GrowthSchool

DEVOPS

COMPLETE NOTES

Seven complete courses - beginner to pro - in one edition

DevOps Engineer

Everything you need to become a DevOps Engineer.

BUILT FROM THESE TOOLS

Linux | Git | Docker | Kubernetes | Terraform | AWS

CODE - SHIP - OPERATE

Diagrams, real commands, labs and revision snapshots

growthschool.cc | @growthschool.cc

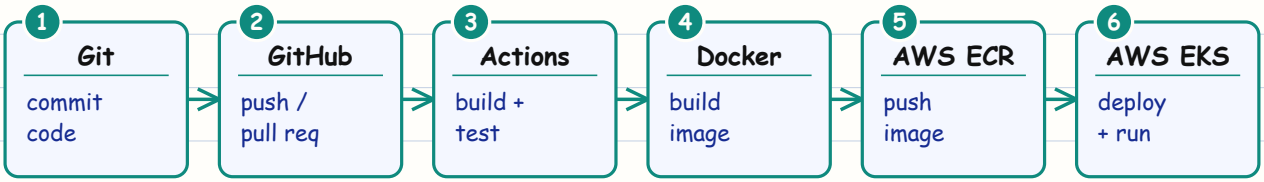
Master Contents & Learning Path

GrowthSchool

This combined edition bundles all seven GrowthSchool handwritten notebooks. Each part is a complete beginner-to-advanced course with diagrams, real commands, labs and revision snapshots. They share one running example: an ecommerce "Shop API".

* The Seven Parts

* The CI/CD Pipeline: from Git to AWS EKS



One push flows all the way to a running app - automatically.

HOW TO USE THIS EDITION

New here? Read the parts in order: Linux, Git, Docker, Kubernetes, Terraform, AWS.
Short on time? Jump to any part's cheat sheet and revision pages.
Type every command yourself as you go.

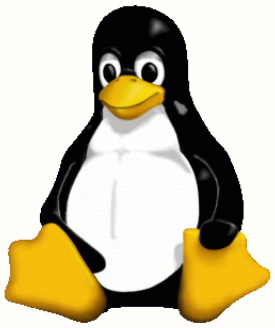
WHAT YOU WILL BE ABLE TO DO

By the end you can package an app in Docker, provision cloud infra with Terraform, ship it Git -> GitHub -> ECR, and run it on AWS EKS.

GrowthSchool

LINUX

COMPLETE NOTES



LEARN - ADMIN - AUTOMATE

Filesystem, permissions, processes, networking, scripting and ops

growthschool.cc | @growthschool.cc

Learning Roadmap

GrowthSchool

Work through these in order. Every page has explanation, real commands, a diagram or example, common mistakes, a student lab and a revision snapshot. Examples target a Shop API server.

HOW TO USE THESE NOTES

Practice on a real Linux box or a container (docker run -it ubuntu bash). Reading is not doing - type every command. Use pages 30-35 for fast revision before interviews.

1. Why Linux for DevOps

GrowthSchool

IN SIMPLE TERMS
Linux is a free operating system that runs most servers, containers, and cloud machines. As a DevOps engineer you will operate it every day, usually through the command line.

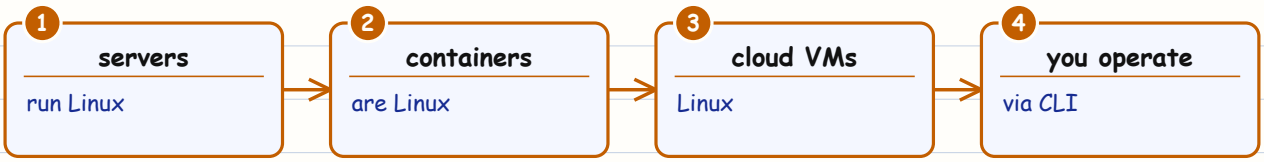
* It Runs the World

- Most servers, containers and cloud instances run Linux. Docker images are Linux userlands.
- If you deploy software, you operate Linux - directly or through containers and CI runners.

* What You Actually Do

- Navigate the filesystem, manage permissions, read logs, control services and write small scripts.
- The command line is faster, scriptable and works identically over SSH on a remote server.

Linux everywhere



REAL-WORLD EXAMPLE

```
uname -a          # kernel + arch
whoami; id        # who am I, my groups
cat /etc/os-release   # which distro + version
uptime            # load average + how long up
```

COMMON MISTAKE
Expecting a GUI on servers. Production Linux is headless; you work over SSH in a terminal. Get comfortable without a mouse.

REVISION SNAPSHOT

ASK YOURSELF Why is Linux fluency non-negotiable for DevOps?

DO THIS Identify your distro, kernel and current user with one command each.

2. The Shell & Command Line

GrowthSchool

IN SIMPLE TERMS
The shell is the program that reads the commands you type and runs them; the command line is the text screen where you type them.

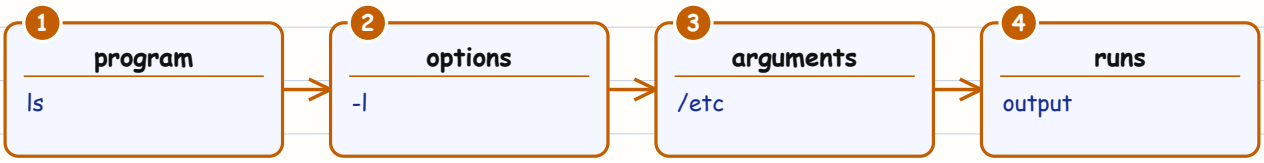
* What the Shell Is

- The shell (usually bash) reads commands and runs programs. A prompt shows user@host:path$.
- A command is: program + options (flags) + arguments, e.g. ls -l /etc.

* Work Faster

- Tab completes names; up-arrow recalls history; Ctrl-R searches it; man/--help explain commands.
- Ctrl-C cancels a command; Ctrl-L clears the screen; Ctrl-A/E jump to line start/end.

A command



REAL-WORLD EXAMPLE

```
ls -l /etc
man ls            # full manual
ls --help         # quick options
history | tail    # recent commands
type ls; which python3
```

COMMON MISTAKE
Retyping long commands. Use history, Ctrl-R and tab completion. Also read man pages instead of guessing flags - the answer is usually right there.

REVISION SNAPSHOT

ASK YOURSELF	What are the three parts of a typical command?
DO THIS	Use Ctrl-R to find and rerun a command from earlier in your session.

3. Filesystem Hierarchy

GrowthSchool

IN SIMPLE TERMS
The filesystem hierarchy is how Linux organises everything under one tree starting at "/", with fixed folders for config, logs, users, and programs.

* One Tree

- Linux has a single tree rooted at /. There are no drive letters; devices mount into the tree.
- Everything is a file - including devices (/dev) and kernel info (/proc, /sys).

* Know These Homes

- Config lives in /etc, logs in /var/log, users in /home, programs in /usr, temp in /tmp.
- Knowing where things live turns "where is that?" into an instant answer.

KEY DIRECTORIES

/ root of everything
/etc system configuration
/home user home directories
/var logs, spools, variable data
/usr installed programs + libs

MORE PATHS

/bin /sbin essential binaries
/tmp temporary files
/dev device files
/proc /sys kernel + process info
/opt /srv add-on + served data

REAL-WORLD EXAMPLE

```
ls /            # top-level layout
ls /etc         # system config files
ls /var/log     # system + app logs
df -h           # mounted filesystems
```

REVISION SNAPSHOT

ASK YOURSELF Where do config, logs and user homes live?

DO THIS Explore /etc, /var/log and /home and name one file in each.

4. Navigation & Paths

GrowthSchool

IN SIMPLE TERMS
Navigation is moving around that folder tree - knowing where you are (pwd) and changing directory (cd) using full or relative paths.

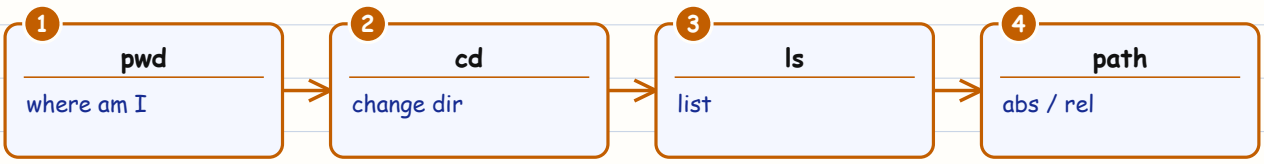
* Move Around

- pwd prints where you are; cd changes directory; ls lists. Absolute paths start with /.
- Relative paths are from your current directory. . is here, .. is up, ~ is your home.

* Speed Moves

- cd - jumps to the previous directory; cd with no argument goes home.
- ls -lah shows long listing, hidden files and human-readable sizes.

Move around



REAL-WORLD EXAMPLE

```
pwd
cd /var/log && ls -lah
cd ~            # home
cd -            # previous directory
cd ../../etc    # up two, then into etc
```

COMMON MISTAKE
Confusing absolute and relative paths. /log is not the same as ./log. When unsure, run pwd and use an absolute path.

REVISION SNAPSHOT

ASK YOURSELF	What do . , .. , ~ and - mean in cd?
DO THIS	Navigate to /var/log using an absolute path, then home using a shortcut.

5. File & Directory Operations

GrowthSchool

IN SIMPLE TERMS
File operations are the everyday actions of creating, copying, moving, and deleting files and folders.

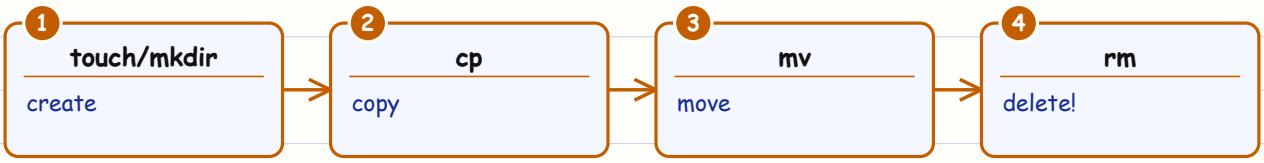
* Create, Copy, Move, Remove

- touch makes an empty file; mkdir -p makes nested dirs; cp copies; mv moves/renames; rm deletes.
- cp -r and rm -r work recursively on directories.

* Be Careful

- rm is permanent - there is no recycle bin. Double-check paths, especially with -r and wildcards.

File actions



REAL-WORLD EXAMPLE

```
mkdir -p shop/api/logs
touch shop/api/app.py
cp -r shop shop-backup
mv shop/api/app.py shop/api/server.py
rm -i shop-backup/api/server.py   # -i asks first
```

COMMON MISTAKE
Running rm -rf with a wrong or empty variable, e.g. rm -rf "$DIR"/ where DIR is unset -> rm -rf /.
Quote variables, avoid trailing slashes on variables, and prefer rm -i when unsure.

REVISION SNAPSHOT

ASK YOURSELF	Why is rm especially dangerous with -r and wildcards?
DO THIS	Create a nested folder tree, copy it, rename a file, then remove the copy safely.

6. Viewing & Editing Files

GrowthSchool

IN SIMPLE TERMS
Viewing and editing files means reading their contents (cat, less, tail) and changing them in a terminal editor like nano or vim.

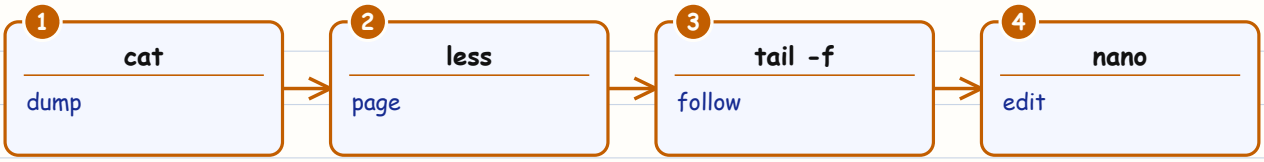
* Read Without Editing

- cat dumps a file; less pages through it (q to quit); head/tail show the start/end.
- tail -f follows a growing file live - essential for watching logs.

* Edit In Place

- nano is the easy editor; vim is powerful but has a learning curve (i to insert, Esc, :wq to save).
- On servers you edit config with these; know at least nano well.

View a file



REAL-WORLD EXAMPLE

```
less /var/log/syslog
head -n 20 access.log
tail -f /var/log/nginx/access.log
nano /etc/hosts
grep -n ERROR app.log | less
```

COMMON MISTAKE
cat-ing a huge log and flooding the terminal. Use less, head, tail, or grep to look at just the part you need.

REVISION SNAPSHOT

ASK YOURSELF	Which command follows a log file as new lines arrive?
DO THIS	Tail -f a log while triggering activity in another terminal.

7. Permissions & Ownership

GrowthSchool

IN SIMPLE TERMS
Permissions decide who can read, write, or run a file; ownership says which user and group a file belongs to.

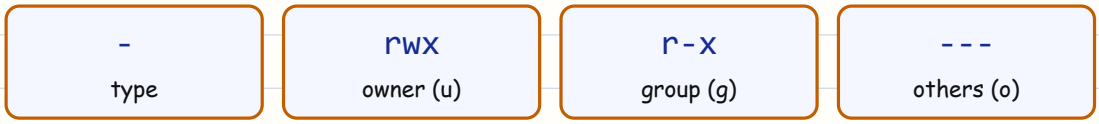
* The Model

- Every file has an owner (user), a group, and permissions for user/group/others.
- Permissions are read (r), write (w), execute (x). ls -l shows them as rwxr-x---.

* On Directories

- x on a directory means "can enter it"; r means "can list it"; w means "can create/delete inside".
- This is why a dir often needs x even just to cd into it.

Reading rwxr-x--- (chmod 750)



r=4 w=2 x=1 -> rwx=7, r-x=5, ---=0 -> 750

REAL-WORLD EXAMPLE

```
ls -l server.py
# -rwxr-x--- 1 asha devs 240 ... server.py
stat server.py       # detailed metadata
namei -l /home/asha/shop/app.py  # path perms
```

REVISION SNAPSHOT

ASK YOURSELF What does x mean on a file vs on a directory?

DO THIS Read ls -l output and state who can read/write/execute a file.

8. chmod, chown & umask

GrowthSchool

IN SIMPLE TERMS
chmod changes permissions, chown changes ownership, and umask sets the default permissions given to newly created files.

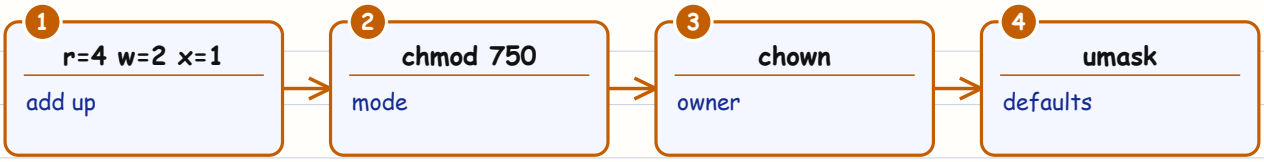
* Change Permissions

- chmod sets permissions numerically (750) or symbolically (u+x, go-w).
- r=4, w=2, x=1; add them per class: rwx=7, r-x=5, r--=4.

* Ownership & Defaults

- chown user:group file changes ownership (usually needs sudo).
- umask sets default permissions for new files; a umask of 022 yields 644 files, 755 dirs.

Set permissions



REAL-WORLD EXAMPLE

```
chmod 750 deploy.sh        # rwx r-x ---
chmod u+x,go-w deploy.sh   # symbolic
sudo chown asha:devs app.log
chmod -R 755 /var/www/shop
umask         # show current default mask
```

COMMON MISTAKE
chmod 777 to "fix" a permission error. That makes files world-writable - a security hole. Grant the least permission that works, usually 644 for files and 755 for dirs/scripts.

REVISION SNAPSHOT

ASK YOURSELF	What numeric mode is rwxr-x--- and how did you get it?
DO THIS	Make a script executable for owner+group only, then run it.

9. Users, Groups & sudo

GrowthSchool

IN SIMPLE TERMS
Users are individual accounts, groups bundle users together, and sudo lets a normal user run a single command with admin power.

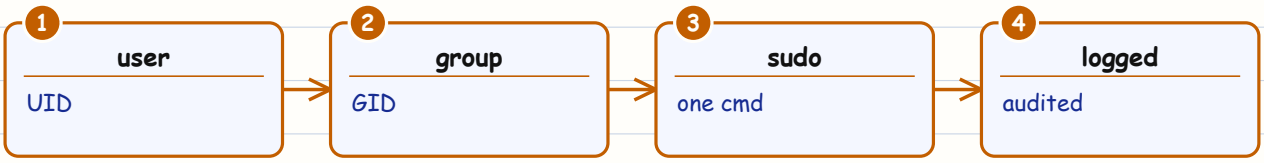
* Accounts

- Users have a UID; groups have a GID. /etc/passwd lists users, /etc/group lists groups.
- Add users with useradd/adduser; add to a group with usermod -aG group user.

* sudo

- sudo runs one command as another user (usually root). It is safer than logging in as root.
- Every sudo use is logged - accountability and a smaller blast radius than a root shell.

Accounts & sudo



REAL-WORLD EXAMPLE

```
id asha
sudo adduser deploy
sudo usermod -aG docker deploy   # add to docker group
groups deploy
sudo -l          # what can I run as sudo?
```

COMMON MISTAKE
Working as root all day "to avoid permission errors". One typo can wreck the system. Use a normal user and sudo only for the specific commands that need it.

REVISION SNAPSHOT

ASK YOURSELF	Why prefer sudo over a persistent root login?
DO THIS	Create a user, add it to a group, and confirm with id and groups.

10. Processes & Signals

GrowthSchool

IN SIMPLE TERMS
A process is a running program; a signal is a message you send it, for example to ask it to stop gracefully or force it to quit.

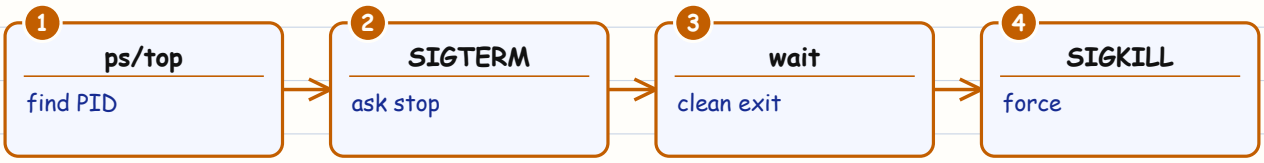
* Seeing Processes

- Every running program is a process with a PID. ps and top/htop list them with CPU/memory use.
- A parent process spawns children; killing a parent can orphan or end its children.

* Signals

- kill sends signals: SIGTERM (15) asks nicely to stop; SIGKILL (9) forces it; SIGHUP (1) reloads.
- Always try SIGTERM first so the app can shut down cleanly and flush data.

Manage a process



REAL-WORLD EXAMPLE

```
ps aux | grep shop-api
top            # live view (q to quit)
kill 4821      # SIGTERM (graceful)
kill -9 4821   # SIGKILL (last resort)
pkill -f shop-api
```

COMMON MISTAKE
Reaching for kill -9 first. It gives the process no chance to clean up, risking corrupt data or leftover lock files. Try SIGTERM, wait, then escalate.

REVISION SNAPSHOT

ASK YOURSELF	What is the difference between SIGTERM and SIGKILL?
DO THIS	Start a long-running command, find its PID, and stop it gracefully.

11. Jobs & Background Control

GrowthSchool

IN SIMPLE TERMS
Job control lets you run tasks in the background, pause and resume them, and keep them running after you log out.

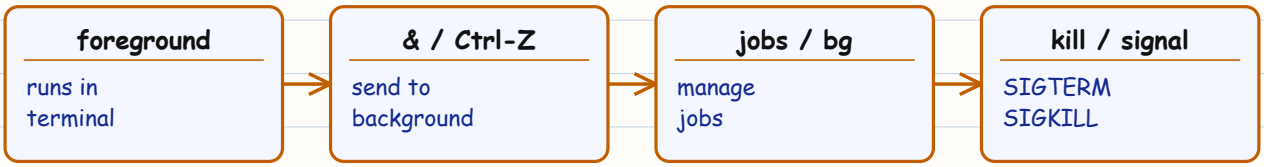
* Run in Background

- Append & to run a command in the background; Ctrl-Z suspends the foreground job.
- jobs lists them; fg brings one to the foreground; bg resumes a suspended job in the background.

