

The Cloud Engineer Course

From zero to job-ready cloud engineer — a B4LCILC course

August 13, 2026

The Cloud Engineer Course

From zero knowledge to genuinely job-ready as a cloud engineer.

How this course works

You are training to *be* a cloud engineer: the person who builds servers with code, keeps systems alive at 3 a.m., automates away boring work, and can look at a broken system and calmly find out why. That is a hands-on trade, and this course treats it like one. You will spend far more time typing than reading — every module has labs, and the labs are the course. Reading about the cloud builds recognition; building in the cloud builds a career.

The course has five stages. **Stage 0 (Weeks 1–4): Foundations** — how computers and networks actually work, and Linux, the operating system the cloud runs on. **Stage 1 (Weeks 5–12): Core cloud, hands-on** — the five AWS services that appear in every job description, learned as labs you build and tear down. **Stage 2 (Weeks 13–20): Automation** — scripting, Git, Terraform, CI/CD, and containers: the skills that separate an engineer from a console-clicker. **Stage 3 (Weeks 21–28): Operations** — monitoring, incidents, security, and cost: the skills you're actually paid for. **Stage 4 (Months 8–12): Job-readiness** — three portfolio projects, three certifications, and interview training.

Rules for the whole course: study **1.5–2 hours a day, six days a week** — consistency beats intensity, and this is a trade learned by daily repetition, like a musical instrument. Every module ends with a **Vocabulary table** (the words you must own), a **Say-it-out-loud drill** (sentences you repeat until they're natural — job interviews are spoken, not written), hands-on **Exercises**, and a **Milestone** that gates progress: do not move on until you can do it, because every stage stands on the one before. From Week 1, keep an **Engineering Journal**: every lab, every error message, every fix, one honest paragraph. By Month 10 that journal becomes your portfolio's raw material and your interview stories.

One promise in return: nothing in this course assumes you know anything. Every term is defined the first time it appears. If you can use a web browser and you're willing to type commands that feel alien for the first two weeks, you have all the prerequisites there are.

The job description you are training to

In August 2026 we pulled real Cloud Engineer job descriptions from employer templates and hiring guides (Arc.dev, Wiz, DevsData, XOPA, Betterteam — full list in Sources). Strip away the company names and the same requirements appear again and again. This table is your contract with the course — every bullet an employer asks for maps to the stage that teaches it:

What real job descriptions ask for (near-verbatim)	Where this course teaches it
“Knowledge of Linux/Unix operating systems”	Stage 0, Module 1
“Expertise in cloud networking including VPCs, subnets, load balancers, DNS”	Stage 0 Module 2 + Stage 1 Lab 3
“Demonstrated expertise in core AWS services, including EC2, S3, RDS, VPC, IAM”	Stage 1 (Labs 1–5)
“Design, develop, and deploy cloud infrastructure using infrastructure as code tools such as Terraform, CloudFormation”	Stage 2, Module 7
“Proficiency in scripting languages such as Python, Bash, PowerShell, or Go”	Stage 2, Module 6
“Build and maintain CI/CD pipelines using tools like Jenkins, GitLab CI, or GitHub Actions”	Stage 2, Module 8
“Experience with containerization technologies including Docker and Kubernetes”	Stage 2, Module 8
“Monitor infrastructure health and performance using cloud-native monitoring tools”	Stage 3, Module 9
“Participate in incident response, including log analysis”; “troubleshooting and analytical skills”	Stage 3, Module 10
“Implement and enforce security controls including encryption, identity and access management”; “least-privilege access”	Stage 1 Lab 5 + Stage 3 Module 10
“Manage cloud costs through rightsizing resources, implementing auto-scaling, resource tagging”	Stage 3, Module 10
“Maintaining, testing and implementing disaster recovery procedures”	Stage 3 + Portfolio Project 3
“AWS certifications preferred”	Stage 4 cert schedule (CCP → SAA → Terraform Associate)
“Provide technical guidance and documentation”; “good communication and collaboration skills”	Journal + runbooks + Stage 4 interview prep

Salary context, so you know what you’re working toward: US cloud engineer salaries cluster around a **median near \$104,000**, with a typical range of **\$85K–\$140K** and senior/specialist AWS roles advertised well above that. Entry-level roles exist under many names — cloud

support associate, junior cloud engineer, cloud operations engineer — and this course targets exactly their requirements.

STAGE 0 — FOUNDATIONS (Weeks 1-4)

Module 1 (Weeks 1-2): How computers work, and Linux — the language of servers

The big idea: a **server** is just a computer whose job is to serve other computers. The laptop in front of you and the machines running Netflix differ in size and reliability, not in kind: both are a **CPU** (the part that does the work), **memory/RAM** (fast short-term workspace, wiped on restart), **disk** (slow long-term storage that survives restarts), and a **network card** (the connection to everything else), all coordinated by an **operating system (OS)**. Your laptop probably runs Windows or macOS. Servers overwhelmingly run **Linux** — a free, open-source OS that is stable, scriptable, and controlled entirely by typed commands. That last part is the point: you cannot automate mouse clicks, but you can automate commands, and cloud engineering is automation. So your first fluency is the **terminal**.

The terminal, demystified. The terminal (or “shell” — the program inside it is usually **Bash**) is a text conversation with the computer. You type a command; it answers. That’s all. The \$ you’ll see in examples is the **prompt** — the shell saying “your turn.” Everything in Linux is a file, files live in a single tree starting at the root /, and your personal folder is /home/yourname (nickname: ~). A **path** is a file’s address in that tree: /home/anna/notes.txt.

Get a Linux to practice on (pick one, ten minutes): on Windows, install **WSL** (Windows Subsystem for Linux — a real Ubuntu Linux inside Windows: <https://learn.microsoft.com/en-us/windows/wsl/install>); on a Mac, the built-in Terminal app is close enough to start (macOS is a Unix cousin); or wait for Week 5 when you’ll rent a real Linux server from AWS for free. WSL is the best answer for most people.

The top 25 commands — this table is Weeks 1-2. Type every one of them, many times:

Command	What it does	Example
pwd	Print working directory — “where am I?”	pwd
ls	List files here (-l long detail, -a include hidden)	ls -la
cd	Change directory — move around the tree	cd /var/log
mkdir	Make a directory (folder)	mkdir projects
touch	Create an empty file	touch notes.txt
cp	Copy a file (-r for folders)	cp a.txt backup.txt

Command	What it does	Example
mv	Move or rename	mv old.txt new.txt
rm	Remove — permanently, no trash can. Respect it.	rm notes.txt
cat	Print a file's whole contents	cat notes.txt
less	Read a long file page by page (q to quit)	less /var/log/syslog
head / tail	First / last lines of a file. tail -f watches a log live — an ops classic	tail -f app.log
grep	Search text for a pattern — the single most-used ops command	grep "ERROR" app.log
find	Find files by name/size/age	find / -name "*.conf"
echo	Print text (often into files or variables)	echo "hello"
nano	A friendly in-terminal text editor	nano notes.txt
man	The manual for any command (q to quit)	man grep
sudo	Run one command as the all-powerful admin ("root"). With respect.	sudo apt update
apt	Install/update software (Ubuntu/Debian family)	sudo apt install htop
chmod	Change a file's permissions	chmod 644 notes.txt
chown	Change a file's owner	sudo chown anna file
ps	List running processes (ps aux for all)	ps aux
top (or htop)	Live dashboard of CPU/memory/processes — "why is the server slow?" starts here	top
df -h / du -sh	Disk space free / space a folder uses — "the disk is full" starts here	df -h
ssh	Log into another machine's terminal over the network — <i>the</i> cloud engineer's front door	ssh anna@server-ip
curl	Make a web request from the	curl

Command	What it does	Example
	terminal — “is the site up?”	https://example.com

Permissions, the 60-second version. Every file has an owner and a mode like `rwxr-xr--`: three triplets — owner, group, everyone — of read, **w**rite, **e**xecute. In numbers: `r=4, w=2, x=1`, so `chmod 755 script.sh` means “owner can do everything ($7=4+2+1$); everyone else can read and run it ($5=4+1$).” When a program “doesn’t have permission,” this is the system saying no — and now you can read why.

SSH, the 60-second version. `ssh` opens a secure remote terminal on another machine — from your laptop into a server in Virginia as if you were sitting at it. Instead of passwords, professionals use a **key pair**: a private key (a secret file on your laptop, never shared) and a public key (placed on the server). They fit like key and lock. Every AWS server you launch in Stage 1 will hand you exactly this.

Processes and services: a **process** is a running program; a **service** (or “daemon”) is a process that runs forever in the background — web servers, databases. `systemctl status nginx` asks Linux “is the nginx service healthy?” — a sentence you will type professionally for years.

