DevOps 101

Complete Beginner-to-Intermediate Course

From Linux fundamentals to Kubernetes, CI/CD, Terraform, and Monitoring

10 Modules • Labs • 1 Capstone Project

MODULE 1 — Introduction to DevOps

1.1 What is DevOps?

DevOps is a cultural and technical movement that breaks down the wall between software Development and IT Operations teams. Instead of developers writing code in isolation and then 'throwing it over the wall' to operations to deploy and manage, DevOps teams collaborate throughout the entire software lifecycle.

ANALOGY	Traditional development is like building a house room-by-room in secrecy, then handing it to the client all at once. DevOps is open-plan construction — the client walks through every week, all trades work in parallel, and problems are caught early.

Three pillars of DevOps:

- Culture — shared responsibility, no blame culture, psychological safety
- Practices — CI/CD, automation, monitoring, Infrastructure as Code
- Tools — Git, Docker, Kubernetes, Terraform, Prometheus, and many more

1.2 The Software Development Lifecycle (SDLC)

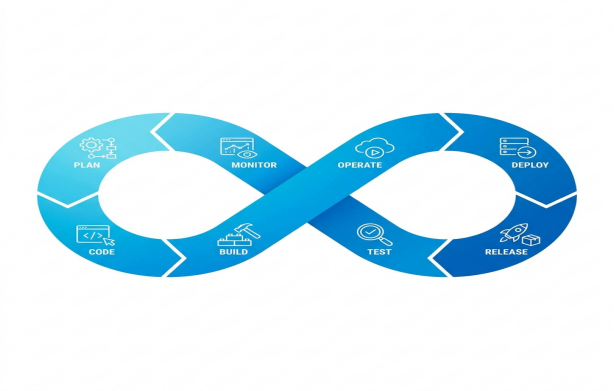
The SDLC is the process a team follows to plan, create, test, and deploy software. Understanding it is essential because DevOps practices improve every stage of this cycle.

Phase	Traditional (Waterfall)	DevOps Approach
Plan	6-month upfront planning	Agile sprints, continuous planning
Code	Dev team works in isolation	Collaborative, small frequent commits
Build	Manual, quarterly releases	Automated builds on every commit
Test	QA team tests after dev	Automated tests run continuously
Release	Big-bang deployments	Small, frequent, low-risk releases
Deploy	Manual, error-prone	Fully automated pipelines
Operate	Ops team scrambles at 2am	Shared on-call, runbooks, alerts

Phase	Traditional (Waterfall)	DevOps Approach
Monitor	Reactive (after outage)	Proactive, dashboards, SLOs

1.3 The DevOps Infinity Loop

The DevOps lifecycle is represented as an infinity loop (figure-8) showing that software development and operations are a continuous, never-ending cycle:



Each stage feeds into the next. A monitoring alert might spark a new plan item. A failed test stops a build. Automation ties it all together.

1.4 CI/CD Overview

CI/CD stands for Continuous Integration and Continuous Delivery/Deployment. It is the backbone of modern DevOps.

Continuous Integration (CI):

- Every developer commits code frequently (multiple times per day)
- Each commit triggers an automated pipeline: lint, test, build
- Broken code is caught within minutes, not days
- Goal: always have a working, tested version of the software

Continuous Delivery (CD):

- After CI passes, the app is automatically packaged and ready to deploy
- Deployment to production requires a human approval button
- The system is always in a deployable state

Continuous Deployment:

- Goes one step further — every passing commit is deployed automatically to production
- Used by companies like Netflix, Amazon, Etsy
- Requires extremely high test coverage and confidence

1.5 Agile Basics for DevOps Engineers

DevOps teams almost always work within an Agile framework. You do not need to be an Agile expert, but you must understand the vocabulary:

Term	Definition	DevOps Relevance
Sprint	A time-boxed iteration (1-4 weeks)	Each sprint ships working software
Standup	Daily 15-min team sync meeting	Where deployment issues are raised
Backlog	Prioritized list of work items	Infrastructure tasks live here too
Retrospective	Sprint review of what went wrong/right	Where DevOps improvements are born
Definition of Done	Criteria for 'this feature is complete'	Includes deployed + monitored

1.6 Why DevOps Matters in Industry

INDUSTRY FACT	According to DORA (DevOps Research & Assessment), elite DevOps teams deploy 973x more frequently, recover from failures 6,570x faster, and have 3x lower change failure rates than low-performing teams.

- Reduced time from idea to production (days instead of months)
- Higher software quality through automation
- Better team morale — less firefighting, more building
- Lower operational costs — automation replaces manual toil
- Competitive advantage — faster feature delivery

1.7 Hands-on Demo: Visualizing a Deployment Workflow

Demo — Manual deployment pain points simulation

Walk through a manual deployment process:

1. Developer finishes a feature on their laptop
2. Creates a ZIP file of the code or pushes to repository
3. Emails it to the ops team telling them he/she pushed the code
4. Ops team SSHs into the server manually
5. Stops the running application
6. Uploads the ZIP, extracts it / Pull new changes from the SCM.
7. Restarts the app — but it crashes (missing dependency!)
8. Dev and Ops teams blame each other. Production is down for 4 hours.

Contrast this with a DevOps pipeline: developer pushes code, automated pipeline runs in 8 minutes, app deploys with zero downtime.

1.8 Lab Exercise — Map Your First DevOps Workflow

Objective: Understand the DevOps lifecycle by applying it to a real scenario

9. Pick a simple web app (e.g., a blog, a to-do list)
10. Draw the complete DevOps lifecycle for that app using the 8-stage loop
11. For each stage, write: (a) who is responsible, (b) what tool could automate it
12. Identify 3 places where things could go wrong in a manual process
13. Share your diagram

EXPECTED OUTPUT	A diagram on paper or digital tool showing all 8 lifecycle stages with annotations for team roles and automation tools at each stage.

1.9 Common Beginner Mistakes

- Thinking DevOps is only about tools — it is culture first, tools second
- Treating CI/CD and DevOps as synonyms — CI/CD is one practice within DevOps
- Believing DevOps eliminates operations — it integrates dev and ops, does not remove either
- Skipping the cultural change and only implementing tools — this always fails

1.10 Module 1 Quiz Questions

14. What does the acronym CI/CD stand for and what is the difference between Continuous Delivery and Continuous Deployment?
15. Name the 8 stages of the DevOps lifecycle in order.
16. What are the 3 pillars of DevOps?
17. In a traditional waterfall model, why does the 'wall' between dev and ops create problems?
18. A company deploys software once every 3 months and has frequent production outages. List 3 DevOps practices that would improve this situation.

MODULE 2 — Linux Fundamentals for DevOps

WHY THIS MATTERS	Over 90% of cloud servers run Linux. Every tool in DevOps — Docker, Kubernetes, Terraform, CI runners — runs on Linux. If you cannot navigate Linux confidently, you cannot work in DevOps.

2.1 The Linux Filesystem Hierarchy

Linux organizes everything in a tree structure starting from the root directory /. Unlike Windows which uses drive letters, everything in Linux is a file or directory hanging from /.

Directory	Purpose	DevOps Relevance
/	Root — top of the entire filesystem	Starting point for all paths
/etc	System configuration files	Nginx config, SSH config, cron jobs
/var	Variable data — logs, caches	Log files: /var/log/nginx/
/home	User home directories	Your shell scripts and projects
/tmp	Temporary files (wiped on reboot)	Scratch space for scripts
/opt	Optional/third-party software	Jenkins, custom tooling
/usr/bin	User executables	Where installed commands live
/proc	Virtual filesystem — system info	ps, top read from here

Essential navigation commands:

```
pwd                    # Print working directory
ls -la                 # List all files with permissions and size
cd /etc/nginx          # Change directory
mkdir -p /opt/app/logs # Create nested directories
find /var/log -name '*.log' -mtime -1  # Find logs modified in last 24h
tree /etc/nginx        # Visual directory tree
```

2.2 File Permissions

Linux permissions control who can read, write, or execute a file. Every file has three permission sets: Owner, Group, and Others.

```
# ls -la output example:
-rwxr-xr-- 1 deploy nginx 2048 Jan 15 10:30 deploy.sh
#  ^^^         owner: rwx (read+write+execute)
#     ^^^      group: r-x (read+execute)
#        ^^^   others: r-- (read only)
```

Changing permissions:

```
chmod 755 deploy.sh     # rwxr-xr-x — owner full, others read+exec
chmod 600 ~/.ssh/id_rsa # rw------- — only owner can read private key
chmod +x script.sh      # Add execute bit to script
chown deploy:nginx app/ # Change owner to 'deploy', group to 'nginx'
chown -R www-data:www-data /var/www/ # Recursive ownership change
```

COMMON MISTAKE Running chmod 777 on everything 'because it works' creates serious security vulnerabilities. Apply least-privilege: give only the permissions actually needed.

2.3 Users, Groups, and sudo

```
# Create a user for your application (never run apps as root!)
sudo useradd -m -s /bin/bash appuser
sudo passwd appuser
sudo usermod -aG sudo appuser   # Add to sudo group

# Check who you are and your groups
whoami
id
groups

# Switch users
su - appuser
sudo -i   # Become root (avoid!)
```

2.4 SSH — Secure Shell

SSH is the primary way you access remote servers. As a DevOps engineer, you will use SSH dozens of times per day.