* Survive Logout

- A background job dies when the shell exits, unless you use nohup or a terminal multiplexer (tmux).
- For real services, use systemd instead of nohup - it supervises and restarts them.

Process & Job Control Flow



REAL-WORLD EXAMPLE

```
./long-task.sh &     # background
jobs
fg %1                # foreground job 1
nohup ./worker.sh &  # survive logout
tmux new -s deploy   # persistent session
```

COMMON MISTAKE
Using nohup for a production service. It has no restart, no logging discipline and no status. Package long-running services as systemd units instead.

REVISION SNAPSHOT

ASK YOURSELF	How do you keep a task running after you log out?
DO THIS	Background a task, suspend and resume it, then run one with nohup.

12. systemd & Services

GrowthSchool

IN SIMPLE TERMS
systemd is the manager that starts and supervises background services on modern Linux; systemctl is how you control them.

* systemd Basics

- systemd is PID 1: it starts and supervises services (units). Manage them with systemctl.
- A unit file (.service) declares how to start, restart and depend on other units.

* Everyday Control

- start/stop/restart control a service now; enable/disable set whether it starts at boot; status shows state.

Linux Boot Sequence

REAL-WORLD EXAMPLE

```
sudo systemctl status nginx
sudo systemctl restart shop-api
sudo systemctl enable --now shop-api   # start + on boot
systemctl list-units --type=service
systemctl cat shop-api      # show the unit file
```

COMMON MISTAKE
Confusing start with enable. start runs it now; enable makes it start on boot. You usually want both: systemctl enable --now.

REVISION SNAPSHOT

ASK YOURSELF	What is the difference between systemctl start and enable?
DO THIS	Check a service status, restart it, and ensure it starts on boot.

13. Logs & journalctl

GrowthSchool

IN SIMPLE TERMS
Logs are the records of what the system and apps did; journalctl reads the logs collected by systemd.

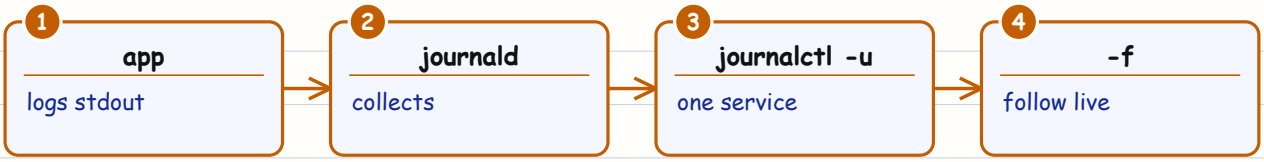
* Where Logs Live

- Traditional logs are text files in /var/log. systemd stores logs in the journal (journalctl).
- Good apps log to stdout/stderr; systemd/containers capture that automatically.

* Query the Journal

- journalctl -u service shows one service; -f follows live; --since filters by time.
- This is your first stop when a service misbehaves.

Read the logs



REAL-WORLD EXAMPLE

```
journalctl -u shop-api -f
journalctl -u shop-api --since "10 min ago"
journalctl -p err -b        # errors this boot
tail -f /var/log/nginx/error.log
journalctl --disk-usage
```

COMMON MISTAKE
Ignoring logs and guessing at causes. Nearly every failure explains itself in journalctl -u <service> or the app log. Read first, change second.

REVISION SNAPSHOT

ASK YOURSELF	How do you follow a single service's logs in real time?
DO THIS	Restart a service and watch its journal show the startup messages.

14. Networking Basics

GrowthSchool

IN SIMPLE TERMS
Networking basics are the tools to check your machine's addresses, routes, and whether it can reach other machines.

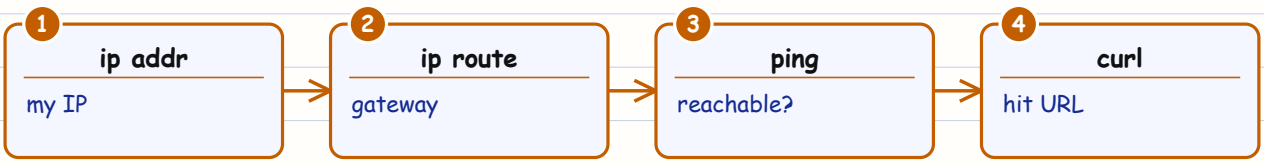
* Interfaces & IPs

- ip addr shows interfaces and IP addresses; ip route shows the routing table and default gateway.
- ping tests reachability; traceroute shows the path packets take.

* Test Connectivity

- curl and wget fetch URLs and are perfect for testing an API from the server itself.
- A failed request could be DNS, routing, firewall or the app - isolate each layer.

Test the network



REAL-WORLD EXAMPLE

```
ip addr
ip route
ping -c 3 growthschool.cc
curl -I https://growthschool.cc
curl -s http://localhost:8080/health
```

COMMON MISTAKE
Assuming "the network is down" when a single layer failed. Test in order: interface up? IP assigned? gateway reachable? DNS resolving? port open? app responding?

REVISION SNAPSHOT

ASK YOURSELF	Which command shows your IP addresses and which shows your default gateway?
DO THIS	Curl your own API health endpoint and a public site from the terminal.

15. DNS, Ports & Sockets

GrowthSchool

IN SIMPLE TERMS
DNS turns names into IP addresses; ports and sockets are the numbered doors that services listen on for connections.

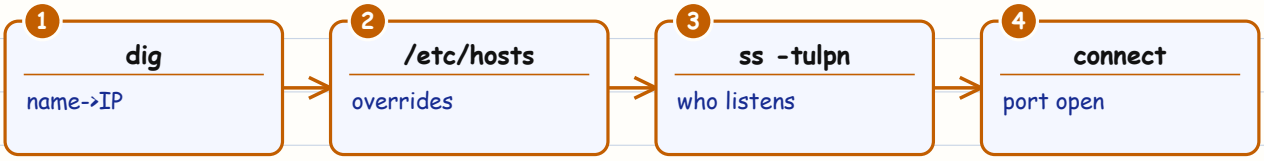
* Name Resolution

- DNS maps names to IPs. dig and nslookup query it; /etc/hosts overrides names locally.
- getent hosts <name> resolves using the systems full configuration.

* Listening Ports

- ss -tulpn lists listening TCP/UDP sockets and the process behind each - the modern netstat.
- If a service is "up" but unreachable, check it is actually listening on the expected port/address.

Name to port



REAL-WORLD EXAMPLE

```
dig +short growthschool.cc
getent hosts db.internal
ss -tulpn | grep 8080
cat /etc/resolv.conf     # configured DNS servers
```

COMMON MISTAKE
A service bound to 127.0.0.1 only, then wondering why remote clients cannot reach it. Bind to 0.0.0.0 (or the right interface) to accept external connections - carefully.

REVISION SNAPSHOT

ASK YOURSELF	Which command lists listening ports and the owning process?
DO THIS	Find which process is listening on a port with ss -tulpn.

16. SSH, SCP & rsync

GrowthSchool

IN SIMPLE TERMS
SSH gives you a secure remote terminal on another machine; SCP and rsync copy files to and from it over that secure link.

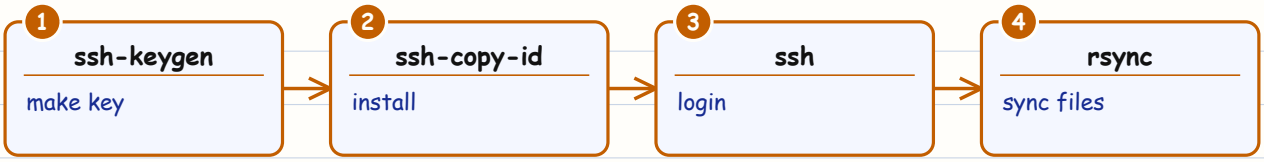
* SSH

- ssh gives a secure remote shell. Use key pairs, not passwords: ssh-keygen then copy the public key.
- Config aliases in ~/.ssh/config save typing host, user and key for each server.

* Copy Files

- scp copies files over SSH; rsync copies efficiently, transferring only differences.
- rsync -avz is the workhorse for syncing directories to/from servers.

Work remotely



REAL-WORLD EXAMPLE

```
ssh-keygen -t ed25519 -C "asha@laptop"
ssh-copy-id deploy@shop-server
ssh deploy@shop-server
scp app.tgz deploy@shop-server:/tmp/
rsync -avz --delete ./dist/ deploy@shop-server:/var/www/shop/
```

COMMON MISTAKE
Using rsync --delete against the wrong target - it removes files not present in the source. Dry-run first with -n, and double-check source/destination order.

REVISION SNAPSHOT

ASK YOURSELF	Why is rsync often better than scp for directories?
DO THIS	Generate an SSH key, add it to a server, and rsync a folder over.

17. Package Managers

GrowthSchool

IN SIMPLE TERMS
A package manager installs, updates, and removes software for you and handles its dependencies (apt, dnf, or apk depending on the distro).

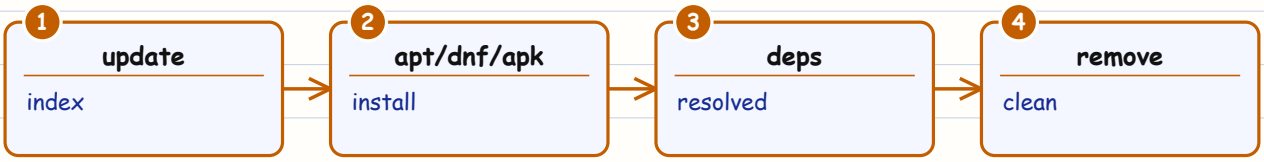
* Install Software

- Debian/Ubuntu use apt (.deb); RHEL/Fedora use dnf/yum (.rpm); Alpine uses apk.
- Update the index, then install/upgrade/remove. Package managers handle dependencies for you.

* Keep It Clean

- Prefer distro packages over random curl-to-bash installs; they are signed and updatable.
- Pin critical versions where reproducibility matters (e.g. in Docker images).

Install software



DISTRO-SPECIFIC
Commands differ per distro family: apt (Debian/Ubuntu), dnf/yum (RHEL/Fedora), apk (Alpine). The concept - indexed, signed, dependency-aware packages - is the same.

REAL-WORLD EXAMPLE

```
sudo apt update && sudo apt install -y nginx   # Debian/Ubuntu
sudo dnf install -y nginx                       # RHEL/Fedora
apk add --no-cache curl                         # Alpine
apt list --installed | grep nginx
```

REVISION SNAPSHOT

ASK YOURSELF	Which package manager belongs to which distro family?
DO THIS	Install, query and remove a small package on your distro.

18. Disk & Storage

GrowthSchool

IN SIMPLE TERMS
Disk and storage commands show how much space you have, what is using it, and which disks are mounted where.

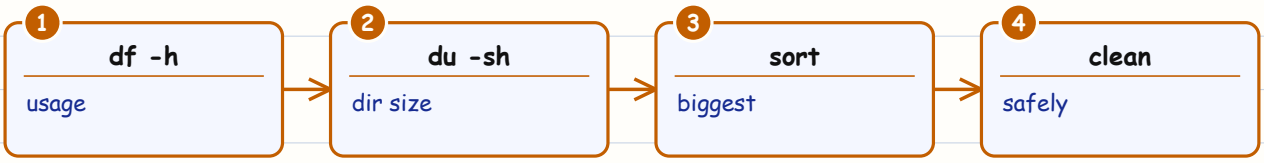
* See Usage

- df -h shows filesystem usage; du -sh shows the size of a directory. Watch for full disks.
- lsblk shows block devices and mounts; a full / disk breaks logging, builds and services.

* Find the Hog

- du -xh --max-depth=1 / | sort -h ranks directories by size to find what filled the disk.

Find the disk hog



REAL-WORLD EXAMPLE

```
df -h
du -sh /var/log
du -xh --max-depth=1 / 2>/dev/null | sort -h | tail
lsblk
ncdu /var        # interactive disk usage (if installed)
```

COMMON MISTAKE
Deleting a large log while a process still holds it open - space is not freed until the process closes the file. Truncate the file or restart the service instead.

REVISION SNAPSHOT

ASK YOURSELF	Which command finds which directory is filling the disk?
DO THIS	Locate the largest directory under / with du and sort.

19. Memory & CPU Diagnostics

GrowthSchool

IN SIMPLE TERMS
Memory and CPU diagnostics are the commands that show how busy your machine is and which processes use the most resources.

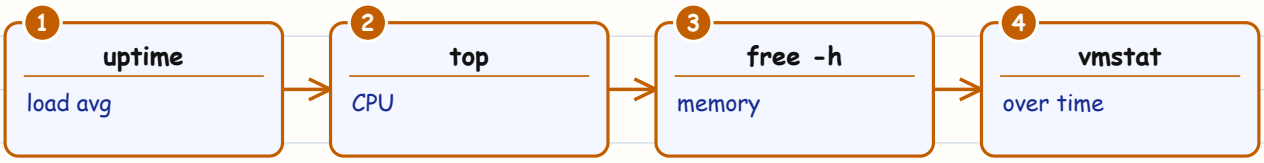
* Snapshot & Live

- free -h shows memory; top/htop show live CPU and per-process usage; uptime shows load average.
- Load average near the CPU count is busy; far above it means saturation or blocked I/O.

* Dig Deeper

- vmstat and iostat reveal CPU, memory, swap and disk I/O over time - useful for sustained issues.

Read the load



REAL-WORLD EXAMPLE

```
free -h
uptime
top -o %CPU
vmstat 1 5
ps aux --sort=-%mem | head
```

COMMON MISTAKE
Panicking about "low free memory". Linux uses free RAM for cache and releases it on demand. Look at available memory and swap activity, not just free.

REVISION SNAPSHOT

ASK YOURSELF	Why is high cached memory usually a good sign, not a problem?
DO THIS	Sort processes by memory and by CPU, and read the load average.

20. grep & Regex

GrowthSchool

IN SIMPLE TERMS
grep searches text for lines that match a pattern; a regex (regular expression) is the mini-language you use to describe that pattern.

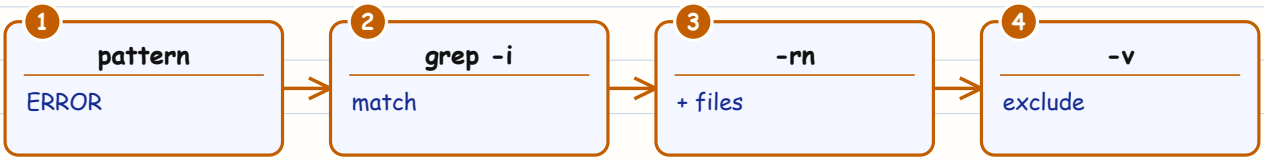
* Search Text

- grep finds lines matching a pattern. -i ignores case, -r recurses, -n shows line numbers, -v inverts.
- grep -E enables extended regex for alternation and quantifiers.

* Regex Building Blocks

- . any char, * zero-or-more, ^ start, $ end, [abc] class, \d-like via [0-9], | alternation (with -E).

grep a log



REAL-WORLD EXAMPLE

```
grep -i error app.log
grep -rn "TODO" src/
grep -E "40[0-9]|50[0-9]" access.log   # 4xx/5xx
grep -c "GET /health" access.log       # count matches
grep -v "healthcheck" access.log       # exclude noise
```

COMMON MISTAKE
Forgetting to quote patterns with special characters, letting the shell expand * or $ before grep sees them. Single-quote your regex.

REVISION SNAPSHOT

ASK YOURSELF	What do -r, -n, -i and -v do in grep?
DO THIS	Count how many 5xx responses appear in a sample access log.

21. find & locate

GrowthSchool

IN SIMPLE TERMS
find searches the filesystem for files by name, size, age, or type, and can run an action on each match.

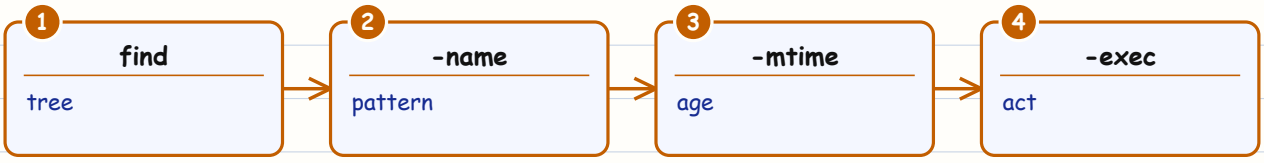
* find

- find searches the tree by name, type, size, time and more, and can act on results with -exec.
- It is precise and real-time (no index), so it is the reliable choice on servers.

* Act on Results

- find ... -delete removes matches; -exec runs a command per file. Test with -print first.

find files



REAL-WORLD EXAMPLE

```
find /var/log -name "*.log" -mtime +7
find . -type f -size +100M
find /tmp -type f -name "*.tmp" -delete
find src -name "*.py" -exec grep -l "TODO" {} +
```

COMMON MISTAKE
Running find ... -delete without first running it with -print to see what matches. Always preview the match set before you delete.

REVISION SNAPSHOT

ASK YOURSELF	How do you find files older than 7 days and act on them safely?
DO THIS	List all files over 100 MB under a directory, then preview a -delete.

22. sed & awk

GrowthSchool

IN SIMPLE TERMS
sed edits text streams (great for find-and-replace); awk processes text column by column for reports and sums.

* sed - Stream Editor

- sed transforms text line by line. s/old/new/g substitutes; -i edits files in place (use with care).
- Great for quick find-replace across files and config tweaks in scripts.

* awk - Field Processor

- awk splits each line into fields ($1, $2, ...) and is perfect for columns, sums and reports.
- Example: sum a column, print specific fields, or filter by a condition.

REAL-WORLD EXAMPLE

```
sed "s/DEBUG/INFO/g" app.log > app.info.log
sed -i "s/localhost/db/" config.env    # in place
awk '{print $1, $7}' access.log        # ip + path
awk '$9 >= 500 {c++} END {print c}' access.log   # 5xx count
```

COMMON MISTAKE
sed -i on the only copy of an important file with an untested expression. Test without -i first, or keep a backup: sed -i.bak "...".

REVISION SNAPSHOT

ASK YOURSELF	When do you reach for sed vs awk?
DO THIS	Use awk to print two columns from a log, then sum a numeric column.

23. Pipes & Redirection

GrowthSchool

IN SIMPLE TERMS
A pipe sends one command's output into another; redirection sends output into (or reads input from) a file.

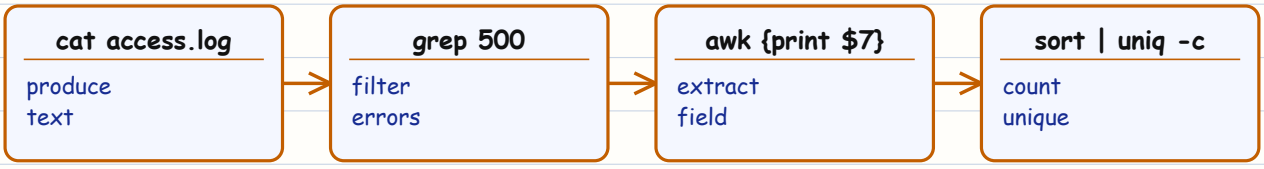
* Compose Small Tools

- A pipe | sends one command's output into the next. Chaining tools solves big problems simply.
- This "do one thing well, then compose" idea is the heart of the Unix philosophy.

* Redirect Streams

- > writes stdout to a file (overwrite), >> appends, 2> redirects stderr, 2>&1 merges them.
- /dev/null discards output; use it to silence noise you do not need.

Pipes: small tools chained into one job



REAL-WORLD EXAMPLE

```
cat access.log | grep " 500 " | awk "{print $7}" | sort | uniq -c | sort -rn | head
command > out.txt 2>&1     # stdout + stderr to file
noisy-cmd 2>/dev/null      # drop errors
echo "$LINE" >> report.txt
```

COMMON MISTAKE
Using > when you meant >>, silently truncating a file you wanted to append to. Also, > creates the file before the command runs, so redirecting a file into itself destroys it.