Vocabulary:

Term	Definition
Server	A computer whose job is to serve other computers. In the cloud, one you rent.
CPU / RAM / disk	The worker, the fast temporary workspace (wiped on restart), and the slow permanent storage.
Operating system (OS)	The software that runs the machine and hosts programs. Servers run Linux.
Linux / distribution	The free, open-source server OS. A “distro” (Ubuntu, Amazon Linux, Debian) is one packaged flavor of it.
Terminal / shell / Bash	The text interface to the OS / the program interpreting your commands / the standard shell’s name.
Prompt	The <code>\$</code> — the shell waiting for your command.
Directory / path	A folder / a file’s full address in the single tree that starts at <code>/</code> .
Root (two meanings)	The top of the file tree (<code>/</code>) <i>and</i> the all-powerful admin user. Context tells you which.
<code>sudo</code>	“Superuser do” — run one command with admin power.
Permissions (<code>rw</code> x)	Per-file rules for who may read, write, execute — shown as triplets for

Term	Definition
	owner/group/everyone.
Process / service (daemon)	A running program / one that runs forever in the background (web servers, databases).
SSH / key pair	Secure remote login to another machine's terminal / the private+public key files that replace passwords.
Log	A text file where software writes what happened — the first place you look when anything breaks.
Package manager	The OS's installer (apt, yum) — software by command, not by download page.

Videos for this module (links verified):

Video	Channel	Length	Link
Linux Operating System — Crash Course for Beginners	freeCodeCamp	~2 hr	https://www.youtube.com/watch?v=ROjZy1WbCIA
WSL install guide (reference, not video)	Microsoft Learn	—	https://learn.microsoft.com/en-us/windows/wsl/install

Say it out loud until natural: “Let me SSH in and check the logs.” · “Grep the log for the error, then tail -f it while we retry.” · “It’s a permissions problem — who owns the file and what’s the mode?” · “Check top — is it CPU, memory, or disk?”

Exercises: (1) In your Linux terminal, build a small project tree with `mkdir` and `touch`, copy and move things, then delete it — narrating each command out loud. (2) Create `hello.sh` containing `echo "hello from $(whoami)"`, make it executable with `chmod +x`, run it with `./hello.sh`. (3) Run `tail -f` on a log file (on Ubuntu: `sudo tail -f /var/log/syslog`) and watch lines arrive. (4) Journal: explain to an imaginary friend why servers run Linux, in three sentences.

Milestone: without notes, you can navigate anywhere in the file tree, create/copy/move/delete files, explain `chmod 755`, and use `grep` to find a word in a file. If any of those requires looking something up, spend two more days here. This module is load-bearing for everything.

Module 2 (Weeks 3–4): Networking fundamentals + your AWS account

The big idea: a network is computers passing addressed envelopes. Every machine gets an **IP address** (like `172.31.8.14` — a numeric street address). Data is chopped into **packets** (envelopes) and routed hop by hop toward the destination address. On arrival, a **port** number

says which program the envelope is for — the same building, thousands of numbered doors: port **22** is SSH, **80** is unencrypted web (**HTTP**), **443** is encrypted web (**HTTPS**), **5432** is PostgreSQL. “Open port 443” means “allow envelopes addressed to door 443.”

Private vs public: your home and every cloud network reuse **private IP ranges** (10.x.x.x, 172.16–31.x.x, 192.168.x.x) that only work inside the local network; a **public IP** is reachable from the whole internet. This split is the bedrock of cloud security: things that don’t need to face the internet get no public address at all.

DNS — the internet’s phone book. Humans use names (example.com); packets need numbers. **DNS** translates: your machine asks a DNS server “what’s the IP for example.com?”, gets the number, then connects. Half of all mysterious outages involve DNS; the industry joke “it’s always DNS” exists because it’s often true. `nslookup example.com` performs the lookup by hand.

HTTP — how the web talks. A **client** (browser) sends a request — a **method** (GET = fetch, POST = submit) plus a path — and the **server** replies with a **status code**: 200 OK, 301 moved, 403 forbidden, 404 not found, 500 server error, 502/503 “the server behind me is broken/overloaded.” Memorize those six; as an engineer you’ll read them daily. `curl -I https://example.com` shows you a live status line and headers.

Firewalls: a rules list deciding which packets may pass, by source, destination, and port — “allow 443 from anywhere; allow 22 only from the office; deny the rest.” In AWS the per-server firewall is called a **security group**, and misconfigured ones are the #1 beginner’s security hole. **Latency** (delay, in milliseconds) and **bandwidth** (capacity per second) round out the vocabulary: distance creates latency, which is why clouds have regions worldwide.

Now, your AWS account — the second half of this module. Go to <https://aws.amazon.com/free> and create a Free Tier account (email, phone, credit/debit card for identity — the card is not meaningfully charged if you follow this course’s teardown discipline). The **Free Tier** gives monthly allowances of the basics (including 750 hours/month of a small EC2 server in your first year) and every lab in this course is designed to fit inside it. Then, before anything else, three non-negotiable safety steps — doing these is the first exercise of your security career:

1. **MFA on the root user.** The root user is the account’s master key. Add multi-factor authentication (a phone authenticator app) in IAM → Security credentials. Then stop using root for daily work.
2. **Create an admin IAM user** (Lab 5 explains IAM deeply; for now: console → IAM → Users → create user with admin access) and log in as that user from now on.
3. **A billing alert.** Console → Billing → Budgets → create a **zero-spend budget** (AWS’s template that emails you the moment anything bills at all). Also turn on Free Tier usage alerts. An engineer who can’t control cost is a liability; you’re 20 minutes into your account and already ahead of many professionals. (Reference: <https://docs.aws.amazon.com/cost-management/latest/userguide/budgets-managing-costs.html>)

Vocabulary:

Term	Definition
IP address	A machine's numeric network address, e.g. 172.31.8.14.
Packet	One addressed envelope of data; all traffic is streams of them.
Port	A numbered door on a machine identifying which program traffic is for: 22 SSH, 80 HTTP, 443 HTTPS.
Private / public IP	An address valid only inside a local network / one reachable from the whole internet.
DNS	The system translating names (example.com) into IP addresses. "It's always DNS."
HTTP / HTTPS	The web's request-response protocol / the same, encrypted with TLS.
Status code	The server's reply summary: 200 OK, 404 not found, 500 server error, 503 overloaded.
Firewall	The rules list deciding which packets pass, by source, destination, port.
Security group	AWS's per-server firewall. Misconfiguring one is the classic beginner hole.
Latency / bandwidth	Delay (ms) / capacity (per second). Distance creates latency.
Client / server (roles)	The asker and the answerer in any network conversation.
AWS Free Tier	The monthly free allowance on a new AWS account — this course's entire budget.
Root user	The AWS account's master identity. MFA it, then stop using it.
Billing alert / budget	The automatic email when spend crosses a threshold. Yours is set to \$0.

Videos for this module:

Video	Channel	Length	Link
What is AWS?	Amazon Web Services (official)	~2 min	https://www.youtube.com/watch?v=a9_D53WsUs
Top 50+ AWS Services Explained in 10 Minutes	Fireship	~10 min	https://www.youtube.com/watch?

Video	Channel	Length	Link
AWS Networking Basics — VPC & Subnets	KodeKloud	~30 min	v=JlbiYCM48to https://www.youtube.com/watch?v=QM63dyA_4Pc

Say it out loud: “What’s the IP, and is it public or private?” · “Is port 443 open in the security group?” · “Curl it — what status code do you get?” · “Did DNS resolve? Check with nslookup before blaming the server.”

Exercises: (1) `ping google.com` (note the latency), `nslookup google.com` (note DNS returning several IPs), `curl -I https://aws.amazon.com` (read the status line and three headers, then look each up). (2) Find your machine’s private IP (`ip addr` on Linux) and your public IP (search “what is my IP”) — explain in your journal why they differ. (3) Complete the AWS account setup: MFA, admin user, zero-spend budget — screenshot the budget for your journal. (4) Draw from memory: laptop — DNS lookup — HTTPS request over port 443 — firewall — web server. Three times, until it’s automatic.

Milestone — end of Stage 0: you can narrate what happens when you type a URL and press Enter — DNS, IP, port, firewall, HTTP request, status code — in under two minutes, using every term correctly; and your AWS account exists with MFA and a \$0 budget alert. You now know more about how the internet works than most people who use it for a living. Stage 1 is where you start building on it.

STAGE 1 — CORE CLOUD, HANDS-ON (Weeks 5–12)

How this stage works. Each of the five services below is a **Lab**: a goal, an outline of steps (detailed enough to follow, short enough that you must think — the thinking is the learning), what you learned, and — always — **teardown**. The teardown discipline matters twice over: it keeps you inside the Free Tier, and “leaves nothing running that isn’t needed” is a professional reflex interviewers genuinely probe for. Rebuild each lab at least twice: once following the outline, once from memory. The second build is where the knowledge moves into your hands. Budget roughly a week and a half per lab; use the slack for breakage, because things will break, and debugging them is the best teaching this course can’t script.