SSH key pair setup (do this once per machine):

```
# Generate an SSH key pair
ssh-keygen -t ed25519 -C 'your.email@company.com'
# Creates: ~/.ssh/id_ed25519 (private key — NEVER share)
#          ~/.ssh/id_ed25519.pub (public key — put on servers)

# Copy your public key to a server
ssh-copy-id -i ~/.ssh/id_ed25519.pub user@192.168.1.100

# Connect
ssh user@192.168.1.100
ssh -i ~/.ssh/my-key.pem ubuntu@ec2-1-2-3-4.amazonaws.com

# ~/.ssh/config — save connection shortcuts
Host myserver
  HostName 192.168.1.100
  User ubuntu
  IdentityFile ~/.ssh/id_ed25519
# Then just: ssh myserver
```

2.5 Package Management

Package managers install, update, and remove software. Ubuntu uses apt (Advanced Package Tool).

```
sudo apt update                     # Refresh package list
sudo apt upgrade -y                 # Upgrade all packages
sudo apt install nginx curl git -y # Install packages
sudo apt remove nginx               # Remove package
sudo apt autoremove                 # Clean unused dependencies

# Check what is installed
dpkg -l | grep nginx
which nginx                         # Find binary location
nginx -v                            # Check version
```

2.6 Process and Service Management

```
# View running processes
```

```
ps aux                              # All processes
ps aux | grep nginx                 # Find specific process
top                                 # Interactive process viewer
htop                                # Better version of top

# Manage system services with systemctl
sudo systemctl start nginx          # Start service
sudo systemctl stop nginx           # Stop service
sudo systemctl restart nginx        # Restart
sudo systemctl reload nginx         # Reload config without restart
sudo systemctl enable nginx         # Start on boot
sudo systemctl status nginx         # Check status

# Kill a process
kill 1234                           # Graceful (SIGTERM)
kill -9 1234                        # Force kill (SIGKILL)
pkill nginx                         # Kill by name
```

2.7 Networking Basics

```
# Check your IP addresses
ip addr show                        # or: ifconfig

# Test connectivity
ping google.com                     # ICMP ping
curl -I https://google.com          # HTTP headers
wget https://example.com/file       # Download file

# Check listening ports
ss -tulnp                           # Modern way
netstat -tulnp                      # Classic way

# DNS lookup
nslookup google.com
dig google.com

# Check what is using a port
sudo lsof -i :80
sudo ss -tulnp | grep :8080
```

2.8 Shell Scripting

Shell scripts automate repetitive tasks. This is the foundation of DevOps automation.

Full example — server health check script:

```
#!/bin/bash
# health-check.sh — check server health and log results

DATE=$(date '+%Y-%m-%d %H:%M:%S')
LOGFILE='/var/log/health-check.log'
THRESHOLD_CPU=80
THRESHOLD_DISK=90

# Function to check CPU usage
check_cpu() {
  CPU=$(top -bn1 | grep 'Cpu(s)' | awk '{print $2}' | cut -d'%' -f1)
  echo "CPU: ${CPU}%"
  if (( $(echo "$CPU > $THRESHOLD_CPU" | bc -l) )); then
    echo "WARNING: CPU usage is ${CPU}%" | tee -a $LOGFILE
  fi
}

# Function to check disk usage
check_disk() {
  DISK=$(df -h / | awk 'NR==2{print $5}' | cut -d'%' -f1)
  echo "Disk: ${DISK}%"
  if [ $DISK -gt $THRESHOLD_DISK ]; then
    echo "CRITICAL: Disk usage is ${DISK}%" | tee -a $LOGFILE
  fi
}

# Function to check memory
check_memory() {
  MEM=$(free | awk 'NR==2{printf "%.0f", $3*100/$2}')
  echo "Memory: ${MEM}%"
}

echo "[$DATE] Health check started" >> $LOGFILE
check_cpu
check_disk
```

```
check_memory
echo "[$DATE] Health check completed" >> $LOGFILE
```

2.9 Lab Exercises — Linux Fundamentals

Lab 2A: SSH and User Management (45 minutes)

19. SSH into a provided VM: ssh ubuntu@<IP>
20. Create a new user called 'deployer': sudo useradd -m -s /bin/bash deployer
21. Set a password for deployer
22. Add deployer to the sudo group
23. Create a directory /opt/myapp owned by deployer
24. Create a file /opt/myapp/config.txt readable only by deployer
25. Verify permissions: ls -la /opt/myapp/

Lab 2B: Bash Automation Script

26. Write a script called system-report.sh
27. Script must output: hostname, uptime, CPU %, disk %, memory %, top 5 processes by CPU
28. Add a timestamp to every line
29. Save output to /tmp/report-$(date +%F).txt
30. Schedule it to run every hour using cron: crontab -e
31. Verify the cron job: crontab -l

EXPECTED OUTPUT	A working bash script that produces a timestamped report file. The cron job entry should be visible in crontab -l output.

2.10 Module 2 Interview Questions

32. How do you find which process is listening on port 8080?
33. What is the difference between kill and kill -9?
34. Explain the permission string: -rw-r--r-- 1 root root 1024 file.txt
35. How do you make a bash script executable?
36. What does sudo systemctl enable nginx do, and how is it different from start?
37. How do you find all log files modified in the last 24 hours?

MODULE 3 — Git and GitHub

ANALOGY	Git is a time machine for your code. Every commit is a save point you can return to. Branches are parallel universes where you can experiment without breaking the main timeline.

3.1 Why Version Control?

- Track every change ever made to your code, with who made it and why
- Revert to any previous state if something breaks
- Multiple developers can work on the same codebase simultaneously
- Code review process improves quality before changes merge
- Complete audit trail for compliance and debugging

3.2 Core Git Concepts

Concept	Definition	Analogy
Repository (repo)	The project folder tracked by Git	Your project's folder with full history
Commit	A snapshot of changes with a message	A save point in a video game
Branch	An independent line of development	A parallel universe copy of the code
Mergel	Combining changes from one branch to another	Merging two timelines back together
Remote	A copy of the repo on a server (GitHub)	The cloud backup of your project
Clone	Copy a remote repo to your machine	Downloading the project to work on
Pull	Download latest changes from remote	Syncing your local copy with the team's
Push	Upload your local commits to remote	Sharing your work with the team

3.3 Essential Git Commands

Setup and first commit:

```
git config --global user.name 'Your Name'
git config --global user.email 'you@example.com'

git init my-project          # Start a new repo
cd my-project
echo '# My App' > README.md
git add README.md            # Stage the file
git commit -m 'Initial commit: add README'

# Connect to GitHub and push
git remote add origin https://github.com/you/my-project.git
git branch -M main
git push -u origin main
```

Daily workflow commands:

```
git status                   # What has changed?
git diff                     # See exact changes
git log --oneline --graph    # Visual commit history
git add .                    # Stage all changes
git add src/app.js           # Stage specific file
git commit -m 'feat: add user authentication'
git push
git pull                     # Get latest from team
```

3.4 Branching Strategy

Branches allow features to be developed in isolation without affecting the main codebase.

```
# Create and switch to a feature branch
git checkout -b feature/user-login
# or newer syntax:
git switch -c feature/user-login

# Work on your feature, commit...
git add . && git commit -m 'feat: add login form'
git add . && git commit -m 'test: add login unit tests'
```

```
# Push branch to GitHub
git push origin feature/user-login
# Then open a Pull Request on GitHub

# After PR is merged, clean up
git checkout main
git pull
git branch -d feature/user-login
```

Branch naming conventions used in industry:

- feature/user-authentication — new feature development
- bugfix/login-null-pointer — fixing a specific bug
- hotfix/critical-payment-crash — urgent production fix
- release/v2.1.0 — release preparation branch
- chore/update-dependencies — maintenance tasks

3.5 Resolving Merge Conflicts

A merge conflict happens when two people edit the same lines in a file. Git cannot auto-resolve this and asks you to decide.

```
# Git marks conflicting sections like this:
<<<<<<< HEAD (your branch)
const greeting = 'Hello World';
=======
const greeting = 'Hi there!';
>>>>>>> feature/update-greeting

# Resolution steps:
# 1. Open the file and find the conflict markers
# 2. Edit the file to keep the correct version (or combine both)
# 3. Remove ALL conflict markers (<<<<, ====, >>>>)
# 4. Stage the resolved file
git add src/greeting.js
# 5. Complete the merge
git commit -m 'fix: resolve greeting conflict — use formal greeting'
```

PRO TIP	Use VS Code's built-in merge conflict editor. It shows Accept Current / Accept Incoming / Accept Both buttons to resolve conflicts visually without manual marker deletion.

3.6 Pull Requests and Code Review

A Pull Request (PR) is a formal request to merge your branch into main. It is the primary collaboration and quality gate mechanism in modern teams.

Pull Request best practices:

- Keep PRs small — ideally fewer than 400 lines changed
- Write a clear description: what changed, why, and how to test
- Reference the issue number: 'Closes #42'
- Ensure all CI checks pass before requesting review
- Respond to review comments professionally and promptly
- Never merge your own PR without a second reviewer (on production code)

Good PR description template:

```
## What changed
Added email validation to the user registration form.

## Why
Users could register with invalid emails, causing downstream errors.
Fixes #42

## How to test
1. Run: npm test
2. Try registering with 'notanemail' — should see validation error
3. Try registering with 'valid@email.com' — should succeed
```

3.7 Git Workflows

Workflow	Description	Best for
GitHub Flow	Branch from main, PR, merge. Simple.	Most teams, continuous deployment

Workflow	Description	Best for
GitFlow	develop, release, hotfix branches. Complex.	Versioned software with release schedule
Trunk-based	Everyone commits to main frequently, feature flags	Large teams, very high CI maturity

3.8 Mini-Project — Team Collaboration Simulation

Objective: Practice the full GitHub collaboration workflow

This exercise requires 2 people. If working alone, use two browser tabs or two terminal sessions.