REVISION SNAPSHOT

ASK YOURSELF	What is the difference between >, >> and 2>&1?
DO THIS	Build a one-line pipeline that ranks the top request paths in a log.

24. Environment & Shell Config

GrowthSchool

IN SIMPLE TERMS
Environment variables carry settings to programs; shell config files (like ~/.bashrc) set up your aliases and preferences.

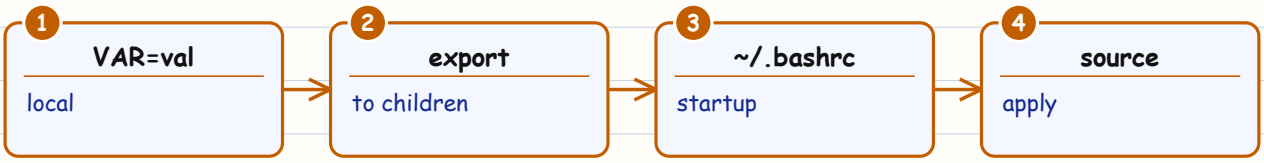
* Variables

- Shell variables (NAME=value) are local; export makes them environment variables for child processes.
- PATH lists where the shell looks for programs; HOME, USER and PWD are common env vars.

* Startup Files

- ~/.bashrc runs for interactive shells; put aliases, exports and prompt tweaks there.
- source a file to apply changes without logging out.

Environment



REAL-WORLD EXAMPLE

```
export APP_ENV=production
echo $PATH
alias ll="ls -lah"
echo "export PATH=\$PATH:/opt/shop/bin" >> ~/.bashrc
source ~/.bashrc
```

COMMON MISTAKE
Setting a variable without export and expecting a child process (or script) to see it. Without export it stays in the current shell only.

REVISION SNAPSHOT

ASK YOURSELF	What does export do that a plain assignment does not?
DO THIS	Add an alias and a PATH entry to ~/.bashrc and source it.

25. Bash Scripting Basics

GrowthSchool

IN SIMPLE TERMS
A bash script is a text file of commands you can run as one program to automate repetitive work.

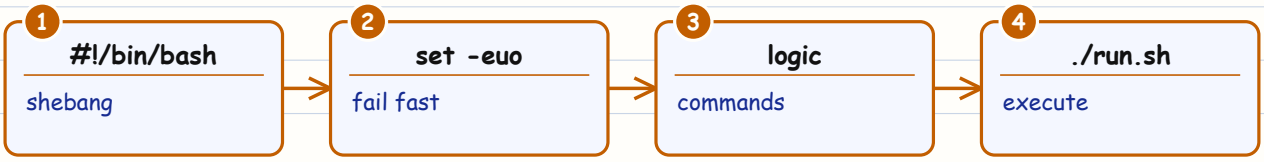
* Anatomy of a Script

- Start with a shebang #!/usr/bin/env bash, make it executable (chmod +x), then run ./script.sh.
- Add set -euo pipefail to fail fast on errors, unset variables and broken pipes.

* Input & Output

- Read arguments as $1, $2, "$@"; read user input with read; print with echo or printf.

A bash script



REAL-WORLD EXAMPLE

```
#!/usr/bin/env bash
set -euo pipefail
NAME="${1:-world}"
echo "Deploying ${NAME}..."
# run:  chmod +x deploy.sh && ./deploy.sh shop-api
```

COMMON MISTAKE
Omitting set -euo pipefail. Scripts then plough on after a failed command, doing damage with half-set variables. Fail fast and loud.

REVISION SNAPSHOT

ASK YOURSELF	What does set -euo pipefail protect you from?
DO THIS	Write a script that greets a name passed as the first argument.

26. Bash: Loops & Conditions

GrowthSchool

IN SIMPLE TERMS
Loops repeat commands over a list, and conditions run commands only when a test is true - the logic of any script.

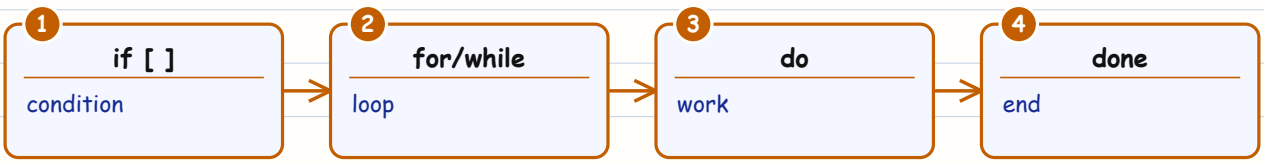
* Conditions

- if ["$x" = "y"]; then ... fi. Use [[...]] in bash for safer tests and pattern matching.
- Test files with -f (exists/file), -d (dir), -z (empty string), -n (non-empty).

* Loops

- for item in list; do ... done iterates; while [cond]; do ... done repeats until false.
- Loop over files, lines or command output to automate repetitive work.

Loops & tests



REAL-WORLD EXAMPLE

```
for f in *.log; do echo "rotating $f"; gzip "$f"; done
if [[ -f /etc/shop.conf ]]; then echo "config found"; fi
while read -r line; do echo ">> $line"; done < hosts.txt
if ! curl -sf localhost:8080/health; then echo DOWN; fi
```

COMMON MISTAKE
Looping over ls output (for f in $(ls)) which breaks on spaces/newlines. Glob directly (for f in *.log) or read files safely with a while-read loop.

REVISION SNAPSHOT

ASK YOURSELF	Why loop over a glob instead of the output of ls?
DO THIS	Write a loop that gzips every .log file in a directory.

27. Bash: Functions & Arguments

GrowthSchool

IN SIMPLE TERMS
Functions are reusable named blocks inside a script; arguments are the values you pass into a script or function.

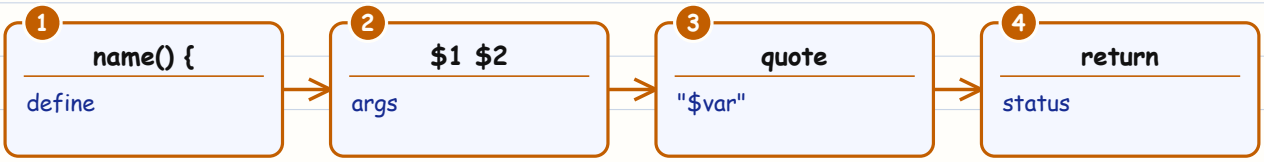
* Functions

- Define reusable blocks: name() { ...; }. Inside, $1..$n are the function's arguments.
- return sets an exit status (0 = success); echo returns data to a caller via command substitution.

* Robust Scripts

- Quote all variable expansions ("$var"); check argument counts; give clear usage/error messages.
- Exit non-zero on failure so callers and CI can detect it.

Functions



REAL-WORLD EXAMPLE

```
deploy() {
  local env="$1"
  [[ -z "$env" ]] && { echo "usage: deploy <env>"; return 1; }
  echo "deploying to ${env}"
}
deploy production
```

COMMON MISTAKE
Leaving variables unquoted so a value with spaces splits into multiple arguments. Always quote: "$1", "$env", "${arr[@]}".

REVISION SNAPSHOT

ASK YOURSELF	Why must you quote variable expansions in bash?
DO THIS	Write a function that validates its arguments and returns non-zero on misuse.

28. cron & Scheduling

GrowthSchool

IN SIMPLE TERMS
cron is the built-in scheduler that runs commands automatically at set times, like a daily backup.

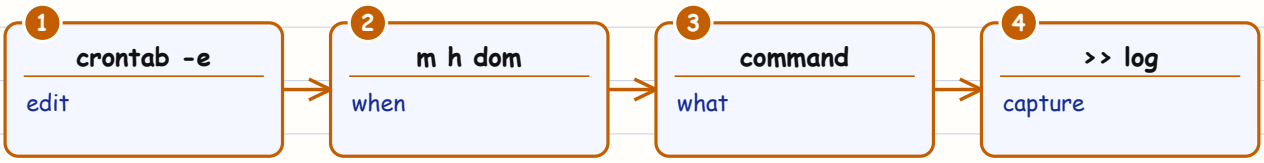
* cron Syntax

- crontab -e edits your schedule. Five fields: minute hour day-of-month month day-of-week + command.
- Example: 0 2 * * * runs daily at 02:00. Use * for "every".

* Reliable Jobs

- Use absolute paths and redirect output to a log; cron runs with a minimal environment.
- For systemd hosts, timers are a more observable alternative to cron.

Schedule with cron



REAL-WORLD EXAMPLE

```
crontab -e
# m h dom mon dow  command
0 2 * * *  /opt/shop/backup.sh >> /var/log/backup.log 2>&1
*/5 * * * * /opt/shop/healthcheck.sh
crontab -l        # list your jobs
```

COMMON MISTAKE
Assuming cron has your interactive PATH and env. It does not. Use absolute paths and set any needed variables explicitly, and always capture output to a log.

REVISION SNAPSHOT

ASK YOURSELF	What do the five cron time fields mean?
DO THIS	Schedule a script to run every 5 minutes and confirm it logs output.

29. Security Basics

GrowthSchool

IN SIMPLE TERMS
Security basics are the essential habits - SSH keys, firewalls, least privilege, and patching - that keep a server safe.

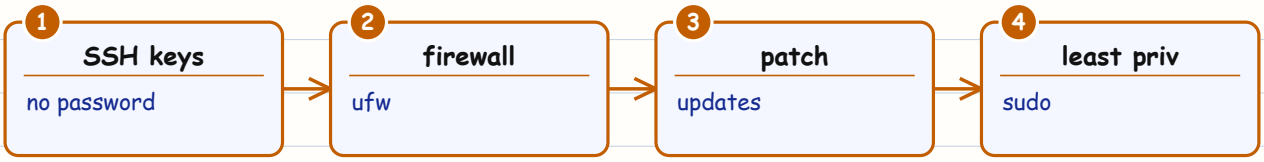
* Harden the Basics

- Use SSH keys and disable password login; keep packages patched; run services as non-root.
- Open only the ports you need with a firewall (ufw/firewalld); use least privilege everywhere.

* Watch & Limit

- Review auth logs for failed logins; use sudo (audited) instead of shared root; rotate credentials.
- Never commit secrets; store them in a secrets manager or protected env files.

Harden a server



REAL-WORLD EXAMPLE

```
sudo ufw allow 22/tcp && sudo ufw allow 443/tcp && sudo ufw enable
grep "Failed password" /var/log/auth.log | tail
sudo apt update && sudo apt upgrade -y
# /etc/ssh/sshd_config: PasswordAuthentication no
```

COMMON MISTAKE
Leaving password SSH login enabled on an internet-facing box. Bots brute-force it constantly. Use keys, disable password auth, and restrict the firewall.

REVISION SNAPSHOT

ASK YOURSELF	Name three quick wins that harden a fresh Linux server.
DO THIS	Enable a firewall allowing only SSH + HTTPS, then verify with ss.

30. Performance Troubleshooting

GrowthSchool

IN SIMPLE TERMS
Performance troubleshooting is a repeatable method for finding what is slowing a server down: CPU, memory, disk, or network.

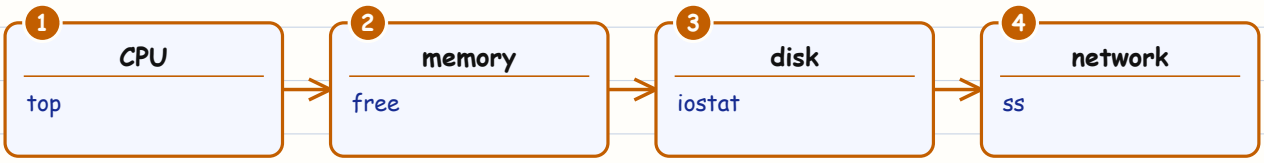
* A Repeatable Method

- Define the symptom, then check the four resources: CPU, memory, disk I/O, network - in that order.
- Use load average + top for CPU, free for memory, iostat for disk, ss/ip for network.

* From Symptom to Cause

- Slow app? Check CPU saturation, memory/swap, disk wait, then dependency latency (DB, network).
- Change one thing at a time and confirm the metric improves before moving on.

Find the bottleneck



REAL-WORLD EXAMPLE

```
uptime                  # load average trend
top -o %CPU             # who burns CPU
free -h                 # memory + swap
iostat -xz 1 3          # disk wait (%util, await)
ss -s                   # socket summary
```

COMMON MISTAKE
Changing several settings at once, so you never learn which one helped (or hurt). Isolate variables and measure after each change.

REVISION SNAPSHOT

ASK YOURSELF	What four resources do you check, and in what order?
DO THIS	Run the CPU/memory/disk/network checks and note a baseline for your box.

31. Real Incident: High CPU

GrowthSchool

IN SIMPLE TERMS
This page walks through a real incident - a server pinned at high CPU - and how to find and fix the cause step by step.

* The Scenario

- Shop API latency spikes. Alerts show load average of 12 on a 4-core box - CPU saturated.
- Users see timeouts. You SSH in to investigate calmly using a method, not guesswork.

* The Investigation

- top shows one shop-api worker at 380% CPU. ps and logs reveal a tight retry loop on a failing call.
- Fix: patch the loop / add backoff, restart the service, and confirm load returns to normal.

REAL-WORLD EXAMPLE

```
uptime
top -o %CPU            # find the hot process
ps -p 4821 -o pid,ppid,%cpu,cmd
journalctl -u shop-api --since "15 min ago" | tail
sudo systemctl restart shop-api
```

COMMON MISTAKE
Rebooting the whole server first. That hides the cause and it recurs. Identify the process, read its logs, fix the root cause, then restart just that service.

REVISION SNAPSHOT

ASK YOURSELF	Walk the steps from "high load alert" to root cause.
DO THIS	Simulate CPU load (e.g. yes >/dev/null &) and locate it with top, then stop it.

32. Real Incident: Disk Full

GrowthSchool

IN SIMPLE TERMS
This page walks through a real incident - a full disk - and how to find what filled it and safely reclaim space.

* The Scenario

- Writes start failing with "No space left on device". The app cannot log or accept uploads.
- df -h shows / at 100%. You must find and safely reclaim space without deleting needed data.

* The Investigation

- du ranks the biggest directories; a runaway /var/log file or old Docker images is a common culprit.
- Truncate/rotate logs, prune unused images, then add rotation/monitoring so it does not recur.

REAL-WORLD EXAMPLE

```
df -h
du -xh --max-depth=1 / 2>/dev/null | sort -h | tail
sudo truncate -s 0 /var/log/huge.log   # free space, keep file
docker system df && docker system prune
journalctl --vacuum-size=200M
```

COMMON MISTAKE
rm on a huge log a running process still holds open - space is not freed until the process releases it. Truncate the file (or restart the writer) instead.

REVISION SNAPSHOT

ASK YOURSELF	Why can deleting an open log file fail to free space?
DO THIS	Fill a test file with fallocate, watch df, then reclaim the space.

33. Linux Command Cheat Sheet

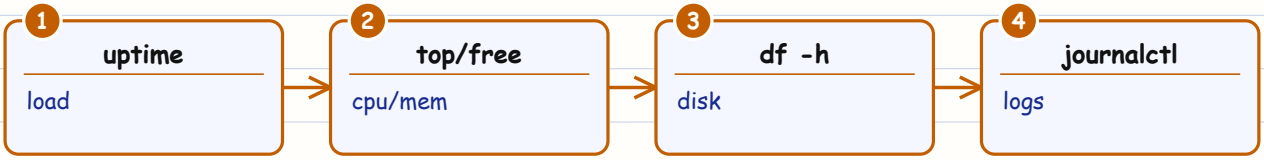
GrowthSchool

IN SIMPLE TERMS
This is a quick-reference list of the Linux commands you will use every day.

* Daily Drivers

- ls/cd/pwd, cp/mv/rm, chmod/chown, ps/top/kill, systemctl, journalctl, grep/find/awk/sed, ss, df/du.

Triage a server



TRIAGE LOOP
uptime -> top -> free -h -> df -h -> journalctl -u <svc>. These five commands orient you on almost any misbehaving Linux server in under a minute.

ONE-PAGE COMMAND REFERENCE

```
ls -lah     cd -          cp -r  mv  rm -i      mkdir -p
chmod 750   chown u:g     stat   umask          sudo -l
ps aux      top/htop      kill -TERM  pkill     free -h
systemctl status/restart/enable --now          journalctl -u x -f
grep -rn    find -name -mtime -exec            sed -i  awk
ss -tulpn   ip addr/route  dig  curl -I        ping -c 3
df -h  du -sh  lsblk       tar czf/xzf         rsync -avz
crontab -e    for/while/if    export  source   chmod +x
```

COMMON MISTAKE
Memorising exotic flags while fumbling the daily ones. Drill ls/cd/grep/ps/systemctl/journalctl until they are reflex - those carry 90% of real server work.

STUDENT LAB
Cover this page and, from memory, write the command to: follow a service log, find the biggest directory, list listening ports, and make a script executable. Check yourself against the sheet.

REVISION SNAPSHOT

ASK YOURSELF	Which five commands triage a struggling server fastest?
DO THIS	Recall fifteen commands from memory grouped by purpose.

34. Capstone Challenge

GrowthSchool

IN SIMPLE TERMS
This capstone ties everything together by setting up and operating a real Linux server for the Shop API.

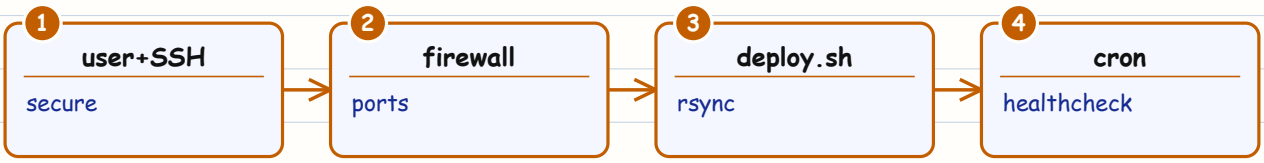
* Provision a Shop API Host

- Create a deploy user with sudo, SSH-key-only login, and a firewall allowing SSH + HTTPS.
- Install nginx, set correct ownership/permissions on /var/www/shop, and enable services on boot.

* Automate & Operate

- Write a bash deploy script (set -euo pipefail) that rsyncs a build and restarts the service.
- Add a cron healthcheck that logs status, and a log-rotation config so the disk never fills.

Capstone steps



REAL-WORLD EXAMPLE

```
sudo adduser deploy && sudo usermod -aG sudo deploy
sudo ufw allow 22,443/tcp && sudo ufw enable
rsync -avz --delete ./dist/ deploy@host:/var/www/shop/
sudo systemctl restart nginx && systemctl is-active nginx
```

STUDENT LAB
Deliverables: key-only SSH, a firewall, a running enabled service, an idempotent deploy script, a cron healthcheck writing to a log, and log rotation configured.

REVISION SNAPSHOT

ASK YOURSELF	Is your server key-only, firewalled, and self-healing on reboot?
DO THIS	Complete every capstone deliverable on a VM or container and document it.

35. Interview & Revision Notes

GrowthSchool

IN SIMPLE TERMS
This page gathers the key ideas and common interview questions so you can revise quickly.

* Explain Clearly

- Permissions (rwx + numeric), SIGTERM vs SIGKILL, systemctl start vs enable, pipes + redirection.
- How you triage high CPU or a full disk, and why you avoid running as root.

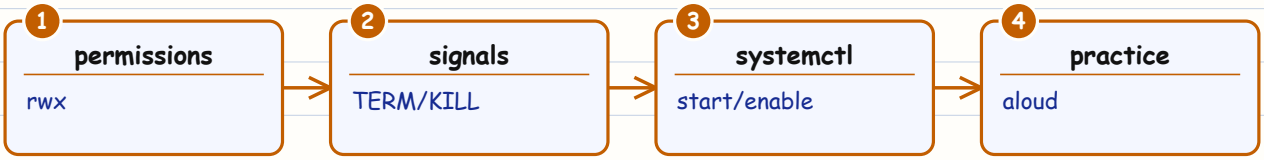
* Common Questions

- How do you find what is using a port / the disk / the CPU? How do you follow a service's logs?
- What does set -euo pipefail do? How do SSH keys work? What is the difference between > and >>?