First, three ideas that frame everything you’re about to build. **Regions and Availability Zones:** a Region is a geographic cluster of AWS data centers (pick one close to you and stay in it — resources in one region are invisible from another, the #1 “where did my server go?” confusion). An **Availability Zone (AZ)** is an isolated data center within the region; serious systems run in two so one building’s failure doesn’t take them down. **The console vs the CLI:** the console (<https://console.aws.amazon.com>) is AWS’s web control panel — great for learning and looking; the **AWS CLI** (`aws` in your terminal) does everything the console does, scriptably. You’ll start in the console and graduate to the CLI, because Stage 2 automates everything you do here by hand.

Lab 1 (Weeks 5–6): EC2 — your first server

EC2 (Elastic Compute Cloud) rents virtual machines, called **instances**. This is the primal act of cloud engineering: a real Linux server, on the internet, in 60 seconds, for free.

Goal: launch a Linux server, SSH into it, make it serve a web page to the world, then destroy it.

Steps outline: 1. Console — EC2 — Launch instance. Name it. Choose **Amazon Linux 2023** as the **AMI** (Amazon Machine Image — the template disk your server boots from) and **t2.micro** or **t3.micro** as the **instance type** (the size; these are Free Tier). 2. Create a **key pair**; a .pem private-key file downloads. That’s your SSH key from Module 1 — guard it, `chmod 400` it. 3. In network settings, allow SSH (port 22) *from “My IP” only* — you know exactly what this security-group rule means now — and allow HTTP (port 80) from anywhere. 4. Launch, wait for “running,” copy the public IP, then from your terminal: `ssh -i mykey.pem ec2-user@<public-ip>`. Take a breath: you are inside a computer in an AWS data center. 5. On the server: `sudo dnf install -y nginx && sudo systemctl start nginx && sudo systemctl enable nginx`. Then browse to `http://<public-ip>` — that’s *your* web server answering the world. 6. Replace the default page: `echo "<h1>Built by me, on EC2</h1>" | sudo tee /usr/share/nginx/html/index.html`. Refresh. Screenshot for the journal. 7. Poke around like an engineer: `top`, `df -h`, `sudo tail -f /var/log/nginx/access.log` while you refresh the page — watch your own visits arrive in the log.

What you learned: AMIs, instance types, key pairs, security groups in anger, SSH to a real server, installing and running a Linux service, reading its logs — i.e., the daily physical motions of the job.

Teardown: EC2 — select instance — Instance state — **Terminate**. Confirm it says “terminated.” Free Tier gives 750 hours/month of one micro instance, so even leaving it up wouldn’t have billed — but tear it down anyway. Habit is the point.

Lab 2 (Week 7): S3 — files that never disappear

S3 (Simple Storage Service) is object storage: a bottomless, eleven-nines-durable bucket for files. It’s the default answer to “where do we put files?” and, misconfigured, the source of history’s most famous data leaks — which is why this lab is half storage, half security.

Goal: create a bucket, work it from the CLI, host a tiny static website, and understand exactly what “public bucket” means.

Steps outline: 1. Console — S3 — Create bucket (names are globally unique — `yourname-lab-2026` works). Note **Block Public Access** is on by default. Upload any file through the console; download it back. 2. Install the AWS CLI locally and run `aws configure` with an **access key** you create for your IAM admin user (an access key is a programmatic username+password for the API — treat it like a password, never put it in code; you’ll internalize this rule in Lab 5). 3. From your terminal: `aws s3 ls · aws s3 cp notes.txt s3://yourname-lab-2026/ · aws s3 sync ./myfolder s3://yourname-lab-2026/backup/`. Feel the difference: the console is visiting; the CLI is engineering. 4. Static website: make a second bucket, enable static website hosting, upload an `index.html`, and add

the documented public-read **bucket policy** (a JSON permissions document — read it line by line: *who* may do *what* to *which* bucket). Your page is now on the internet without any server at all. 5. Explore **storage classes** (Standard — Infrequent Access — Glacier: cheaper per GB, slower/costlier to retrieve) and set a **lifecycle rule** (“move objects to IA after 30 days”) — your first taste of automated cost management.

What you learned: object storage vs disks, the CLI and access keys, bucket policies and public access (deliberately, not accidentally), storage tiers as a cost lever.

Teardown: empty both buckets, delete both buckets, and — important — deactivate any access key you’re not using. S3’s free allowance is small (5 GB) but these files are kilobytes; the discipline, again, is the point.

Lab 3 (Weeks 8–9): VPC — the network you own

VPC (Virtual Private Cloud) is your private, fenced-off slice of AWS’s network. Until now you used the default one without looking; engineers build their own, because the sentence “*the database sits in a private subnet with no internet route*” is half of cloud security, and you’re about to make it true with your own hands.

Goal: build a two-tier network — public subnet for a web server, private subnet for a future database — and prove the private half is unreachable from the internet.

Steps outline: 1. Console — VPC — Create VPC. Give it the address block `10.0.0.0/16` — **CIDR notation**, where `/16` means “the first 16 bits are fixed, the rest are mine”: 65,536 private addresses. 2. Create two **subnets**: `10.0.1.0/24` (public, in AZ-a) and `10.0.2.0/24` (private, in AZ-b). A subnet is a smaller block within the VPC, living in exactly one AZ. 3. Create an **Internet Gateway** (the VPC’s door to the internet) and attach it. Create a **route table** with the rule `0.0.0.0/0 → internet gateway` (“anything not local goes to the internet door”) and associate it with the *public* subnet only. The private subnet keeps only the local route — that absence of a route *is* the security. 4. Launch one micro EC2 instance in each subnet (public one with a public IP, private one without). 5. The proof: SSH to the public instance — works. Try the private instance’s address from your laptop — hangs forever, and now you can say precisely why. Then SSH *from* the public instance *to* the private one (it’s reachable from inside the VPC): the public box is acting as a **bastion host**, an industry-standard pattern you just discovered by building it. 6. Bonus concept to look up and journal: a **NAT gateway** lets private machines reach *out* (for updates) while remaining unreachable from outside — but it bills per hour, so read about it, don’t build it.

What you learned: CIDR, subnets, route tables, internet gateways, the public/private split, bastion hosts — the exact networking bullet in every cloud JD, as muscle memory.

Teardown: terminate both instances first, then delete the VPC (which sweeps subnets, route tables, and the gateway with it). Verify in EC2 that nothing says “running.”

Lab 4 (Week 10): RDS — a database you don’t babysit

RDS (Relational Database Service) is a managed database: AWS runs the database engine (PostgreSQL, MySQL...) and handles backups, patching, and failover, while you own the data

and the queries. “Managed service” is the cloud’s core bargain — trade some control for a lot of undifferentiated labor — and this lab is where you feel that bargain.

Goal: launch a PostgreSQL database in a private subnet, connect to it from an EC2 instance, and understand backups and multi-AZ — then tear it down promptly, because RDS is the easiest lab to leave running by accident.

Steps outline: 1. Rebuild Lab 3’s VPC quickly (second build from memory — this is deliberate spaced repetition), adding a second *private* subnet in another AZ, because RDS requires a **subnet group** spanning two AZs. 2. Console → RDS → Create database → PostgreSQL → **Free tier template** (this preselects `db.t3.micro/db.t4g.micro`, single-AZ). Set the master password. Place it in your VPC, **Public access: No**, in a security group that allows port 5432 *only from the web server’s security group* — a rule that references another rule rather than an IP. Elegant, and standard. 3. Launch a micro EC2 in the public subnet, install the `postgresql` client, and connect: `psql -h <rds-endpoint> -U postgres`. The **endpoint** is a DNS name, not an IP — AWS may move the underlying machine, and the name follows it. (Module 2 paying rent already.) 4. At the `psql` prompt: create a table, insert three rows, select them back. You don’t need SQL depth today; you need to have touched it. 5. Tour, don’t enable: the **automated backups** setting (point-in-time restore from nightly snapshots + logs), and the **Multi-AZ** option (a live standby in another AZ with automatic **failover** — roughly double cost, which is why prod says yes and labs say no). Take a manual **snapshot**, find it in the console, understand you could restore a clone from it. 6. Journal question worth ten minutes: what, exactly, is AWS doing for you here that you’d otherwise do at 2 a.m. yourself? (Patching, backups, failover, hardware.) That answer is the managed-services interview answer.

What you learned: managed databases, subnet groups, security-group-to-security-group rules, endpoints, snapshots, multi-AZ/failover — plus a live demonstration of paying with control to buy reliability.

Teardown: delete the RDS instance (decline the final snapshot for a lab; note that prod would take one), delete the manual snapshot (snapshots bill for storage!), terminate EC2, delete the VPC. Check your billing dashboard the next day — reading it weekly is a Stage 3 habit starting now.

