, Azure, or Google Cloud instead of physical hardware.

Rather than racking servers in a data center, they provision virtual machines and managed services through a web console or an API.

Cloud Engineers also handle networking, storage, and security in the cloud environment.

DevOps Engineer

DevOps sits at the intersection of software development and infrastructure operations.

A DevOps Engineer focuses on automating the process of building, testing, and deploying software so that teams can ship changes quickly and reliably.

They build CI/CD pipelines, manage infrastructure as code, and ensure developers and operations teams work efficiently together.

How Modern Companies Run Infrastructure

In the early days of computing, every company owned and managed its own physical servers stored in its own data center.

Today most companies use a combination of their own infrastructure and cloud services. Some workloads run in AWS or Azure. Others run in a private data center. Many companies use both, which is called a hybrid environment.

What has not changed is the need for people who understand how these systems work and can keep them running. Whether the server is physical or virtual, on-premise or in the cloud, it still needs to be configured, monitored, secured, and maintained.

PRACTICAL — Before Session 2

Install Visual Studio Code from code.visualstudio.com
Install Git from git-scm.com
Create a free AWS account at aws.amazon.com/free
These three things are prerequisites for almost every lab in this course. Do not skip them.

Hardware : What Is Inside a Computer?

Before you can manage servers, you need to understand what a computer actually is at the hardware level. A computer is a machine made up of components that each perform a specific role.

CPU — Central Processing Unit

The CPU is the brain of the computer. It executes instructions.

When you run a program, the CPU reads the instructions in that program and carries them out, billions of times per second. The speed of a CPU is measured in GHz (gigahertz).

The number of cores determines how many tasks it can work on simultaneously. A quad-core CPU can work on four things at the same time.

RAM — Random Access Memory

RAM is short-term, fast memory. It holds the data that is currently being used by running programs.

When you open a browser, the browser is loaded into RAM. When you close it, that memory is freed. RAM is volatile, meaning it is wiped when the machine powers off.

The more RAM a server has, the more processes it can run simultaneously without slowing down.

Storage — Hard Drive or SSD

Storage is long-term memory. Data written to disk persists after the machine powers off.

This is where your operating system, applications, and files live. HDDs (hard disk drives) use spinning magnetic platters and are slower. SSDs (solid-state drives) have no moving parts and are significantly faster.

In cloud environments, storage is usually network-attached, meaning it is connected over the network rather than physically inside the machine.

Network Interface Card (NIC)

The NIC connects the computer to a network. Every machine that communicates over a network needs one.

On a server, the NIC handles all incoming and outgoing traffic.

Video references
What is a computer: https://www.youtube.com/watch?v=Cu3R5it4cQs
Basic Parts of a Computer: https://www.youtube.com/watch?v=mLgTnkw558w
Inside a computer: https://www.youtube.com/watch?v=HB4I2CgkcCo

Computer software

Types of Software: System vs. Application

System Software:

- Manages hardware resources
- Provides a platform for running applications

Examples: Operating systems, device drivers, utilities

Application Software:

- Performs specific tasks for users

Examples: Word processors, web browsers, games

Operating Systems

An operating system (OS) is the software layer that sits between hardware and applications.

It manages the CPU, RAM, storage, and network on behalf of the programs, resources and processes running on the machine.

Without an OS, a computer is just hardware with no way to run anything useful.

- **Windows:** dominant on desktops and common in enterprise environments, especially for office applications
- **Linux:** the operating system that runs the majority of web servers, cloud infrastructure, containers, and DevOps tooling. This is what you will spend most of your time on in this course.
- **macOS:** Apple's OS, built on a Unix foundation. Common among developers.

Video reference
Understanding Operating systems: https://www.youtube.com/watch?v=fkGCLIQx1MI

Virtualization basics

Virtualization is one of the most important concepts in modern IT. A single physical machine is powerful enough to run multiple independent virtual machines simultaneously.

Each virtual machine (VM) behaves as if it is its own dedicated computer, with its own OS, its own RAM allocation, its own storage, and its own network configuration.

The software that makes this possible is called a hypervisor. There are two types:

- Type 1 (Bare-metal): Runs directly on the physical hardware with no host OS underneath. Examples: VMware ESXi, Microsoft Hyper-V, KVM. This is what cloud providers like AWS use at their data centers.
- Type 2 (Hosted): Runs on top of an existing OS. Examples: VirtualBox, VMware Workstation. This is what you might use on your laptop for learning.

Cloud computing is essentially virtualization at massive scale. When you launch an EC2 instance on AWS, you are getting a virtual machine running on AWS's physical hardware somewhere in one of their data centers.

Video references
What is the Cloud: https://www.youtube.com/watch?v=4OO77HFcCUs
Virtualization explained: https://www.youtube.com/watch?v=FZR0rG3HKIk

Applications vs Services

An application is a program that a user runs interactively: a browser, a text editor, a chat client.

A service (called a daemon on Linux) is a program that runs in the background without a user interface, waiting for requests. A web server is a service. A database is a service.

When you start a Linux server, dozens of services start automatically before anyone logs in.

PRACTICAL — Session 2

Open Task Manager (Windows) or Activity Monitor (Mac) and look at CPU, RAM, and disk usage
Find your machine's specs: how many CPU cores, how much RAM, how much disk space
Log into AWS and launch your first Ubuntu EC2 instance. Choose t2.micro (free tier eligible)
Try to identify each hardware concept from this session inside the AWS console

Video reference
Understanding Applications: https://www.youtube.com/watch?v=3gMOYZoMtEs

Week 1, Session 2: AI BASICS

Week 2, Session 3: Operating Systems Fundamentals

What an Operating System Actually Does

An operating system is the most fundamental software on a computer. Every application you run, every file you open, every network connection you make goes through the OS. It is the layer that makes the hardware usable.

- Process management: The OS decides which program gets CPU time and for how long, switching between processes so fast that everything feels simultaneous.
- Memory management: The OS allocates RAM to processes, prevents processes from reading each other's memory (security), and manages swap space on disk when RAM runs low.
- Storage management: The OS reads and writes to disk through the file system, providing a consistent interface for programs to open, read, write, and delete files.
- Network management: The OS provides network interfaces and the TCP/IP stack that applications use to communicate over a network.
- Security: The OS enforces permissions, manages users, and controls what each process is allowed to do.

Windows vs Linux — A Direct Comparison

Area	Windows	Linux
Interface	Primarily GUI (graphical)	Primarily CLI (command line)
Server use	Common for Active Directory, .NET	Dominant for web servers, cloud, DevOps
File system	NTFS (C:\, D:\)	ext4, everything under / (root)
Package management	Manual installers, Chocolatey	apt (Ubuntu/Debian), yum (RHEL/CentOS)
Cost	Requires licensing	Free and open source
Server market share	Minority	Majority (~70%+ of web servers)

Processes

A process is a running instance of a program. Every running program is a process. Each process has a unique Process ID (PID) assigned by the OS. Processes consume CPU and RAM. When a process hangs or misbehaves, you identify its PID and kill it.

```
# List all running processes:
ps aux

# Columns explained:
# USER    - which user owns the process
# PID     - process ID
# %CPU    - CPU usage
# %MEM    - memory usage
# COMMAND - the command that started the process

# Kill a process gracefully (allows it to clean up):
kill 1234
```

```
# Force kill (cannot be ignored by the process):
kill -9 1234
```

Linux System Startup Sequence

When a Linux machine boots, the following sequence happens in order:

1. BIOS or UEFI runs and performs a POST (Power-On Self Test) to check hardware.
2. The bootloader (usually GRUB) loads the Linux kernel into memory.
3. The kernel initializes hardware drivers and mounts the root file system.
4. systemd starts as PID 1. It is the init system responsible for starting all other services.
5. systemd reads unit files and starts all configured services in the correct order.
6. The login prompt appears.

Understanding this sequence helps you troubleshoot boot failures, service startup issues, and recovery scenarios.

Week 2, Session 4: Virtualization & Server Fundamentals

What Is a Server?

A server is a computer that provides services to other computers (clients) over a network. The word server refers to hardware (a physical machine) or software (a program that listens for and responds to requests). Physical servers are designed to run continuously without stopping, handle high loads, be maintained without powering down, and fit into racks in data centers. But fundamentally they have the same components as any computer: CPU, RAM, storage, and a network interface.

SSH — Secure Shell

SSH is the protocol you use to connect to and manage remote Linux servers. Instead of sitting in front of a machine, you open a terminal on your laptop and connect over the network. SSH encrypts all traffic so credentials and commands cannot be intercepted.

How SSH Authentication Works

A key pair consists of two mathematically linked keys: a private key (which stays on your laptop and never leaves it) and a public key (which you place on the server). When you connect, the server uses your public key to verify that you hold the matching private key, without you ever sending the private key itself. This is more secure than passwords.

```
# Generate an SSH key pair on your laptop:
ssh-keygen -t rsa -b 4096 -C 'your-email@example.com'

# Creates two files:
# ~/.ssh/id_rsa       (private key — never share this with anyone)
# ~/.ssh/id_rsa.pub   (public key — this goes on servers)

# Connect to a server:
```

```
ssh ubuntu@54.123.45.67
# ubuntu is the username, the IP 54.123.45.67 is the server's public IP

# Connect using a specific key file (AWS .pem files):
ssh -i ~/.ssh/my-key.pem ubuntu@54.123.45.67

# Copy your public key to a server:
ssh-copy-id ubuntu@54.123.45.67
```

PHASE 1 CHECKPOINT

Before moving to Phase 2, confirm you can do all of the following:

1. Launch an Ubuntu EC2 instance on AWS
2. Connect to it via SSH from your laptop
3. Identify the server's private and public IP addresses
4. Explain the difference between a VM and a physical server
5. Explain what DNS does in plain language and trace the steps

If any of these are unclear, revisit the session and repeat the lab before continuing.

Week 3, Session 5: Networking Fundamentals I

Why Networking Matters

Networking is the foundation of everything in infrastructure. Servers do not exist in isolation. They communicate with each other, with databases, with load balancers, with users on the internet. If you do not understand how data moves between machines, you will struggle to configure systems, troubleshoot connectivity problems, or implement security properly.

Types of Networks

Network Type	Full Name	What It Is
LAN	Local Area Network	A network within a single location, like your home or an office. All devices share the same local network.
WAN	Wide Area Network	A network that spans large distances, connecting multiple LANs. The internet is the largest WAN.
Internet	Interconnected Networks	A global system of interconnected networks. When you access a website, your request travels across the internet to reach a server somewhere in the world.

IP Addresses

Every device on a network needs an address so other devices can find it and send data to it. This address is called an IP address. Think of it exactly like a postal address: for data to reach your machine, something has to know where to send it.

IPv4 Addresses

IPv4 is the most common version of IP addressing. An IPv4 address looks like this:

```
192.168.1.100
```

It is made up of four numbers separated by dots, each between 0 and 255. That means each number is 8 bits and the whole address is 32 bits. There are about 4.3 billion possible IPv4 addresses.

Public vs Private IP Addresses

- Public IP addresses are globally unique and used to communicate across the internet. Your router has a public IP assigned by your Internet Service Provider.
- Private IP addresses are for use within local networks and are not routable on the public internet. Multiple homes and offices can use the same private IP ranges without conflict because they never appear directly on the internet.