* Your Next Step

- Run everything on a real box. Recreate the two incidents deliberately and fix them from memory.

Revise out loud



REAL-WORLD EXAMPLE

```
# 30-second self test
chmod 640 file  # who can read/write now?
ss -tulpn | grep 8080   # what is listening?
journalctl -u svc -f    # follow logs
# SIGTERM vs SIGKILL? start vs enable? > vs >>?
```

REVISION SNAPSHOT

ASK YOURSELF	Can you explain rwx numeric permissions in 30 seconds?
DO THIS	Answer every "Common Question" above out loud without notes.

GrowthSchool

GIT & GITHUB

COMPLETE NOTES

TRACK - BRANCH - SHIP

Version control, branching, PRs, CI/CD and team workflow

growthschool.cc | @growthschool.cc

Learning Roadmap

GrowthSchool

Follow this order. Each page has explanation, real commands, a diagram or example, common mistakes, a student lab and a revision snapshot. We build one ecommerce "Shop API" throughout.

HOW TO USE THESE NOTES

Read top to bottom the first time, then use pages 26-30 for fast revision before interviews. Type every command yourself in a throwaway repo. Reading is not remembering.

1. Why Version Control & Git

GrowthSchool

IN SIMPLE TERMS
Version control is a tool that saves the full history of your project so you can see every change, undo mistakes, and work with others without overwriting each other. Git is the most popular version-control tool.

* The Problem

- Copying folders like project-final-FINAL-v2 does not scale and loses history.
- Teams need to work in parallel, review changes, and undo mistakes without fear.

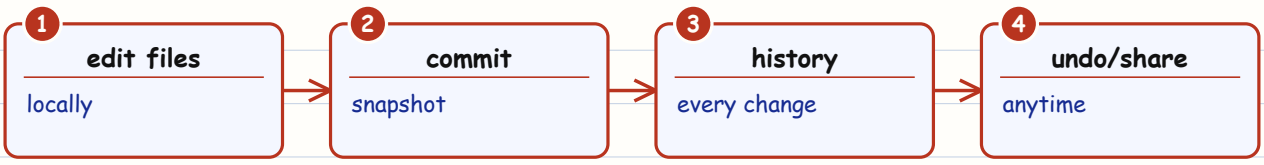
* What Git Is

- Git is a distributed version control system: every clone has the full history.
- It records snapshots (commits), not diffs, and links them into a directed history.

* Git vs GitHub

- Git is the tool that runs locally. GitHub is a hosting platform for Git repositories.
- GitHub adds pull requests, reviews, issues, Actions (CI/CD) and access control on top of Git.

Why version control



FIRST-TIME SETUP

```
# check your version and set identity
git --version
git config --global user.name  "Asha Dev"
git config --global user.email "asha@shop.example"
```

COMMON MISTAKE
Committing as the wrong identity. Set user.name and user.email before your first commit, or history will show the wrong author on every commit you make.

REVISION SNAPSHOT

ASK YOURSELF What makes Git "distributed", and how is it different from GitHub?

DO THIS Explain snapshot vs diff, and name three things GitHub adds on top of Git.

2. The Git Mental Model

GrowthSchool

IN SIMPLE TERMS
The Git mental model is understanding the four places your work lives: your files, the staging area, your local history, and the remote copy on a server.

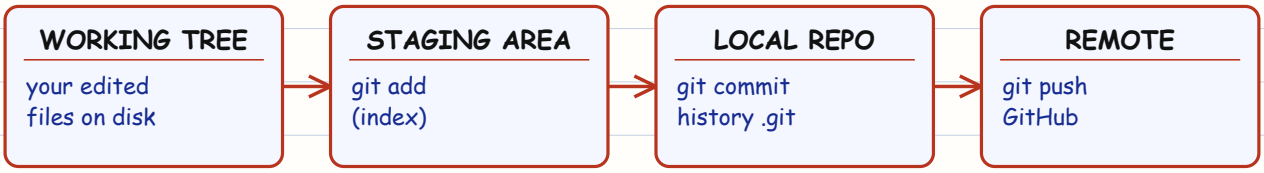
* Four Places Your Code Lives

- Working tree: the real files you edit. Staging area (index): what will go into the next commit.
- Local repository: committed history in the hidden .git folder. Remote: the shared copy (GitHub).

* Why Staging Exists

- Staging lets you craft a clean, logical commit instead of dumping every change at once.
- You can stage part of your work now and leave the rest for a separate, meaningful commit.

The Four Areas of Git



add -> commit -> push moves work right; fetch/checkout brings it back left.

REAL-WORLD EXAMPLE

```
git status            # what changed, what is staged
git add app/checkout.py   # stage one file
git add -p            # stage selected chunks interactively
git commit -m "add checkout total calculation"
```

STUDENT LAB
In a new repo, edit two files. Stage only one with git add, run git status, and confirm the other file stays "not staged". Commit, then observe how status changes.

REVISION SNAPSHOT

ASK YOURSELF	Name the four areas and the command that moves work between each.
DO THIS	Draw the model from memory: working -> staging -> local -> remote.

3. Start a Repository

GrowthSchool

IN SIMPLE TERMS
A repository (repo) is a project folder that Git is tracking. You either create a new one (init) or copy an existing one from a server (clone).

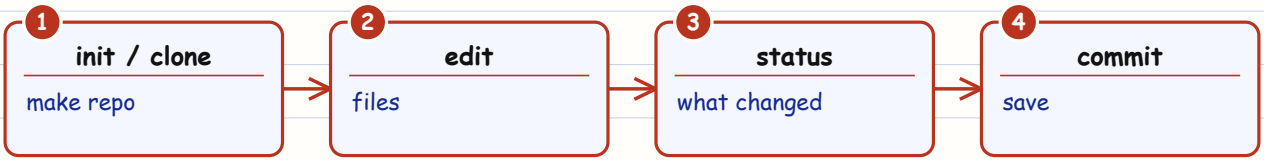
* Two Ways to Begin

- git init turns an existing folder into a repo. git clone <url> copies an existing remote repo.
- Cloning also sets up a remote named origin pointing back to where you cloned from.

* Inspect Before You Act

- git status is the command you will run most: it shows branch, staged, and unstaged changes.
- git log shows history; add --oneline --graph --all for a compact visual of all branches.

Start a repo



REAL-WORLD EXAMPLE

```
git init shop-api            # new local repo
git clone git@github.com:org/shop-api.git
git status
git log --oneline --graph --all --decorate
```

COMMON MISTAKE
Running git init inside an existing repo (nested repos) or in your home folder by accident. Check with git status / git rev-parse --show-toplevel before creating a new repo.

REVISION SNAPSHOT

ASK YOURSELF	When do you use init vs clone, and what does clone set up automatically?
DO THIS	Clone any public repo and read its history with git log --oneline --graph.

4. Staging, Commits, Log & Diff

GrowthSchool

IN SIMPLE TERMS
Committing is saving a snapshot of your staged changes into history, each with a message; diff and log let you see what changed and when.

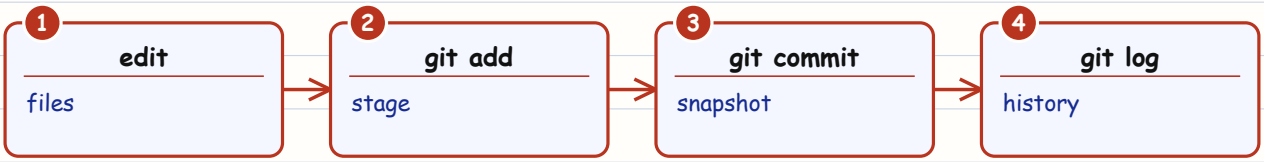
* Make Good Commits

- A commit is a snapshot plus author, timestamp and message. Keep each commit one logical change.
- Write messages in the imperative: "add", "fix", "refactor" - describing what the change does.

* See What Changed

- git diff shows unstaged changes; git diff --staged shows what is about to be committed.
- git log -p shows the patch for each commit; git log --stat summarises files changed.

Commit a change



REAL-WORLD EXAMPLE

```
git add .
git commit -m "fix: prevent negative cart quantity"
git diff              # working tree vs staged
git diff --staged     # staged vs last commit
git log --oneline -5
```

COMMON MISTAKE
Giant commits like "misc fixes" that mix features, formatting and bug fixes together. They make review, rollback and git bisect painful. Commit small and often.

REVISION SNAPSHOT

ASK YOURSELF What is the difference between git diff and git diff --staged?

DO THIS Make three small, well-described commits instead of one large one.

5. Exploring History

GrowthSchool

IN SIMPLE TERMS
Exploring history means reading the story of a project - who changed which line, when, and why - using log, show, and blame.

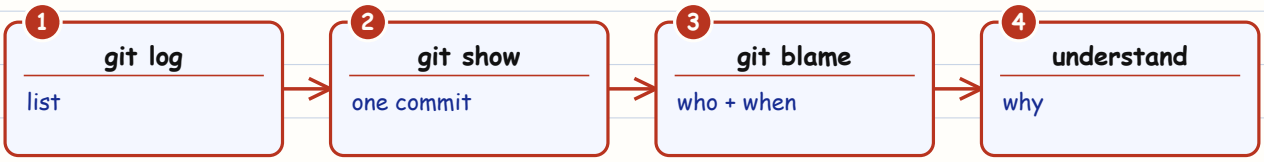
* Read the Story of the Code

- git show <sha> displays one commit in full. git log --oneline gives a scannable list.
- git blame <file> shows who last changed each line and in which commit - great for context.

* Find Things Fast

- git log --grep "checkout" searches commit messages. git log -S"functionName" finds when code appeared.
- git log -- path/to/file limits history to a single file or folder.

Read the history



REAL-WORLD EXAMPLE

```
git show a1b2c3d
git blame app/checkout.py
git log --oneline --grep "refund"
git log -S"calculateTax" --oneline
```

STUDENT LAB
Pick any file in a repo, run git blame, and trace one line back to its introducing commit with git show. Write one sentence explaining why that line exists.

REVISION SNAPSHOT

ASK YOURSELF	Which command tells you WHEN a specific function first appeared?
DO THIS	Use blame + show to explain the origin of one confusing line of code.

6. Branches

GrowthSchool

IN SIMPLE TERMS
A branch is a separate line of work, like a parallel copy of the project, so you can build a feature without disturbing the main version.

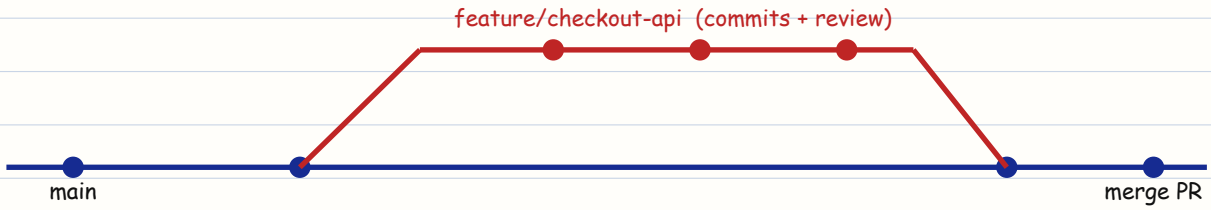
* What a Branch Really Is

- A branch is just a movable pointer to a commit. Creating one is instant and cheap.
- HEAD points to your current branch. Switching branches moves your working tree to that snapshot.

* Everyday Branching

- Create a branch per feature or fix so main always stays releasable.
- git switch is the modern, clearer command; git checkout still works for older habits.

Branch & Pull-Request Flow



REAL-WORLD EXAMPLE

```
git switch -c feature/checkout-api   # create + switch
git branch                # list local branches
git switch main
git merge feature/checkout-api
```

COMMON MISTAKE
Doing all work directly on main. One bad commit then blocks everyone and every release. Branch per change; keep main deployable at all times.

REVISION SNAPSHOT

ASK YOURSELF	If a branch is "just a pointer", what actually moves when you commit on it?
DO THIS	Create feature/x, commit twice, merge to main, then delete the branch.

7. Merge vs Rebase

GrowthSchool

IN SIMPLE TERMS
Merging and rebasing are two ways to combine branches: merge joins them keeping both histories, rebase re-writes your commits on top for a straight line.

* Merge

- git merge joins two branches and, when histories diverged, creates a merge commit.
- It preserves the true history but can produce a busy, non-linear graph.

* Rebase

- git rebase replays your commits on top of another branch, producing a clean, linear history.
- It rewrites commit SHAs, so never rebase commits that others have already pulled.

* Golden Rule

- Rebase local, private work to tidy it. Merge (or PR-merge) shared branches to preserve history.

REAL-WORLD EXAMPLE

```
# tidy your feature branch on top of latest main
git switch feature/checkout-api
git fetch origin
git rebase origin/main
# resolve conflicts, then: git rebase --continue
```

COMMON MISTAKE
Rebasing a branch that teammates already based work on - it rewrites shared history and forces everyone into painful conflicts. Rebase only private, unpushed commits.

REVISION SNAPSHOT

ASK YOURSELF	When is rebase safe and when is it dangerous?
DO THIS	Rebase a local branch onto main, then compare the graph to a merge result.

8. Merge Conflicts (Full Example)

GrowthSchool

IN SIMPLE TERMS
A merge conflict happens when two branches changed the same lines, so Git asks you to choose the correct final result by hand.

* Why Conflicts Happen

- Two branches changed the same lines. Git cannot decide which is correct, so it asks you.
- Conflicts are normal - not errors. Stay calm and resolve them deliberately.

* The Resolution Steps

- 1) Run the merge/rebase. 2) Open conflicted files. 3) Edit to the correct final result.
- 4) Remove the <<<< ==== >>>> markers. 5) git add the file. 6) commit or continue.

REAL-WORLD EXAMPLE

```
<<<<<<< HEAD
    total = price * qty
=======
    total = price * qty * (1 - discount)
>>>>>>> feature/checkout-api
# keep the correct line, delete markers, then:
git add app/checkout.py && git commit
```

COMMON MISTAKE
Committing the conflict markers (<<<<, ====, >>>>) still in the file. Always search the project for these markers before you finish, and re-run tests after resolving.

REVISION SNAPSHOT

ASK YOURSELF	What do the three marker sections HEAD / ==== / branch mean?
DO THIS	Force a conflict on purpose in a test repo and resolve it end to end.

9. cherry-pick

GrowthSchool

IN SIMPLE TERMS
cherry-pick copies one specific commit from one branch onto another, without merging everything else.

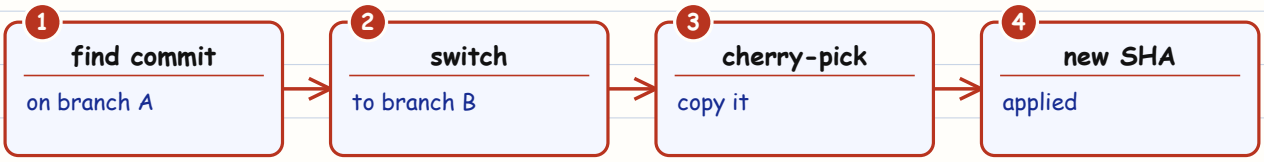
* What It Does

- git cherry-pick <sha> copies a single commit from one branch onto your current branch.
- Useful to bring one urgent fix to a release branch without merging everything else.

* Use Sparingly

- Cherry-picking duplicates the change as a new commit with a new SHA.
- Overuse creates divergent, hard-to-track history. Prefer proper merges when possible.

cherry-pick



REAL-WORLD EXAMPLE

```
git switch release/1.4
git cherry-pick 9f8e7d6      # bring hotfix commit only
git cherry-pick A^..B        # a range of commits
git cherry-pick --abort      # bail out on conflict
```

STUDENT LAB
Create a fix on main, then cherry-pick just that commit onto a release/1.0 branch. Confirm with git log that only that single change was applied.

REVISION SNAPSHOT

ASK YOURSELF	Why does a cherry-picked commit get a new SHA?
DO THIS	Cherry-pick one hotfix commit to a release branch cleanly.

10. reset, revert, restore

GrowthSchool

IN SIMPLE TERMS
reset, revert, and restore are the three ways to undo work in Git - each undoes a different thing in a different, safe or unsafe, way.

* git restore

- Discards changes in the working tree or unstages files. It does not touch commit history.
- git restore <file> reverts edits; git restore --staged <file> unstages.

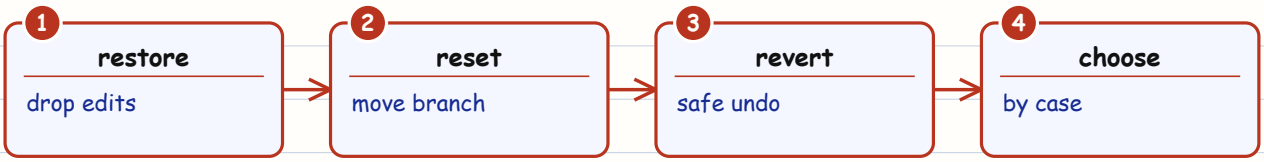
* git reset

- --soft keeps changes staged; --mixed (default) unstages; --hard discards everything.
- reset moves the branch pointer. --hard is destructive - be certain before using it.

* git revert

- Creates a NEW commit that undoes a previous one. Safe for shared/public history.

Pick the right undo



REAL-WORLD EXAMPLE

```
git restore app.py            # drop working edits
git reset --soft HEAD~1       # undo commit, keep changes staged
git reset --hard HEAD~1       # DANGER: discard commit + changes
git revert a1b2c3d            # safe undo on shared branch
```

COMMON MISTAKE
Using git reset --hard on shared branches or before saving needed work. Lost commits may still be recoverable via reflog, but treat --hard as a loaded weapon.

REVISION SNAPSHOT

ASK YOURSELF	Which undo command is safe on a branch others have pulled?
DO THIS	Practice soft vs hard reset in a scratch repo and observe the difference.

11. reflog & Recovery

GrowthSchool

IN SIMPLE TERMS
The reflog is Git's safety net: a private log of everywhere you have been, so you can recover commits that seem lost.

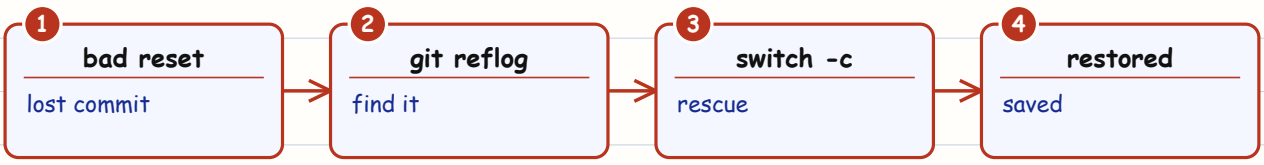
* Your Safety Net

- git reflog records where HEAD has been - even after resets, rebases and branch deletes.
- Most "lost" commits are not gone; they are just no longer referenced by a branch.

* Recover a Commit

- Find the commit SHA in reflog, then create a branch or reset back to it.
- Git eventually garbage-collects truly unreferenced commits, so recover promptly.

Recover with reflog



REAL-WORLD EXAMPLE

```
git reflog
# c0ffee HEAD@{2}: commit: add refund flow
git switch -c recovered c0ffee   # rescue lost work
# or: git reset --hard HEAD@{2}
```

STUDENT LAB
Commit something, run git reset --hard HEAD~1 to "lose" it, then recover it using git reflog. This single skill will save you one day.

REVISION SNAPSHOT

ASK YOURSELF	After an accidental hard reset, what is the first command you run?
DO THIS	Delete a commit with reset, then bring it back from the reflog.

12. git stash

GrowthSchool

IN SIMPLE TERMS
git stash temporarily parks your unfinished changes and gives you a clean workspace, so you can switch tasks and come back later.