Lab 5 (Weeks 11–12): IAM — who may do what

IAM (Identity and Access Management) decides which people and which programs may do what to which resources. It bills nothing, provisions nothing — and it’s the most audited, most interview-probed, most breach-implicated service in AWS. Security bullets in JDs (“least-privilege access controls,” “IAM policies”) mean this lab.

Goal: create users, groups, policies, and — the crucial one — a **role**, and internalize least privilege by feeling AWS deny you.

Steps outline: 1. Concepts first, five minutes: a **user** is an identity for a human or program; a **group** bundles users; a **policy** is a JSON document granting permissions (“allow `s3:GetObject` on `arn:aws:s3:::my-bucket/*`”); a **role** is an identity with policies but *no password* that a trusted party temporarily *assumes* — how servers and services get permissions without any stored secret. 2. Create a user `readonly-rita` in a group with the AWS-managed

ReadOnlyAccess policy. Log in as her in a private browser window: she can see everything, but every create/delete button fails with an explicit denial. Read one of those errors fully — learning to parse “not authorized to perform X on Y” is a daily job skill. 3. Write your first custom policy in the JSON editor: allow `s3:ListBucket` and `s3:GetObject` on one specific bucket. Attach it to a new user; verify she can read that bucket and *nothing else*. You have now implemented least privilege, not just defined it. 4. The role, the payoff: create a role for EC2 with `AmazonS3ReadOnlyAccess`, launch a micro instance with that role attached, SSH in, and run `aws s3 ls` — it works with **no access keys anywhere on the machine**. The instance is assuming the role and receiving short-lived credentials automatically. This is the single most important security pattern in AWS: *roles for machines, never keys on machines*. Say it twice. 5. Audit your own account like a pro: is root MFA’d (Week 3)? Any access keys older than 90 days? Does any user have more permission than they use? Run the IAM **credential report** and read it. This ten-minute ritual, monthly, is Stage 3’s IAM hygiene checklist being born.

What you learned: users/groups/policies/roles, reading and writing policy JSON, least privilege enforced by experiment, roles-not-keys, and your first security audit.

Teardown: delete the test users and their credentials, terminate the instance, keep the role knowledge forever.

Vocabulary for Stage 1 (one table, all five labs):

Term	Definition
Region / Availability Zone	Geographic cluster of AWS data centers / one isolated data center within it. Serious systems span two AZs.
EC2 / instance	The virtual-machine rental service / one rented server.
AMI	Amazon Machine Image — the template disk an instance boots from.
Instance type	The size/spec you chose (t3.micro = tiny, Free Tier).
Security group	Per-resource firewall: which ports, from which sources.
Key pair	The private/public SSH keys for reaching your instance.
S3 / bucket / object	Object storage / one named container / one stored file.
Storage class / lifecycle rule	Price-speed tier for objects / automatic rule moving them to cheaper tiers with age.
Bucket policy	JSON document stating who may do what to a bucket. Public ones make headlines.
AWS CLI / access key	The terminal interface to AWS / programmatic credentials for it (treat as a password).

Term	Definition
VPC / subnet	Your private network slice / a smaller block of it living in one AZ, public or private.
CIDR	Address-block notation: 10.0.0.0/16 = “first 16 bits fixed, 65,536 addresses mine.”
Internet gateway / route table	The VPC’s internet door / the rules deciding where traffic is sent. No route = unreachable = secure.
NAT gateway	Lets private machines call out without being reachable in. Bills hourly — know it, don’t idle it.
Bastion host	The hardened public machine you SSH through to reach private ones.
RDS / endpoint	Managed relational database service / the DNS name you connect to.
Snapshot / Multi-AZ / failover	Point-in-time copy / live standby in a second AZ / the automatic switch to it.
IAM user / group / policy / role	Identity / bundle of identities / JSON permission grant / assumable identity with no password — how machines get permissions.
Least privilege	The golden rule: minimum permissions that do the job, nothing more.
Managed service	AWS runs the undifferentiated labor (patching, backups, failover); you keep the data and the decisions.

Videos for this stage:

Video	Channel	Length	Link
Getting Started with EC2	AWS Developers (official)	27 min	https://www.youtube.com/watch?v=nJ-djerESW0
Introduction to Amazon S3	Amazon Web Services (official)	~5 min	https://www.youtube.com/watch?v=ecv-19sYL3w
AWS Networking Basics — VPC & Subnets	KodeKloud (rewatch after Lab 3 — it lands differently now)	~30 min	https://www.youtube.com/watch?v=QM63dyA_4Pc
The AWS Shared	Digital Cloud	4 min	https://

Video	Channel	Length	Link
Responsibility Model	Training		www.youtube.com/watch?v=ESPBBEK-cvo

Say it out loud: “It’s in a private subnet; there’s no route to the internet gateway.” · “The security group only allows 5432 from the web tier’s security group.” · “The instance uses a role — there are no keys on the box.” · “Terminate it when you’re done; nothing idles in this account.”

Milestone — end of Stage 1: the gauntlet build. In one sitting, from memory: VPC with public/private subnets — EC2 web server (public, role-attached, serving a page) — RDS PostgreSQL (private, reachable only from the web server) — an S3 bucket the instance can read via its role — then tear all of it down clean. Under three hours means you’re ready for Stage 2. Also: **start CLF-C02 prep now** (the freeCodeCamp full course — <https://www.youtube.com/watch?v=7HKot-brXFE> — at 1.25× as revision; the classic 14-hour edition is <https://www.youtube.com/watch?v=NhDYbskXRgc>), and sit the **AWS Cloud Practitioner** exam around Week 14–16.

STAGE 2 — AUTOMATION (Weeks 13–20)

The idea of this stage: everything you clicked in Stage 1, you will now do with code. The industry’s mild insult for console-clicking is **ClickOps**; the job description phrase for what replaces it is “*design, develop, and deploy cloud infrastructure using infrastructure as code.*” This stage is the difference between someone who has *used* AWS and someone a company will pay to *run* it.

Module 6 (Weeks 13–15): Scripting — Bash and Python for ops, plus Git

Bash scripting is putting the Module 1 commands into a file so the computer repeats them perfectly. Learn, in this order: **variables** (`NAME="web-1"`), **command substitution** (`TODAY=$(date +%F)`), **if** (`if [ -f "$FILE" ]; then ... fi`), **loops** (`for f in *.log; do gzip "$f"; done`), **exit codes** (every command returns 0 for success, non-zero for failure; `&&` chains on success — scripts make decisions with this), and reading **arguments** (`$1, $2`). That’s 90% of real ops Bash. Write your first genuinely useful script this week: `backup.sh` — tar a directory, name the archive with today’s date, `aws s3 cp` it to a bucket, delete local archives older than 7 days, and echo a success/failure line to a log file. That one script is variables, substitution, conditionals, exit codes, and the CLI in one artifact — and a bullet on your résumé (“automated backups to S3”).

Python is Bash’s bigger sibling: better for anything involving logic, data, or talking to APIs. You need functional ops-Python, not software-engineering depth: variables and types, lists and dictionaries, `if/for`, functions, reading/writing files, `try/except` error handling, and installing libraries with `pip`. Then meet **boto3**, the AWS library for Python: `import boto3; ec2 = boto3.client("ec2"); ec2.describe_instances()` — suddenly your Stage 1

knowledge is programmable. Write `audit.py`: list every EC2 instance in the account with its name, type, state, and launch time, and print a warning line for anything running longer than 24 hours. Congratulations — you’ve written a real cost-control tool employers pay for.

Git is the version-control system all engineering work lives in. Concepts: a **repository** (a folder whose entire history is recorded), a **commit** (one saved snapshot with a message), a **branch** (a parallel line of work), a **remote** (the copy on GitHub), and a **pull request** (asking for a branch to be reviewed and merged). The daily loop is six commands: `git init` / `git clone`, `git status`, `git add`, `git commit -m "message"`, `git push`, `git pull`. Create a GitHub account today, create a repo called `cloud-journey`, and from now on **every script and config you write in this course gets committed to it**. In ten months, that commit history is dated, public proof of everything this course claims you can do.

Vocabulary:

Term	Definition
Script	A file of commands run top to bottom — automation’s atom.
Variable / argument	A named value in a script / a value passed in when you run it (<code>\$1</code>).
Exit code	Every command’s success (0) or failure (non-zero) signal — how scripts make decisions.
Cron	Linux’s scheduler: run a script every night at 2 a.m., forever.
Python / pip	The ops world’s favorite programming language / its package installer.
boto3	The Python library for driving AWS — the console, as code.
try/except	Python’s “attempt this; if it fails, do that instead” — how scripts fail gracefully.
Git / repository / commit	Version control / one project’s recorded history / one saved snapshot with a message.
Branch / merge	A parallel line of work / bringing it back into the main line.
GitHub / remote / pull request	The hosting site / your repo’s cloud copy / a reviewed, discussed request to merge a branch.
README	The front-page document of a repo explaining what it is and how to run it. Recruiters read these.