The reserved private IP ranges are:

```
10.0.0.0    to  10.255.255.255
172.16.0.0  to  172.31.255.255
192.168.0.0 to  192.168.255.255
```

When you look at the IP address of your laptop right now, it is probably something like 192.168.1.x or 10.0.0.x. That is your private IP. Your public IP is the address your router presents to the internet.

Practical Commands

```
# Find your IP on Linux or macOS:
ip addr show
# or
ifconfig

# On Windows:
ipconfig

# Test connectivity to another machine:
ping 8.8.8.8
# 8.8.8.8 is Google's public DNS server. If this responds, you have internet connectivity.

# Find your public IP from the terminal:
curl ifconfig.me
```

TIP When you run ping, notice the response times in milliseconds. Low latency (under 20ms) means the target is geographically close. High latency (200ms+) means it is far away or there is congestion on the path.

Week 3, Session 6: Networking Fundamentals II

DNS — Domain Name System

DNS is one of the most fundamental services on the internet. Without it, you would have to remember the IP address of every website you visit. DNS translates human-readable domain names like google.com into IP addresses that computers use to communicate.

Here is exactly what happens when you type a domain name into your browser:

1. Your browser checks its local DNS cache. If it has recently looked up this domain, it already knows the IP.
2. If not, it asks your operating system, which checks its own cache and the local hosts file (/etc/hosts on Linux).
3. If still not resolved, the OS queries your DNS resolver — usually your router or your ISP's DNS server.
4. The resolver queries authoritative DNS servers for that domain and gets back the IP address.
5. Your browser uses that IP to connect to the web server.

This entire process typically completes in milliseconds. You never notice it happening.

```
# Query DNS for a domain:
nslookup google.com

# Output example:
# Server:   8.8.8.8
```

```
# Address:  8.8.8.8#53

# Non-authoritative answer:
# Name:     google.com
# Address:  142.250.185.78

# More detailed DNS query with dig:
dig google.com
# Shows the full DNS response including TTL (time to live)
```

DHCP — Dynamic Host Configuration Protocol

When a device joins a network, it needs an IP address. DHCP is the service that assigns one automatically. Your home router runs a DHCP server.

When your phone connects to Wi-Fi, it broadcasts a request for an IP. The DHCP server responds with an IP address, a subnet mask, a default gateway address, and the DNS server addresses to use. All of this happens automatically.

NAT — Network Address Translation

IPv4 only supports about 4.3 billion addresses and the world has more than 4.3 billion devices. NAT allows multiple devices on a private network to share a single public IP address.

Your home router does NAT. Your laptop, phone, and tablet all have private IPs. When any of them makes a request to the internet, the router translates the private source IP to its own public IP, forwards the request, receives the response, and sends it back to the correct private device.

Firewalls

A firewall controls what network traffic is allowed into or out of a system. It evaluates traffic against a set of rules. Firewalls operate on IP addresses and port numbers. A port is a number that identifies a specific service on a machine:

```
Port 22    SSH (remote server access)
Port 80    HTTP (web traffic, unencrypted)
Port 443   HTTPS (web traffic, encrypted)
Port 3306  MySQL database
Port 5432  PostgreSQL database
```

```
# Check firewall status on Ubuntu:
sudo ufw status

# Allow SSH:
sudo ufw allow 22

# Allow web traffic:
sudo ufw allow 80
sudo ufw allow 443

# Enable the firewall:
sudo ufw enable
```

```
# Deny a specific port:
sudo ufw deny 3306
# Never expose your database port to the internet
```

VPN — Virtual Private Network

A VPN creates an encrypted tunnel between two points over the internet. This lets a remote worker connect to a company's internal network as if they were physically in the office. All traffic through the VPN is encrypted, so it cannot be intercepted and read by anyone in the middle.

Video reference
VPN Explained: https://www.youtube.com/watch?v=R-JUOpCgTZc

Routing and traceroute

When data leaves your machine, routers along the path determine how to forward it toward its destination. The traceroute command shows you each hop your data takes across the network:

```
traceroute google.com   # Linux/macOS
tracert google.com      # Windows

# Each line is one router hop along the path.
# The time values show latency to each hop.
# A * means the router did not respond (does not mean the path is broken).
```

PHASE 2

Linux System Administration

Weeks 4 to 6 • Sessions 7 to 12

Phase 2 is where you become comfortable inside a Linux server. The command line will feel unfamiliar at first. That is normal. Every experienced engineer felt the same way. The key is repetition. Run every command in these notes on your own server. Make mistakes. Break things and fix them. That is exactly how this sticks.

Week 4, Session 7: Linux Fundamentals

Linux Distributions

Linux is not a single operating system. It is a kernel — the core software that manages hardware.

A Linux distribution (distro) is the kernel bundled with tools, utilities, and package repositories into a complete OS.

Distribution	Based On	Common Use Case
Ubuntu	Debian	Cloud servers, development, beginners. This course uses Ubuntu.
Debian	Debian	Stable hosting servers
CentOS / RHEL	Red Hat	Enterprise environments, long-term support
Alpine	Independent	Containers — very small footprint
Amazon Linux	RHEL	AWS EC2 instances

The Linux File System Structure

Unlike Windows which uses drive letters (C:\, D:\), Linux organizes everything under a single root directory represented by /.

Every file, directory, device, and even hardware components appear somewhere in this **tree.** *(you can install a tool call Tree)*

```
/              Root of the entire file system
├── etc/       System-wide configuration files
├── var/       Variable data — logs, caches, databases
│   └── log/   System and application log files
├── home/      Home directories for regular users
│   └── ubuntu/ Home directory for the 'ubuntu' user
├── root/      Home directory for the root user
├── tmp/       Temporary files (cleared on reboot)
├── usr/       User-installed programs and libraries
│   ├── bin/   Common user commands
│   └── local/ Locally compiled/installed software
```

```
├── bin/        Essential system binaries
├── sbin/       System administration binaries
└── proc/       Virtual file system showing kernel and process info
```

Linux File Permissions

Every file and directory in Linux has three permission sets: owner, group, and others. Each set has three permission types: read (r), write (w), and execute (x).

```
# When you run 'ls -la' you see permissions like this:
  -rwxr-xr--  1  ubuntu  ubuntu  4096  Jun 1  script.sh

# Breaking down -rwxr-xr--:
# -    = file type (- = regular file, d = directory, l = symlink)
# rwx  = owner can read, write, and execute
# r-x  = group can read and execute but not write
# r--  = others can only read

# Permissions as numbers (used with chmod):
# r = 4,  w = 2,  x = 1
# rwx = 4+2+1 = 7
# r-x = 4+0+1 = 5
# r-- = 4+0+0 = 4

# Set permissions:
chmod 755 script.sh    # rwxr-xr-x  — scripts meant to be run by others
chmod 644 config.txt   # rw-r--r--  — config files, readable by all
chmod 600 ~/.ssh/id_rsa  # rw-------  — SSH private key, owner only

# Change file owner:
sudo chown ubuntu:ubuntu /var/www/mysite
```

Week 4, Session 8: Linux CLI Essentials & Text Editors

Text Editors on Linux

You will edit configuration files on remote servers constantly throughout this course and throughout your career. There is no graphical file manager. You need to be comfortable editing files directly in the terminal. There are two editors you need to know: nano and vim.

nano — The Beginner-Friendly Editor

References:
https://www.nano-editor.org/dist/latest/cheatsheet.html
https://itsfoss.com/nano-editor-guide/

nano is simple and approachable. The commands are shown at the bottom of the screen so you never have to memorise them. Use nano for most of your day-to-day editing.

```
# Open a file in nano:
nano /etc/nginx/nginx.conf

# If the file does not exist, nano creates it.

# Key shortcuts inside nano:
# Ctrl+O       Save the file (O = Write Out)
# Ctrl+X       Exit nano
# Ctrl+K       Cut the current line
# Ctrl+U       Paste the cut line
# Ctrl+W       Search for text
# Ctrl+G       Show help

# The ^ symbol in nano means Ctrl.
# So ^O means Ctrl+O, ^X means Ctrl+X.
```

vim — The Editor You Will Be Forced Into

References:
https://devhints.io/vim
https://vim.rtorr.com/

vim is powerful but has a steep learning curve. The most important thing to know is that vim has modes. When you first open vim, you are in Normal mode where keypresses are commands, not text input. If you start typing and random things happen, you are probably in Normal mode by accident.

```
# Open a file in vim:
vim /etc/ssh/sshd_config

# vim starts in NORMAL MODE — keys are commands, not text

# THE MOST IMPORTANT COMMANDS:
i            Enter INSERT mode (now you can type text)
Esc          Return to NORMAL mode (press this if something feels wrong)

# In NORMAL mode (type these, then press Enter):
  :w          Save the file
  :q          Quit vim
  :wq         Save and quit
  :q!         Quit WITHOUT saving (force quit)

# Navigation in NORMAL mode:
h j k l      Move left, down, up, right (or use arrow keys)
gg           Go to the top of the file
G            Go to the bottom of the file
  /word      Search for 'word' in the file
n            Jump to the next search result
```

```
# The most common beginner trap:
# You opened vim, it looks stuck, you cannot type.
# Press Esc, then type  :q!  and press Enter to force-quit without saving.
```

EDITOR ADVICE Use nano for everything in this course unless you are forced into vim. The goal right now is to manage servers confidently, not to master a text editor. You will naturally pick up more vim over time as you encounter it in the wild.

Navigating the File System

Linux-commands-101-ebook-light:

https://drive.google.com/file/d/1F05CdDvVSEBYl4xBQlOOSLvuXptYBeJM/view?usp=sharing

```
# Where am I right now?
pwd
# Output: /home/ubuntu

# List files in the current directory:
ls

# List with full details (permissions, owner, size, date):
ls -la

# Change to a specific directory:
cd /etc

# Go up one level:
cd ..

# Go back to your home directory:
cd ~
# or just: cd

# Go to the previous directory:
cd -
```

Working with Files and Directories

```
# Create a directory:
mkdir /tmp/myproject

# Create nested directories in one command:
mkdir -p /tmp/myproject/config/ssl

# Create an empty file:
```

```
touch /tmp/myproject/notes.txt

# Write text into a file (overwrites existing content):
echo 'Hello world' > /tmp/myproject/notes.txt

# Append text to a file (keeps existing content):
echo 'Second line' >> /tmp/myproject/notes.txt

# View file contents:
cat /tmp/myproject/notes.txt

# create a file with the cat command:
cat > file_name

# Append the contents of file1 to file2:
cat file1 >> file2

# View file contents in a reverse form:
tac <specified_file_name>

# View a long file one page at a time:
less /var/log/syslog
# Press space to go forward, b to go back, q to quit

# Copy a file:
cp source.txt destination.txt

# Copy a directory and its contents:
cp -r /tmp/myproject /tmp/myproject-backup

# Move or rename a file:
mv oldname.txt newname.txt

# Delete a file:
rm file.txt

# Delete a directory and everything inside it:
rm -rf /tmp/myproject
# WARNING: rm -rf is permanent. There is no recycle bin on Linux.

# display the help texts for a command and exit:
Eg:
cat --help
tac --help
ls --help
touch --help
less --help
cp --help
etc
```