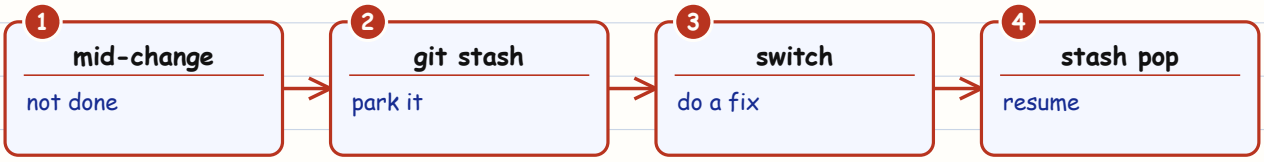
* Park Work Safely

- git stash saves uncommitted changes and gives you a clean working tree instantly.
- Perfect when an urgent fix arrives while you are mid-change on something else.

* Manage the Stash

- git stash list shows saved stashes; git stash pop re-applies and removes the latest.
- Use git stash push -m "msg" to label stashes and avoid a confusing pile of them.

Park work with stash



REAL-WORLD EXAMPLE

```
git stash push -m "wip: checkout ui"
git switch hotfix/login
# ...fix and commit...
git switch feature/checkout-ui
git stash pop
```

COMMON MISTAKE
Treating stash as long-term storage. Stashes are easy to forget and lose. For anything important, make a commit on a branch instead.

REVISION SNAPSHOT

ASK YOURSELF	When is stash better than a commit, and when is it worse?
DO THIS	Stash mid-work, switch branches to do a fix, then pop and continue.

13. Tags, Releases & SemVer

GrowthSchool

IN SIMPLE TERMS
A tag marks an important commit (usually a release); semantic versioning is the MAJOR.MINOR.PATCH numbering that tells users how big a change is.

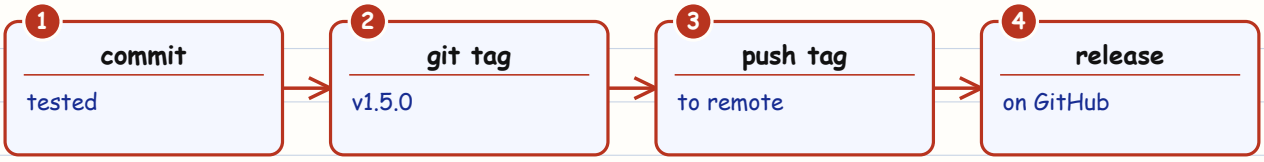
* Tags

- A tag marks a specific commit, usually a release point. Annotated tags store author + message.
- Push tags explicitly: git push origin v1.2.0 (or git push --tags).

* Semantic Versioning

- MAJOR.MINOR.PATCH: MAJOR = breaking change, MINOR = new feature, PATCH = backward-safe fix.
- Example: 1.4.2 -> 1.5.0 adds a feature; 1.4.2 -> 2.0.0 breaks the public API.

Tag a release



REAL-WORLD EXAMPLE

```
git tag -a v1.5.0 -m "Shop API: coupon support"
git push origin v1.5.0
# GitHub Releases can attach notes + build artifacts to a tag
```

COMMON MISTAKE
Bumping only PATCH for a breaking change. Consumers pin versions expecting SemVer rules; a wrong bump silently breaks their builds.

REVISION SNAPSHOT

ASK YOURSELF	What version bump does a breaking API change require?
DO THIS	Tag a commit as v1.0.0 and create a GitHub Release from it.

14. .gitignore & Cleaning

GrowthSchool

IN SIMPLE TERMS
A .gitignore file lists files Git should never track, like build output and secrets, keeping your repo clean and safe.

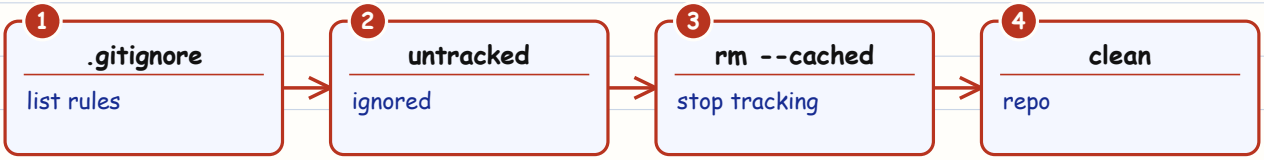
* Ignore the Right Things

- Add build output, dependencies, secrets and local config to .gitignore so they never commit.
- Common entries: node_modules/, __pycache__/, .env, dist/, *.log, .DS_Store.

* Already Tracked?

- .gitignore only affects untracked files. To stop tracking something already committed,
- run git rm --cached <file> then commit the removal.

Ignore the right files



REAL-WORLD EXAMPLE

```
# .gitignore
node_modules/
.env
*.log
dist/
# stop tracking a committed secret file:
git rm --cached .env && git commit -m "chore: stop tracking .env"
```

COMMON MISTAKE
Committing a .env or credentials file, then just adding it to .gitignore. The secret is still in history - you must rotate the secret and scrub history (git filter-repo).

REVISION SNAPSHOT

ASK YOURSELF	Why does adding a tracked file to .gitignore not remove it from the repo?
DO THIS	Add a .gitignore, then untrack an accidentally committed log file.

15. Remotes: fetch, pull, push

GrowthSchool

IN SIMPLE TERMS
A remote is a shared copy of your repo on a server (like GitHub); fetch, pull, and push move commits between your machine and it.

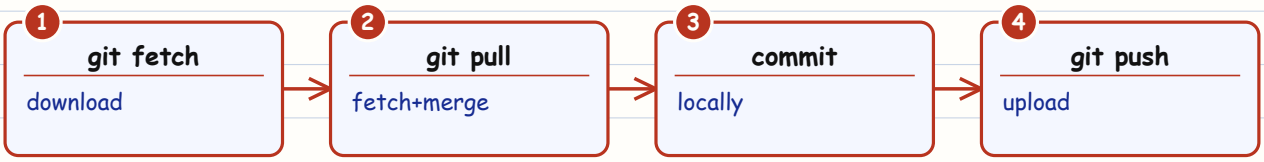
* Remotes

- A remote is a named URL to a shared repo. origin is the default created by clone.
- git remote -v lists remotes; git remote add upstream <url> adds another (common in forks).

* Sync Direction

- git fetch downloads remote changes without touching your files. git pull = fetch + merge.
- git push uploads your commits. Set upstream once with git push -u origin <branch>.

Sync with remote



REAL-WORLD EXAMPLE

```
git remote -v
git fetch origin
git pull --rebase origin main    # linear history on pull
git push -u origin feature/checkout-api
```

COMMON MISTAKE
Blindly git pull creating surprise merge commits. Prefer git pull --rebase for a clean linear history, and always fetch before you assume you are up to date.

REVISION SNAPSHOT

ASK YOURSELF	What is the difference between git fetch and git pull?
DO THIS	Add a second remote (upstream) and fetch from it without merging.

16. GitHub & Collaboration

GrowthSchool

IN SIMPLE TERMS
GitHub is a website that hosts Git repositories and adds teamwork features like issues, reviews, and access control on top of Git.

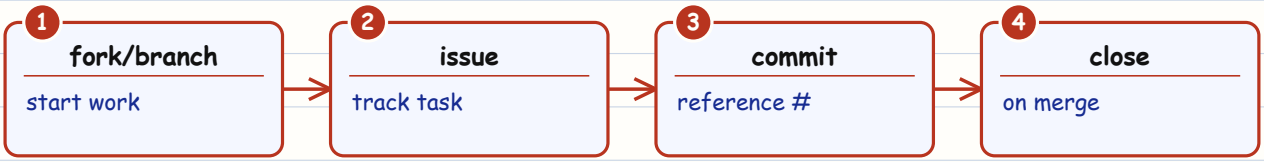
* What GitHub Adds

- Hosting plus pull requests, code review, issues, projects, Actions (CI/CD) and fine-grained access.
- Issues track work; labels, milestones and projects organise it; discussions handle Q&A.

* Fork vs Branch

- Team members with write access branch inside the same repo.
- Outside contributors fork the repo, push to their fork, then open a PR back to the original.

Collaborate on GitHub



REAL-WORLD EXAMPLE

```
# authenticate and work with GitHub from the CLI
gh auth login
gh repo clone org/shop-api
gh issue create --title "Cart total wrong with coupons"
```

STUDENT LAB
Create a GitHub repo for a small Shop API, add a README and an issue describing one feature, then reference that issue number in a commit message (e.g. "fixes #1").

REVISION SNAPSHOT

ASK YOURSELF	When do you fork instead of branching directly?
DO THIS	Open an issue, then close it automatically via a commit that says "closes #N".

17. Pull Requests & Code Review

GrowthSchool

IN SIMPLE TERMS
A pull request (PR) proposes merging your branch into another and opens it for teammates to review before it is accepted.

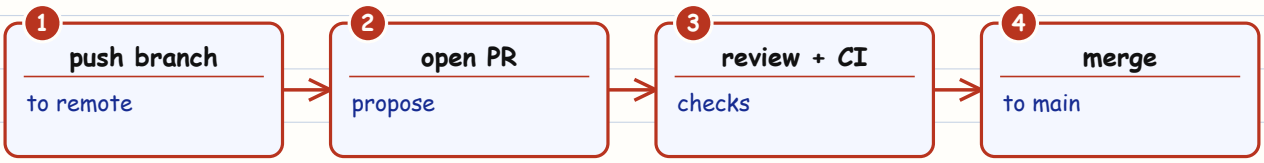
* The PR

- A pull request proposes merging your branch into a target branch and opens it for review.
- Keep PRs small and focused; add a clear description of what changed and why, plus how to test.

* Good Review Culture

- Reviewers check correctness, readability, tests and security - not personal style preferences.
- Respond to comments, push follow-up commits, and let CI pass before requesting final approval.

Open a pull request



REAL-WORLD EXAMPLE

```
git push -u origin feature/checkout-api
gh pr create --base main --title "feat: coupon support" \
  --body "Adds coupon discount. Tests added. Closes #12."
gh pr status
```

COMMON MISTAKE
Massive PRs touching 50 files. They get rubber-stamped, hide bugs, and are impossible to review well. Split work into small, reviewable PRs.

REVISION SNAPSHOT

ASK YOURSELF	What makes a pull request easy (or hard) to review?
DO THIS	Open a PR from a feature branch and request review from a teammate.

18. Branch Protection & CODEOWNER

GrowthSchool

IN SIMPLE TERMS
Branch protection and CODEOWNERS are GitHub rules that stop unreviewed or broken code from reaching your main branch.

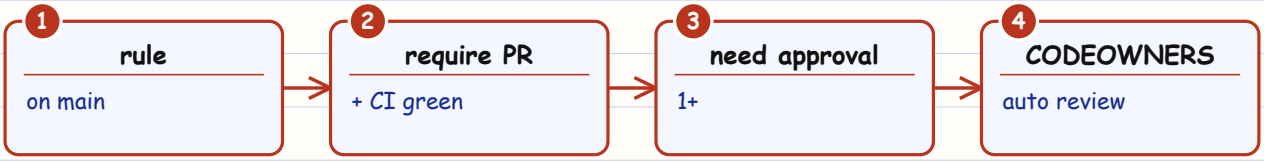
* Protect main

- Branch protection rules block direct pushes and require PRs, passing checks and approvals.
- This keeps the release branch healthy and enforces the review process automatically.

* CODEOWNERS

- A CODEOWNERS file auto-requests specific reviewers when matching paths change.
- Example: /infra/ requires the platform team; /payments/ requires the payments team.

Protect main



WHY IT MATTERS
Rules > good intentions. Automated protection ensures no unreviewed or failing code reaches main,
even under deadline pressure when humans cut corners.

REAL-WORLD EXAMPLE

```
# .github/CODEOWNERS
*                @shop/maintainers
/payments/       @shop/payments-team
/infra/          @shop/platform-team
```

REVISION SNAPSHOT

ASK YOURSELF	What three checks would you require before allowing a merge to main?
DO THIS	Enable "require PR + passing CI + 1 approval" on a test repo.

19. Conventional Commits

GrowthSchool

IN SIMPLE TERMS
Conventional Commits is a simple standard for writing commit messages (like "feat:" or "fix:") that machines can read to automate versioning.

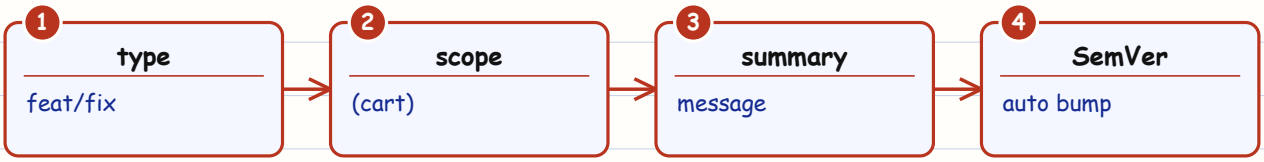
* A Commit Message Standard

- Format: type(scope): summary. Types include feat, fix, docs, refactor, test, chore, ci.
- Example: feat(checkout): apply coupon discount to cart total.

* Why Bother

- Machine-readable messages enable automatic changelogs and SemVer version bumps.
- feat -> MINOR bump, fix -> PATCH bump, and a "BREAKING CHANGE:" footer -> MAJOR bump.

Conventional commits



REAL-WORLD EXAMPLE

```
feat(cart): support multiple coupons
fix(auth): reject expired session tokens
refactor(api): extract price calculator
feat(api)!: rename /v1/checkout to /v2/checkout

BREAKING CHANGE: clients must use /v2/checkout
```

STUDENT LAB
Rewrite five of your recent commit messages in Conventional Commit format and identify which SemVer bump each one implies.

REVISION SNAPSHOT

ASK YOURSELF	Which commit type triggers a MINOR vs a PATCH release?
DO THIS	Write one feat, one fix and one breaking-change commit correctly.

20. Branching Strategies

GrowthSchool

IN SIMPLE TERMS
A branching strategy is the agreed team pattern for how branches are created, reviewed, and merged - such as feature branches or trunk-based.

* Feature Branch (recommended start)

- Short-lived branch per change -> PR -> review -> merge to main. Simple and effective for most teams.

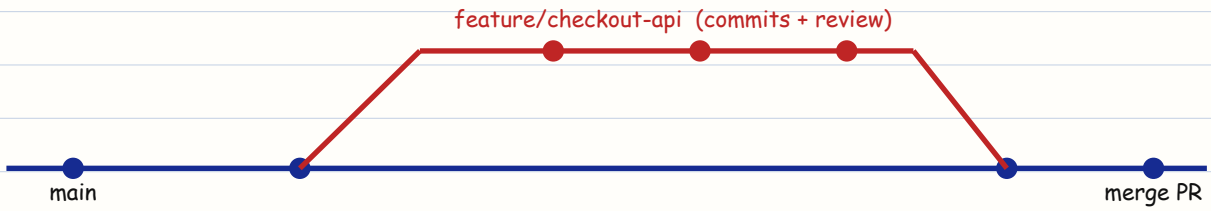
* GitFlow

- Long-lived develop + release + hotfix branches. Powerful but heavy; suits scheduled releases.

* Trunk-Based

- Everyone commits small changes to main behind feature flags with strong CI. Enables fast delivery.

Branch & Pull-Request Flow



COMMON MISTAKE
Adopting heavy GitFlow for a tiny team that ships continuously. Match the strategy to your release cadence and team size - complexity is not maturity.

REVISION SNAPSHOT

ASK YOURSELF	Which strategy fits a team that deploys many times per day?
DO THIS	Describe your current project's branching model and one way to simplify it.

21. GitHub Actions Fundamentals

GrowthSchool

IN SIMPLE TERMS
GitHub Actions is GitHub's built-in automation: it runs your build and tests automatically whenever you push code or open a PR.

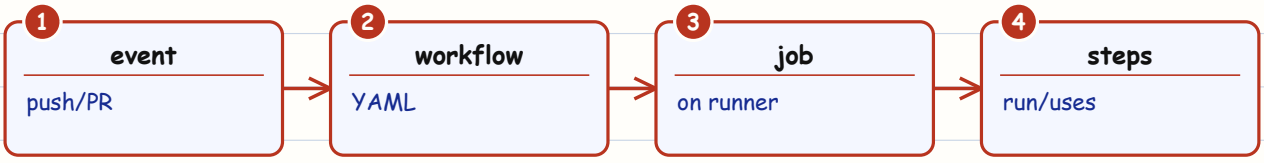
* The Building Blocks

- Workflow: a YAML file in .github/workflows. Event: what triggers it (push, pull_request, schedule).
- Job: a set of steps on a runner. Step: a shell command or a reusable action (uses:).

* Why CI

- Continuous Integration runs build + tests on every push/PR so bugs are caught in minutes, not days.
- A green check on a PR gives reviewers confidence the change actually works.

GitHub Actions parts



REAL-WORLD EXAMPLE

```
name: ci
on: [push, pull_request]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: echo "run build and tests here"
```

REVISION SNAPSHOT

ASK YOURSELF	Name the four core concepts: workflow, event, job, step.
DO THIS	Add a workflow that just prints "hello" on every push and watch it run.

22. CI Workflow for Shop API

GrowthSchool

IN SIMPLE TERMS
Continuous Integration (CI) means automatically building and testing every change, so bugs are caught in minutes instead of days.

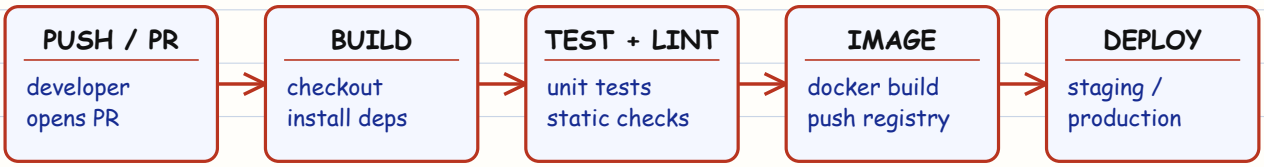
* A Realistic Pipeline

- On every PR: check out code, set up runtime, install dependencies, run lint, run tests.
- Cache dependencies to keep the pipeline fast, and fail the build if any step fails.

* Read the YAML Carefully

- Pin action versions (e.g. @v4) for reproducibility. Keep secrets in GitHub Secrets, never in YAML.

GitHub Actions CI/CD Pipeline (Shop API)



REAL-WORLD EXAMPLE

```
jobs:
  build-test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with: { node-version: "20", cache: "npm" }
      - run: npm ci
      - run: npm run lint
      - run: npm test
```

REVISION SNAPSHOT

ASK YOURSELF	Why cache dependencies and pin action versions?
DO THIS	Add lint + test steps to a workflow and make a PR fail on a bad test.

23. CD & Release Automation

GrowthSchool

IN SIMPLE TERMS
Continuous Delivery (CD) means automatically shipping tested changes toward production, often with an approval step for safety.

* From Green Check to Deploy

- After tests pass on main, a deploy job can build a Docker image and push it to a registry.
- Use environments and required approvals for production; deploy staging automatically.

* Secrets & Environments

- Store credentials in GitHub Secrets or an OIDC-based cloud role - never hard-code them.
- Gate production behind a protected environment that requires a manual approval.

REAL-WORLD EXAMPLE

```
on:
  push: { branches: [main] }
jobs:
  deploy:
    environment: production   # requires approval
    steps:
      - run: docker build -t registry/shop-api:$GITHUB_SHA .
      - run: docker push registry/shop-api:$GITHUB_SHA
```

COMMON MISTAKE
Printing secrets with echo in a workflow, or committing a token to test CI. Logs are visible; use masked GitHub Secrets and rotate anything ever exposed.

REVISION SNAPSHOT

ASK YOURSELF	How do you require a human approval before a production deploy?
DO THIS	Add a deploy job that only runs on pushes to main.

24. Git Internals

GrowthSchool

IN SIMPLE TERMS
Git internals are the simple building blocks under the hood - objects and pointers - that explain why Git behaves the way it does.

* Objects

- Git stores four object types: blob (file contents), tree (directory), commit, and tag.
- Each object is content-addressed by a SHA hash, so identical content is stored once.