Videos for this module:

Video	Channel	Length	Link
Learn Python — Full	freeCodeCamp	~4.5 hr (watch	https://

Video	Channel	Length	Link
Course for Beginners		across the 3 weeks)	www.youtube.com/watch?v=rfscVS0vtbw

Say it out loud: “I’ll script it — it’ll be wrong once, then never again.” · “Check the exit code before the next step runs.” · “Commit it with a message that says why, not what.” · “It’s in the repo — pull the latest.”

Exercises: (1) `backup.sh`, as specified above, scheduled nightly with cron on a lab EC2 instance. (2) `audit.py`, as specified above. (3) Both committed to `cloud-journey` with a README explaining each. (4) Break your own script on purpose (a wrong path, a missing bucket) and make it fail *loudly and clearly* — error handling is an ops love language.

Milestone: your backup script survives being run twice in a row and with a missing folder (no crash, clear message); your Python audit runs against your real account; your repo shows a week of commits.

Module 7 (Weeks 16–17): Terraform — infrastructure as code

The big idea: instead of clicking resources into existence, you write text files *declaring* what should exist — “one VPC, two subnets, one instance, these tags” — and **Terraform** makes AWS match the file. Declarative, not imperative: you state the destination, not the turns. Why every JD asks for it: the files live in Git, so infrastructure gets **reviewed** (pull requests before changes), **repeatable** (the same file builds dev, staging, and prod identically), **reversible** (roll back by reverting the commit), and **auditable** (history says who changed what, when, why). The console is a workshop; Terraform is a factory.

The core loop you’ll run hundreds of times: `terraform init` (download the AWS **provider** — the plugin that translates your files into API calls) — `terraform plan` (a dry run printing exactly what would be created/changed/destroyed — read every line; engineers who apply without reading plans cause outages) — `terraform apply` (make it real) — `terraform destroy` (unmake it — teardown as a single command, which by now you’ll find beautiful). Terraform tracks what it built in a **state file** — its memory of reality; lose it or hand-edit reality behind its back (“drift”) and pain follows. Resources are declared in **HCL**, a readable config language:

```
resource "aws_instance" "web" {
  ami           = "ami-0abcdef1234567890"
  instance_type = "t3.micro"
  tags = { Name = "web-1", Project = "lab" }
}
```

The lab (this is the module): rebuild Stage 1’s gauntlet in Terraform, incrementally. (1) Init a repo `terraform-labs`; write config for just a tagged S3 bucket; plan, read, apply, check the console — it’s there; destroy. (2) Grow the config: VPC, two subnets, internet gateway, route table — the Lab 3 build, now ~60 lines of HCL. (3) Add the security group and an EC2 instance with `variables.tf` (region, instance size) and `outputs.tf` (print the public IP after apply). (4) Change the instance type in the file and re-apply — watch Terraform compute the

difference and change *only that*. This diff-driven behavior is the whole magic. (5) Destroy everything, confirm the console is empty, commit, and tag the repo v1.0. Reference documentation lives at <https://developer.hashicorp.com/terraform> — bookmark it; reading provider docs is half of real Terraform work.

Vocabulary:

Term	Definition
IaC (Infrastructure as Code)	Declaring infrastructure in versioned text files a tool turns into reality.
Terraform / HCL	The dominant multi-cloud IaC tool / its configuration language.
Provider	The plugin translating your files into one cloud's API calls (here: AWS).
plan / apply / destroy	The dry-run diff / making it real / unmaking it all. Read every plan.
State file	Terraform's record of what it built — its memory of reality. Protect it.
Drift	Reality changed behind Terraform's back (someone clicked). The enemy.
Variable / output / module	A config input / a printed result (an IP, a URL) / a reusable packaged chunk of config.
Declarative vs imperative	Stating the destination vs scripting the turns. Terraform is declarative.
CloudFormation	AWS's own IaC service — same idea, AWS-only. JDs accept either; learn Terraform first.

Videos for this module:

Video	Channel	Length	Link
Terraform explained in 15 mins	TechWorld with Nana	18 min	https://www.youtube.com/watch?v=l5k1ai_GBDE
Terraform Course — Automate your AWS cloud infrastructure	freeCodeCamp	~2.5 hr	https://www.youtube.com/watch?v=SLB_c_ayRMo

Say it out loud: “Nothing changes in prod except through Terraform.” · “Show me the plan before you apply.” · “That was clicked in manually — that’s drift; let’s import it or remove it.” · “It’s in the module; reuse it, don’t rewrite it.”

Exercises: the lab above, plus: (1) deliberately create drift — change the instance’s tag by hand in the console, run `terraform plan`, and watch Terraform notice; journal what it proposed.

(2) Write a plain-English paragraph in the repo's README: "why this beats clicking." If you can't write it, you don't have it yet.

Milestone: from an empty directory, you can bring up the VPC + instance stack with `init/plan/apply` and remove it with `destroy`, explaining out loud what each command is doing — without notes.

Module 8 (Weeks 18–20): CI/CD, Docker, and a first look at Kubernetes

CI/CD (Continuous Integration / Continuous Delivery) is the robot conveyor belt attached to your Git repo: on every push, it automatically checks, tests, builds, and — when configured — deploys your change. The point is small, frequent, boring releases instead of rare, terrifying ones. **GitHub Actions** is the CI/CD system built into GitHub: a **workflow** is a YAML file in `.github/workflows/` saying "on push, run these steps on a fresh **runner** (a temporary VM)." Lab: add a workflow to `terraform-labs` that runs `terraform fmt -check` and `terraform validate` on every push — push a deliberately malformed file and watch the red **X**, fix it and watch the green **✓**. You now have a pipeline; small as it is, it's the same species as the ones in every JD. (Second step, in Project 1: a workflow that runs `terraform plan` on pull requests, so proposed infrastructure changes show their diff in the review.)

Docker packages an application with everything it needs into a **container** — a sealed lunchbox that runs identically on your laptop, on EC2, and everywhere else, ending the ancient plague of "works on my machine." An **image** is the frozen recipe (built from a **Dockerfile**, a short text file of build steps: start FROM a base image, COPY in your app, define the start command); a **container** is one running instance of an image; a **registry** (Docker Hub, or AWS's **ECR**) is where images are pushed and pulled. Containers are *not* VMs: they share the host's Linux kernel, so they start in about a second and you can run dozens on a micro instance. Lab: on an EC2 instance, install Docker, `docker run hello-world`, then `docker run -d -p 80:8080 <a sample web image>` and browse to it; then write a five-line Dockerfile serving your Lab 2 static page from an `nginx` base image, build it, run it, and push it to Docker Hub. Learn the daily verbs: `docker ps`, `docker logs`, `docker exec -it <id> bash` (a shell *inside* the container), `docker stop`.

Kubernetes (K8s) — for this course, a reading-level skill, honestly labeled. When a company runs hundreds of containers across many machines, something must schedule them, restart the crashed, scale the busy, and route traffic among them: that orchestrator is Kubernetes. Learn the concept map now — a **cluster** of **nodes** runs **pods** (the smallest deployable unit, usually one container); a **deployment** declares "keep 3 replicas of this pod alive" and the cluster continuously makes it true (declarative again — Kubernetes is Terraform's philosophy applied to running software); a **service** gives pods a stable address. AWS's managed offering is **EKS**. Junior JDs want exactly this literacy plus Docker fluency; real K8s operating skill (and the CKA certification) is a strong Year 2 goal, not a Month 5 one. This course tells you honestly which is which.

Vocabulary:

Term	Definition
CI/CD	The automated pipeline that checks, builds,

Term	Definition
	tests, and ships every change.
GitHub Actions / workflow / runner	GitHub's built-in CI/CD / the YAML file defining a pipeline / the temporary VM executing it.
YAML	The indentation-based config format of pipelines and Kubernetes. Indentation is meaning — mind it.
Docker / image / container	The container toolkit / the frozen, layered recipe / one running instance of it.
Dockerfile	The short text file of steps that builds an image.
Registry / ECR	Where images are pushed and pulled / AWS's registry.
Kubernetes (K8s) / cluster / node	The container orchestrator / its group of machines / one machine in it.
Pod / deployment / service	Smallest deployable unit / “keep N replicas alive,” continuously enforced / a stable address in front of pods.
EKS	AWS's managed Kubernetes control plane.
Rollback	Reverting to the previous version fast when a release misbehaves — the safety net CI/CD makes routine.