Searching for Files and Content

```
# Find a file by name (searches from root /):
find / -name 'sshd_config' 2>/dev/null
# 2>/dev/null hides permission errors so output is clean

# Find files in a specific directory:
find /etc -name '*.conf'

# Find files modified in the last 24 hours:
find /var/log -mtime -1

# Search inside a file for a pattern:
grep 'error' /var/log/syslog
# Prints every line containing the word 'error'

# Case-insensitive search:
grep -i 'error' /var/log/syslog

# Search recursively through all files in a directory:
grep -r 'database_host' /etc/

# Show line numbers with matches:
grep -n 'failed' /var/log/auth.log

# Count how many lines match:
grep -c 'error' /var/log/syslog
```

Week 5, Session 9: Linux Administration

User Management

Linux is a multi-user operating system. Every person or service that needs access to the system should have its own account. This is important for security and accountability: if something goes wrong, you can trace activity back to a specific user.

```
# Create a new user with a home directory:
sudo useradd -m -s /bin/bash alice
# -m creates /home/alice
# -s sets bash as the default shell

# Set a password for the user:
sudo passwd alice

# Create a group:
sudo groupadd developers

# Add a user to a group:
sudo usermod -aG developers alice
# -a = append (do not remove from other groups)
# -G = add to supplementary group

# List the groups a user belongs to:
groups alice

# View all users on the system:
cat /etc/passwd

# View all groups:
cat /etc/group

# Delete a user and their home directory:
sudo userdel -r alice

# Switch to another user:
su - alice

# Run a single command as another user:
sudo -u alice cat /home/alice/.bashrc
```

Understanding sudo

sudo (superuser do) allows a regular user to run a single command with root (administrator) privileges. It is safer than logging in as root directly because every sudo command is logged, you only elevate when necessary, and if the account is compromised, damage is more limited than if root itself were compromised.

```
# Run a command as root:
sudo apt update

# Open a root shell:
sudo -i
# Type 'exit' to return to your regular user

# See who can use sudo:
sudo visudo
# ALWAYS use visudo to edit the sudoers file — it validates syntax.
# A syntax error in /etc/sudoers can lock you out of sudo permanently.
```

Package Management with apt

```
# ALWAYS run update before installing anything:
sudo apt update
# This refreshes the package list. It does NOT install or upgrade anything yet.

# Upgrade all installed packages:
sudo apt upgrade

# Install a package:
sudo apt install nginx

# Install multiple packages at once:
sudo apt install git curl wget vim tree htop

# Remove a package:
sudo apt remove nginx

# Remove package plus its configuration files:
sudo apt purge nginx

# Remove packages that are no longer needed:
sudo apt autoremove

# Search for a package:
apt search nginx

# Get information about a package:
apt show nginx
```

Managing Services with systemctl

```
# Start a service:
sudo systemctl start nginx
```

```
# Stop a service:
sudo systemctl stop nginx

# Restart a service (stop then start):
sudo systemctl restart nginx

# Reload configuration without restarting:
sudo systemctl reload nginx
# Not all services support reload. nginx does. mysql does not.

# Check service status (most useful command):
sudo systemctl status nginx
# Shows: is it running, recent log lines, PID, memory usage

# Enable a service to start on boot:
sudo systemctl enable nginx

# Disable autostart:
sudo systemctl disable nginx

# List all failed services:
systemctl --failed

# List all active services:
systemctl list-units --type=service --state=active
```

Week 5, Session 10: Monitoring & Troubleshooting

Real-Time Resource Monitoring

```
# Basic process viewer:
top
# Press 'q' to quit, 'k' to kill a process by PID

# Better process viewer (install if needed):
sudo apt install htop
htop
# Color-coded bars for CPU cores and memory
# F9 to kill a process, F10 to quit

# Check disk space on all partitions:
df -h
# -h = human-readable (GB/MB instead of bytes)
# Look at the 'Use%' column. Above 90% is a problem.
```

```
# Check disk usage of a specific directory:
du -sh /var/log
# -s = summary total only, -h = human-readable

# Find the top 10 largest directories:
du -h /var | sort -rh | head -10

# Check RAM usage:
free -m
# -m = show in megabytes
# The 'available' value is what the OS can actually give to new processes
# 'used' includes memory used for caching (Linux uses spare RAM for cache)
```

Reading Log Files

When something goes wrong on a Linux server, log files are almost always where you find the answer. Get into the habit of checking logs first when troubleshooting.

```
# System log (general system messages):
sudo cat /var/log/syslog

# Follow a log file in real time (new lines appear as written):
sudo tail -f /var/log/syslog
# Press Ctrl+C to stop following

# Show last 100 lines:
sudo tail -n 100 /var/log/syslog

# Authentication log (SSH logins, sudo usage, failed logins):
sudo tail -f /var/log/auth.log

# View logs for a specific service using systemd journal:
sudo journalctl -u nginx

# Follow service logs in real time:
sudo journalctl -u nginx -f

# Show logs from the last 30 minutes:
sudo journalctl --since '30 min ago'

# Show only error-level and above:
sudo journalctl -p err -b
```

TROUBLESHOOTING SEQUENCE

When something breaks on a server, work through this order every time:
1. sudo systemctl status <service> — is the service running?
2. sudo journalctl -u <service> -n 50 — what does the log say?
3. df -h — is the disk full? (a full disk causes many mysterious failures)

4. free -m — is the server out of memory?
5. ps aux | grep <service> — is the process actually running?
6. Check application-specific logs in /var/log/<service>/

This sequence resolves the majority of common server problems.

Week 6, Session 11: Bash Scripting Fundamentals

What Is a Bash Script?

A Bash script is a plain text file containing a sequence of commands that the Bash shell executes in order. Anything you can type into a terminal, you can put in a script. Scripts let you automate repetitive tasks so you do not have to type the same commands repeatedly.

Introduction-to-bash-scripting-light -
https://drive.google.com/file/d/1L16-OhkrfZz1A0i5qFmT9i8iHzGa9UEF/view?usp=sharing

Structure of a Bash Script

```
#!/bin/bash
# This first line is called the shebang.
# It tells the OS which interpreter to use. Without it, the script will not run correctly.

# Comments start with #. They are ignored by the interpreter.
# Use them to explain what your script does and why.

# Make a script executable (before running it):
chmod +x myscript.sh

Then run it
./myscript.sh

Or
bash myscript.sh
```

Variables

```
#!/bin/bash

# Define a variable — no spaces around the = sign:
SERVER_NAME='web-01'
PORT=8080

# Reference a variable with $:
echo "Server: $SERVER_NAME is listening on port $PORT"

# Capture the output of a command into a variable:
CURRENT_USER=$(whoami)
HOSTNAME=$(hostname)
TODAY=$(date '+%Y-%m-%d')

echo "Running as: $CURRENT_USER on $HOSTNAME at $TODAY"

# Read-only variable:
```

```
readonly MAX_RETRIES=3

# Check if a variable is set:
if [ -z "$MY_VAR" ]; then
    echo 'MY_VAR is not set'
fi
```

Conditionals

```
#!/bin/bash

DISK_USAGE=$(df / | tail -1 | awk '{print $5}' | tr -d '%')
# This pipeline:
# df /           — gets disk info for the root partition
# tail -1        — takes the last line (the data line)
# awk '{print $5}' — extracts the 5th column (use percentage)
# tr -d '%'      — removes the % sign so it is a plain number

if [ $DISK_USAGE -gt 90 ]; then
    echo "CRITICAL: Disk is $DISK_USAGE% full — immediate action required"
elif [ $DISK_USAGE -gt 75 ]; then
    echo "WARNING: Disk is $DISK_USAGE% full — monitor closely"
else
    echo "OK: Disk usage is $DISK_USAGE%"
fi

# Common comparison operators for numbers:
# -eq   equal to
# -ne   not equal to
# -gt   greater than
# -lt   less than
# -ge   greater than or equal to
# -le   less than or equal to

# String comparisons:
# = or ==  equal
# !=       not equal
# -z       string is empty
# -n       string is not empty
```

Loops

```
#!/bin/bash

# For loop — iterate over a list:
for USER in alice bob charlie dave; do
    echo "Processing user: $USER"
```

```
    sudo useradd -m -s /bin/bash $USER
    echo "  Created: /home/$USER"
done

# For loop — over a numbered range:
for i in {1..5}; do
    echo "Backup number $i"
done

# While loop — run while condition is true:
COUNT=0
MAX=5
while [ $COUNT -lt $MAX ]; do
    echo "Attempt $COUNT of $MAX"
    COUNT=$((COUNT + 1))
    sleep 1
done

# Loop over files matching a pattern:
for LOG in /var/log/*.log; do
    SIZE=$(du -sh $LOG | awk '{print $1}')
    echo "$LOG: $SIZE"
done
```

Functions

```
#!/bin/bash

# Define a function:
check_service() {
    local SERVICE=$1   # $1 is the first argument passed to the function
    # 'local' keeps this variable scoped to the function

    if systemctl is-active --quiet $SERVICE; then
        echo "  [OK]  $SERVICE is running"
    else
        echo "  [FAIL] $SERVICE is NOT running"
        echo "  Attempting to start $SERVICE..."
        sudo systemctl start $SERVICE
    fi
}

# Call the function:
echo 'Checking critical services...'
check_service nginx
check_service mysql
check_service ssh
echo 'Done.'
```

Are you stuck on Bash Scripting?
Find help here using any of the links

- **https://youtube.com/playlist?list=PLS1QulWo1RIYmaxcEqw5JhK3b-6rgdWO_&si=qjT2Mz-xmnIxu4lg**
- **https://youtube.com/playlist?list=PL2qzCKTbjutJRM7K_hhNyvf8sfGCLklXw&si=g2MmEZ0N36rvq_Q9**
- **https://youtube.com/playlist?list=PLT98CRl2KxKGj-VKtApD8-zCqSaN2mD4w&si=YgYf7-DfJMULCbpO**

Week 6, Session 12: Bash Automation Project

Cron Jobs — Scheduling Scripts

The Cron software is a job scheduler in Linux operating systems that allows Linux users to run a specific command or script at a given time and date. Generally, it is used by system administrators to automate backup tasks, directory cleaning, notifications, etc.