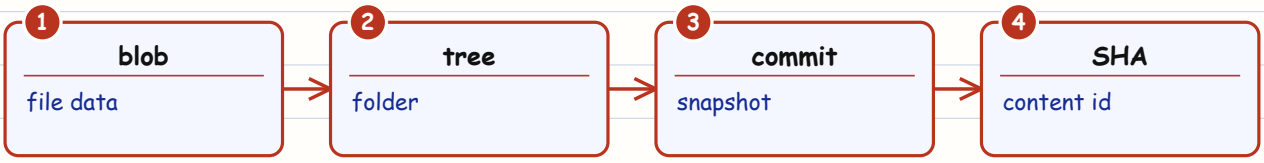
* Refs & HEAD

- A branch is a ref: a file containing a commit SHA. HEAD points to the current branch/commit.
- This is why branching is instant - it just writes a small pointer, not a copy of files.

* Why This Matters

- Understanding objects explains why commits are immutable and why history rewrites make new SHAs.
- It also explains git gc (garbage collection) removing unreachable objects after some time.

Git objects



REAL-WORLD EXAMPLE

```
git cat-file -t a1b2c3d     # type: commit
git cat-file -p a1b2c3d     # show commit contents
cat .git/HEAD               # ref: refs/heads/main
git rev-parse HEAD          # current commit SHA
git count-objects -v        # repo object stats
```

STUDENT LAB
Run git cat-file -p on a commit, then on the tree SHA it points to, then on a blob SHA.
You have now walked the object graph from commit down to file contents by hand.

REVISION SNAPSHOT

ASK YOURSELF	What are the four Git object types, and why is each commit immutable?
DO THIS	Inspect a commit object with cat-file and follow it down to a blob.

25. Undo Decision Guide

GrowthSchool

IN SIMPLE TERMS
The undo decision guide helps you pick the right, safe way to undo something based on whether your work has been shared yet.

* Ask One Question First

- Has this commit been pushed/shared? If NO, you may safely rewrite history (reset, amend, rebase).
- If YES, do not rewrite shared history - use git revert to add a safe, inverse commit.

* Pick the Right Tool

- Discard file edits -> restore. Undo last commit, keep work -> reset --soft. Undo public commit -> revert.

LOCAL / NOT PUSHED

git restore <file> : discard working change
git reset --soft HEAD~1 : keep changes staged
git commit --amend : fix last message/content
git reflog : find any lost commit

ALREADY PUSHED / SHARED

git revert <sha> : safe inverse commit
never rewrite public history casually
communicate before force operations
push --force-with-lease if truly needed

COMMON MISTAKE
Rewriting pushed history to "clean up", forcing everyone to re-sync and often losing work. Once shared, prefer revert. Reserve force-push for branches only you use.

REVISION SNAPSHOT

ASK YOURSELF	First question before any undo: has it been shared?
DO THIS	For three scenarios, name the exact command you would run.

26. Troubleshooting Playbook

GrowthSchool

IN SIMPLE TERMS
Troubleshooting is the calm, step-by-step way to fix common Git problems like rejected pushes or a detached HEAD.

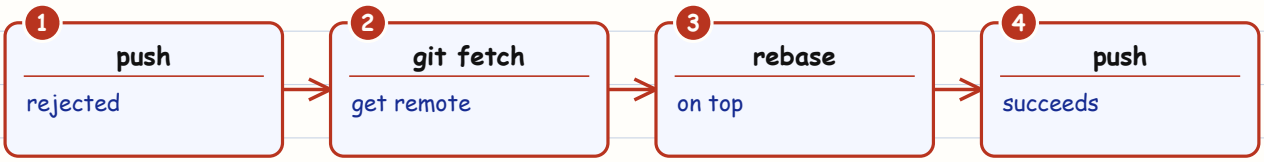
* Common Situations

- detached HEAD: you checked out a commit, not a branch. Create a branch to keep new work.
- "rejected - non-fast-forward" on push: remote has commits you lack; fetch + rebase, then push.

* More Fixes

- Accidentally committed to main: create a branch there, then reset main back with reset (if local).
- Wrong author on last commit: git commit --amend --author="Name <email>".

Fix a rejected push



REAL-WORLD EXAMPLE

```
# push rejected? sync first
git fetch origin
git rebase origin/main
git push
# detached HEAD? rescue your work:
git switch -c fix/rescue
```

COMMON MISTAKE
Force-pushing to "fix" a rejected push without looking at what is on the remote. You can overwrite a teammate's commits. Always fetch and inspect first.

REVISION SNAPSHOT

ASK YOURSELF	What causes a non-fast-forward push rejection, and how do you fix it safely?
DO THIS	Reproduce a detached HEAD and recover by creating a branch.

27. End-to-End Team Workflow

GrowthSchool

IN SIMPLE TERMS
The team workflow is the full loop real teams use: branch, commit, push, open a PR, review, merge, and release.

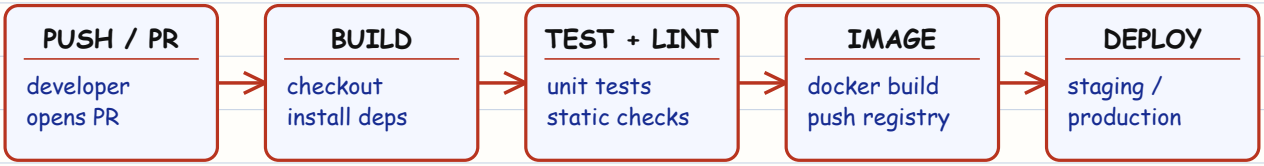
* The Shop API Flow

- Pull latest main -> create feature branch -> commit small -> push -> open PR -> CI runs.
- Review + approve -> merge to main -> tag release -> CD deploys -> monitor -> repeat.

* Keeping main Healthy

- Protected main, required checks, small PRs and fast reviews keep the team unblocked and shipping.

GitHub Actions CI/CD Pipeline (Shop API)



STUDENT LAB
Run the full loop once: branch, commit, push, PR, self-review, merge, and tag a v0.1.0 release on a practice Shop API repo. Note how long each step took.

REVISION SNAPSHOT

ASK YOURSELF	List the workflow from "pull main" to "deployed" in order.
DO THIS	Complete one full branch-to-release cycle end to end.

28. Git & GitHub Cheat Sheet

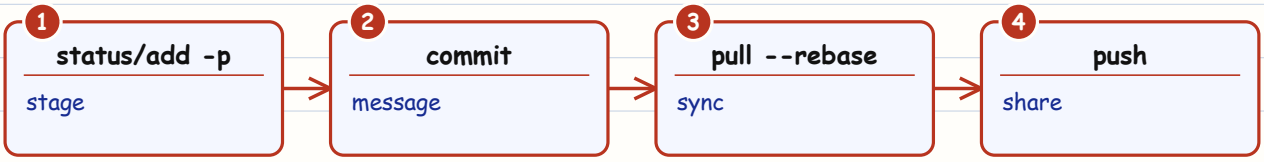
GrowthSchool

IN SIMPLE TERMS
This is a quick-reference list of the Git and GitHub commands you will use every day.

* Daily Drivers

- status, add -p, commit -m, switch -c, pull --rebase, push -u, log --oneline --graph.
- These cover 90% of daily work. Master them before reaching for anything exotic.

Everyday Git



DEBUG LOOP
status -> diff -> log --oneline --graph -> reflog. These four commands answer "what is going on?" in almost every confusing Git situation.

ONE-PAGE COMMAND REFERENCE

```
git switch -c feature/x        git restore <file>
git add -p                     git reset --soft HEAD~1
git commit -m "feat: ..."      git revert <sha>
git pull --rebase origin main  git reflog
git push -u origin feature/x   git stash push -m "wip"
git merge <branch>             git tag -a v1.0.0 -m "..."
git rebase origin/main         git cherry-pick <sha>
gh pr create                   git log --oneline --graph --all
```

COMMON MISTAKE
Memorising rare commands you will Google anyway, while fumbling the daily ones. Drill the top row until switch/add -p/commit/pull --rebase/push are pure muscle memory.

STUDENT LAB
Time yourself: branch, stage a hunk with add -p, commit with a Conventional message, rebase onto main and push - all from memory. Repeat until it takes under a minute.

REVISION SNAPSHOT

ASK YOURSELF	Which four commands orient you when Git is confusing?
DO THIS	Pin this page. Recall five commands from memory without looking.

29. Capstone Challenge

GrowthSchool

IN SIMPLE TERMS
This capstone ties everything together by shipping the Shop API using a complete Git and GitHub workflow.

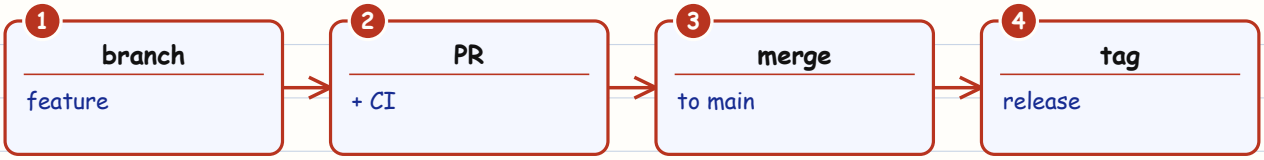
* Ship the Shop API with Git

- Init the repo, add a README and .gitignore, and make three well-scoped Conventional commits.
- Create feature/coupons, implement a change, push and open a PR with a clear description.

* Automate & Release

- Add a GitHub Actions CI workflow that runs lint + tests on every PR.
- Merge via PR, tag v1.0.0, and create a GitHub Release with notes.
- Stretch: add branch protection (CI + 1 approval) and a CODEOWNERS file.

Capstone flow



REAL-WORLD EXAMPLE

```
git switch -c feature/coupons
git commit -m "feat(cart): apply coupon discount"
git push -u origin feature/coupons
gh pr create --base main --fill
git tag -a v1.0.0 -m "Shop API first release" && git push origin v1.0.0
```

STUDENT LAB
Deliverables: a repo with protected main, a green CI check on a merged PR, a v1.0.0 tag/release, and a history that reads cleanly with git log --oneline --graph.

REVISION SNAPSHOT

ASK YOURSELF	Can your commit history be understood by a new teammate in 2 minutes?
DO THIS	Complete every capstone deliverable, then review your own history critically.

30. Interview & Revision Notes

GrowthSchool

IN SIMPLE TERMS
This page gathers the key ideas and common interview questions so you can revise quickly.

* Explain Clearly

- merge vs rebase, reset vs revert, fetch vs pull, and what a branch/HEAD actually is.
- Describe how you resolve a conflict and how you recover a lost commit with reflog.

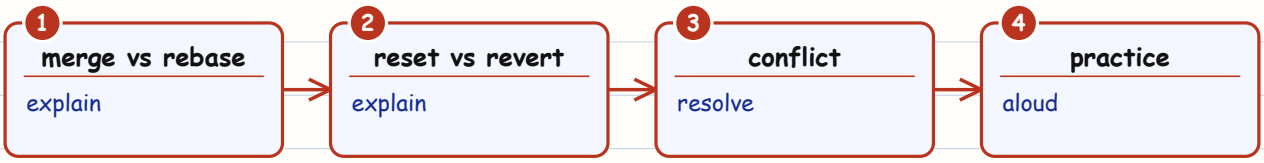
* Common Questions

- How do you undo a pushed commit? (revert) How do you tidy local history? (rebase -i).
- What is in a commit object? (tree, parent, author, message). Why is Git distributed?

* Your Next Step

- Keep a scratch repo. Every time you learn a new command, break something and fix it there.

Revise out loud



REAL-WORLD EXAMPLE

```
# rapid-fire self test - answer before running
git reset --soft HEAD~1   # vs --mixed vs --hard?
git revert <sha>          # why safe on shared branches?
git rebase -i HEAD~3      # squash/reword - local only!
git reflog                # recover after a bad reset
```

STUDENT LAB
Whiteboard the four Git areas and the commands between them, then explain merge vs rebase to a friend. If they understand it, you understand it.

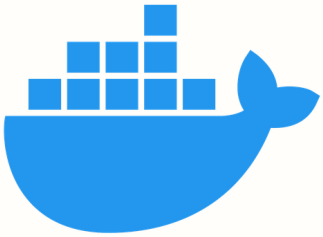
REVISION SNAPSHOT

ASK YOURSELF	Can you explain reset vs revert to a beginner in 30 seconds?
DO THIS	Answer all "Common Questions" above out loud without notes.

GrowthSchool

DOCKER

COMPLETE NOTES



BUILD - SHIP - RUN

Images, containers, Dockerfiles, Compose and production packaging

growthschool.cc | @growthschool.cc

Learning Roadmap

GrowthSchool

Learn Docker in order. Every page has explanation, real commands, a diagram or example, common mistakes, a student lab and a revision snapshot. We package one ecommerce "Shop API" throughout.

HOW TO USE THESE NOTES

Type every command in a real terminal. Break an image on purpose and fix it - that is how the concepts stick. Use pages 25-29 for fast revision before interviews.

1. Containers vs Virtual Machines

GrowthSchool

IN SIMPLE TERMS
Docker packages an app and everything it needs into a container. A container shares the host's operating system, so it is far smaller and faster to start than a full virtual machine.

* The Old Way (VMs)

- Each VM ships a full guest OS on top of a hypervisor - heavy, slow to boot, large images.
- Great isolation, but you pay for a whole OS per application instance.

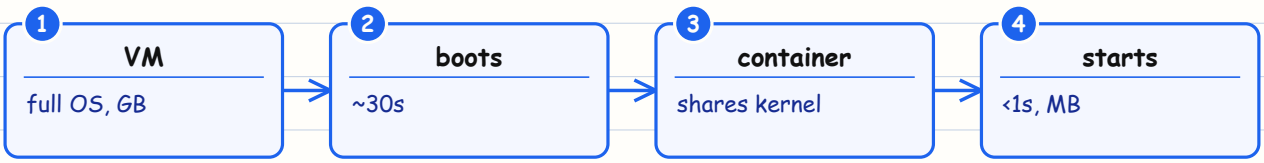
* The Container Way

- Containers share the host OS kernel and isolate processes, filesystem and network.
- They start in milliseconds, are megabytes not gigabytes, and pack far more density per host.

* When Each Fits

- Containers for app packaging and scaling. VMs for strong OS-level isolation or different kernels.

Container vs VM



REAL-WORLD EXAMPLE

```
# same app, radically different footprint
# VM image: ~1-4 GB, boots in ~30s
# container image: ~20-150 MB, starts in <1s
docker run --rm hello-world
```

COMMON MISTAKE
Thinking a container is a lightweight VM. It is an isolated process, not a machine.
It has no init/systemd by default and should run one main process.

REVISION SNAPSHOT

ASK YOURSELF	Why does a container start far faster than a VM?
DO THIS	Run hello-world, then docker info to see the host kernel it shares.

2. Docker Architecture

GrowthSchool

IN SIMPLE TERMS
Docker's architecture has three parts: the command you type (client), the background service that does the work (daemon), and the registry that stores images.

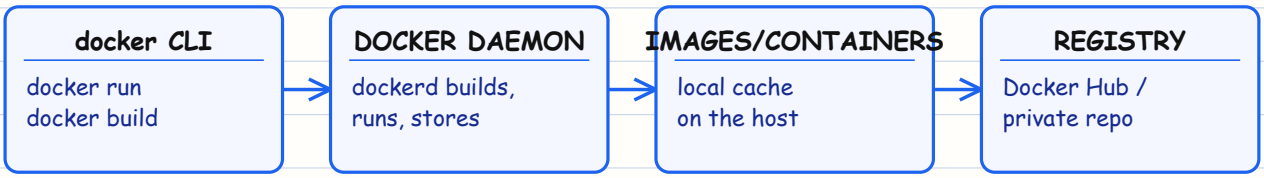
* Client and Daemon

- The docker CLI is a thin client that talks to the Docker daemon (dockerd) over an API.
- The daemon does the real work: building images, running containers, managing storage and networks.

* Registries

- A registry stores and distributes images. Docker Hub is the default public registry.
- docker pull downloads images; docker push uploads them to a registry you can authenticate to.

Docker Architecture: client -> daemon -> registry



REAL-WORLD EXAMPLE

```
docker version        # client + daemon versions
docker info           # daemon status, storage driver
docker pull nginx:1.27
docker images
```

REVISION SNAPSHOT

ASK YOURSELF What are the three pieces: client, daemon, registry - and who does the work?

DO THIS Run docker info and identify the storage driver and running container count.

3. Images, Layers & Union Filesystem

GrowthSchool

IN SIMPLE TERMS
An image is a read-only template built from stacked layers; because layers are cached and shared, rebuilds and downloads are fast.

* What an Image Is

- An image is a read-only stack of layers plus metadata describing how to run it.
- Each Dockerfile instruction that changes the filesystem creates a new cached layer.

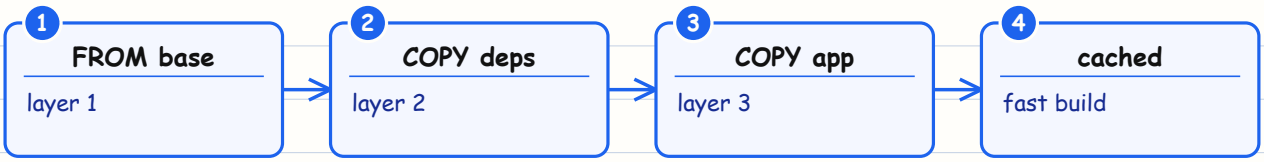
* Layers & Caching

- Layers are shared between images and cached between builds, making rebuilds fast.
- Order matters: put rarely-changing steps first so the cache is reused as often as possible.

* Tags

- A tag names a version, e.g. shop-api:1.4.2. "latest" is just a default tag, not "newest".

Image layers



REAL-WORLD EXAMPLE

```
docker history shop-api:1.0    # see the layers
docker image inspect shop-api:1.0
# a tag is a pointer to an image id:
docker tag shop-api:1.0 registry.example.com/shop-api:1.0
```

COMMON MISTAKE
Relying on the latest tag in production. It is mutable and ambiguous; two hosts may run different code. Always pin an explicit version or digest.

REVISION SNAPSHOT

ASK YOURSELF	Why does instruction order in a Dockerfile affect build speed?
DO THIS	Run docker history on any image and explain what each layer came from.

4. Container Lifecycle

GrowthSchool

IN SIMPLE TERMS
The container lifecycle is the stages a container moves through: created, running, stopped, and finally removed.

* States

- A container is created from an image, then running, then it can be paused, stopped, or removed.
- A stopped (exited) container keeps its writable layer until you docker rm it.

* The Writable Layer

- Each container adds a thin writable layer on top of the image; changes there are lost on rm.
- Persist important data in volumes, not the container layer.

Image -> Container Lifecycle

REAL-WORLD EXAMPLE

```
docker run -d --name api shop-api:1.0
docker stop api
docker start api
docker rm -f api        # force remove (stop + rm)
docker ps -a           # include stopped containers
```

COMMON MISTAKE
Assuming files written inside a running container survive removal. They do not - the writable layer is discarded. Use a volume for anything that must persist.

REVISION SNAPSHOT

ASK YOURSELF	What happens to container data when you docker rm it?
DO THIS	Create a file inside a container, remove it, recreate, and prove the file is gone.

5. Registries & Docker Hub

GrowthSchool

IN SIMPLE TERMS
A registry is a store for images. Docker Hub is the default public one; you pull images from it and push your own to it.

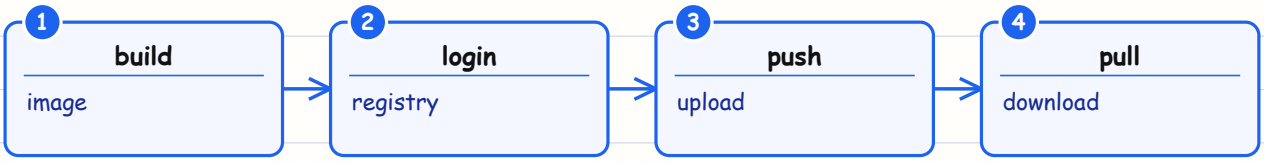
* Pull and Push

- docker pull fetches an image; docker push uploads to a registry after docker login.
- Image names encode the registry: registry.example.com/team/shop-api:1.0.