38. Person A creates a GitHub repository called 'team-sim' with a README
39. Person A invites Person B as a collaborator
40. Both persons clone the repo to their machines
41. Person A creates branch feature/homepage, adds an index.html with a heading
42. Person B creates branch feature/about, adds an about.html
43. Both persons push their branches and open Pull Requests
44. Person A reviews and approves Person B's PR — Person B does the same
45. Both PRs are merged into main
46. Both persons do git pull on main and verify they see each other's files
47. Instructor introduces a conflict: both persons edit README.md on new branches
48. Resolve the conflict together and document the resolution in a comment

3.9 Common Beginner Mistakes

- Committing directly to main — always use feature branches
- Writing vague commit messages like 'fix stuff' or 'changes' — be specific
- Committing secrets (passwords, API keys) to the repo — use .gitignore and environment variables
- Forgetting to pull before creating a branch — always git pull first
- Force-pushing to shared branches — destroys teammates' history

Essential .gitignore entries for every project:

```
.env                    # Environment variables / secrets
node_modules/           # Dependencies (install from package.json)
```

```
__pycache__/          # Python compiled files
*.log                 # Log files
.terraform/           # Terraform provider downloads
*.tfstate             # Terraform state — use remote backend
```

3.10 Module 3 Interview Questions

49. What is the difference between git merge and git rebase? When would you use each?
50. How do you recover a deleted branch that was not yet pushed?
51. What is the difference between git pull and git fetch?
52. How do you remove a file from Git tracking without deleting it from disk?
53. A developer accidentally committed a password to GitHub. What do you do?

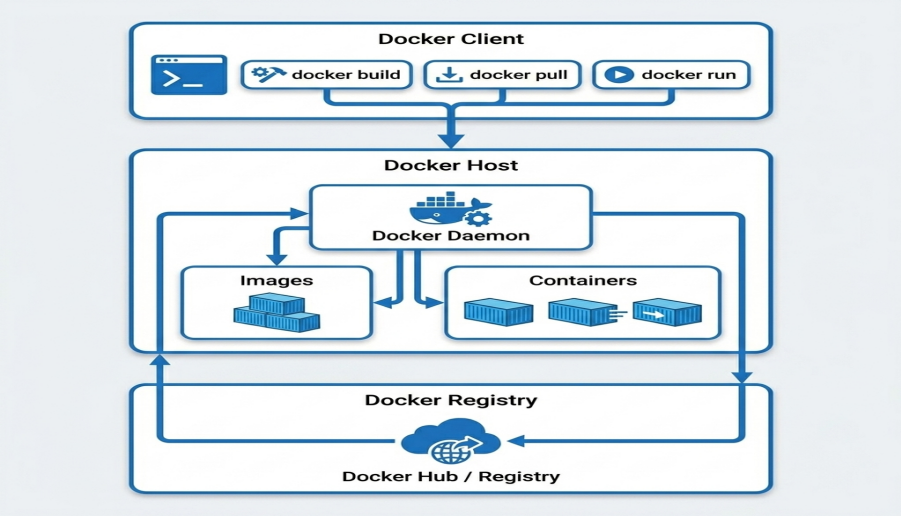
MODULE 4 — Docker Fundamentals

ANALOGY	A Docker container is like a shipping container. Whether the cargo ship is in China, Germany, or the USA, the container behaves identically — same contents, same temperature, same configuration. Your app runs the same way in dev, staging, and production.

4.1 Containers vs Virtual Machines

Aspect	Virtual Machine	Container
What it includes	Full OS + kernel + libraries + app	App + libraries only (shares host kernel)
Startup time	Minutes (boots full OS)	Seconds (process starts)
Size	Gigabytes (10–50GB)	Megabytes (50–500MB)
Isolation	Strong (separate kernel)	Process-level (shared kernel)
Use case	Running different OS types	Microservices, consistent deployments
Resource overhead	High	Very low

4.2 Docker Architecture



- Docker Client — the CLI you type commands into
- Docker Daemon — the background service that does the actual work
- Docker Images — read-only templates used to create containers
- Docker Containers — running instances of images

- Docker Registry — storage for images (Docker Hub is the public one)

4.3 Working with Images and Containers

```
# Pull an image from Docker Hub
docker pull nginx:latest
docker pull node:18-alpine

# List local images
docker images

# Run a container
docker run -d -p 8080:80 –nam-nnmy-nginx nginx
#           ^   ^          ^
#      detached  port map   container name

# Run interactively (for debugging)
docker run -it ubuntu:22.04 /bin/bash

# Manage containers
docker ps                    # Running containers
docker ps -a                 # All containers (including stopped)
docker stop my-nginx
docker start my-nginx
docker rm my-nginx           # Delete container
docker logs -f my-nginx      # Follow logs
docker exec -it my-nginx bash  # Open shell in running container
```

4.4 Writing a Dockerfile

A Dockerfile is a script of instructions that builds a Docker image. Each instruction creates a layer.

Node.js application Dockerfile:

```
# Layer 1: Base image (Alpine = minimal Linux, ~5MB)
FROM node:18-alpine

# Layer 2: Set working directory inside container
WORKDIR /app
```

```
# Layer 3: Copy dependency files first (cache optimization)
COPY package*.json ./

# Layer 4: Install dependencies
RUN npm install --production

# Layer 5: Copy application code
COPY . .

# Layer 6: Expose the port the app listens on
EXPOSE 3000

# Layer 7: Create non-root user for security
RUN addgroup -S appgroup && adduser -S appuser -G appgroup
USER appuser

# Layer 8: Command to run when container starts
CMD ["node", "server.js"]
```

Python Flask application Dockerfile:

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
EXPOSE 5000
ENV FLASK_APP=app.py
ENV FLASK_ENV=production
CMD ["flask", "run", "--host=0.0.0.0"]
```

Build and publish:

```
# Build the image
docker build -t my-app:v1 .
docker build -t my-app:v1 -f Dockerfile.prod .  # Custom Dockerfile

# Tag for Docker Hub
docker tag my-app:v1 yourusername/my-app:v1
```

```
# Login and push
docker login
docker push yourusername/my-app:v1
```

4.5 Docker Volumes — Persisting Data

Containers are ephemeral — when a container is deleted, its data is lost. Volumes persist data outside the container lifecycle.

```
# Named volume (Docker manages it)
docker volume create mydata
docker run -d -v mydata:/var/lib/postgresql/data do-m

# Bind mount (maps host directory to container)
docker run -d -v $(pwd)/nginx.conf:/etc/nginx/nginx.conf nginx
docker run -d -v $(pwd)/app:/app node:18 node /app/server.js

# List and inspect volumes
docker volume ls
docker volume inspect mydata
```

4.6 Docker Compose — Multi-Container Applications

Most real applications have multiple services: an API, a database, a cache, maybe a message queue. Docker Compose defines and runs all of them together.

Full docker-compose.yml example — web app with database and cache:

```
version: '3.8'

services:
  # 1. The application
  app:
    build: .
    container_name: myapp
    ports:
      - '3000:3000'
    environment:
      - NODE_ENV=production
      - DATABASE_URL=postgresql://user:pass@db:5432/mydb
```

```
      - REDIS_URL=redis://cache:6379
    depends_on:
      db:
        condition: service_healthy
    volumes:
      - ./logs:/app/logs
    restart: unless-stopped

  # 2. PostgreSQL database
  db:
    image: postgres:15-alpine
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: pass
      POSTGRES_DB: mydb
    volumes:
      - postgres_data:/var/lib/postgresql/data
    healthcheck:
      test: ['CMD-SHELL', 'pg_isready -U user']
      interval: 10s
      timeout: 5s
      retries: 5

  # 3. Redis cache
  cache:
    image: redis:7-alpine
    volumes:
      - redis_data:/data

volumes:
  postgres_data:
  redis_data:
```

Docker Compose commands:

```
docker compose up -d          # Start all services in background
docker compose down           # Stop and remove containers
docker compose logs -f app    # Follow app logs
docker compose ps             # Status of all services
docker compose exec app bash  # Shell into running app
docker compose restart app    # Restart one service
docker compose pull           # Pull latest images
```

4.7 Common Beginner Mistakes

- Using the 'latest' tag in production — always pin versions (node:18.20.3, not node:latest)
- Running the app as root inside the container — creates security risk
- Copying node_modules or .env into the image — use .dockerignore
- Not ordering Dockerfile layers for cache efficiency — COPY package.json before COPY . .
- Hardcoding secrets in Dockerfile ENV — use runtime environment injection

Essential .dockerignore file:

```
node_modules
.env
*.log
.git
Dockerfile
docker-compose*.yml
README.md
.gitignore
```

4.8 Lab Exercise — Containerize a Real Application

Lab: Node.js containerization (60 minutes)

54. Create a simple Express.js API with two endpoints: GET / and GET /health
55. Write a Dockerfile following the pattern from section 4.4
56. Build the image: docker build -t myapi:v1 .
57. Run it: docker run -d -p 3000:3000 myapi:v1
58. Test: curl http://localhost:3000 and curl http://localhost:3000/health
59. View logs: docker logs <container-id>
60. Push to Docker Hub: docker push yourusername/myapi:v1

Lab: Docker Compose with database

61. Add a PostgreSQL service to your docker-compose.yml
62. Update your app to connect to PostgreSQL using a connection string from environment

63. Add a GET /users endpoint that queries the database
64. Run: docker compose up -d and test all endpoints
65. Verify data persists: stop containers, restart, check data is still there

4.9 Module 4 Interview Questions

66. What is the difference between CMD and ENTRYPOINT in a Dockerfile?
67. How do you reduce Docker image size?
68. What is a multi-stage Docker build and when would you use it?
69. How does Docker networking work? Explain the bridge network.
70. What happens to data in a container when the container is deleted?

MODULE 5 — CI/CD Pipelines

ANALOGY	A CI/CD pipeline is like an automated car assembly line. Raw materials (code) enter one end and a finished, tested, quality-checked product exits the other — automatically, every time, without human intervention.

5.1 Why CI/CD?

- Without CI: broken code sits undetected for days. Integration is painful and slow.
- Without CD: deployments are manual, error-prone, and terrifying. Deploys happen rarely.
- With CI/CD: every commit is validated in minutes. Deployments are boring because they happen 10+ times per day.

Without CI/CD	With CI/CD
Monthly 'big bang' releases	Daily or hourly small releases
Integration problems found after weeks	Caught within minutes of commit
Manual testing takes days	Automated tests run in minutes
Deployments require all-hands war room	Automated, unattended, boring
Rollback takes hours	Rollback in 2 minutes

5.2 CI/CD Pipeline Stages

5.3 GitHub Actions

GitHub Actions is GitHub's built-in CI/CD system. Workflows are defined in YAML files inside .github/workflows/. They are free for public repos and have a generous free tier for private repos.

Complete GitHub Actions pipeline:

```
# .github/workflows/pipeline.yml
name: CI/CD Pipeline

# Trigger: run on push to main or any pull request
on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

env:
  IMAGE_NAME: myrepo/my-app

jobs:
  test:
    name: Test
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '18'
          cache: 'npm'

      - name: Install dependencies
        run: npm ci

      - name: Run linter
        run: npm run lint
```

```
    - name: Run unit tests
      run: npm test -- --coverage

build-push:
  name: Build and Push Docker Image
  runs-on: ubuntu-latest
  needs: test                   # Only runs if test job passes
  if: github.ref == 'refs/heads/main'  # Only on main branch
  steps:
    - uses: actions/checkout@v4

    - name: Login to Docker Hub
      uses: docker/login-action@v3
      with:
        username: ${{ secrets.DOCKER_USERNAME }}
        password: ${{ secrets.DOCKER_TOKEN }}

    - name: Build and push
      uses: docker/build-push-action@v5
      with:
        push: true
        tags: |
          ${{ env.IMAGE_NAME }}:latest
          ${{ env.IMAGE_NAME }}:${{ github.sha }}

    - name: Scan image for vulnerabilities
      uses: aquasecurity/trivy-action@master
      with:
        image-ref: '${{ env.IMAGE_NAME }}:${{ github.sha }}'
        severity: 'HIGH,CRITICAL'
        exit-code: '1'          # Fail pipeline if critical vuln found

deploy-staging:
  name: Deploy to Staging
  runs-on: ubuntu-latest
  needs: build-push
  environment: staging
  steps:
    - name: Deploy to staging server
      run: |
        echo 'Deploying ${{ github.sha }} to staging'
```

```
          # SSH to staging server and pull new image
```

5.4 Secrets Management in Pipelines

NEVER put passwords, tokens, or API keys directly in your YAML files. They will be visible in your Git history forever.

Setting up GitHub secrets:

71. Go to your GitHub repo → Settings → Secrets and variables → Actions
72. Click 'New repository secret'
73. Name: DOCKER_USERNAME, Value: your Docker Hub username
74. Name: DOCKER_TOKEN, Value: Docker Hub access token (not password)
75. Reference in YAML: ${{ secrets.DOCKER_USERNAME }}

SECURITY RULE	Always use access tokens (not passwords) for CI/CD. Tokens can be scoped to specific permissions and revoked without changing your password. Rotate them every 90 days.

5.5 Jenkins Pipeline

Jenkins is a self-hosted, open-source CI/CD server. It gives you full control but requires more setup than GitHub Actions.

Jenkinsfile (Declarative Pipeline):

```
pipeline {
  agent any

  environment {
    IMAGE_NAME = 'myrepo/my-app'
    DOCKER_CREDS = credentials('docker-hub-creds')
  }

  stages {
    stage('Checkout') {
      steps {
        git branch: 'main', url: 'https://github.com/you/my-app.git'
      }
```

```
    }
    stage('Test') {
      steps {
        sh 'npm install && npm test'
      }
    }
    stage('Build') {
      steps {
        sh 'docker build -t ${IMAGE_NAME}:${BUILD_NUMBER} .'
      }
    }
    stage('Push') {
      steps {
        sh '''
          echo $DOCKER_CREDS_PSW | docker login -u $DOCKER_CREDS_USR --
password-stdin
          docker push ${IMAGE_NAME}:${BUILD_NUMBER}
        '''
      }
    }
    stage('Deploy') {
      steps {
        sh 'kubectl set image deployment/my-app
app=${IMAGE_NAME}:${BUILD_NUMBER}'
      }
    }
  }
  post {
    failure {
      mail to: 'team@company.com', subject: 'Pipeline FAILED', body:
"${env.JOB_NAME} failed"
    }
  }
}
```

5.6 Lab Exercise — Build a CI/CD Pipeline

Lab 5A: GitHub Actions CI

76. Take the Node.js app from Module 4
77. Create .github/workflows/ci.yml
78. Pipeline must: checkout → install → lint → test → build Docker image

79. Add DOCKER_USERNAME and DOCKER_TOKEN to GitHub secrets
80. Extend pipeline to push image to Docker Hub on merge to main
81. Test it: push a commit and watch the Actions tab run live

Lab 5B: Breaking and fixing the pipeline

82. Introduce a deliberate bug in a test (make an assertion fail)
83. Push to a branch and open a PR — watch CI fail
84. Fix the bug, push again — watch CI pass
85. Merge the PR — watch the full pipeline run including Docker push
86. Discuss: what would have happened without CI?

5.7 Module 5 Interview Questions

87. What is the difference between Continuous Delivery and Continuous Deployment?
88. A test fails in your pipeline at 3am on an automated push. What happens next?
89. How do you prevent secrets from appearing in pipeline logs?
90. What is the purpose of the needs keyword in GitHub Actions?
91. What is a blue/green deployment and why is it used?
92. How do you roll back a bad deployment in a CI/CD pipeline?

MODULE 6 — Cloud Fundamentals (AWS)

ANALOGY	The cloud is like renting a fully serviced office versus buying a building. You pay for what you use, the landlord (AWS) handles maintenance, and you can expand or shrink overnight.

6.1 Cloud Service Models

Model	You manage	Provider manages	Examples
IaaS	OS, runtime, app, data	Hardware, networking, hypervisor	AWS EC2, Azure VM, GCP Compute
PaaS	App code and data only	OS, runtime, middleware, scaling	Heroku, AWS Elastic Beanstalk, Google App Engine
SaaS	Configuration and data	Everything else	Gmail, Salesforce, GitHub

6.2 Core AWS Services

Compute:

- EC2 (Elastic Compute Cloud) — virtual machines in the cloud
- Lambda — run code without managing servers (serverless)
- ECS/EKS — run Docker containers or Kubernetes

Storage:

- S3 (Simple Storage Service) — object storage for files, images, backups
- EBS (Elastic Block Store) — persistent disk storage attached to EC2
- EFS (Elastic File System) — shared file system across multiple EC2 instances

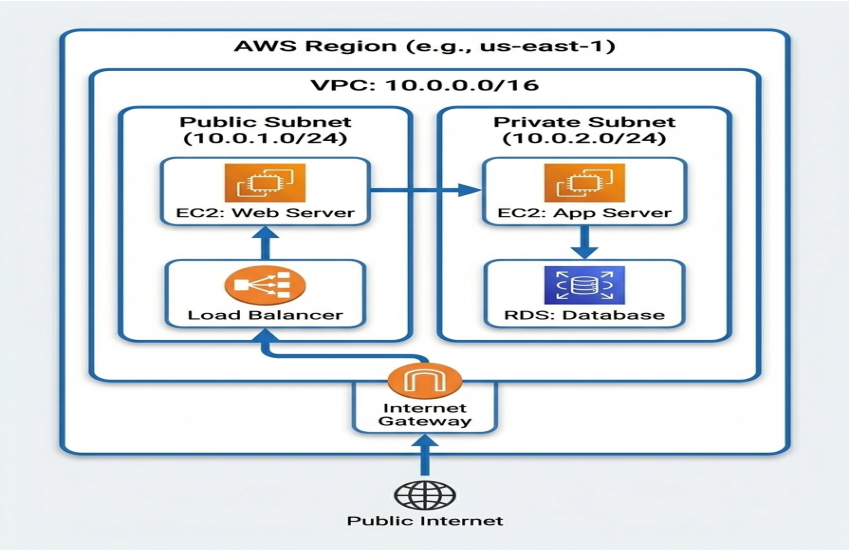
Networking:

- VPC (Virtual Private Cloud) — your isolated network in AWS
- Security Groups — stateful firewalls at the instance level
- Route 53 — DNS service
- ALB (Application Load Balancer) — distribute traffic across instances

Database:

- RDS — managed relational database (PostgreSQL, MySQL, etc.)
- DynamoDB — managed NoSQL database
- ElastiCache — managed Redis/Memcached

6.3 VPC — Virtual Private Cloud



- Public subnet — resources with internet access (web servers, load balancers)
- Private subnet — resources with NO direct internet access (databases, internal services)
- Internet Gateway — the door between your VPC and the internet
- NAT Gateway — allows private subnet resources to initiate outbound connections

6.4 Security Groups

Security groups are virtual firewalls that control traffic to EC2 instances. They are stateful — if you allow inbound traffic, the response is automatically allowed back out.

Rule type	Protocol	Port	Source	Purpose
Inbound	TCP	22	Your IP only (x.x.x.x/32)	SSH access

Rule type	Protocol	Port	Source	Purpose
Inbound	TCP	80	0.0.0.0/0	HTTP web traffic
Inbound	TCP	443	0.0.0.0/0	HTTPS web traffic
Inbound	TCP	5432	App security group ID	PostgreSQL from app only
Outbound	All	All	0.0.0.0/0	All outbound traffic (default)

SECURITY RULE NEVER open port 22 (SSH) to 0.0.0.0/0 (the whole internet). Always restrict SSH to your specific IP address or use a bastion host/VPN.

6.5 IAM — Identity and Access Management

IAM controls who can do what in your AWS account. The golden rule is least privilege — give only the permissions needed.

- IAM User — a person or application that needs AWS access
- IAM Role — a set of permissions that can be assumed temporarily (by EC2, Lambda, etc.)
- IAM Policy — a JSON document defining allowed/denied actions
- MFA (Multi-Factor Authentication) — always enable on root and admin accounts

IAM Policy example — S3 read-only for a specific bucket:

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Action": ["s3:GetObject", "s3:ListBucket"],
    "Resource": [
      "arn:aws:s3:::my-app-bucket",
      "arn:aws:s3:::my-app-bucket/*"
    ]
  }]
}
```