The Cron job runs in background and continuously checks **/etc/crontab**, **/etc/cron.*/** and **/var/spool/cron/** directories for any jobs and execute them at a pre-defined interval of time.

https://crontab.guru/
https://crontab.guru/examples.html
https://crontab-generator.org/

Install Cron

Before starting, make sure the Cron package is installed in your system. If not installed, you can install it using the following command:

```
sudo apt-get install cron -y
```

OR

```
sudo apt update && sudo apt install cron
```

After installing the Cron package, start the Cron service and enable it to start at system reboot:

sudo systemctl start cron && sudo systemctl enable cron

```
# Edit your crontab:
crontab -e

# Cron schedule format:
# ┌──────────── minute       (0-59)
# │ ┌────────── hour         (0-23)
# │ │ ┌──────── day of month  (1-31)
# │ │ │ ┌────── month         (1-12)
```

```
# | | | | ┌─ day of week    (0-7, 0 and 7 = Sunday)
# | | | | |
# * * * * *  command

# Examples:
  0 2 * * *   /home/ubuntu/scripts/backup.sh       # Every day at 2:00 AM
  */15 * * * * /home/ubuntu/scripts/health.sh       # Every 15 minutes
  0 9 * * 1   /home/ubuntu/scripts/weekly-report.sh # Mondays at 9:00 AM
  0 0 * * 0   /home/ubuntu/scripts/cleanup.sh       # Sundays at midnight

# View current crontab:
crontab -l

# Redirect cron output to a log file:
  0 2 * * * /home/ubuntu/scripts/backup.sh >> /var/log/backup.log 2>&1
# 2>&1 means redirect stderr (error output) to the same place as stdout
```

Mini Project: Server Health Check Script

```
#!/bin/bash
# ================================================
# health-check.sh
# Server Health Report — FirstByte DevOps Bootcamp
# ================================================

REPORT_DATE=$(date '+%Y-%m-%d %H:%M:%S')
HOSTNAME=$(hostname)

divider() {
    echo '------------------------------------------------'
}

echo ''
echo '================================================'
echo "  SERVER HEALTH REPORT"
echo "  Host:      $HOSTNAME"
echo "  Generated: $REPORT_DATE"
echo '================================================'

# ---- UPTIME ----
echo ''
echo '[ UPTIME ]'
divider
uptime

# ---- DISK USAGE ----
echo ''
echo '[ DISK USAGE ]'
```

```
divider
df -h | grep -v tmpfs

# Alert if any partition is above 85%
df / | tail -1 | awk '{print $5}' | tr -d '%' | while read USAGE; do
    if [ $USAGE -gt 85 ]; then
        echo "  WARNING: Root partition is $USAGE% full!"
    fi
done

# ---- MEMORY ----
echo ''
echo '[ MEMORY ]'
divider
free -m

# ---- TOP PROCESSES BY CPU ----
echo ''
echo '[ TOP 5 PROCESSES BY CPU ]'
divider
ps aux --sort=-%cpu | head -6

# ---- FAILED SERVICES ----
echo ''
echo '[ FAILED SERVICES ]'
divider
systemctl --failed --no-legend

# ---- RECENT SSH LOGINS ----
echo ''
echo '[ RECENT SSH LOGINS ]'
divider
sudo last | head -10

echo ''
echo '================================================'
echo '  END OF REPORT'
echo '================================================'

# Usage:
#   chmod +x health-check.sh
#   ./health-check.sh
#   To save output: ./health-check.sh > /tmp/health-report.txt
#   To schedule: 0 * * * * /home/ubuntu/scripts/health-check.sh >>
/var/log/health.log 2>&1
```

Bash Scripting Assignment

1. Create an account on Exercism - https://exercism.org/users/sign_up
2. Sign In - https://exercism.org/users/sign_in
3. Choose Bash track - https://exercism.org/tracks/bash

You are to complete all 91 exercises.

There are 7 achievements, which you are to unlock by the time you complete all tasks.

PHASE 3

Systems Engineering & Web Services

Weeks 7 to 8 • Sessions 13 to 16

Phase 3 puts software on top of the Linux servers you can now manage. Web servers, databases, and monitoring are what turn a bare server into something useful. Every company that runs software on the internet uses all three.

Week 7, Session 13: Web Server Fundamentals

How the Web Works — The Full Request Lifecycle

When you type a URL into a browser and press Enter, a precise sequence of events happens. Understanding this end-to-end is essential for infrastructure work.

1. You type **https://example.com** into your browser.
2. Your browser checks its DNS cache for the IP address of example.com.
3. If not cached, it queries a DNS resolver which returns an IP, say 93.184.216.34.
4. Your browser opens a TCP connection to port 443 (HTTPS) on that IP.
5. A TLS handshake happens: the server presents its SSL certificate, your browser verifies it, and an encrypted session is established.
6. Your browser sends an HTTP GET request: 'Give me the resource at /'
7. The web server receives the request, finds the resource, and sends back an HTTP 200 response with the HTML content.
8. Your browser renders the HTML into the page you see.

HTTP Status Codes

Status Code	Meaning	When You Encounter It
200 OK	Request succeeded	Page loaded successfully
301 Moved Permanently	Resource has a new permanent URL	Domain redirects
403 Forbidden	Server refused the request	No permission to access the resource
404 Not Found	Resource does not exist	Broken link, wrong URL
500 Internal Server Error	Application crashed or misconfigured	Bug in the application
502 Bad Gateway	Reverse proxy got invalid response from upstream	Nginx cannot reach your app
503 Service Unavailable	Server temporarily unable to handle requests	Overloaded or in maintenance

Reverse Proxies

A reverse proxy sits in front of your application servers and forwards client requests to them. Clients talk to the proxy, not directly to the application.

- Security: The application server's internal IP is never exposed to the internet.
- SSL termination: The proxy handles HTTPS so your application only needs to speak plain HTTP internally.
- Load balancing: The proxy distributes requests across multiple application servers.

- Caching: The proxy can serve cached responses for repeated identical requests.
- Centralized logging: All request logs in one place regardless of how many app servers exist.

Week 7, Session 14: Nginx Administration

Installing and Starting Nginx

```
sudo apt update
sudo apt install nginx
sudo systemctl start nginx
sudo systemctl enable nginx     # Start automatically on boot
sudo systemctl status nginx

# Visit your server's public IP in a browser.
# You should see the default Nginx welcome page.

# Key Nginx file locations:
# /etc/nginx/nginx.conf             Main configuration file
# /etc/nginx/sites-available/       Site config files
# /etc/nginx/sites-enabled/         Symlinks to enabled sites
# /var/www/html/                    Default web root
# /var/log/nginx/access.log         Request log
# /var/log/nginx/error.log          Error log
```

Serving a Static Website

```
# Create a directory for your site:
sudo mkdir -p /var/www/mysite
sudo chown -R ubuntu:ubuntu /var/www/mysite

# Create a basic HTML page:
echo '<h1>Hello from FirstByte!</h1>' > /var/www/mysite/index.html

# Create the Nginx site configuration:
sudo nano /etc/nginx/sites-available/mysite.conf

# Paste this content:
# server {
#     listen 80;
#     server_name mysite.com www.mysite.com;
#     root /var/www/mysite;
#     index index.html;
#
#     location / {
#         try_files $uri $uri/ =404;
```

```
#     }
# }

# Enable the site:
sudo ln -s /etc/nginx/sites-available/mysite.conf /etc/nginx/sites-enabled/

# Disable the default site:
sudo rm /etc/nginx/sites-enabled/default

# Test configuration syntax:
sudo nginx -t
# If it says 'test is successful', you are good to reload

# Reload Nginx to apply changes:
sudo systemctl reload nginx
```

Note:

One server can host hundreds or even thousands of websites on a single IP address. The key technology is called **virtual hosting**.

Example: Suppose your server has **IP:** 203.0.113.10

DNS records can be set to:

```
mysite.com        -> 203.0.113.10
myblog.com         -> 203.0.113.10
```

All three domains point to the same server.

```
# In Nginx
server {
    server_name mysite.com;
    root /var/www/mysite;
}

server {
    server_name myblog.com;
    root /var/www/myblog;
}
```

When someone visits:

```
mysite.com → it gets served from /var/www/mysite
myblog.com → it gets served from /var/www/myblog
```

even though they all resolve to the same IP.

there may be thousands of domains sharing the same IP address. The web server, load balancer, or reverse proxy routes requests based on the hostname.

Quick test On Linux:

```
nslookup google.com
nslookup gmail.com
```

You may notice multiple Google services sometimes resolve to the same IP ranges. The server infrastructure determines which application to serve based on the hostname and request details.

```
Browser
   ↓
DNS resolves domain to IP
   ↓
Connect to server IP
   ↓
Send hostname (Host header / SNI)
   ↓
Web server selects correct website
   ↓
Returns content
```

This is why a single VPS **(Virtual Private Server)** with one public IP can comfortably host multiple websites, APIs, and applications.

Nginx as a Reverse Proxy

```
# Scenario: Node.js app running on port 3000.
# Goal: Nginx forwards requests from port 80 to port 3000.

# /etc/nginx/sites-available/myapp.conf:
#
# server {
#     listen 80;
#     server_name myapp.example.com;
#
#     location / {
#         proxy_pass http://localhost:3000;
#         proxy_http_version 1.1;
#         proxy_set_header Upgrade $http_upgrade;
#         proxy_set_header Connection 'upgrade';
#         proxy_set_header Host $host;
#         proxy_set_header X-Real-IP $remote_addr;
#         proxy_cache_bypass $http_upgrade;
#     }
# }

# Test and reload:
sudo nginx -t && sudo systemctl reload nginx
```

Reading Nginx Logs

```
# Watch all incoming requests in real time:
sudo tail -f /var/log/nginx/access.log
# Each line shows: IP, date, request, status code, response size, user agent
```

```
# Watch errors in real time:
sudo tail -f /var/log/nginx/error.log

# Filter for specific status codes:
sudo grep ' 502 ' /var/log/nginx/access.log

# Count 404s in the last hour of logs:
sudo awk '/404/{count++} END{print count}' /var/log/nginx/access.log
```

Week 8, Session 15: Databases for Sysadmins

What a Sysadmin Needs to Know About Databases

Almost every application your servers run will have a database behind it. Your job as a sysadmin is not to design database schemas or write application queries — that is the developer's responsibility. Your job is to install and secure the database correctly, keep it running, and make sure backups are happening reliably. That is what this session focuses on.

Installing and Securing MySQL

```
sudo apt update
sudo apt install mysql-server
sudo systemctl start mysql
sudo systemctl enable mysql    # Start automatically on boot

# Verify it is running:
sudo systemctl status mysql

# Run the security hardening wizard — always do this after install:
sudo mysql_secure_installation

# This will:
#   Set a strong root password
#   Remove anonymous users
#   Remove the test database
#   Disable remote root login
# Answer 'Y' to all prompts.
```

Connecting to MySQL and Basic Navigation

```
# Connect as root:
sudo mysql -u root -p

# Once inside the MySQL shell:
SHOW DATABASES;      -- List all databases
SHOW TABLES;         -- List tables in the current database
USE myapp_db;        -- Switch to a database
EXIT;                -- Leave the MySQL shell

# Connect as a specific application user:
mysql -u appuser -p myapp_db
```

User and Permission Management

Never let your application connect to the database as root. Create a dedicated user with only the permissions that application actually needs.

```
-- Inside the MySQL shell:
```

```
-- Create a dedicated application user:
CREATE USER 'appuser'@'localhost' IDENTIFIED BY 'StrongPassword123!';

-- Grant that user access only to the specific database it needs:
GRANT ALL PRIVILEGES ON myapp_db.* TO 'appuser'@'localhost';

-- Apply the changes:
FLUSH PRIVILEGES;

-- View a user's permissions:
SHOW GRANTS FOR 'appuser'@'localhost';

-- Create a read-only backup user:
CREATE USER 'backup_user'@'localhost' IDENTIFIED BY 'BackupPass456!';
GRANT SELECT, LOCK TABLES ON myapp_db.* TO 'backup_user'@'localhost';
FLUSH PRIVILEGES;
```

Backups and Restoration

A database with no backup strategy is a liability. This is non-negotiable in any production environment.

```
# Backup a single database (run in Linux terminal, not the MySQL shell):
mysqldump -u root -p myapp_db > /var/backups/myapp_db_$(date +%F).sql
# $(date +%F) appends today's date, e.g. myapp_db_2024-06-15.sql

# Backup all databases at once:
mysqldump -u root -p --all-databases > /var/backups/all_dbs_$(date +%F).sql

# Verify the backup file is not empty:
ls -lh /var/backups/myapp_db_$(date +%F).sql

# Restore a database from backup:
mysql -u root -p myapp_db < /var/backups/myapp_db_2024-06-15.sql

# Automate daily backups with cron:
# Add this line with: crontab -e
# 0 2 * * * mysqldump -u backup_user -pBackupPass456! myapp_db >
/var/backups/myapp_db_$(date +\%F).sql

# Rotate old backups — delete files older than 30 days:
# 0 3 * * * find /var/backups -name '*.sql' -mtime +30 -delete
```

Checking Database Health

```
# Check if MySQL is running:
sudo systemctl status mysql


# Check MySQL error log:
sudo tail -f /var/log/mysql/error.log
```

```
# Check how much disk space the databases are using:
sudo du -sh /var/lib/mysql

# Connect and check running processes (useful when DB feels slow):
sudo mysql -u root -p -e 'SHOW PROCESSLIST;'
```

SCOPE NOTE SQL syntax for querying and manipulating data (SELECT, INSERT, UPDATE, DELETE, JOIN) is part of a developer or DBA role. If you want to go deeper in that direction, that is a separate learning path. For now, knowing how to install, secure, back up, and monitor a MySQL instance is what your job requires.