Videos for this module:

Video	Channel	Length	Link
DevOps CI/CD Explained in 100 Seconds	Fireship	2 min	https://www.youtube.com/watch?v=scEDHsr3APg
What is DevOps? REALLY understand it	TechWorld with Nana	~15 min	https://www.youtube.com/watch?v=0yWAtQ6wYNM
Docker in 100 Seconds	Fireship	2 min	https://www.youtube.com/watch?v=Gjnup-PuquQ
Docker Tutorial for Beginners [FULL COURSE in 3 Hours]	TechWorld with Nana	3 hr	https://www.youtube.com/watch?v=3c-iBn73dDE

Video	Channel	Length	Link
Kubernetes explained in 15 mins	TechWorld with Nana	~16 min	https://www.youtube.com/watch?v=VnvRFRk_5lk

Say it out loud: “Don’t merge until the pipeline is green.” · “It’s containerized — same image in dev and prod.” · “Exec into the container and check its logs.” · “K8s keeps three replicas up; kill one and watch it come back.”

Exercises: the two labs above, plus: (1) journal a one-paragraph answer to “VM vs container vs serverless — when each?” (Fireship’s Serverless in 100 Seconds — https://www.youtube.com/watch?v=W_VV2Fx32_Y — rounds out the third option). (2) Terminate the Docker lab instance; confirm your images live on in the registry — notice that the *artifact* now outlives the *server*, which is the entire modern deployment worldview in one observation.

Milestone — end of Stage 2: your GitHub shows: a repo of working Bash + Python ops scripts, a Terraform repo that builds and destroys a real stack, a green Actions workflow, and a Dockerfile pushed to a registry — and you can explain every file in an interview. That GitHub profile is no longer a student’s. It’s a junior engineer’s.

STAGE 3 — OPERATIONS (Weeks 21–28)

The idea of this stage: building systems gets you hired; *running* them is the actual job. The JD bullets this stage answers are the operational half — “monitor infrastructure health,” “participate in incident response, including log analysis,” “identifying, analyzing, and resolving infrastructure vulnerabilities,” “manage cloud costs.” A cloud engineer’s week is mostly this stage.

Module 9 (Weeks 21–24): Monitoring, logging, and incidents

Monitoring — knowing before the users do. CloudWatch is AWS’s built-in observability service. Three primitives: **metrics** (numbers over time — CPU %, disk %, request count, error count), **alarms** (a rule watching a metric: “if CPU > 80% for 5 minutes — notify”), and **dashboards** (metrics arranged on one screen). Notifications flow through **SNS** (Simple Notification Service — a topic you publish to, subscribers get emailed/paged). The craft is *what* to alarm on: page a human only for what needs a human, or people learn to ignore the pager (**alert fatigue** — the failure mode that has preceded many famous outages). The four golden signals worth memorizing: **latency, traffic, errors, saturation** — how slow, how busy, how broken, how full.

Logging — knowing why. Metrics say something is wrong; logs say what happened. **CloudWatch Logs** centralizes them: an **agent** on each instance ships files like `/var/log/nginx/access.log` to **log groups**, where **Logs Insights** lets you query across

machines (“count 5xx responses by minute for the last hour”). Centralization matters because servers are disposable now (you proved that in Module 8) — logs must outlive the machines that wrote them.

Lab (Weeks 21–22): stand up a small monitored web server, all in Terraform (your stack from Module 7, extended): EC2 + nginx + the CloudWatch agent; an SNS topic emailing you; an alarm on high CPU and another on instance status-check failure. Then attack yourself: SSH in and run a CPU-burner (yes `> /dev/null &` a few times); watch the metric climb, the alarm fire, the email arrive; kill the processes; watch it recover. Then query your own access logs in Logs Insights. You have now witnessed the full detect→notify→diagnose→resolve loop on a system you built.

Incident response — the human half. An **incident** is an unplanned “the system is not okay”; **severity** levels (sev-1 = customers down, all hands) set the response scale. The professional loop: **detect** (the alarm, not a customer email, ideally) → **triage** (how bad, who’s needed) → **mitigate** (stop the bleeding *first* — roll back, restart, failover; root cause comes later) → **resolve** → **post-mortem**: a *blameless* written review of what happened, why, and what will prevent a repeat. Blameless is not softness; it’s engineering: punished people hide information, and hidden information causes repeat outages. **On-call** is the rotation of who carries the pager; JDs mention it, interviewers ask about it, and your honest answer after this module is “I’ve simulated it and I know the loop.”

Runbooks — the documentation bullet, made concrete. A runbook is a step-by-step recipe for one operational situation, written so a stressed person at 3 a.m. can follow it: symptoms → checks (exact commands) → fixes (exact commands) → escalation (who to wake if it doesn’t work). **Lab (Weeks 23–24):** write two runbooks in your repo — “web server down” and “disk filling up” — then *drill* them: break the thing, follow your own document literally, and fix every step that proved vague. Then run one full **game day**: have a friend (or an AI) break your lab stack in secret; you get paged, diagnose from metrics and logs, mitigate, and write the post-mortem in your journal. That post-mortem is an interview story, and a good one.

Vocabulary:

Term	Definition
CloudWatch	AWS’s monitoring service: metrics, alarms, dashboards, logs.
Metric / alarm / dashboard	A number over time / a rule that fires on it / one screen of them.
SNS	The notification service alarms publish to — email, SMS, pager.
Golden signals	Latency, traffic, errors, saturation — the four numbers that describe any service’s health.
Alert fatigue	Too many non-actionable pages → ignored pages → missed real ones. Alarm design’s enemy.
Log group / Logs Insights	Where centralized logs land / the query

Term	Definition
	language over them.
Incident / severity	An unplanned degradation / its ranked badness (sev-1 = worst).
Triage / mitigate / resolve	Assess fast / stop the bleeding first / actually fix.
Post-mortem	The blameless written review: what, why, what prevents a repeat.
Runbook	The 3-a.m.-proof recipe for one situation: symptoms, checks, fixes, escalation.
On-call / game day	The pager rotation / a deliberate practice incident.
MTTR	Mean time to recovery — the ops metric mature teams optimize.

Say it out loud: “Did we find out from the alarm or from a customer?” · “Mitigate first — root cause after we’re stable.” · “Is there a runbook for this? There will be by tomorrow.” · “What did the post-mortem conclude, and what’s the prevention item?”

Milestone: your game-day post-mortem exists, is blameless, and names one concrete prevention you then actually implemented (an alarm added, a runbook fixed).

Module 10 (Weeks 25–28): Security operations and cost management

Security operations — hygiene, not heroics. The **shared responsibility model** first (rewatch: <https://www.youtube.com/watch?v=ESPBBEK-cvo>): AWS secures the cloud itself; *you* secure what you put in it — and most real breaches are customer misconfigurations, not AWS failures. Your operational security checklist, practiced until boring:

1. **IAM hygiene** (monthly): MFA everywhere; no long-lived access keys where a role would do; run the credential report; delete unused users and keys; question every * in every policy. You built this reflex in Lab 5 — now it’s a calendar entry.
2. **Patching:** unpatched software is how most break-ins start. On your instances: apply updates on a schedule (and know that **SSM Patch Manager** automates this fleet-wide); in the container world, patching often means rebuilding the image from an updated base — connect that to Module 8.
3. **Backups — tested ones:** an untested backup is a hope, not a plan. RDS snapshots, S3 **versioning** (every overwrite keeps the old version — the anti-oops and anti-ransomware control). Lab: snapshot your lab database, *restore it* to a new instance, verify the rows, tear down. Now you may say “tested” in interviews.
4. **Guardrails:** turn on **CloudTrail** (the audit log of every API call in the account — who did what, when) and skim yesterday’s events once; enable **GuardDuty**’s free trial (automated threat detection) and read what it watches for. Know the phrase **encryption at rest and in transit** and that in AWS it’s mostly a checkbox backed by **KMS** — table stakes, always on.

Cost management — the skill that makes juniors look senior. The JD bullet is verbatim: “manage cloud costs through rightsizing resources, implementing auto-scaling, resource tagging.” How the meter runs: compute bills per second it’s *on* (idle ≠ free — “we left it running” is the classic waste), storage per GB-month, and **egress** (data *out* of AWS) per GB while data in is free — the famous bill surprise. The levers, in the order a junior can pull them: **turn it off** (dev systems at night; your teardown habit, industrialized) — **rightsize** (most servers are oversized; check CloudWatch CPU history and shrink) — **tag everything** (Project, Owner, Environment — untagged spend is unaccountable spend; enforce tags in your Terraform) — **tier storage** (lifecycle rules from Lab 2) — know that **reserved capacity/Savings Plans** (commit 1–3 years, save 30–70%) and **Spot** (up to 90% off, interruptible) exist for steady and batch workloads respectively. Tools: **Cost Explorer** (the account’s spend, graphed — read it weekly, forever) and **AWS Budgets** (your \$0 alert from Week 3, now understood as the smallest member of a serious family). The wider discipline is called **FinOps** — worth two minutes: https://www.youtube.com/watch?v=Y-c_xw9bHFw.