6.6 Lab Exercise — Launch and Deploy on AWS

Lab 6A: Launch EC2 and deploy Nginx (90 minutes)

93. Log into AWS Console (ensure billing alerts are set at $5)
94. Navigate to EC2 → Launch Instance
95. Choose: Ubuntu 22.04 LTS, t2.micro (free tier), create new key pair
96. Configure security group: allow SSH from your IP, HTTP from anywhere
97. In Advanced Details → User Data, paste this script:

```
#!/bin/bash
apt update -y
apt install -y nginx
echo '<h1>DevOps 101 Lab 6 - Hello from EC2!</h1>' >
/var/www/html/index.html
systemctl enable nginx && systemctl start nginx
```

98. Launch the instance, wait 2 minutes, open the public IP in browser
99. SSH in: ssh -i your-key.pem ubuntu@<public-ip>
100. Verify Nginx is running: systemctl status nginx
101. IMPORTANT: Stop (not terminate) the instance after lab. Set a reminder to terminate it.

6.7 Module 6 Interview Questions

102. What is the difference between a Security Group and a Network ACL?
103. What is the difference between an IAM Role and an IAM User?
104. Explain the difference between S3 and EBS.
105. What is a VPC and why do you need one?
106. How would you make an EC2 instance accessible on the internet?

MODULE 7 — Infrastructure as Code with Terraform

ANALOGY	Clicking through the AWS console is configuring infrastructure with Post-it notes. Terraform is having a blueprint. The same blueprint always produces the same building, you can version it in Git, and rebuilding from scratch takes 5 minutes.

7.1 What is Infrastructure as Code?

- Define infrastructure (servers, networks, databases) in code files
- Version infrastructure alongside application code in Git
- Recreate entire environments automatically and consistently
- Review infrastructure changes before applying them (like code review for infra)
- Destroy and rebuild environments easily — great for ephemeral dev/test environments

Without IaC	With IaC (Terraform)
Click through console manually	Run terraform apply
Config undocumented — only in someone's head	Configuration in version-controlled .tf files
Prod and staging environments drift	Identical environments from same code
Hard to reproduce after disaster	Full rebuild in minutes from state + code
No audit trail of infrastructure changes	Git history shows every change

7.2 Terraform Core Concepts

Concept	Definition	Example
Provider	Plugin that talks to a cloud/service API	hashicorp/aws, hashicorp/google
Resource	A cloud resource Terraform manages	aws_instance, aws_s3_bucket
Data source	Read existing infrastructure (don't create)	Get latest Ubuntu AMI ID

Concept	Definition	Example
Variable	Input values to make configs reusable	var.region, var.instance_type
Output	Values to display after apply	Public IP of created instance
State	File tracking what Terraform created	terraform.tfstate
Module	Reusable package of Terraform resources	vpc module, ec2 module

7.3 Terraform Workflow

```
terraform init      # Download providers, set up backend
terraform plan      # Preview changes (like a diff — shows what WILL
happen)
terraform apply     # Create/modify infrastructure
terraform destroy   # Delete all managed infrastructure
terraform fmt       # Format .tf files
terraform validate  # Check syntax
terraform state list # Show managed resources
```

7.4 Complete Terraform Example — AWS EC2 + VPC

File structure:

```
terraform-project/
├── main.tf           # Resource definitions
├── variables.tf      # Input variable declarations
├── outputs.tf        # Output value declarations
├── providers.tf      # Provider configuration
└── terraform.tfvars # Variable values (do NOT commit secrets)
```

providers.tf:

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

```
  # Remote state backend — prevents state file in Git
  backend "s3" {
    bucket = "my-terraform-state"
    key    = "devops101/terraform.tfstate"
    region = "us-east-1"
  }
}

provider "aws" {
  region = var.region
}
```

variables.tf:

```
variable "region" {
  description = "AWS region to deploy into"
  type        = string
  default     = "us-east-1"
}

variable "instance_type" {
  description = "EC2 instance type"
  type        = string
  default     = "t2.micro"
}

variable "environment" {
  description = "Environment name"
  type        = string
}
```

main.tf:

```
# Get the latest Ubuntu 22.04 AMI
data "aws_ami" "ubuntu" {
  most_recent = true
  owners      = ["099720109477"] # Canonical
  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-*"]
  }
}
```

```
# Security group
resource "aws_security_group" "web" {
  name        = "${var.environment}-web-sg"
  description = "Allow web traffic"

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

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["${var.my_ip}/32"]
  }

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

# EC2 instance
resource "aws_instance" "web" {
  ami                    = data.aws_ami.ubuntu.id
  instance_type          = var.instance_type
  vpc_security_group_ids = [aws_security_group.web.id]

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

  tags = {
```

```
    Name        = "${var.environment}-web"
    Environment = var.environment
    ManagedBy   = "Terraform"
  }
}
```

outputs.tf:

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

output "instance_id" {
  value = aws_instance.web.id
}
```

7.5 Common Beginner Mistakes

- Committing terraform.tfstate to Git — contains sensitive data, use S3 remote backend
- Running apply without reviewing plan first — always read the plan carefully
- Hardcoding AWS credentials in .tf files — use environment variables or IAM roles
- Not tagging resources — you will not know what each resource is for when bill arrives
- Forgetting to run terraform destroy after labs — costs money

7.6 Module 7 Interview Questions

107. What is Terraform state and why is it important?
108. What happens if two engineers run terraform apply at the same time?
109. How do you import existing AWS resources into Terraform management?
110. What is the difference between terraform destroy and removing a resource from your .tf file?
111. How would you manage different environments (dev, staging, prod) with Terraform?

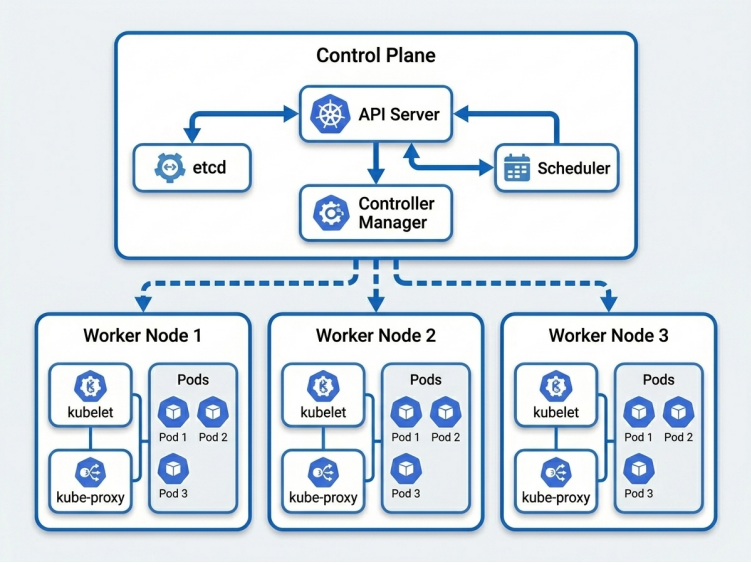
MODULE 8 — Kubernetes Fundamentals

ANALOGY	Docker runs a single container like one musician playing alone. Kubernetes is the conductor of an orchestra — it decides how many musicians play (replicas), automatically replaces sick musicians (self-healing), and ensures the concert never stops (zero downtime).

8.1 Why Kubernetes?

- Docker alone cannot automatically restart crashed containers in production
- Docker alone cannot distribute traffic across multiple containers
- Docker alone has no built-in health checking or self-healing
- Docker alone cannot roll out updates with zero downtime
- Kubernetes solves all of these — it is the industry standard for container orchestration

8.2 Kubernetes Architecture



8.3 Core Kubernetes Objects

Pod — the smallest unit:

```yaml
# pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-app
  labels:
    app: my-app
spec:
  containers:
  - name: app
    image: myrepo/my-app:v1
    ports:
    - containerPort: 3000
    resources:
      requests:
        memory: '64Mi'
        cpu: '100m'
      limits:
        memory: '128Mi'
        cpu: '200m'
```

Deployment — manages replicas and rolling updates:

```yaml
# deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  namespace: production
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  template:
    metadata:
      labels:
```

```
        app: my-app
    spec:
      containers:
      - name: app
        image: myrepo/my-app:v2
        ports:
        - containerPort: 3000
        livenessProbe:
          httpGet:
            path: /health
            port: 3000
          initialDelaySeconds: 10
          periodSeconds: 30
        readinessProbe:
          httpGet:
            path: /ready
            port: 3000
          initialDelaySeconds: 5
          periodSeconds: 10
```

Service — exposes pods to network traffic:

```
# service.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  selector:
    app: my-app     # Routes to all pods with this label
  ports:
  - port: 80        # Port on the service
    targetPort: 3000 # Port on the pod
  type: LoadBalancer # ClusterIP | NodePort | LoadBalancer
```

ConfigMap and Secret:

```
# configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
```

```
data:
  LOG_LEVEL: 'info'
  ENVIRONMENT: 'production'
  MAX_CONNECTIONS: '100'

# secret.yaml (values must be base64 encoded)
apiVersion: v1
kind: Secret
metadata:
  name: app-secrets
type: Opaque
data:
  # echo -n 'mypassword' | base64
  DB_PASSWORD: bXlwYXNzd29yZA==
  API_KEY: c2VjcmV0a2V5MTIz
```

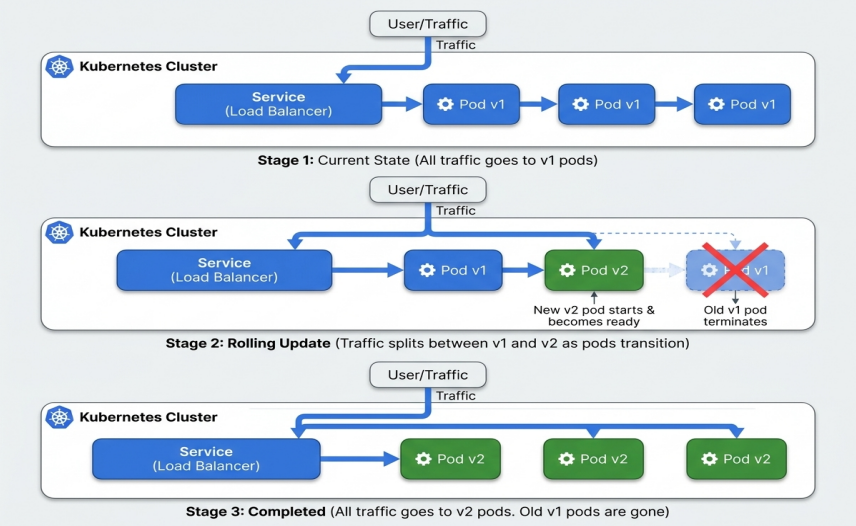
8.4 Essential kubectl Commands

```
# Apply manifests
kubectl apply -f deployment.yaml
kubectl apply -f ./k8s/          # Apply all files in directory

# View resources
kubectl get pods
kubectl get pods -o wide          # Show which node each pod is on
kubectl get pods -w               # Watch in real time
kubectl get all                   # Get everything
kubectl get events --sort-by=.lastTimestamp

# Debug
kubectl logs my-app-7d8f-abc12    # View pod logs
kubectl logs -f my-app-7d8f-abc12 # Follow logs
kubectl logs my-app-7d8f-abc12 --previous  # Crashed container logs
kubectl describe pod my-app-7d8f-abc12      # Full pod details
kubectl exec -it my-app-7d8f-abc12 -- bash  # Shell into pod

# Scaling
kubectl scale deployment my-app --replicas=5
kubectl autoscale deployment my-app --min=3 --max=10 --cpu-percent=70

# Rolling updates and rollbacks
```

```
kubectl set image deployment/my-app app=myrepo/my-app:v3
kubectl rollout status deployment/my-app
kubectl rollout history deployment/my-app
kubectl rollout undo deployment/my-app
kubectl rollout undo deployment/my-app --to-revision=2
```

8.5 Rolling Update — Zero Downtime Deployment

Kubernetes performs rolling updates by replacing pods one at a time, ensuring some replicas are always running.



8.6 Lab Exercises — Kubernetes

Lab 8A: Deploy to Minikube (120 minutes)

112. Install Minikube: minikube start
113. Apply the deployment.yaml from section 8.3
114. Apply the service.yaml — use type: NodePort for Minikube
115. Get the service URL: minikube service my-app-service --url
116. Test the app in browser
117. Scale to 5 replicas: kubectl scale deployment my-app --replicas=5
118. Watch pods come up: kubectl get pods -w
119. Delete one pod and watch Kubernetes restart it automatically

Lab 8B: Rolling update and rollback (60 minutes)

120. Update the image to a new version: kubectl set image deployment/my-app app=myrepo/my-app:v2

121. Watch the rolling update: kubectl rollout status deployment/my-app
122. View rollout history: kubectl rollout history deployment/my-app
123. Simulate bad deployment: update to a non-existent image tag
124. Watch pods get stuck in ImagePullBackOff
125. Roll back: kubectl rollout undo deployment/my-app
126. Verify the app is running the previous version again

8.7 Module 8 Interview Questions

127. What is the difference between a Pod, a ReplicaSet, and a Deployment?
128. What is the difference between a liveness probe and a readiness probe?
129. How does Kubernetes decide which node to schedule a pod on?
130. What happens when you delete a Pod that is managed by a Deployment?
131. What is the difference between a ClusterIP, NodePort, and LoadBalancer service?
132. How do you debug a pod that is stuck in CrashLoopBackOff?

MODULE 9 — Monitoring and Logging

ANALOGY	Running an application without monitoring is flying a plane without instruments. You might be perfectly on course — or 10,000 feet too low. You will not know until it is too late.

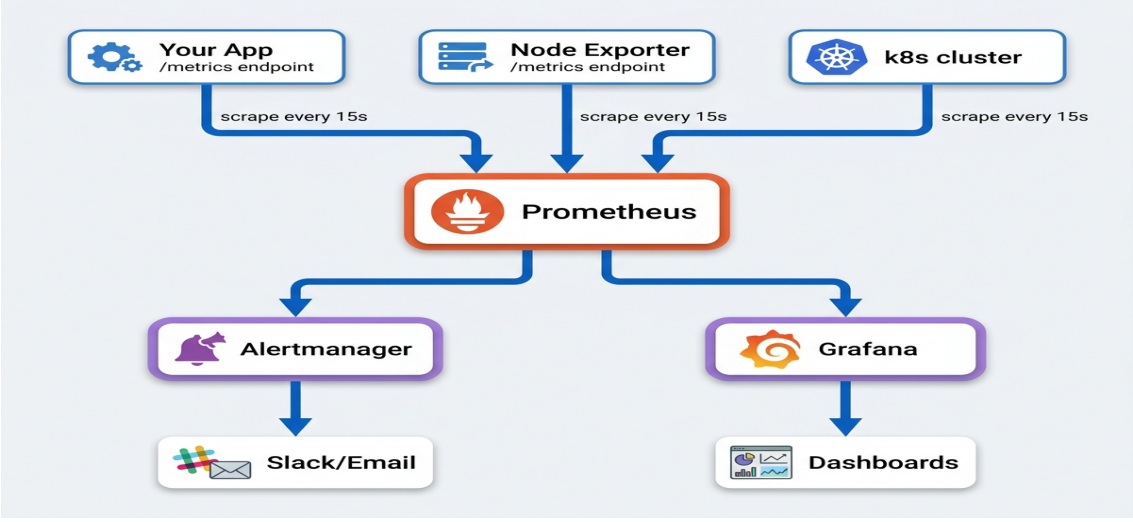
9.1 The 3 Pillars of Observability

Pillar	Definition	Tools	Example
Metrics	Numeric measurements over time	Prometheus, Datadog, CloudWatch	CPU: 78%, Error rate: 0.1%, RPS: 1200
Logs	Timestamped text records of events	ELK stack, Loki, Splunk	2024-01-15 10:30:01 ERROR: DB connection failed
Traces	End-to-end request journey	Jaeger, Zipkin, AWS X-Ray	Request spent 200ms in auth, 800ms in DB query

9.2 Prometheus — Metrics Collection

Prometheus is an open-source monitoring system that scrapes metrics from applications and stores them in a time-series database.

Prometheus architecture:



Metric types:

- Counter — always increases. Total requests served, errors. Example: http_requests_total
- Gauge — can go up or down. Current memory usage, active connections. Example: memory_used_bytes
- Histogram — distribution of values. Request latency buckets. Example: http_request_duration_seconds
- Summary — similar to histogram but calculated on client side

prometheus.yml — scrape configuration:

```
global:
  scrape_interval: 15s
  evaluation_interval: 15s

rule_files:
  - 'alerts.yml'

alerting:
  alertmanagers:
  - static_configs:
    - targets: ['alertmanager:9093']

scrape_configs:
  - job_name: 'my-app'
    static_configs:
      - targets: ['app:3000']

  - job_name: 'node-exporter'
    static_configs:
      - targets: ['node-exporter:9100']

  - job_name: 'kubernetes-pods'
    kubernetes_sd_configs:
      - role: pod
```

Key PromQL queries:

```
# Request rate per second (5 min window)
rate(http_requests_total[5m])
```

```
# Error rate
rate(http_requests_total{status=~'5..'}[5m])

# 95th percentile latency
histogram_quantile(0.95, rate(http_request_duration_seconds_bucket[5m]))

# Targets that are down
up == 0

# Memory usage as percentage
(node_memory_MemTotal_bytes - node_memory_MemFree_bytes) /
node_memory_MemTotal_bytes * 100
```

9.3 Grafana — Dashboards and Visualization

Grafana connects to Prometheus (and many other data sources) to create beautiful, actionable dashboards.

Essential dashboard panels for a web application:

- Request rate (requests per second) — is the app receiving traffic?
- Error rate (errors per second) — is the app returning errors?
- Latency (p50, p95, p99) — how fast is the app responding?
- CPU usage — is the server overloaded?
- Memory usage — is the app leaking memory?
- Pod health (for Kubernetes) — how many pods are running?

Complete Docker Compose monitoring stack:

```
# monitoring/docker-compose.yml
version: '3.8'
services:
  prometheus:
    image: prom/prometheus:latest
    ports:
      - '9090:9090'
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
      - ./alerts.yml:/etc/prometheus/alerts.yml
```

```
      - prometheus_data:/prometheus
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--storage.tsdb.retention.time=30d'

  grafana:
    image: grafana/grafana:latest
    ports:
      - '3001:3000'
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=admin123
    volumes:
      - grafana_data:/var/lib/grafana

  alertmanager:
    image: prom/alertmanager:latest
    ports:
      - '9093:9093'
    volumes:
      - ./alertmanager.yml:/etc/alertmanager/alertmanager.yml

  node-exporter:
    image: prom/node-exporter:latest
    ports:
      - '9100:9100'

volumes:
  prometheus_data:
  grafana_data:
```

9.4 Alerting

Alert rules (alerts.yml):

```
groups:
- name: application
  rules:
  - alert: HighErrorRate
    expr: rate(http_requests_total{status=~'5..'}[5m]) > 0.05
    for: 2m
    labels:
```

```
      severity: critical
    annotations:
      summary: 'High error rate on {{ $labels.job }}'
      description: 'Error rate is {{ $value | humanizePercentage }}'

  - alert: AppDown
    expr: up{job='my-app'} == 0
    for: 1m
    labels:
      severity: critical
    annotations:
      summary: 'Application is down'

  - alert: HighMemoryUsage
    expr: (node_memory_MemTotal_bytes - node_memory_MemFree_bytes) /
node_memory_MemTotal_bytes > 0.9
    for: 5m
    labels:
      severity: warning
    annotations:
      summary: 'Memory usage above 90%'
```

9.5 Instrumenting Your Application

Node.js app with Prometheus metrics (prom-client):

```
const express = require('express');
const client = require('prom-client');

const app = express();

// Create a Registry
const register = new client.Registry();
client.collectDefaultMetrics({ register });

// Custom counter for HTTP requests
const httpRequestsTotal = new client.Counter({
  name: 'http_requests_total',
  help: 'Total HTTP requests',
  labelNames: ['method', 'route', 'status'],
  registers: [register]
```

```
});

// Custom histogram for request duration
const httpDuration = new client.Histogram({
  name: 'http_request_duration_seconds',
  help: 'Request duration in seconds',
  labelNames: ['method', 'route'],
  registers: [register]
});

// Middleware to record metrics
app.use((req, res, next) => {
  const end = httpDuration.startTimer();
  res.on('finish', () => {
    httpRequestsTotal.inc({ method: req.method, route: req.path, status:
res.statusCode });
    end({ method: req.method, route: req.path });
  });
  next();
});

// Expose metrics endpoint for Prometheus
app.get('/metrics', async (req, res) => {
  res.set('Content-Type', register.contentType);
  res.end(await register.metrics());
});
```

9.6 Lab Exercise — Full Monitoring Stack

Lab 9A: Deploy Prometheus + Grafana (120 minutes)

133. Use the docker-compose.yml from section 9.3
134. Run: docker compose up -d
135. Open Prometheus at http://localhost:9090 — verify targets are up
136. Open Grafana at http://localhost:3001 — login admin/admin123
137. Add Prometheus as data source in Grafana
138. Create a new dashboard with 4 panels: CPU, memory, request rate, error rate
139. Trigger a high error rate by calling a bad endpoint repeatedly
140. Watch the error rate metric spike in real time on your dashboard

9.7 Module 9 Interview Questions

141. What are the 3 pillars of observability?
142. What is the difference between a Prometheus Counter and a Gauge?
143. What is alert fatigue and how do you prevent it?
144. How would you debug a production issue where users report the app is 'slow'?
145. What is SLO, SLA, and SLI? Give an example of each.

MODULE 10 — DevOps Best Practices and Security

10.1 Secrets Management

CRITICAL	Secrets in code repos, Docker images, or CI/CD logs are one of the most common causes of security breaches. Treat secrets like a house key — never leave them lying around.

Secrets management approaches:

- Environment variables — inject at runtime, never bake into images
- GitHub Actions Secrets — encrypted, not visible in logs
- AWS Secrets Manager — centralized, audited, automatic rotation
- HashiCorp Vault — open-source, self-hosted secrets management
- Kubernetes Secrets — base64 encoded (not encrypted by default, use sealed secrets)

How to scan for secrets accidentally committed:

```
# Install git-secrets or use GitHub's secret scanning
pip install detect-secrets
detect-secrets scan > .secrets.baseline
git diff HEAD | detect-secrets-hook --baseline .secrets.baseline

# Use Trivy to scan Docker images
trivy image myrepo/my-app:latest
```

10.2 Container Security Best Practices

```
# Good Dockerfile security practices
FROM node:18-alpine  # Use minimal base images

# Never run as root
RUN addgroup -S appgroup && adduser -S appuser -G appgroup

# Set permissions before switching user
COPY --chown=appuser:appgroup . .
USER appuser
```

```
# Read-only filesystem (add in docker run or compose)
# docker run --read-only ...

# Scan image before pushing
trivy image --severity HIGH,CRITICAL myapp:v1
```

10.3 Backup Strategies

Strategy	Description	Recovery Time
3-2-1 Rule	3 copies, 2 media types, 1 offsite	Hours
Daily snapshots	EBS snapshots, RDS automated backups	30-60 minutes
Continuous replication	Multi-region database replication	Minutes
Point-in-time recovery	Restore to any second in the last 35 days	Minutes

- RTO (Recovery Time Objective) — how long can you be down? Defines backup strategy.
- RPO (Recovery Point Objective) — how much data can you lose? Defines backup frequency.

10.4 Incident Response

The on-call response process:

146. Alert fires → on-call engineer is paged
147. Acknowledge alert (stop escalation)
148. Assess severity: P1 (site down) → P2 (degraded) → P3 (minor)
149. Communicate: post in incident Slack channel 'Investigating X'
150. Investigate: check logs, metrics, recent deployments
151. Mitigate: rollback, scale up, restart service
152. Resolve: confirm service restored, close incident
153. Post-mortem: within 48 hours, blameless analysis

Blameless post-mortem template:

```
# Incident Post-Mortem
Date: 2024-01-15
Duration: 47 minutes
Severity: P1

## Summary
Database connection pool exhausted, causing 503 errors for all users.

## Timeline
10:15 - Alertmanager fires HighErrorRate alert
10:17 - On-call engineer acknowledges
10:25 - Root cause identified: connection pool limit hit
10:32 - Connection limit increased, service restored

## Root Cause
5-why analysis:
1. Users got 503 errors
2. App could not connect to database
3. DB connection pool was exhausted (limit: 10)
4. New feature increased DB queries per request by 5x
5. No load testing was done before deployment