* Public vs Private

- Docker Hub hosts public and private repos. Companies also run private registries (see page 21).
- Use official or verified base images and pin versions to reduce supply-chain risk.

Registry flow



REAL-WORLD EXAMPLE

```
docker login
docker tag shop-api:1.0 myuser/shop-api:1.0
docker push myuser/shop-api:1.0
docker pull myuser/shop-api:1.0
```

STUDENT LAB
Create a free Docker Hub repo, push a small image you built, then pull it on a clean machine or after docker image rm to prove it round-trips.

REVISION SNAPSHOT

ASK YOURSELF	How does an image name encode which registry it lives in?
DO THIS	Push one image to a repo and pull it back after deleting the local copy.

6. Running Containers

GrowthSchool

IN SIMPLE TERMS
Running a container means starting an app from an image, usually publishing a port and passing in some settings.

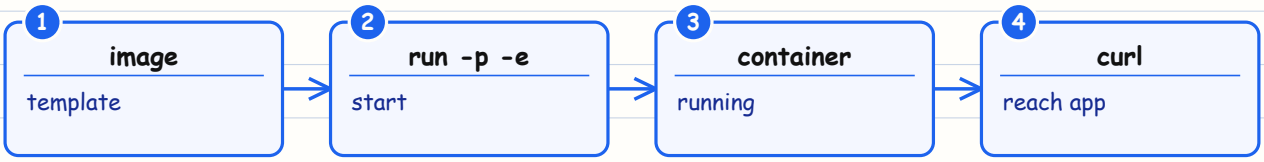
* Core Flags

- -d run detached; --name give a stable name; -p publish a port; -e set an env var; --rm auto-clean.
- -it gives an interactive terminal, useful for shells and debugging.

* Ports

- Containers are isolated: publish a port with -p host:container to reach the app from the host.
- Example: -p 8080:8080 maps host 8080 to the container app port 8080.

Run a container



REAL-WORLD EXAMPLE

```
docker run -d --name api -p 8080:8080 \
  -e APP_ENV=production shop-api:1.0
curl http://localhost:8080/health
docker run --rm -it ubuntu:24.04 bash   # throwaway shell
```

COMMON MISTAKE
Forgetting -p and wondering why localhost:8080 refuses the connection. Without publishing, the port is only reachable inside Docker networks, not from your host.

REVISION SNAPSHOT

ASK YOURSELF	What does -p 8080:8080 actually connect?
DO THIS	Run a web image, publish its port, and load it in your browser.

7. Dockerfile Basics

GrowthSchool

IN SIMPLE TERMS
A Dockerfile is a recipe of instructions (FROM, COPY, RUN, CMD) that tells Docker how to build your image step by step.

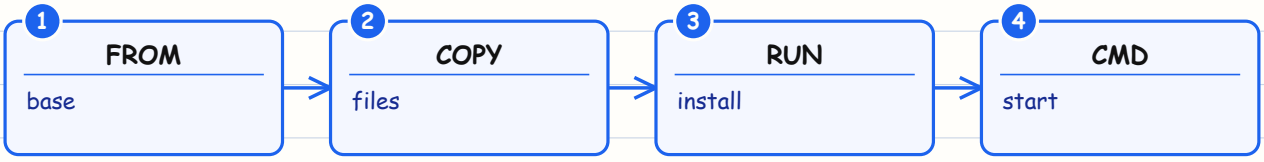
* The Essential Instructions

- FROM sets the base image. WORKDIR sets the working directory. COPY adds files from the build context.
- RUN executes build-time commands. CMD sets the default command when the container starts.

* Build and Run

- docker build -t name:tag . reads the Dockerfile and build context in the current folder.
- Then docker run name:tag starts a container from the image you built.

Dockerfile basics



REAL-WORLD EXAMPLE

```
FROM node:20-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci --omit=dev
COPY . .
EXPOSE 8080
CMD ["node", "server.js"]
```

COMMON MISTAKE
Doing COPY . . before installing dependencies. Any code change then busts the dependency cache and reinstalls everything. Copy manifests + install first, then copy the rest.

REVISION SNAPSHOT

ASK YOURSELF	Why copy package files and install BEFORE copying the whole app?
DO THIS	Write a minimal Dockerfile for a small app and build it successfully.

8. Dockerfile Deep Dive

GrowthSchool

IN SIMPLE TERMS
The deeper Dockerfile instructions control how your container starts and behaves - things like ENTRYPOINT, ENV, USER, and HEALTHCHECK.

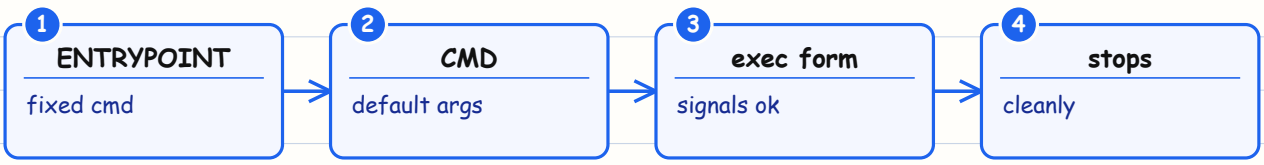
* CMD vs ENTRYPOINT

- ENTRYPOINT sets the fixed executable; CMD provides default arguments you can override at runtime.
- Use exec form ["cmd","arg"] so signals reach your process and it stops cleanly.

* More Instructions

- ARG = build-time variable; ENV = runtime variable; USER = drop root; HEALTHCHECK = liveness signal.
- EXPOSE documents a port (it does not publish it); LABEL adds metadata.

CMD vs ENTRYPOINT



REAL-WORLD EXAMPLE

```
FROM python:3.12-slim
ENV PYTHONUNBUFFERED=1
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
USER 1000
HEALTHCHECK CMD curl -f http://localhost:8080/health || exit 1
ENTRYPOINT ["gunicorn", "app:app", "-b", "0.0.0.0:8080"]
```

COMMON MISTAKE
Using shell form CMD gunicorn ... so the app runs as a child of /bin/sh. Then SIGTERM hits the shell, not your app, and stops become slow, ungraceful kills. Prefer exec form.

REVISION SNAPSHOT

ASK YOURSELF	When do you use ENTRYPOINT vs CMD, and why the exec form?
DO THIS	Add a non-root USER and a HEALTHCHECK to your Dockerfile.

9. Build Context & .dockerignore

GrowthSchool

IN SIMPLE TERMS
The build context is the folder sent to Docker when building; a .dockerignore file keeps unwanted files (and secrets) out of it.

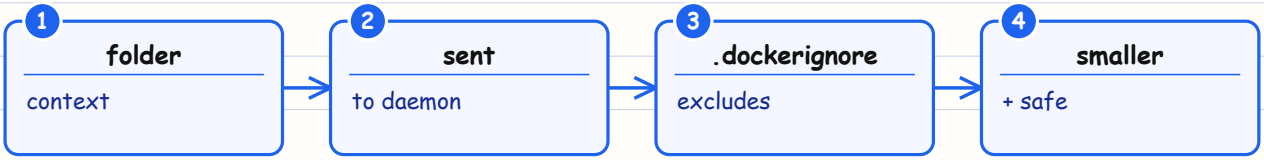
* The Build Context

- docker build sends the whole folder (the context) to the daemon. Big contexts = slow builds.
- Only files in the context can be COPYed. Keep it lean.

* .dockerignore

- Like .gitignore, it excludes files from the context: node_modules, .git, .env, dist, logs.
- This speeds builds, shrinks images, and stops secrets leaking into image layers.

Build context



REAL-WORLD EXAMPLE

```
# .dockerignore
.git
node_modules
.env
dist
*.log
Dockerfile
```

COMMON MISTAKE
Shipping node_modules or a .env into the image because there is no .dockerignore. This bloats the image and can bake secrets into a layer that anyone with the image can extract.

REVISION SNAPSHOT

ASK YOURSELF	Why can a missing .dockerignore leak secrets into an image?
DO THIS	Add a .dockerignore and compare build time and image size before/after.

10. Environment & Configuration

GrowthSchool

IN SIMPLE TERMS
Configuration means passing settings into a container at run time (environment variables) instead of baking them into the image.

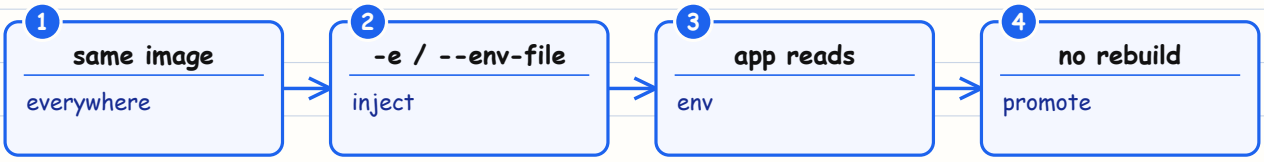
* Passing Config

- Use -e KEY=value or --env-file to inject configuration; read it from the environment in the app.
- Keep the same image across environments; only the injected config changes.

* Secrets Care

- Do not bake secrets into the image or commit .env. Inject at runtime or use a secrets manager.
- Anyone who can pull the image can read anything baked into its layers.

Config at runtime



REAL-WORLD EXAMPLE

```
docker run --env-file ./prod.env shop-api:1.0
docker run -e DB_HOST=db -e DB_PORT=5432 shop-api:1.0
# 12-factor: config comes from the environment, not the image
```

COMMON MISTAKE
Building separate images per environment (shop-api-dev, shop-api-prod). Build once, promote the same artifact, and vary only the injected configuration.

REVISION SNAPSHOT

ASK YOURSELF	Why build one image and inject config, instead of one image per environment?
DO THIS	Run the same image twice with different --env-file values.

11. Volumes & Bind Mounts

GrowthSchool

IN SIMPLE TERMS
A volume is Docker-managed storage that outlives a container; a bind mount maps a folder from your computer straight into the container.

* Why Persistence

- The container writable layer is disposable. Databases and uploads must live in a volume.
- Named volumes are managed by Docker; bind mounts map a host path into the container.

* Choosing

- Named volume for production data (portable, backed up). Bind mount for local dev live-reload.
- tmpfs keeps sensitive data in memory only, never on disk.

NAMED VOLUME	BIND MOUNT
managed by Docker in /var/lib/docker	maps a host path into the container
best for databases + app state	great for local dev live-reload
survives container removal	tied to host filesystem layout
-v shopdata:/var/lib/postgresql/data	-v $(pwd)/src:/app/src

REAL-WORLD EXAMPLE

```
docker volume create shopdata
docker run -d --name db -v shopdata:/var/lib/postgresql/data postgres:16
# dev live-reload with a bind mount:
docker run -v ${PWD}/src:/app/src shop-api:dev
```

COMMON MISTAKE
Storing database files in the container layer, then losing all data on docker rm. Always mount a named volume for stateful services.

REVISION SNAPSHOT

ASK YOURSELF	When do you pick a named volume vs a bind mount?
DO THIS	Put a database on a named volume, remove the container, recreate, verify data persists.

12. Docker Networking

GrowthSchool

IN SIMPLE TERMS
Docker networking is how containers get IP addresses, reach the internet, and talk to each other - by name on a shared network.

* Default Bridge

- By default containers join a bridge network with private IPs and reach the outside via NAT.
- Publish ports with -p to accept traffic from the host.

* User-Defined Networks

- Create a user-defined bridge so containers resolve each other by name via built-in DNS.
- This is how Compose lets the api reach the db as the hostname "db".

NETWORK DRIVERS

bridge : default, private subnet per host
host : share host network, no isolation
none : no networking at all
user-defined bridge : DNS by container name

KEY IDEAS

containers on one user network resolve each other by name (built-in DNS)
publish ports with -p host:container
never rely on changing container IPs

REAL-WORLD EXAMPLE

```
docker network create shopnet
docker run -d --name db --network shopnet postgres:16
docker run -d --name api --network shopnet -e DB_HOST=db shop-api:1.0
# api reaches the database at hostname "db"
```

COMMON MISTAKE
Hard-coding a container IP address in config. IPs change on restart. Use the container/service name on a user-defined network and let Docker DNS resolve it.

REVISION SNAPSHOT

ASK YOURSELF	Why do you get name-based DNS on a user-defined bridge but not the default one?
DO THIS	Put two containers on one network and curl one from the other by name.

13. Docker Compose Basics

GrowthSchool

IN SIMPLE TERMS
Docker Compose lets you define and run several containers together with one YAML file and one command.

* One File, Many Services

- A compose.yaml describes services, networks and volumes declaratively so you run them together.
- docker compose up -d starts everything; docker compose down stops and cleans it up.

* Why Compose

- It replaces long, error-prone docker run commands and documents your local stack in Git.
- Services share a network automatically and can reference each other by service name.

REAL-WORLD EXAMPLE

```
services:
  api:
    build: .
    ports: ["8080:8080"]
    environment: { DB_HOST: db }
  db:
    image: postgres:16
    volumes: ["shopdata:/var/lib/postgresql/data"]
volumes: { shopdata: {} }
```

REVISION SNAPSHOT

ASK YOURSELF	What three things does a compose file usually declare?
DO THIS	Write a two-service compose file and bring it up with docker compose up.

14. Compose for the Shop API

GrowthSchool

IN SIMPLE TERMS
This page builds a real multi-service Compose stack (web, API, database) for the Shop API on one network.

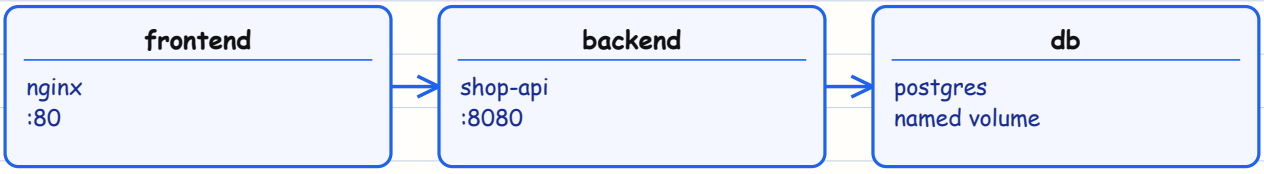
* The Full Local Stack

- Frontend (nginx), backend (shop-api) and database (postgres) on one network with a data volume.
- depends_on plus a healthcheck ensures the API waits for the database to be ready.

* Everyday Commands

- compose up -d --build to rebuild and start; compose logs -f api to tail; compose ps to see status.

Shop API with Compose: one network, three services



REAL-WORLD EXAMPLE

```
docker compose up -d --build
docker compose ps
docker compose logs -f api
docker compose exec db psql -U shop
docker compose down -v      # also remove volumes
```

COMMON MISTAKE
Assuming depends_on waits for the DB to be READY. It only waits for it to START. Add a healthcheck and a condition, or retry the DB connection in the app.

REVISION SNAPSHOT

ASK YOURSELF	Does depends_on guarantee the database is ready to accept queries?
DO THIS	Bring up the three-service Shop API stack and hit the API through nginx.

15. Multi-Stage Builds

GrowthSchool

IN SIMPLE TERMS
A multi-stage build compiles your app in a big "builder" image, then copies only the finished output into a tiny final image.

* The Problem

- Build tools (compilers, dev dependencies) make images huge and expand the attack surface.
- You need them to build, but not to run.

* The Solution

- Use multiple FROM stages: build in a fat stage, then COPY only the artifacts into a slim runtime.
- The final image contains just what runs, often 5-10x smaller.

Multi-Stage Build: fat builder -> slim runtime

REAL-WORLD EXAMPLE

```
# stage 1: build
FROM node:20 AS build
WORKDIR /app
COPY . . && RUN npm ci && npm run build
# stage 2: runtime
FROM nginx:alpine
COPY --from=build /app/dist /usr/share/nginx/html
```

COMMON MISTAKE
Shipping the build stage as the final image, dragging along compilers and dev deps. Always end on a minimal runtime stage and copy only built artifacts into it.

REVISION SNAPSHOT

ASK YOURSELF	What do you COPY --from the build stage into the runtime stage?
DO THIS	Convert a single-stage Dockerfile to multi-stage and compare image sizes.

16. Image Optimization

GrowthSchool

IN SIMPLE TERMS
Image optimization is the set of techniques - small base, fewer layers, good caching - that make images smaller and faster.

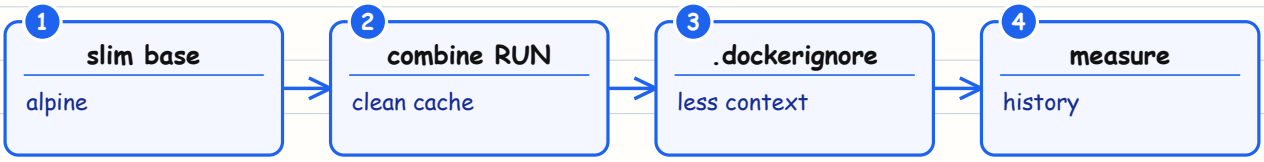
* Smaller & Faster

- Pick a slim/alpine base, combine RUN steps, clean package caches, and use .dockerignore.
- Order layers from least- to most-frequently changed to maximise cache hits.

* Measure It

- Use docker history and docker images to find fat layers; use dive-style analysis to inspect them.

Shrink the image



REAL-WORLD EXAMPLE

```
# combine + clean in one layer (Debian example)
RUN apt-get update && apt-get install -y --no-install-recommends curl \
    && rm -rf /var/lib/apt/lists/*
docker images shop-api        # check final size
docker history shop-api:1.0   # find the biggest layers
```

COMMON MISTAKE
Running apt-get update in one RUN and install in another. The cache can serve a stale package index. Combine update + install + cleanup in a single RUN.

REVISION SNAPSHOT

ASK YOURSELF	Name three concrete ways to shrink an image.
DO THIS	Reduce one of your images by at least 30% and record what changed.

17. Image & Supply-Chain Security

GrowthSchool

IN SIMPLE TERMS
Image security means shrinking the attack surface: minimal trusted bases, scanning for vulnerabilities, and never running as root.

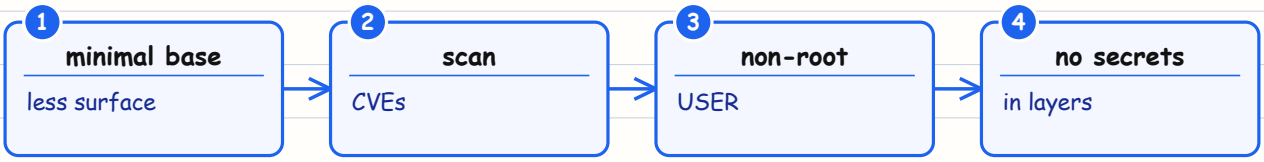
* Reduce Risk

- Use minimal, trusted base images; pin versions/digests; run as non-root; scan for CVEs.
- Fewer packages means fewer vulnerabilities and a smaller attack surface.

* Scan & Verify

- Scan images in CI and fail on critical CVEs; sign images so consumers can verify provenance.
- Never bake credentials into layers - they are trivially extractable.

Secure the image



REAL-WORLD EXAMPLE

```
docker scout cves shop-api:1.0     # or trivy image shop-api:1.0
# Dockerfile hardening:
USER 1000
FROM gcr.io/distroless/base-debian12   # no shell, tiny surface
```

COMMON MISTAKE
Running containers as root by default. If the app is compromised, root inside the container is a far bigger problem. Add a non-root USER.

REVISION SNAPSHOT

ASK YOURSELF	Why does running as non-root and using a minimal base reduce risk?
DO THIS	Scan one of your images and remediate at least one high/critical finding.

18. Debugging Containers

GrowthSchool

IN SIMPLE TERMS
Debugging containers is inspecting a misbehaving container using its logs, a shell inside it, and its full configuration.

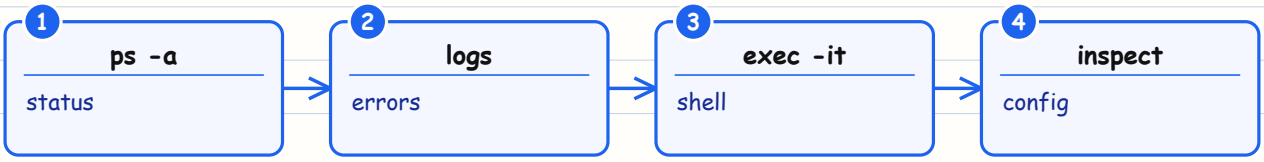
* First Four Commands

- docker logs shows stdout/stderr; docker exec opens a shell; docker inspect dumps full config.
- docker stats shows live CPU/memory/network per container.