Week 8, Session 16: Monitoring & Logging

Why Monitoring Matters

In production, you do not find out about problems when users complain. You find out because your monitoring alerted you before users even noticed. Monitoring tells you the health of your systems in real time and over time. Logging tells you what happened and when.

Concept	What It Is	Example
Metric	A numeric measurement captured over time	CPU at 73%, 200 requests/sec, 5ms response time
Log	A timestamped record of a specific event	2024-01-01 12:00:01 ERROR: Connection refused
Alert	Notification triggered when a metric crosses a threshold	Email when CPU > 90% for 5 consecutive minutes
Dashboard	Visual display of metrics over time	Grafana board showing all server health metrics

Structured Logging with journalctl

```
# All logs from the current boot:
sudo journalctl -b

# Logs for a specific service:
sudo journalctl -u nginx

# Follow live:
sudo journalctl -u nginx -f

# Logs from the last hour:
sudo journalctl --since '1 hour ago'
```

```
# Logs between specific timestamps:
sudo journalctl --since '2024-01-01 08:00' --until '2024-01-01 09:00'

# Only errors and above:
sudo journalctl -p err

# See how much disk space the journal is using:
sudo journalctl --disk-usage

# Limit journal size in /etc/systemd/journald.conf:
# SystemMaxUse=500M
```

GRAFANA INTRO

Grafana is a visualization tool for building dashboards from time-series data. In class you will get an introduction to it. In production, Grafana is typically paired with Prometheus (metrics collection) to give you a full observability stack. Focus first on understanding what monitoring is and why it matters. The tooling becomes straightforward once you have that foundation.

PHASE 4

Cloud Engineering

Weeks 9 to 10 • Sessions 17 to 20

Cloud computing is not a different technology from what you have learned. It is the same Linux servers, networking, storage, and services — managed through a provider's API and console instead of physical hardware. AWS is the largest cloud provider and the most relevant for job seekers right now.

Week 9, Session 17: AWS Fundamentals

Video reference
What is Cloud Computing | AWS: https://www.youtube.com/watch?v=mxT233EdY5c
Cloud computing Explained: https://www.youtube.com/watch?v=_a6us8kaq0g

Useful links for further education:

- **AWS Educate - Introduction to Cloud 101:** https://awseducate.instructure.com/courses/891/modules
- **AWS Skill Builder: Cloud Essentials - Knowledge Badge Readiness Path:**

https://explore.skillbuilder.aws/learn/lp/82/Cloud%2520Essentials%2520-%2520Knowledge%2520Badge%2520Readiness%2520Path

- **NDG Linux Unhatched English:** NDG Linux Unhatched

Regions and Availability Zones

AWS operates data centers in geographic locations around the world called Regions (for example, us-east-1 in Northern Virginia, eu-west-1 in Ireland). Within each Region there are multiple Availability Zones (AZs). An AZ is a physically separate data center with its own power, cooling, and networking. If one AZ has a power failure, your application keeps running in another AZ. Always deploy production workloads across at least two AZs.

The AWS Shared Responsibility Model

This is one of the most tested concepts in cloud. Responsibility for security is divided between AWS and you:

AWS Is Responsible For	You Are Responsible For
Physical security of data centers	Operating system patching and updates
Hardware maintenance and networking	Application security and code quality
Hypervisor security	Identity and access management (IAM)
Managed service patching (e.g., RDS)	Data encryption at rest and in transit
Physical network infrastructure	Security Group and firewall configuration

EC2 — Elastic Compute Cloud

EC2 is AWS's virtual machine service. You choose an instance type (CPU and RAM), an AMI (Amazon Machine Image — the OS template), storage, and networking. Within minutes you have a running Linux server.

```
# Common instance types to know:
t2.micro    1 vCPU,  1 GB RAM   Free tier eligible — good for labs
t3.small    2 vCPU,  2 GB RAM   Small production workloads
t3.medium   2 vCPU,  4 GB RAM   General purpose
m5.large    2 vCPU,  8 GB RAM   Memory-balanced workloads
c5.large    2 vCPU,  4 GB RAM   Compute-optimized

# Connect after launch:
ssh -i ~/.ssh/my-key.pem ubuntu@<PUBLIC_IP>
```

S3 — Simple Storage Service

S3 stores files as objects in containers called buckets. Every object has a key (its filename) and lives in a bucket. S3 is designed for 99.999999999% (eleven nines) durability — data is replicated across multiple AZs automatically. Use it for backups, static websites, logs, and media files.

```
# Install AWS CLI:
sudo apt install awscli
aws configure    # Enter your Access Key ID, Secret Key, region

# Create a bucket:
aws s3 mb s3://my-company-backup-2024

# Upload a file:
aws s3 cp myfile.txt s3://my-company-backup-2024/

# List bucket contents:
aws s3 ls s3://my-company-backup-2024/

# Sync a local directory to S3:
aws s3 sync /var/backups/ s3://my-company-backup-2024/backups/

# Download a file:
aws s3 cp s3://my-company-backup-2024/myfile.txt ./
```

IAM — Identity and Access Management

IAM controls who can do what in your AWS account. Apply the principle of least privilege at all times: grant each user, role, or service only the permissions it actually needs.

IAM Concept	What It Is	When to Use
User	Long-term identity for a person	Human users who need console or API access

Group	Collection of users	Apply the same permissions to multiple users
Role	Temporary identity that can be assumed	Give an EC2 instance or Lambda AWS permissions
Policy	JSON document defining allowed actions	Attached to users, groups, or roles

NEVER DO THIS

Never put AWS Access Keys directly in code or scripts.
If those credentials get pushed to a public GitHub repository, attackers scan GitHub automatically and will exploit them within minutes — potentially generating thousands of dollars in charges.
Use IAM Roles for EC2 instances. Use environment variables or AWS Secrets Manager for applications.

Week 9, Session 18: AWS Networking

VPC — Virtual Private Cloud

A VPC is your own isolated network inside AWS. When you create one, you define a private IP range using CIDR notation. All your resources live inside the VPC and are isolated from other AWS customers.

```
# CIDR notation defines an IP address range:
# 10.0.0.0/16 means:
#   First 16 bits are fixed: 10.0.x.x
#   Gives you 10.0.0.0 to 10.0.255.255 (65,536 addresses)

# /24 subnet:
#   10.0.1.0/24 = 10.0.1.0 to 10.0.1.255 (256 addresses)

# Typical VPC layout:
VPC:               10.0.0.0/16
Public Subnet 1:   10.0.1.0/24   (AZ: us-east-1a)
Public Subnet 2:   10.0.2.0/24   (AZ: us-east-1b)
Private Subnet 1:  10.0.10.0/24  (AZ: us-east-1a)
Private Subnet 2:  10.0.20.0/24  (AZ: us-east-1b)
```

Public vs Private Subnets

- A public subnet has a route to an Internet Gateway. Resources in it can send and receive traffic from the internet. Your web servers and load balancers live here.
- A private subnet has no route to the internet. Resources cannot be reached from outside. Your databases and internal services live here. For outbound internet access (software updates), use a NAT Gateway.

Security Groups

Security Groups are stateful firewalls attached to EC2 instances. You define inbound and outbound rules by port, protocol, and source/destination IP range. 'Stateful' means if you allow an inbound connection, the response is automatically allowed back out.

```
# Example Security Group for a web server:

# INBOUND:
# Type      Protocol  Port Range  Source
# SSH       TCP       22          Your office IP only: 203.0.113.5/32
# HTTP      TCP       80          Anywhere: 0.0.0.0/0
# HTTPS     TCP       443         Anywhere: 0.0.0.0/0

# OUTBOUND:
# All traffic  All  All  Anywhere: 0.0.0.0/0

# Example Security Group for a database:
# INBOUND:
# MySQL     TCP  3306  Web server Security Group ID (not 0.0.0.0/0!)
# Only the web server should be able to reach the database.

# /32 at the end of an IP means exactly that single IP address.
```

Week 10, Session 19: AWS Operations & Infrastructure Design

CloudWatch — Monitoring and Alerting

CloudWatch is AWS's native monitoring service. It collects metrics from virtually every AWS service automatically. You configure alarms to notify you when a metric exceeds a threshold.

```
# CloudWatch collects these automatically from EC2:
# CPUUtilization, NetworkIn, NetworkOut, DiskReadBytes, DiskWriteBytes

# Create a CPU alarm via AWS CLI:
aws cloudwatch put-metric-alarm \
  --alarm-name 'High-CPU' \
  --metric-name CPUUtilization \
  --namespace AWS/EC2 \
  --statistic Average \
  --period 300 \
  --threshold 80 \
  --comparison-operator GreaterThanThreshold \
  --evaluation-periods 2 \
  --dimensions Name=InstanceId,Value=i-1234567890abcdef0 \
  --alarm-actions arn:aws:sns:us-east-1:123456789:alerts-topic

# This alarm fires if average CPU exceeds 80%
# for two consecutive 5-minute evaluation periods.
```

Load Balancers

An Elastic Load Balancer (ELB) distributes incoming traffic across multiple EC2 instances. It also performs health checks and automatically stops sending traffic to unhealthy instances. Use an Application Load Balancer (ALB) for HTTP/HTTPS traffic and a Network Load Balancer (NLB) for TCP/UDP at very high throughput.

Auto Scaling

Auto Scaling automatically adjusts the number of EC2 instances based on demand. When traffic increases, it adds instances. When traffic drops, it removes them. You define minimum, maximum, and desired capacity, and the scaling policies that trigger changes.

Auto Scaling Term	What It Means
Minimum capacity	The fewest instances that should ever be running (e.g., 2)
Maximum capacity	The most instances Auto Scaling will ever create (e.g., 10)
Desired capacity	The target number of instances under normal load (e.g., 3)
Scale-out	Add instances — triggered when CPU stays above 70% for 5 minutes
Scale-in	Remove instances — triggered when CPU stays below 30% for 10 minutes

Infrastructure Design Lab

This session closes with a hands-on lab where you build a complete simple production environment from scratch. Here is the target architecture:

```
Internet
    |
Internet Gateway
    |
Public Subnet (10.0.1.0/24)
    |
EC2 Instance — Nginx web server
    Security Group: ports 80, 443 open | port 22 from your IP only
    |
Private Subnet (10.0.10.0/24)
    |
RDS MySQL — database
    Security Group: port 3306 from web server Security Group only

S3 Bucket:    static assets and backups
CloudWatch:   CPU alarm on the EC2 instance
IAM Role:     EC2 instance role with S3 permissions — no hardcoded credentials
```

- Deploy EC2
- Configure security groups
- Use S3
- Set up monitoring
- Manage IAM users

LAB GOAL

Build this environment from scratch in your own AWS account. Do not use the default VPC.
By the end of this lab you should have: a web server accessible in a browser, a private database it can reach, S3 configured for backups, and a CloudWatch alarm in place.
This is the kind of environment you will be asked to build or maintain in a real job.