Vocabulary:

Term	Definition
Shared responsibility model	AWS secures the cloud; you secure what you put in it. Most breaches are the second half.
CloudTrail	The account’s audit log: every API call, by whom, when.
GuardDuty	AWS’s automated threat detection over logs and traffic.
Patching / SSM	Applying security updates / AWS’s Systems Manager, which automates it fleet-wide.
S3 versioning	Keep every overwritten version — the anti-oops, anti-ransomware control.
KMS / encryption at rest & in transit	AWS’s key service / data scrambled on disk and on the wire. Always on.
Egress	Data leaving AWS — billed per GB; inbound is free. The classic bill surprise.
Rightsizing	Shrinking oversized resources to measured need. Free money.
Tagging	Labeling every resource with owner/project/environment so all spend is attributable.
Savings Plans / Spot	1–3-year commitment for 30–70% off steady load / reclaimable spare capacity up to 90% off for batch.
Cost Explorer / Budgets	The spend graphs you read weekly / the alerts that mean you never learn from the invoice.
FinOps	The discipline of making cloud spend visible,

Term	Definition
	allocated, and continuously optimized.
Say it out loud: “Who has access to prod, and when did we last review the list?” · “When did we last <i>restore</i> a backup?” · “What’s untagged, and what died but is still billing?” · “It’s over-provisioned — the CPU history says we can halve it.” Exercises: (1) The restore-test lab above. (2) A monthly-security-checklist file in your repo, then actually run it against your account. (3) In Cost Explorer, find the most expensive thing in your account’s history and explain it in one journal sentence. (4) Add default tags to every resource in your Terraform repo. Milestone — end of Stage 3: run one full self-audit — security checklist, cost review, alarm test, backup restore — and write the one-page report. You can now do the <i>job</i>, not just the build. What remains is proving it to strangers: Stage 4.	

STAGE 4 — JOB-READINESS (Months 8–12)

Module 11 (Months 8–10): Three portfolio projects

Certifications say you studied; projects prove you can build. Three, each in its own GitHub repo, each with an architecture diagram, a README written for a hiring manager (what, why, how to run it, what it costs), and a teardown script. Build — screenshot/record — *destroy* — the repo is the artifact, not a running bill.

Project 1 — Three-tier web app, fully automated (the centerpiece). Terraform builds everything: VPC with public/private subnets across two AZs; an **Application Load Balancer** (the traffic-spreading device — new to you, and the natural next step from Lab 3) in front of an **Auto Scaling Group** of web instances (AWS adds/removes instances with load — look it up, wire it up); RDS in the private subnets; S3 for static assets; CloudWatch alarms and a runbook. GitHub Actions runs `terraform plan` on every pull request and `apply` on merge — a real infrastructure pipeline. This single project demonstrates ten of the fourteen JD bullets in the mapping table; expect every interview to walk through it, and rehearse narrating it in five minutes.

Project 2 — Serverless data pipeline (range). No servers at all: a file landing in an S3 bucket triggers a **Lambda function** (your Python, run by AWS per-event, billed per invocation) that processes it — parse a CSV, summarize it, write results to a second bucket or a DynamoDB table — with failures caught, logged to CloudWatch, and alarmed to your inbox. Deploy it with Terraform. It shows Python, event-driven thinking, and breadth beyond EC2. Two minutes of orientation first: https://www.youtube.com/watch?v=W_VV2Fx32_Y.

Project 3 — Production-style ops showcase (the differentiator). Take Project 1 and operate it like it matters: a CloudWatch dashboard of the golden signals; alarms with a documented paging policy; three runbooks; a tested backup-and-restore procedure with its evidence; a security-hardening pass (IAM audit, patching notes, CloudTrail on) written up; a

cost analysis (“this stack costs \$X/month; here are the three changes that would halve it”); and one game-day post-mortem. Almost no junior candidate has this. It answers the only question interviews really ask — “can this person be trusted with production?” — with documents instead of adjectives.

Module 12 (Months 8–12, parallel): The certification schedule

Certifications open recruiter filters and structure your study. The standard 2026 path for this role, with realistic prep at this course’s pace — note how each exam lands just after the course stages that teach its content, which is why these times are shorter than the internet’s:

Order	Certification	What it proves	Prep time from where you’ll stand	When to sit it
1	AWS Certified Cloud Practitioner (CLF-C02)	Cloud vocabulary, billing, shared responsibility	3–4 weeks (guides say the same for beginners; Stages 0–1 cover most of it)	~ Month 4
2	AWS Solutions Architect Associate (SAA-C03)	Designing real AWS architectures — the credential JDs actually filter on	6–8 weeks of focused prep (industry-standard estimate after CCP)	Months 8–9
3	HashiCorp Terraform Associate (003)	IaC fluency, verified	2–3 weeks — after Module 7 and Project 1 you’re mostly revising (official prep: https://developer.hashicorp.com/terraform/tutorials/certification-003)	Months 10–11

Optional fourth, if targeting ops-titled roles: **AWS SysOps Administrator / CloudOps Associate** — its content is literally Stage 3. For CLF-C02, the freeCodeCamp full course (<https://www.youtube.com/watch?v=7HKot-brXFE>, classic edition <https://www.youtube.com/watch?v=NhDYbskXRgc>) plus practice exams is the well-trodden free path; for SAA-C03, pair a course with many timed practice tests — the exam is scenario-based and stamina matters.

Module 13 (Months 11–12): Résumé, interviews, and the ten questions

The résumé: one page. Lead with skills (AWS, Terraform, Python/Bash, Docker, CI/CD, CloudWatch — mirror the JD's own words; automated filters match keywords) and the three projects, each as two bullets of the form *did X with Y achieving Z*: “Deployed a three-tier web app on AWS with Terraform and GitHub Actions; zero-downtime deploys via ALB + Auto Scaling.” List certifications with dates. Link the GitHub — then assume they’ll actually open it, because the good ones do, and yours now rewards the visit.

Interview prep: three flavors to rehearse — trivia (Stage vocabulary tables), scenarios (the labs you’ve actually done), and behavioral (your journal’s stories, told in **STAR** shape: Situation, Task, Action, Result). Practice *out loud*, daily, for two weeks — ideally with an AI interviewer, mercilessly. The ten questions below cover the classics; the sample answers are deliberately compact — expand each with your own lab details, because “...and when I built this, what actually happened was...” is the sentence that separates you from candidates who only read.

The ten questions, with strong answers:

1. **“Explain the difference between a public and a private subnet.”** *A public subnet’s route table has a route to an internet gateway, so its resources can be reached from the internet; a private subnet has no such route. Web tiers go public, databases go private, and admin access reaches private machines through a bastion or SSM. In my labs I proved the private instance unreachable from my laptop but reachable from the public tier — the missing route is the security.*
2. **“What is IAM, and what is least privilege?”** *IAM controls which identities can perform which actions on which resources, via users, groups, roles, and JSON policies. Least privilege means granting the minimum that does the job — my audit script’s user has read-only access to exactly one service, and my EC2 instances use roles instead of stored keys, so there are no long-lived credentials to leak.*
3. **“An EC2 web server stopped responding. Walk me through your troubleshooting.”** *First scope: one server or everything — check the CloudWatch dashboard and any alarms. Then the layers in order: instance status checks; security group — is 80/443 actually open; SSH in — is nginx running (`systemctl status`), is the disk full (`df -h`), is CPU pinned (`top`); then the logs (`tail` the error log). Mitigate first — restart the service or replace the instance — and do root cause after we’re stable. That’s the order I drill in my runbooks.*
4. **“Why Terraform instead of clicking in the console?”** *The console doesn’t leave a reviewable record and can’t rebuild anything. Terraform is declarative and versioned: changes go through pull requests, the same config builds identical environments, a bad change rolls back by reverting a commit, and `plan` shows the exact diff before anything happens. My pipeline runs `plan` on every PR so the diff is part of the review.*
5. **“What’s the difference between a container and a virtual machine?”** *A VM virtualizes hardware and carries a full OS, taking minutes to boot; a container shares the host’s kernel and packages just the app and its dependencies, starting in about a second. Containers give identical behavior across environments — the same image runs on my laptop and on EC2. VMs still matter for isolation and for running non-Linux workloads.*