## Action Items
- [ ] Increase connection pool limit to 50 (Owner: @alice, Due: Jan 16)
- [ ] Add connection pool metrics to Grafana dashboard
- [ ] Add load testing to CI pipeline
```

10.5 Real-World Case Studies

Case Study 1: GitLab DB deletion (2017)

- What happened: Ops engineer accidentally deleted production database while trying to remove replica. 300GB of data lost.
- Root cause: No tested backup restore procedure. Manual, error-prone process.
- Lesson: Always test your backups. Automated, tested restores. Runbooks for all dangerous operations.

Case Study 2: AWS US-East-1 outage (2021)

- What happened: Network configuration error took down S3 in US-East-1, cascading to many AWS services and internet-connected apps.
- Root cause: Single-region dependencies, cascading failures.
- Lesson: Design for multi-region or at least graceful degradation. Never have a single point of failure.

Case Study 3: Secrets in Git (common, frequent)

- What happens: Developer commits AWS access keys to a public GitHub repo.
- Result: Within minutes, bots scan GitHub and begin mining cryptocurrency in the victim's account. Bills can reach $50,000+.
- Lesson: Use git-secrets pre-commit hooks. Rotate keys immediately if exposed. Use IAM roles instead of keys where possible.

CAPSTONE PROJECT — End-to-End DevOps Pipeline

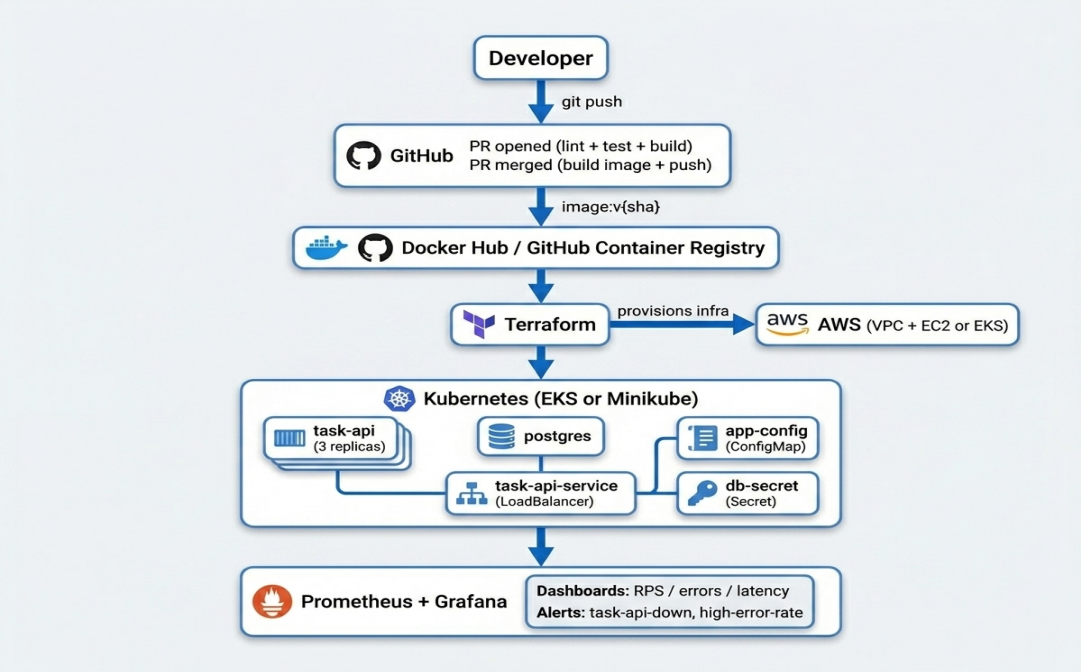
OVERVIEW	You are joining a startup as their first DevOps engineer. Build a production-grade CI/CD pipeline for their Task Manager API from scratch. This simulates a real junior DevOps role.

Application: Task Manager API

Endpoints to implement:

```
GET    /health        -> {status: 'ok', version: '1.0.0'}
GET    /metrics       -> Prometheus metrics
GET    /tasks         -> List all tasks
POST   /tasks         -> Create a task {title, description}
DELETE /tasks/:id     -> Delete a task
```

Capstone Architecture



Capstone Deliverable Checklist

- Create GitHub repo with README documenting the project
- Write the Task Manager API (Node.js or Python Flask)
- Write unit tests with at least 80% coverage
- Write Dockerfile — passes trivy scan with no CRITICAL vulns
- Write docker-compose.yml with app + postgres + prometheus + grafana
- Test locally: docker compose up, all endpoints working

Week 2 — Automation

- Create .github/workflows/ci.yml: lint → test → build image → push
- Pipeline passes on a clean commit and fails on a broken test
- Add security scan (trivy) as a pipeline stage
- Write Terraform to provision: VPC, EC2/EKS, security groups, IAM role
- Apply Terraform: terraform plan shows expected resources, terraform apply succeeds
- Write Kubernetes manifests: deployment.yaml, service.yaml, configmap.yaml, secret.yaml
- Deploy app to Kubernetes: kubectl apply -f k8s/
- Verify: kubectl get pods shows 3/3 pods Running

Week 3 — Observability and Polish

- Deploy Prometheus and Grafana to Kubernetes monitoring namespace
- App exposes /metrics endpoint with http_requests_total counter and latency histogram
- Build Grafana dashboard with: RPS, error rate, p95 latency, pod count
- Create alert rule: fires when app is down for more than 1 minute
- Demonstrate rolling update: change app version, zero downtime confirmed
- Demonstrate rollback: kubectl rollout undo deployment/task-api
- Write 1-page post-mortem for a simulated incident
- Record 5-minute video demo of the full pipeline in action

DevOps Glossary

Term	Definition
Artifact	A deployable output of a build: Docker image, JAR file, zip package
Blue/Green deployment	Running two identical production environments, switching traffic to the new one
Canary deployment	Routing a small percentage of traffic to a new version before full rollout
CNCF	Cloud Native Computing Foundation — oversees Kubernetes, Prometheus, and many other tools
Container	A lightweight, isolated process that packages app code and dependencies
Drift	When actual infrastructure state no longer matches the desired state in code
Feature flag	A toggle that enables/disables a feature without redeployment
GitOps	Using Git as the single source of truth for both code AND infrastructure state
Golden path	The recommended, pre-configured path for developers to ship software
Helm	Kubernetes package manager — installs complex apps with a single command
Idempotent	An operation that produces the same result no matter how many times you run it
Immutable infrastructure	Never modify running servers — replace them entirely with new ones
Ingress	Kubernetes resource that routes external HTTP/S traffic to services
Mean Time to Recovery (MTTR)	Average time to restore service after a failure
Namespace	A Kubernetes virtual cluster for isolating resources within a cluster
Observability	The ability to understand a system's internal state from its external outputs
Operator	Kubernetes controller that automates complex application management

Term	Definition
Pipeline as Code	Defining CI/CD pipelines in version-controlled code files
Pod	The smallest deployable unit in Kubernetes — one or more containers
RBAC	Role-Based Access Control — controlling who can do what in a system
Runbook	Step-by-step documentation for handling a specific operational scenario
Shift left	Moving testing and security earlier in the development process
SLI	Service Level Indicator — a metric measuring service performance
SLO	Service Level Objective — a target value for an SLI (e.g., 99.9% uptime)
SLA	Service Level Agreement — a contract with customers about service performance
Stateless	An application that stores no data locally — can be replaced or scaled freely

Command Cheat Sheets

Linux Cheat Sheet

```
# Navigation
pwd | ls -la | cd / | mkdir -p a/b | cp src dst | mv src dst | rm -rf dir

# Permissions
chmod 755 file | chown user:group file | ls -la

# Processes
ps aux | grep nginx | kill -9 PID | systemctl start/stop/status nginx

# Network
ip addr | ss -tulnp | curl -I url | ping host | nslookup domain

# Files
find / -name '*.log' | grep -r 'error' /var/log | tail -f /var/log/syslog
cat file | head -20 | wc -l | sed 's/old/new/' | awk '{print $1}'
```

Git Cheat Sheet

```
git init | git clone url | git add . | git commit -m 'msg' | git push | git pull
git checkout -b feature/x | git merge main | git rebase main
git log --oneline --graph | git diff | git stash | git stash pop
git reset HEAD~1 | git revert SHA | git tag v1.0.0
```

Docker Cheat Sheet

```
docker build -t name:tag . | docker run -d -p 8080:80 name | docker ps -a
docker logs -f id | docker exec -it id bash | docker stop id | docker rm id
docker images | docker rmi image | docker pull image | docker push image
docker compose up -d | docker compose down | docker compose logs -f service
```

kubectl Cheat Sheet

```
kubectl apply -f file.yaml | kubectl get pods/svc/deploy | kubectl describe pod name
kubectl logs -f pod | kubectl exec -it pod -- bash | kubectl delete pod name
kubectl scale deploy name --replicas=5 | kubectl rollout undo deploy/name
kubectl get events | kubectl top pods | kubectl config get-contexts
```

Terraform Cheat Sheet

```
terraform init | terraform plan | terraform apply | terraform destroy
terraform fmt | terraform validate | terraform state list | terraform output
terraform import aws_instance.web i-1234567890 | terraform taint resource
```

Recommended Resources and Reading

Books

- The Phoenix Project — Gene Kim (DevOps culture in novel form — essential reading)
- The DevOps Handbook — Gene Kim, Patrick Debois, John Willis
- Site Reliability Engineering — Google (free at sre.google/books)
- Kubernetes in Action — Marko Luksa
- Terraform: Up & Running — Yevgeniy Brikman

Online Platforms

- Linux Foundation free courses — training.linuxfoundation.org
- AWS Skill Builder — explore.skillbuilder.aws
- Terraform documentation — developer.hashicorp.com/terraform
- Kubernetes documentation — kubernetes.io/docs
- GitHub Skills — skills.github.com

Communities

- DevOps subreddit — r/devops

- CNCF Slack — cloud-native.slack.com
- HashiCorp Discuss — discuss.hashicorp.com
- Docker Community — forums.docker.com

End of DevOps 101 Course Content