Week 10, Session 20: Docker Fundamentals

What Is a Container?

A container is a lightweight, isolated environment for running an application. It packages the application code together with everything it needs to run: its runtime, libraries, and configuration. The result is a unit that runs identically on any machine that has Docker installed — your laptop, a colleague's laptop, a staging server, or a production EC2 instance.

The key difference from a VM is that containers share the host operating system's kernel. They do not need their own full OS, which makes them start in seconds and use far less memory than VMs. A server that might run 5 VMs can run 50 containers.

	Virtual Machine	Container
OS	Full OS per VM	Shares host kernel
Startup time	Minutes	Seconds
Size	Gigabytes	Megabytes
Isolation	Strong (separate kernel)	Process-level isolation
Use case	Full environment isolation	Application packaging and deployment

Key Concepts

- Image: A read-only template that describes what a container should contain — the OS layer, dependencies, application code, and the command to run. Think of it as a blueprint.
- Container: A running instance of an image. Just like a VM is a running instance of a VM template.
- Dockerfile: A text file with instructions for building a Docker image. You define it, Docker builds it.
- Docker Hub: A public registry where images are stored and shared. Official images exist for Ubuntu, Nginx, MySQL, Node.js, Python, and thousands more.

Installing Docker

```
# Install Docker on Ubuntu:
sudo apt update
sudo apt install -y ca-certificates curl gnupg

# Add Docker's official GPG key and repository:
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o
/usr/share/keyrings/docker.gpg
echo 'deb [arch=amd64 signed-by=/usr/share/keyrings/docker.gpg]
https://download.docker.com/linux/ubuntu focal stable' | sudo tee
/etc/apt/sources.list.d/docker.list

sudo apt update
sudo apt install -y docker-ce docker-ce-cli containerd.io

# Start and enable Docker:
sudo systemctl start docker
sudo systemctl enable docker

# Allow your user to run Docker without sudo:
sudo usermod -aG docker $USER
# Log out and back in for this to take effect

# Verify Docker works:
docker --version
docker run hello-world
```

Core Docker Commands

```
# Pull an image from Docker Hub:
docker pull nginx
docker pull ubuntu:22.04

# List images you have locally:
docker images

# Run a container:
docker run nginx
# This runs nginx but you cannot interact with it. Ctrl+C to stop.

# Run in detached mode (background):
docker run -d nginx

# Run with port mapping (host_port:container_port):
docker run -d -p 8080:80 nginx
# Now visit http://localhost:8080 and you will see the Nginx welcome page

# Run interactively (useful for exploring):
docker run -it ubuntu:22.04 bash
# This drops you into a bash shell inside the container
# Type 'exit' to leave

# List running containers:
docker ps

# List all containers including stopped ones:
docker ps -a

# Stop a running container:
docker stop <container_id>

# Remove a stopped container:
docker rm <container_id>

# Remove an image:
docker rmi nginx

# View logs from a container:
docker logs <container_id>
docker logs -f <container_id>   # Follow in real time

# Run a command inside a running container:
docker exec -it <container_id> bash
```

Writing a Dockerfile

A Dockerfile is how you build your own custom image. Each line is an instruction that adds a layer to the image.

```
# Dockerfile — a simple Node.js web application

# Start from an official base image:
FROM node:18-alpine

# Set the working directory inside the container:
WORKDIR /app

# Copy dependency files first (Docker caches this layer):
COPY package*.json ./

# Install dependencies:
RUN npm install --production

# Copy the rest of the application code:
COPY . .

# Tell Docker which port this app listens on (documentation only):
EXPOSE 3000

# The command to run when the container starts:
CMD ['node', 'server.js']

# --- Build and run the image ---
# Build:  docker build -t myapp:v1 .
#         -t sets the image name and tag
#         . tells Docker to look for the Dockerfile in the current directory

# Run:    docker run -d -p 3000:3000 myapp:v1
# Visit:  http://localhost:3000
```

Volumes — Persisting Data

Containers are ephemeral by default. When a container is removed, any data it wrote inside itself is gone. Volumes solve this by mounting a directory from the host machine into the container.

```
# Mount a host directory into a container:
docker run -d \
  -p 8080:80 \
  -v /home/ubuntu/website:/usr/share/nginx/html \
  nginx
# Files in /home/ubuntu/website on the host are served by Nginx in the container
# Changes to those files appear immediately without restarting the container
```

```
# Create a named volume (Docker manages the location):
docker volume create mydata
docker run -d -v mydata:/var/lib/mysql mysql:8

# List volumes:
docker volume ls

# Inspect a volume:
docker volume inspect mydata
```

Docker Networking Basics

```
# By default, containers on the same Docker network can reach each other by name.

# Create a custom network:
docker network create myapp-network

# Run containers on the same network:
docker run -d --name db --network myapp-network mysql:8
docker run -d --name web --network myapp-network -p 80:80 nginx

# The 'web' container can now reach 'db' using the hostname 'db'
# e.g. mysql -h db -u root -p   (from inside the web container)

# List networks:
docker network ls
```

WHY DOCKER MATTERS FOR DEVOPS

Docker is the entry point to the entire container ecosystem — Kubernetes, ECS, and most CI/CD pipelines use containers as the unit of deployment. When your GitHub Actions pipeline builds and pushes a Docker image, then a container orchestrator runs it, that is the standard production deployment pattern at most companies right now. What you learned here is the foundation for all of it.

PHASE 5

DevOps Fundamentals

Weeks 11 to 12 • Sessions 21 to 24

DevOps is the practice of automating the processes that get software from a developer's laptop into production. The tools change, but the principles do not: automate repetitive work, catch problems early, deploy often, and make the process reliable enough that deploying is not a stressful event.

Week 11, Session 21: Version Control with Git & GitHub

Why Version Control?

Without version control, every software project eventually becomes chaos. Files get overwritten, changes are lost, nobody knows who changed what or when, and rolling back a bad change means hoping someone made a manual backup.

Git solves all of this. It tracks every change to every file, stores a complete history, and lets multiple people work on the same codebase simultaneously without overwriting each other's work.

Git Setup and Core Commands

```
# One-time setup on each machine:
git config --global user.name 'Your Name'
git config --global user.email 'you@example.com'
git config --global core.editor 'nano'   # Set your preferred editor

# Initialize a new repository:
mkdir myproject && cd myproject
git init

# Clone an existing repository from GitHub:
git clone https://github.com/username/repo-name.git
cd repo-name

# Check the current state of your working directory:
git status
# Shows: modified files, staged files, untracked files

# Stage changes for the next commit:
git add filename.txt           # Stage a specific file
git add .                      # Stage all changes in the current directory

# Commit staged changes:
git commit -m 'Add user authentication module'
# A good commit message is short, present tense, and describes WHAT changed

# View commit history:
```

```
git log --oneline

# Push commits to GitHub:
git push origin main

# Pull the latest changes from GitHub:
git pull origin main

# See what changed in a file:
git diff filename.txt

# Undo changes to a file (before staging):
git restore filename.txt
```

Branching

A branch is an independent line of development. The main branch holds stable, production-ready code. New features and fixes go on separate branches so the main branch is never broken by work in progress.

```
# Create a new branch and switch to it:
git checkout -b feature/user-login
# -b creates the branch and switches in one step

# List all branches:
git branch
# * marks your current branch

# Push the branch to GitHub:
git push origin feature/user-login

# Switch between branches:
git checkout main
git checkout feature/user-login

# Merge a branch into main:
git checkout main
git merge feature/user-login

# Delete a branch after merging:
git branch -d feature/user-login

# Delete it on GitHub too:
git push origin --delete feature/user-login
```

Pull Requests

A Pull Request (PR) is a request to merge your branch into another branch.

On GitHub it creates a review page where teammates can comment on specific lines of code, request changes, and approve the merge.

In professional teams, no code goes directly to main without a PR. This is how mistakes are caught before they reach production.

Week 11, Session 22: CI/CD Fundamentals

Continuous Integration

CI means every time a developer pushes code, an automated system immediately runs tests and checks against that code. If the code breaks something, the developer finds out within minutes. The rule in CI is: the main branch is always in a working state. Automated tests enforce this.

Continuous Delivery and Deployment

CD extends CI by automatically deploying code that has passed all tests. Continuous Delivery deploys to a staging environment automatically and waits for a human approval before production. Continuous Deployment goes all the way to production without human intervention — only possible with very mature test coverage.

Pipeline Stage	What Happens	Who or What Does It
Developer pushes code	Code sent to GitHub	Developer
CI triggers	Automated pipeline starts immediately	GitHub Actions (automatic)
Lint and tests run	Code style, unit tests, security scans	GitHub Actions runners
Build	Code compiled or packaged for deployment	GitHub Actions
Deploy to staging	Deploy to a test environment	GitHub Actions (automatic)
Deploy to production	Deploy to live environment	Automatic or after manual approval

Week 12, Session 23: Introduction to YAML

YAML stands for 'YAML Ain't Markup Language' — a slightly tongue-in-cheek name. It is a human-readable data format used for configuration files across almost every modern DevOps tool. GitHub Actions workflows, Docker Compose files, Kubernetes manifests, and Ansible playbooks are all written in YAML.

YAML files use the .yml or .yaml extension. The format relies on indentation (spaces, never tabs) to represent structure — similar to how Python uses indentation instead of curly braces.

YAML Syntax Basics

```
# Comments start with #

# Key-value pairs (the most basic building block):
name: FirstByte
version: 1.0
active: true

# Strings usually do not need quotes, but can have them:
description: 'A DevOps bootcamp'
title: "Systems Engineering Fundamentals"

# Numbers and booleans:
port: 8080
debug: false

# Nested structures use indentation (2 spaces is standard):
database:
  host: localhost
  port: 5432
  name: myapp_db

# Lists use a dash:
fruits:
  - apple
  - banana
  - orange

# Lists of objects (common in GitHub Actions):
steps:
  - name: Checkout code
    uses: actions/checkout@v3
  - name: Run tests
    run: npm test
```

The Golden Rules of YAML

- Indentation matters. Two spaces is the standard convention. Never mix tabs and spaces — this is the single most common cause of YAML errors.
- Indentation defines nesting. A key indented further than another key is nested inside it.
- Lists use a hyphen followed by a space: '- item'.
- Strings generally do not need quotes, but use them when a value contains a colon, starts with a special character, or could be misread as another type (e.g. a version number like '1.0' that you want treated as text, not a number).
- The pipe symbol '|' preserves line breaks in a multi-line string — this is how GitHub Actions runs multi-line shell scripts inside 'run:' blocks.