6. **“How would you reduce our AWS bill?”** *In order of effort: find what’s running that shouldn’t be — idle instances, unattached volumes, old snapshots; Cost Explorer and tags make that visible. Then rightsize against CloudWatch history. Then schedule dev environments off outside hours. Then lifecycle old S3 data to cheaper tiers. Then commit steady workloads to Savings Plans and put interruptible batch on Spot. And watch egress — data out is the classic surprise line.*
7. **“What happens when you type a URL and press Enter?”** *DNS resolves the name to an IP; the browser opens a TCP connection to port 443 and negotiates TLS; it sends an HTTP GET; on the far side that hits a load balancer, which forwards to a healthy instance in a public subnet; the app may query a database in a private subnet; the response returns with a status code — 200 if healthy — and the browser renders it. I can drill into any hop, and I’ve built each of them.*
8. **“A teammate’s change broke production. What happens next?”** *Mitigate first — roll back through the pipeline; recovery beats diagnosis in the moment. Then a blameless post-mortem: what happened, why the system allowed it, and which guardrail was missing — a test, a plan review, an alarm. Blameless matters practically: people who fear punishment hide information, and hidden information causes the repeat.*
9. **“What is the shared responsibility model?”** *AWS secures the cloud — data centers, hardware, hypervisor; the customer secures what’s in it — data, IAM, network rules, patching, encryption settings. Most real cloud breaches are on the customer side, misconfigurations like public buckets or leaked keys. So ‘AWS is secure’ and ‘we are secure on AWS’ are different sentences, and the second one is my job.*
10. **“Tell me about a problem you debugged.”** *(Yours, from the journal, in STAR shape. Example skeleton:) During my monitoring game day, my alarm fired for CPU saturation (S). I had to find and stop the cause before ‘users’ noticed (T). Metrics showed the spike’s start; Logs Insights tied it to runaway processes; I SSH’d in, confirmed with top, killed them, and watched recovery on the dashboard (A). Resolved in eleven minutes, and the post-mortem produced a process-count alarm I then implemented in Terraform (R).*

Job-search mechanics: apply from Month 11 — after SAA, don’t wait for “ready,” because interviews are training. Target titles: cloud engineer (junior/associate), cloud support engineer, cloud operations engineer, junior DevOps engineer, AWS support associate. Every rejection with an interview is a free lesson; journal what they asked.

Vocabulary:

Term	Definition
Portfolio	Public, documented proof you can build — for this trade, GitHub repos with diagrams and READMEs.
Application Load Balancer (ALB)	AWS’s traffic-spreader: distributes requests across healthy instances, drops the sick.
Auto Scaling Group	Keeps N instances alive and adjusts N with load — self-healing and elasticity in one.
Lambda / serverless	Code AWS runs per event, billed per invocation — no servers to manage at all.

Term	Definition
DynamoDB	AWS's serverless NoSQL table — pairs naturally with Lambda.
STAR	Situation, Task, Action, Result — the shape of a good interview story.
ATS	Applicant tracking system — the keyword filter your one-page résumé must pass.
CLF-C02 / SAA-C03 / Terraform Associate 003	Your three exams: vocabulary, architecture, IaC.
SysOps / CloudOps Associate	The optional ops-focused AWS associate — Stage 3, examined.

Milestone — end of course: three repos a stranger can understand, three certifications scheduled or passed, ten answers rehearsed out loud, applications live. You are not “hoping to get into cloud.” You are a junior cloud engineer with evidence, interviewing.

The one-page syllabus

When	Focus	Hands-on proof	External proof
Weeks 1–2	How computers work; Linux, terminal, permissions, SSH	Scripted file ops; first shell script	—
Weeks 3–4	Networking: IP, DNS, ports, HTTP, firewalls; AWS account	Account + MFA + \$0 budget	—
Weeks 5–6	Lab 1: EC2	Web server on the internet, torn down	—
Week 7	Lab 2: S3 + CLI	Static site; CLI fluency	—
Weeks 8–9	Lab 3: VPC	Public/private network, bastion proof	—
Week 10	Lab 4: RDS	Private database, snapshot, teardown	—
Weeks 11–12	Lab 5: IAM; Stage-1 gauntlet rebuild	Full stack from memory, <3 hr	Schedule CCP
Weeks 13–15	Bash, Python/boto3, Git	backup.sh, audit.py, repo history	CCP exam (~Month 4)
Weeks 16–17	Terraform	Stack as code,	—

When	Focus	Hands-on proof	External proof
		plan/apply/destroy	
Weeks 18–20	CI/CD, Docker, K8s literacy	Green pipeline; image in registry	—
Weeks 21–24	CloudWatch, logs, incidents, runbooks	Alarm fired + game-day post-mortem	—
Weeks 25–28	Security ops; cost management	Self-audit report; restore test	—
Months 8–10	Portfolio Projects 1–3	Three documented repos	SAA-C03 (Months 8–9)
Months 10–11	Terraform Associate prep	—	Terraform Associate
Months 11–12	Résumé, interviews, applications	Ten answers, out loud	First offers

A closing word from your teacher. Twelve months is short for a career change and long for a daily habit, so here is the honest deal: the people who finish this course are not the smartest ones — they are the ones who typed the commands on the tired days too. Every error message you hit is the curriculum working, not failing; the engineer who has broken and fixed a hundred small things is exactly what a hiring manager means by “experience.” Keep the journal, tear down what you build, never claim in an interview what your repos can’t back — and a year from now, when the pager goes off at 3 a.m., you’ll feel something unexpected underneath the adrenaline: competence. Go build.

Sources

Job-description research (August 2026): [Arc.dev](#) — [AWS Cloud Engineer Job Description](#) · [Wiz](#) — [Cloud Engineer Job Description Guide](#) · [DevsData](#) — [AWS Cloud Engineer JD Template](#) · [XOPA](#) — [Cloud Engineer JD Template 2026](#) · [Betterteam](#) — [Cloud Engineer Job Description](#).
 Certification path and prep times: [StudyTech](#) — [AWS Certification Roadmap 2026](#) · [Cloud Evolvers](#) — [Cloud Engineer Roadmap 2026](#) · [HashiCorp](#) — [Terraform Associate 003 prep](#). Salary context (US median ≈ \$104K, range \$85K–\$140K): [Wiz](#), above. All YouTube links verified via YouTube metadata at time of writing.

A B4LCILC course — companion volume to “The Cloud Leader Course.”

NEW LESSONS — AUGUST 2026 EDITION

In August 2026, new lessons were added to the three interactive courses at [b4lcilc.org](#) — the Cloud Leader, Cloud Engineer, and Cloud Architect courses. This addendum brings the reading

edition up to date with faithful condensations of that new material, in the same voice as the rest of this book.

A word on what you are holding. These pages are the reading edition: the ideas, the metaphors, and the sentences worth saying out loud. The full interactive versions online carry considerably more — each new lesson ships with quizzes that check your understanding before letting you move on, hands-on labs (including a practice terminal that behaves like the real one but cannot cost money), videos, vocabulary drills, and — for the capstone material — guided design exercises with requirement checklists and worked model answers. Read these condensations to get the shape of each idea into your head; then, when you can, sit the interactive versions to make the knowledge stick.

*The additions fall into three groups. For the **Cloud Leader** course: a gentle introduction to the terminal, Thailand's own privacy law, a leader's guide to AI proposals and AI bills, and a capstone exercise that assembles the entire course into one page. For the **Cloud Engineer** course: serverless computing — the pattern that turns small operational scripts into functions that cost nothing to run. For the **Cloud Architect** course: the arithmetic of availability promises, the Retrieval-Augmented Generation pattern that customers now ask about weekly, and a full costing exercise — pricing a design before anyone builds it.*

For the Cloud Engineer course

What serverless actually is

Everything you have automated so far still assumes a machine: the cron job in this module's lab runs for two minutes a day on a virtual machine you pay for all month, patch, monitor, and one day debug at 3 a.m. Serverless is the cloud's answer to that mismatch. There are still servers — the name is marketing — but they are the provider's problem. You hand over a function, the provider runs it when something triggers it, and you pay per invocation, by the millisecond. The model is called **FaaS — Function as a Service: Lambda** on AWS, **Azure Functions** on Azure. The concepts transfer name-for-name.

The anatomy fits in one hand. A **handler** is the function the platform calls, and it receives an **event** — a JSON (JavaScript Object Notation) description of whatever happened: a file landed in a bucket, an HTTP request arrived, a schedule fired. Sit up at the last one: "a schedule fired" is exactly what cron does — except a Lambda on a schedule needs no machine switched on between runs. Converting one of this module's operational scripts is a classic first serverless project.

The economics are the reason the model exists, and they cut both ways. AWS's free tier includes one million Lambda invocations a month, forever — your lab function could run every minute of the year and cost nothing, making serverless the cheapest home for operations glue. But per-millisecond pricing that is a rounding error at low volume becomes real money at high volume; a function that runs constantly is usually cheaper as a container or a VM. The professional skill is knowing which side of that line a workload sits.

Two honest caveats. First, the **cold start**: a function that has not run recently takes a few hundred milliseconds to wake — irrelevant for a nightly report, a design consideration for a latency-sensitive API. Second, a function still runs as an identity: Lambda assumes an IAM (Identity and Access Management) role exactly as an EC2 instance does, and an over-permissioned function is the same security mistake in a smaller wrapper.