A YAML Example You Have Already Seen

This is a simplified piece of a GitHub Actions workflow. Read it now purely as YAML — ignore what it does for a moment and just trace the structure.

```
name: Deploy Website

on:
  push:
    branches:
      - main

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3
      - name: Deploy to S3
        run: aws s3 sync . s3://my-bucket
```

Trace it level by level: 'name' and 'on' and 'jobs' are top-level keys. Under 'on', 'push' is nested. Under 'push', 'branches' is nested, and it contains a list with one item: 'main'. Under 'jobs', 'deploy' is a job. Under 'deploy', 'runs-on' and 'steps' are nested. 'steps' is a list, and each item in that list is itself an object with 'name' and either 'uses' or 'run' keys.

Common YAML Mistakes

```
# MISTAKE 1: Mixing tabs and spaces (invisible but breaks everything)
# Always configure your editor to insert spaces, not tabs, for .yml files.

# MISTAKE 2: Inconsistent indentation
jobs:
  deploy:
      runs-on: ubuntu-latest   # WRONG — should be 4 spaces, not 6, to align
under 'deploy:'

# MISTAKE 3: Forgetting the space after the colon
name:FirstByte   # WRONG — needs a space: name: FirstByte
```

```
# MISTAKE 4: Forgetting the dash-space for list items
steps:
- name: Checkout   # technically valid but inconsistent with the rest of the file

# MISTAKE 5: Unquoted special characters
time: 12:00   # YAML may misinterpret this — quote it: time: '12:00'
```

VALIDATE BEFORE YOU COMMIT

A free tool worth showing students: yamllint.com or the YAML extension in VS Code.
Both will catch indentation errors before students push a broken workflow file to GitHub and waste a CI run discovering it failed on syntax alone.
Make this part of the routine: write YAML, validate it, then commit.

PRACTICAL — Session 22

Take the GitHub Actions workflow from Session 23 (next session) and, working backwards, identify every YAML construct in it: key-value pairs, nested mappings, lists, and multi-line strings using the pipe symbol.
Write a YAML file from scratch describing a simple personal project: name, version, a list of three dependencies, and a nested 'author' object with name and email fields. Validate it with yamllint before moving on.

Week 12, Session 24: GitHub Actions

How GitHub Actions Works

GitHub Actions is a CI/CD platform built into GitHub. You define workflows in YAML files inside your repository under .github/workflows/. These workflows run automatically on GitHub-hosted virtual machines called runners.

- Workflow: The overall automation defined in a .yml file.
- Trigger (on): The event that starts the workflow. Most common: push, pull_request, schedule.
- Job: A group of steps that runs on one runner. Multiple jobs can run in parallel.
- Step: An individual command or pre-built action inside a job.
- Action: A reusable automation unit. GitHub publishes official ones at github.com/actions.

A Complete Deployment Workflow

```
# File Name: .github/workflows/deploy.yml

name: Deploy Website to S3

on:
  push:
    branches:
      - main    # Only run when code is pushed to main

jobs:
```

```
  deploy:
    runs-on: ubuntu-latest    # GitHub provides this Ubuntu VM for free

    steps:
      # 1. Check out the repository code
      - name: Checkout repository
        uses: actions/checkout@v3

      # 2. Validate HTML files
      - name: Validate HTML
        run: |
          sudo apt-get install -y tidy
          find . -name '*.html' -exec tidy -q -e {} \;

      # 3. Run a simple link check
      - name: Check for obvious issues
        run: |
          echo 'Checking file sizes...'
          find . -name '*.html' -size +1M && echo 'WARNING: Large HTML files found'
          echo 'Pre-deploy checks passed'

      # 4. Configure AWS credentials from GitHub Secrets
      - name: Configure AWS credentials
        uses: aws-actions/configure-aws-credentials@v2
        with:
          aws-access-key-id:     ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region:            us-east-1

      # 5. Sync website files to S3
      - name: Deploy to S3
        run: |
          aws s3 sync . s3://${{ secrets.S3_BUCKET_NAME }} \
            --exclude '.git/*' \
            --exclude '.github/*' \
            --delete

      # 6. Confirm deployment
      - name: Confirm deployment
        run: echo 'Deployment complete at' $(date)

# IMPORTANT:
# AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY, and S3_BUCKET_NAME
# are stored as encrypted secrets in GitHub.
# Go to: Repository Settings > Secrets and variables > Actions > New repository
secret
# Never paste real credentials directly in this file.
```

GITHUB SECRETS The ${{ secrets.MY_SECRET }} syntax pulls values from GitHub's encrypted secrets store. Once stored, secret values are never visible again — not even to you. This is how you use credentials in pipelines without exposing them in your code.

Week 12, Session 25: Final Capstone Project

What You Are Building

The capstone is a practical demonstration that you can bring everything from this course together independently. You will build a complete system from scratch, show it working end to end, and present the decisions you made.

Component	What Is Required
Linux Server	EC2 instance on AWS running Ubuntu, configured with a non-root user, SSH key authentication
Web Server	Nginx installed and serving content, with reverse proxy if you have an application backend
Git Repository	GitHub repo with a clear branching structure and meaningful commit history
CI/CD Pipeline	GitHub Actions workflow that deploys to AWS automatically on push to main
AWS Networking	Custom VPC with public and private subnets, security groups correctly scoped
IAM	IAM role used by the EC2 instance instead of hardcoded credentials, least-privilege policy
Monitoring	CloudWatch alarm configured on the EC2 instance, at minimum CPU utilization
Live Demo	Push code, pipeline runs, website updates — no manual steps required
Presentation	Short walkthrough explaining architecture decisions and what you would improve

CAPSTONE ADVICE
Start at least a week early. Do not try to build everything in the last session.
Build one layer at a time: server first, then web server, then Git, then the pipeline.
When something breaks, use the troubleshooting sequence from Phase 2.
The presentation matters. You will spend your career explaining technical decisions. Practice doing it clearly, even if it is just to yourself out loud.

Review Questions

For each question, write or say your answer out loud without looking at the notes. If you get stuck, go back to the relevant section. These are not academic questions — they are the kinds of things you will be asked in interviews and in real jobs.

Phase 1 — IT Fundamentals & Operations

1. What is the difference between a physical server and a virtual machine? If a company can use a VM, why would they ever need a physical server?
2. Walk through exactly what happens from the moment you type google.com into a browser to the moment the page appears.
3. What is the difference between a public IP and a private IP? Give an example of each. If your laptop has a private IP, how does it reach the internet?
4. A colleague says the firewall is blocking something. What information do you need to diagnose and fix this?
5. What is SSH and why is key-pair authentication better than password authentication for a server?
6. You have just launched an EC2 instance. What three pieces of information do you need before you can SSH into it?
7. What is the difference between a commit and a push in Git?
8. Three developers are working on different features at the same time on the same codebase. How does Git branching manage this?

Phase 2 — Linux System Administration

1. You get a call that a web server is responding slowly. Describe the first three commands you would run and what you would look for in each output.
2. A new developer needs SSH access to a server. Walk through every step from creating their account to them being able to log in.
3. What is the difference between chmod 755 and chmod 600? When would you use each?
4. A service fails to start after a reboot. Write the exact sequence of commands you would run to find out why.
5. Write a crontab entry that runs /home/ubuntu/scripts/backup.sh every day at 3:30 AM.
6. What is the difference between apt update and apt upgrade? What happens if you run upgrade without update first?

Phase 3 — Systems Engineering & Web Services

1. A user reports a 502 Bad Gateway error. What does this error mean technically and what are the three most likely causes?
2. What is a reverse proxy? Describe the path a request takes from a browser through Nginx to an application.
3. Why does every production application need monitoring? What is the difference between logs and metrics?
4. Your database backup script runs nightly. The next morning the backup file is 0 bytes. How do you investigate?
5. Explain the difference between HTTP and HTTPS at a technical level.
6. A web server returns a 502 Bad Gateway error. What could cause this and where do you start?
7. Why is monitoring important and what is the difference between logs and metrics?

Phase 4 — Cloud Engineering

1. Explain the AWS Shared Responsibility Model in your own words. Give two things AWS is responsible for and two things you are responsible for.
2. What is the difference between a public subnet and a private subnet? Which one would you put a database in and why?

3. Your company's web application gets 10x more traffic on Monday mornings. What AWS services would you use to handle this automatically?
4. A developer accidentally committed AWS access keys to a public GitHub repo. What do you do, and in what order?
5. What is an IAM role and how is it different from an IAM user? When would you give an EC2 instance a role?
6. You need to allow only your office IP address to SSH into an EC2 instance. How do you do that?
7. What is Auto Scaling and why would you use it?

Phase 5 — DevOps Fundamentals

1. A GitHub Actions pipeline fails at the 'Deploy to S3' step. What are the three most likely causes and how do you investigate each?
2. Explain what CI/CD means and the problem it solves compared to deploying software manually.
3. Find the indentation error in a short YAML snippet provided to you. Explain why it would cause the pipeline to fail.
4. Why is YAML used for GitHub Actions, Docker Compose, and Kubernetes instead of a format like JSON?
5. Walk through your capstone architecture. Explain why you made the networking decisions you did and what you would change if this were a real production system serving 10,000 users per day.

Glossary of Key Terms

Term	Definition
Auto Scaling	AWS feature that automatically adds or removes EC2 instances based on demand.
Availability Zone (AZ)	A physically isolated data center within an AWS Region. Multiple AZs provide redundancy.
Bash	The default shell on most Linux systems, used for command-line work and scripting.
Branch (Git)	An independent line of development in a Git repository, isolated from the main codebase.
CI/CD	Continuous Integration and Continuous Delivery. Automating the testing and deployment of code.
CIDR	Classless Inter-Domain Routing. Notation for specifying IP address ranges, e.g. 10.0.0.0/16.
CloudWatch	AWS monitoring service that collects metrics and logs from AWS resources and triggers alarms.
Commit (Git)	A saved snapshot of changes with a message describing what changed and why.
Cron job	A scheduled task on Linux that runs a command or script at specified times automatically.
DHCP	Dynamic Host Configuration Protocol. Assigns IP addresses automatically to devices on a network.
DNS	Domain Name System. Translates human-readable domain names into IP addresses.
EC2	Elastic Compute Cloud. AWS virtual machine service.
Firewall	A network security tool that allows or blocks traffic based on configured rules.
Git	A distributed version control system that tracks every change to files over time.
GitHub Actions	A CI/CD automation platform built into GitHub using YAML workflow files.
Hypervisor	Software that creates and manages virtual machines on a physical host machine.
IAM	Identity and Access Management. AWS service that controls who can access what.
Internet Gateway	AWS component that enables communication between a VPC and the internet.
IP Address	A numerical label identifying a device on a network.
LAN	Local Area Network. A network confined to a single physical location.
Load Balancer	Distributes incoming network traffic across multiple servers to prevent overload.
NAT	Network Address Translation. Allows private-network devices to share a single public IP.
Nginx	A high-performance web server and reverse proxy. Widely used in production environments.
PID	Process ID. A unique number assigned by the OS to each running process.
Pull Request (PR)	A request on GitHub to merge one branch into another, typically reviewed by teammates.

Reverse Proxy	A server that forwards client requests to application servers, hiding the backend.
S3	Simple Storage Service. AWS object storage for files, backups, and static websites.
Security Group	A stateful AWS firewall attached to EC2 instances, controlling inbound and outbound traffic.
SSH	Secure Shell. An encrypted protocol for remotely accessing and managing Linux servers.
Subnet	A segment of a VPC. Public subnets route to the internet; private subnets do not.
sudo	Linux command that runs another command with root (administrator) privileges.
systemctl	The command-line tool for managing services on Linux systems running systemd.
VPC	Virtual Private Cloud. An isolated private network in AWS for your resources.
VM (Virtual Machine)	A software-defined computer running on a physical host via a hypervisor.

PHASE 6

Terraform & Infrastructure as Code (Bonus Phase)

No fixed timeline • 2 Sessions • Self-paced

This phase is for students who have completed the 12-week program and want to go further. Terraform is the industry-standard tool for Infrastructure as Code (IaC) — the practice of defining and managing infrastructure through code instead of clicking through a console. It is one of the most in-demand skills in cloud and DevOps roles right now.

PREREQUISITE Complete all 12 weeks including the capstone before starting this phase. You need a solid understanding of AWS networking, EC2, S3, IAM, and Security Groups before Terraform will make sense. If you try to learn Terraform before you understand what it is provisioning, you will get lost.

Bonus Session 1: Terraform Fundamentals

What Is Infrastructure as Code?

Right now, if you want to create an EC2 instance, you log into the AWS console and click through a series of menus. Infrastructure as Code means instead of clicking, you write a configuration file that describes the infrastructure you want, and a tool like Terraform reads that file and creates it for you.

The advantages of this approach are significant:

- Reproducibility: Running the same Terraform config in any AWS account produces the same infrastructure every time. No more 'it works in staging but not in production'.
- Version control: Your infrastructure configuration lives in Git. You can see who changed what and when, roll back to a previous state, and review infrastructure changes the same way you review code.
- Automation: CI/CD pipelines can apply Terraform automatically. No more manual provisioning steps before a deployment.
- Documentation: The Terraform files are a live, accurate description of what infrastructure exists. No more outdated Confluence pages.

Installing Terraform

```
# Install on Ubuntu:
sudo apt update && sudo apt install -y gnupg software-properties-common

wget -O- https://apt.releases.hashicorp.com/gpg | sudo gpg --dearmor -o
/usr/share/keyrings/hashicorp-archive-keyring.gpg

echo 'deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg]
https://apt.releases.hashicorp.com focal main' | sudo tee
/etc/apt/sources.list.d/hashicorp.list

sudo apt update && sudo apt install terraform

# Verify:
terraform --version
```

Core Terraform Concepts

Concept	What It Is
Provider	A plugin that lets Terraform talk to a specific platform. You use the AWS provider to manage AWS resources.
Resource	A specific piece of infrastructure you want Terraform to create: an EC2 instance, an S3 bucket, a Security Group.
State	Terraform keeps a state file that records what infrastructure it has created. It uses this to know what needs to change.
Plan	A preview of what Terraform is about to do before it does it. Always review the plan before applying.
Apply	The command that actually creates, updates, or destroys infrastructure based on your configuration.
Module	A reusable group of resources. Like a function in programming — define it once, use it in many places.

Terraform Workflow

```
# The three commands you will use constantly:

terraform init
# Downloads the required providers and sets up the working directory.
# Run this once when you start a new project or add a new provider.

terraform plan
# Shows you exactly what Terraform will create, modify, or destroy.
# Read this carefully before every apply. It is your safety net.

terraform apply
# Applies the plan and creates the infrastructure.
# Terraform asks for confirmation before doing anything.

# Other useful commands:
terraform destroy    # Destroy all infrastructure in the current config
terraform fmt        # Auto-format your .tf files to standard style
terraform validate   # Check your config for syntax errors
terraform show       # Show the current state
terraform output     # Show output values defined in your config
```

HCL — HashiCorp Configuration Language

Terraform configurations are written in HCL. It is designed to be human-readable and concise. Here is the basic structure:

```
# Every Terraform file ends in .tf
# Terraform reads all .tf files in the current directory
```

```
# 1. Configure the provider:
terraform {
  required_providers {
    aws = {
      source  = 'hashicorp/aws'
      version = '~> 5.0'
    }
  }
}

provider 'aws' {
  region = 'eu-west-1'
  # Terraform uses your AWS CLI credentials automatically
  # Never hardcode access keys here
}

# 2. Define a resource:
resource 'aws_instance' 'web_server' {
  ami           = 'ami-0694d931cee176e7d'   # Ubuntu 22.04 in eu-west-1
  instance_type = 't2.micro'

  tags = {
    Name        = 'FirstByte-Web'
    Environment = 'dev'
  }
}

# 3. Variables make configs reusable:
variable 'environment' {
  description = 'Deployment environment'
  type        = string
  default     = 'dev'
}

# Reference a variable:
# var.environment

# 4. Outputs expose values after apply:
output 'instance_public_ip' {
  description = 'Public IP of the web server'
  value       = aws_instance.web_server.public_ip
}
```

STATE FILE WARNING The Terraform state file (terraform.tfstate) contains sensitive information about your infrastructure. Never commit it to a public Git repository. In team environments, store state remotely in an S3 bucket with a DynamoDB lock table — this prevents two people from applying Terraform simultaneously. You will set this up in the lab.

Bonus Session 2: Provisioning AWS Infrastructure with Terraform

What You Are Building

In this session you provision the same AWS environment you built manually in Phase 4 — but entirely in Terraform code. By the end, you should be able to destroy and recreate the entire environment with a single command.

Project Structure

```
# Recommended file layout for a Terraform project:

terraform-aws/
├── main.tf          # Core resources (EC2, VPC, etc.)
├── variables.tf     # Input variable definitions
├── outputs.tf       # Output value definitions
├── providers.tf     # Provider and backend configuration
└── terraform.tfvars # Variable values (do NOT commit if it contains secrets)
```

providers.tf — Backend and Provider Config

```
terraform {
  required_providers {
    aws = {
      source  = 'hashicorp/aws'
      version = '~> 5.0'
    }
  }

  # Remote state in S3 (recommended for any real project):
  backend 's3' {
    bucket         = 'firstbyte-terraform-state'
    key            = 'dev/terraform.tfstate'
    region         = 'eu-west-1'
    dynamodb_table = 'terraform-lock'
    encrypt        = true
  }
}

provider 'aws' {
  region = var.aws_region
}
```

variables.tf

```
variable 'aws_region' {
  description = 'AWS region to deploy into'
  type        = string
```

```
  default     = 'eu-west-1'
}

variable 'environment' {
  description = 'Environment name (dev, staging, prod)'
  type        = string
  default     = 'dev'
}

variable 'my_ip' {
  description = 'Your public IP for SSH access (format: x.x.x.x/32)'
  type        = string
}
```

main.tf — The Full Infrastructure

```
# ── VPC ──────────────────────────────────────────────
resource 'aws_vpc' 'main' {
  cidr_block           = '10.0.0.0/16'
  enable_dns_hostnames = true

  tags = { Name = '${var.environment}-vpc' }
}

# ── INTERNET GATEWAY ─────────────────────────────────
resource 'aws_internet_gateway' 'main' {
  vpc_id = aws_vpc.main.id
  tags   = { Name = '${var.environment}-igw' }
}

# ── PUBLIC SUBNET ────────────────────────────────────
resource 'aws_subnet' 'public' {
  vpc_id                  = aws_vpc.main.id
  cidr_block              = '10.0.1.0/24'
  availability_zone       = '${var.aws_region}a'
  map_public_ip_on_launch = true
  tags                    = { Name = '${var.environment}-public-subnet' }
}

# ── ROUTE TABLE ──────────────────────────────────────
resource 'aws_route_table' 'public' {
  vpc_id = aws_vpc.main.id

  route {
    cidr_block = '0.0.0.0/0'
    gateway_id = aws_internet_gateway.main.id
  }
```

```
  tags = { Name = '${var.environment}-public-rt' }
}

resource 'aws_route_table_association' 'public' {
  subnet_id      = aws_subnet.public.id
  route_table_id = aws_route_table.public.id
}

# ── SECURITY GROUP ──────────────────────────────────────
resource 'aws_security_group' 'web' {
  name   = '${var.environment}-web-sg'
  vpc_id = aws_vpc.main.id

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = 'tcp'
    cidr_blocks = [var.my_ip]   # SSH from your IP only
  }

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = 'tcp'
    cidr_blocks = ['0.0.0.0/0']
  }

  ingress {
    from_port   = 443
    to_port     = 443
    protocol    = 'tcp'
    cidr_blocks = ['0.0.0.0/0']
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = '-1'
    cidr_blocks = ['0.0.0.0/0']
  }

  tags = { Name = '${var.environment}-web-sg' }
}

# ── EC2 INSTANCE ────────────────────────────────────────
resource 'aws_instance' 'web' {
  ami                    = 'ami-0694d931cee176e7d'   # Ubuntu 22.04 eu-west-1
  instance_type          = 't2.micro'
```

```
  subnet_id              = aws_subnet.public.id
  vpc_security_group_ids = [aws_security_group.web.id]
  key_name               = 'my-key-pair'

  user_data = <<-EOF
    #!/bin/bash
    apt update -y
    apt install -y nginx
    systemctl start nginx
    systemctl enable nginx
  EOF

  tags = { Name = '${var.environment}-web-server' }
}

# ── S3 BUCKET ──────────────────────────────────────────
resource 'aws_s3_bucket' 'backups' {
  bucket = 'firstbyte-${var.environment}-backups'
  tags   = { Name = '${var.environment}-backups' }
}

resource 'aws_s3_bucket_versioning' 'backups' {
  bucket = aws_s3_bucket.backups.id
  versioning_configuration { status = 'Enabled' }
}
```

outputs.tf

```
output 'web_server_public_ip' {
  description = 'Public IP of the web server'
  value       = aws_instance.web.public_ip
}

output 'web_server_public_dns' {
  description = 'Public DNS of the web server'
  value       = aws_instance.web.public_dns
}

output 's3_bucket_name' {
  description = 'Name of the S3 backup bucket'
  value       = aws_s3_bucket.backups.bucket
}
```

Deploying and Destroying

```
# Set your IP as a variable:
export TF_VAR_my_ip=$(curl -s ifconfig.me)/32
```

```
# Initialize:
terraform init

# Preview what will be created:
terraform plan

# Deploy:
terraform apply
# Type 'yes' when prompted

# After apply, Terraform prints your outputs:
# web_server_public_ip = '54.123.45.67'

# SSH in:
ssh -i ~/.ssh/my-key.pem ubuntu@54.123.45.67

# When you are done, destroy everything:
terraform destroy
# Type 'yes' when prompted
# Terraform removes every resource it created, in the correct order.
# This is one of the biggest advantages over manual provisioning.
```

PHASE 6 CHECKPOINT

You have completed Phase 6 when you can:

1. Write a Terraform config from scratch that provisions a VPC, subnet, security group, and EC2 instance
2. Run terraform plan and accurately predict what it will do before it does it
3. Explain what the state file is and why it should never be in a public repo
4. Destroy and recreate the same environment reproducibly
5. Explain to someone else why IaC matters compared to manual provisioning

If you reach this point, you have skills that many people applying for junior DevOps roles do not have.

A Final Word

The engineers who succeed in this field are not the ones who memorised the most commands.

They are the ones who understand why things work, not just how to make them work.

Ask why. Break things deliberately. Read error messages carefully. Google relentlessly.

Welcome to FirstByte.