* A Method

- Is it running? (ps) -> what did it say? (logs) -> get inside (exec) -> why configured so? (inspect).

Debug a container



REAL-WORLD EXAMPLE

```
docker ps -a
docker logs -f --tail 100 api
docker exec -it api sh
docker inspect api | less
docker stats api
```

COMMON MISTAKE
Deleting and recreating a crashing container repeatedly instead of reading its logs. The answer is almost always in docker logs or the exit code from docker ps -a.

REVISION SNAPSHOT

ASK YOURSELF	What is your four-step method for a misbehaving container?
DO THIS	Deliberately break an env var, then diagnose it using logs + inspect.

19. Resource Limits

GrowthSchool

IN SIMPLE TERMS
Resource limits cap how much CPU and memory a container may use, so one container cannot starve the others or the host.

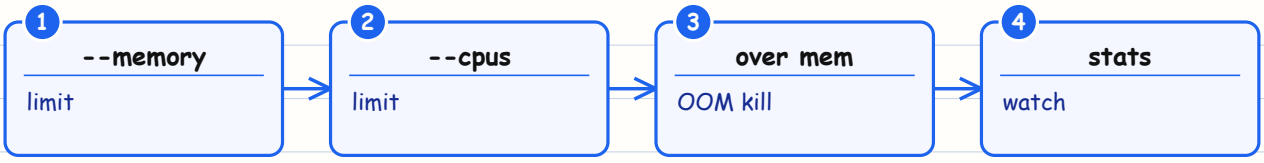
* Cap CPU & Memory

- Use --memory and --cpus to prevent one container from starving the host or its neighbours.
- Without limits, a leak in one container can take down everything else on the host.

* OOM Behaviour

- If a container exceeds its memory limit, the kernel OOM-kills it. Size limits from real usage.

Cap resources



REAL-WORLD EXAMPLE

```
docker run -d --name api \
  --memory=512m --cpus=1.0 shop-api:1.0
docker stats api          # watch actual usage
# in compose: deploy.resources.limits.{cpus,memory}
```

COMMON MISTAKE
Setting a memory limit far below real usage, causing constant OOM kills and restart loops. Measure typical + peak usage first, then set limits with headroom.

REVISION SNAPSHOT

ASK YOURSELF	What happens when a container exceeds its --memory limit?
DO THIS	Run a container with a tight memory limit and observe an OOM kill.

20. Healthchecks & Restart Policy

GrowthSchool

IN SIMPLE TERMS
A healthcheck lets Docker test if a container is truly working; a restart policy tells Docker whether to restart it if it crashes.

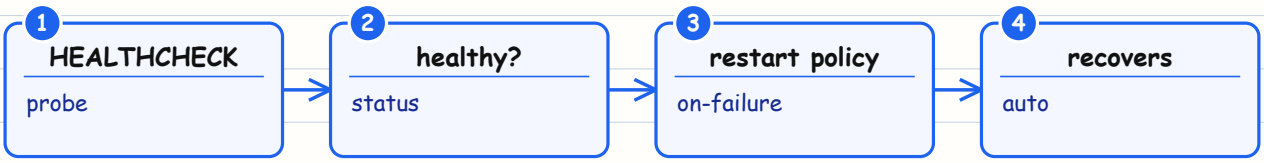
* Healthchecks

- A HEALTHCHECK lets Docker mark a container healthy/unhealthy based on a probe command.
- Orchestrators and Compose can wait for healthy before routing traffic or starting dependents.

* Restart Policies

- --restart=on-failure retries crashed containers; unless-stopped survives daemon restarts.
- Restart policy is basic self-healing for single hosts; Kubernetes does this at cluster scale.

Self-healing



REAL-WORLD EXAMPLE

```
docker run -d --restart=unless-stopped \
  --health-cmd="curl -f http://localhost:8080/health || exit 1" \
  --health-interval=10s shop-api:1.0
docker inspect --format "{{.State.Health.Status}}" api
```

COMMON MISTAKE
Using --restart=always for a container that crashes instantly on a config error - it just restart-loops forever. Fix the root cause; use on-failure with a sane backoff.

REVISION SNAPSHOT

ASK YOURSELF	What is the difference between a healthcheck and a restart policy?
DO THIS	Add a healthcheck to the Shop API and watch its status change.

21. Private Registries & Auth

GrowthSchool

IN SIMPLE TERMS
A private registry stores your company's images securely; you log in with credentials before pushing or pulling.

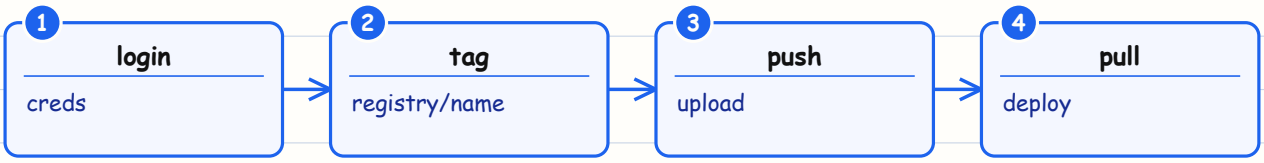
* Why Private

- Companies push internal images to a private registry (Docker Hub private, GHCR, ECR, GCR, ACR).
- Access is controlled with credentials or cloud IAM, keeping proprietary code off public hubs.

* Login & Tag

- Tag images with the registry hostname, docker login, then push. CI usually uses a robot token.

Private registry



PROVIDER-SPECIFIC
Registry auth differs per provider (ECR, GCR, ACR, GHCR). The docker tag/ push flow is the same; only the login step changes.

REAL-WORLD EXAMPLE

```
docker login registry.example.com
docker tag shop-api:1.0 registry.example.com/shop/shop-api:1.0
docker push registry.example.com/shop/shop-api:1.0
# cloud examples (provider-specific auth):
# aws ecr get-login-password | docker login --username AWS ...
```

REVISION SNAPSHOT

ASK YOURSELF	What must a private image name include that a public one may omit?
DO THIS	Push an image to any private registry and pull it from another machine.

22. Docker in CI/CD

GrowthSchool

IN SIMPLE TERMS
Using Docker in CI/CD means your pipeline builds one image, tags it, scans it, and ships that same image to every environment.

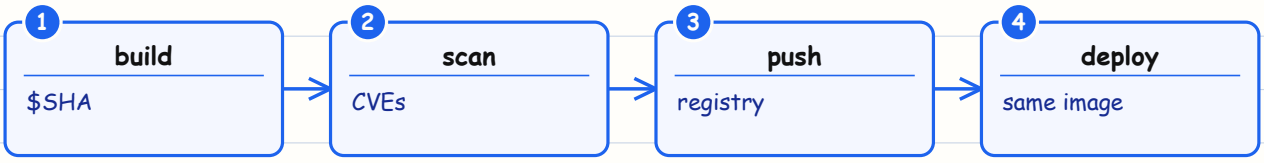
* Build Once, Deploy Many

- CI builds a single immutable image tagged with the commit SHA, then pushes it to a registry.
- The exact same artifact is promoted through staging and production - no rebuilds.

* Speed & Safety

- Cache layers between CI runs; scan the image; only push on green tests.
- Use build args for non-secret build config; use secret stores for credentials.

Build once, ship



REAL-WORLD EXAMPLE

```
docker build -t registry/shop-api:$GIT_SHA .
docker scout cves registry/shop-api:$GIT_SHA
docker push registry/shop-api:$GIT_SHA
# deploy step references the SAME immutable tag
```

COMMON MISTAKE
Rebuilding the image separately for each environment. That risks "works in staging, breaks in prod". Build one artifact, tag by SHA, and promote it unchanged.

REVISION SNAPSHOT

ASK YOURSELF	Why tag CI images by commit SHA instead of latest?
DO THIS	Build and push an image tagged by a fake commit SHA in a script.

23. Data & Persistence Patterns

GrowthSchool

IN SIMPLE TERMS
Persistence patterns are the rules for keeping data safe: keep apps stateless and put data in volumes or managed services.

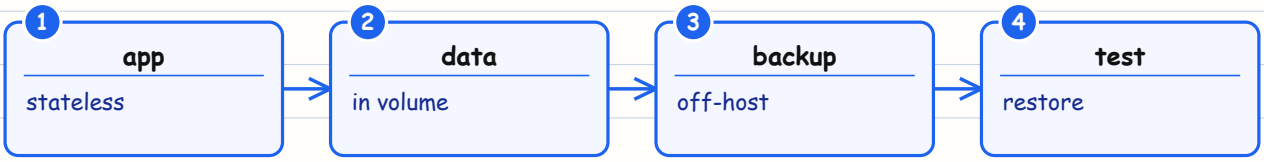
* Stateless vs Stateful

- Keep app containers stateless so they are disposable and horizontally scalable.
- Push state (DB, files, cache) into volumes or managed external services.

* Backups

- A volume is not a backup. Snapshot or dump volume data on a schedule and test restores.
- For heavy production databases, prefer a managed database over self-hosting in a container.

Keep state safe



REAL-WORLD EXAMPLE

```
# back up a postgres volume via a throwaway container
docker run --rm -v shopdata:/data -v ${PWD}:/backup alpine \
  tar czf /backup/shopdata.tgz -C /data .
# restore reverses the tar into the volume
```

COMMON MISTAKE
Treating a Docker volume as a durable backup. Disks fail. Export volume data off-host and periodically test that you can actually restore it.

REVISION SNAPSHOT

ASK YOURSELF	Why keep application containers stateless?
DO THIS	Back up a named volume to a tar file, then restore it into a fresh volume.

24. Best Practices

GrowthSchool

IN SIMPLE TERMS
Best practices are the proven habits - one process per container, small pinned base, non-root, no secrets baked in - that keep images clean.

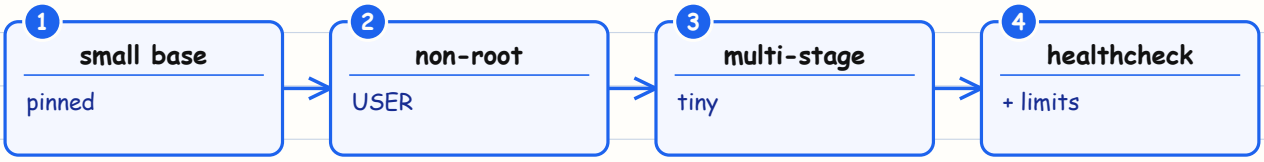
* Images

- One process per container; small trusted base; pin versions; non-root; .dockerignore; multi-stage.
- Log to stdout/stderr so a collector can pick logs up centrally.

* Operations

- Set resource limits and healthchecks; store data in volumes; never bake secrets into images.
- Keep the same image across environments and inject config at runtime.

Best-practice image



CHECKLIST
Small base - pinned tag - non-root USER - .dockerignore - multi-stage - healthcheck - resource limits - stdout logs - config via env - no secrets in layers.

COMMON MISTAKE
Cramming multiple services (nginx + app + cron) into one container. You lose independent scaling, restarts and clean logs. One concern per container.

REVISION SNAPSHOT

ASK YOURSELF	Recite the image checklist from memory.
DO THIS	Audit one of your Dockerfiles against every item in the checklist.

25. Troubleshooting Playbook

GrowthSchool

IN SIMPLE TERMS
Troubleshooting is the calm way to fix common Docker errors like port clashes, out-of-space, and build failures.

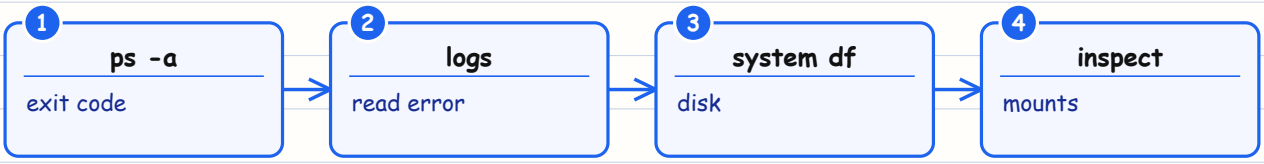
* Common Errors

- port is already allocated: another process/container uses that host port - change -p or free it.
- no space left on device: prune dangling images/volumes with docker system prune.

* More Fixes

- Cannot connect to the Docker daemon: the daemon is not running or you lack permissions.
- COPY failed / file not found: the file is outside the build context or .dockerignore excludes it.

Fix common errors



REAL-WORLD EXAMPLE

```
docker ps -a                  # find exit codes
docker logs <container>       # read the error
docker system df              # what is using disk
docker system prune -a        # reclaim space (careful!)
docker inspect <container>    # full config + mounts
```

COMMON MISTAKE
Running docker system prune -a on a shared/production host without thinking. It deletes all unused images and can remove data you still need. Understand the flags first.

REVISION SNAPSHOT

ASK YOURSELF	What causes "port is already allocated" and how do you fix it?
DO THIS	Reproduce a build "file not found" by excluding a file in .dockerignore.

26. Shop API Docker Project

GrowthSchool

IN SIMPLE TERMS
This page builds the complete Shop API Docker project end to end, from Dockerfile to a running Compose stack.

* Build the Image

- Write a multi-stage Dockerfile with a non-root user, healthcheck and pinned base image.
- Add a .dockerignore and build a SHA-tagged image.

* Run the Stack

- Use Compose to run api + postgres on a network with a named volume for the database.
- Confirm the API is healthy and can read/write the database.

REAL-WORLD EXAMPLE

```
docker build -t shop-api:$(git rev-parse --short HEAD) .
docker compose up -d --build
curl -f http://localhost:8080/health
docker compose exec db psql -U shop -c "\dt"
```

STUDENT LAB
Deliverables: a multi-stage image under 200 MB, running as non-root, with a passing healthcheck,
plus a compose stack whose database survives docker compose down (without -v).

REVISION SNAPSHOT

ASK YOURSELF	Does your Shop API image run as non-root with a healthcheck?
DO THIS	Complete the build + compose deliverables and verify data persistence.

27. Docker Cheat Sheet

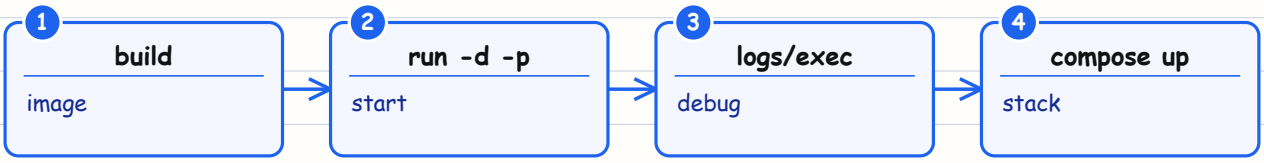
GrowthSchool

IN SIMPLE TERMS
This is a quick-reference list of the Docker commands you will use every day.

* Daily Drivers

- build, run -d -p -e, ps -a, logs -f, exec -it, inspect, stats, compose up/down, image prune.

Everyday Docker



DEBUG LOOP
ps -a -> logs -> exec -> inspect. These four answer "why is my container misbehaving?" in almost every case.

ONE-PAGE COMMAND REFERENCE

```
docker build -t app:1.0 .          docker logs -f api
docker run -d -p 80:80 app:1.0     docker exec -it api sh
docker ps -a                       docker inspect api
docker stop/start/rm api           docker stats
docker images / rmi                docker volume ls
docker compose up -d --build       docker network ls
docker compose logs -f api         docker system prune
docker push/pull registry/img      docker history app:1.0
```

STUDENT LAB
From memory: build an image, run it with a published port and an env var, tail its logs, shell into it, then remove it. Repeat until fluent.

REVISION SNAPSHOT

ASK YOURSELF	Which four commands debug almost any container issue?
DO THIS	Recall ten Docker commands from memory without looking.

28. Capstone Challenge

GrowthSchool

IN SIMPLE TERMS
This capstone ties everything together by packaging and running the Shop API the professional way.

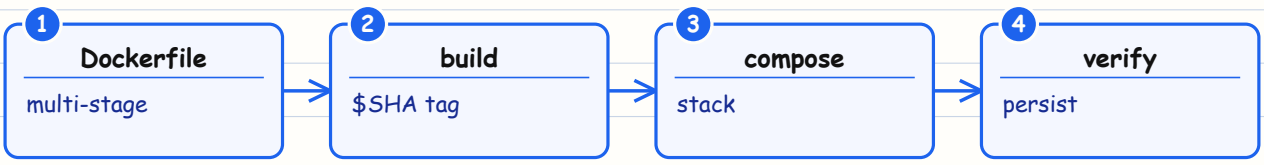
* Package the Shop API End to End

- Multi-stage Dockerfile, non-root, healthcheck, .dockerignore, pinned base, SHA tag.
- Compose stack: nginx -> shop-api -> postgres, one network, named volume for data.

* Prove It Works

- Publish the image to a registry, then deploy the same tag on a second machine.
- Show resource limits, a passing healthcheck and persistent data across restarts.

Capstone steps



REAL-WORLD EXAMPLE

```
docker build -t registry/shop-api:$SHA .
docker push registry/shop-api:$SHA
docker compose up -d
docker compose down && docker compose up -d   # data persists
```

STUDENT LAB
Stretch: add a CI script that builds, scans and pushes the image only when tests pass, then writes the image digest to a file for the deploy step to consume.

REVISION SNAPSHOT

ASK YOURSELF	Is your final image small, non-root, healthchecked and reproducible?
DO THIS	Complete every capstone deliverable and document the final image size.

29. Interview & Revision Notes

GrowthSchool

IN SIMPLE TERMS
This page gathers the key ideas and common interview questions so you can revise quickly.

* Explain Clearly

- Image vs container; layer caching; CMD vs ENTRYPOINT; volume vs bind mount; bridge vs host network.
- Why multi-stage builds, why non-root, and why one process per container.

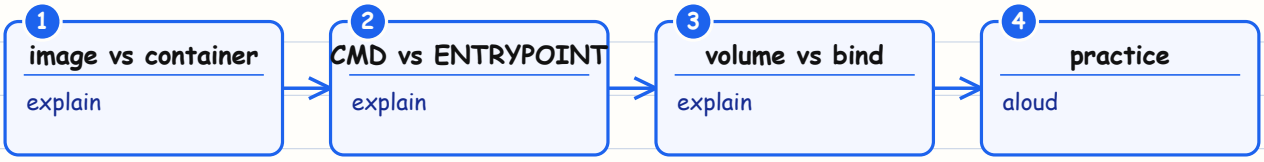
* Common Questions

- How do you shrink an image? Where does container data go? How do containers talk to each other?
- How do you pass secrets safely? What does docker build cache and how do you bust it well?

* Your Next Step

- Containerise a real app of your own, then optimise the image and write a compose stack for it.

Revise out loud



REAL-WORLD EXAMPLE

```
# 30-second self test
docker history <img>   # can you explain every layer?
docker inspect <ctr>   # where are its mounts + network?
# CMD vs ENTRYPOINT? volume vs bind mount? bridge vs host?
```

REVISION SNAPSHOT

ASK YOURSELF	Can you explain image vs container to a beginner in 30 seconds?
DO THIS	Answer every "Common Question" above out loud without notes.

GrowthSchool

KUBERNETES

COMPLETE NOTES

LEARN - BUILD - DEPLOY

Clusters, Pods, Services, storage, security and production ops

growthschool.cc | @growthschool.cc

Learning Roadmap

GrowthSchool

Learn Kubernetes in order. Every page has explanation, real kubectl/YAML, a diagram or example, common mistakes, a student lab and a revision snapshot. We deploy an ecommerce "Shop API" throughout.

HOW TO USE THESE NOTES

Practice on kind or minikube - a free local cluster. Type every command; read Events daily. Use pages 30-34 for fast revision before interviews.

1. Why Kubernetes

GrowthSchool

IN SIMPLE TERMS
Kubernetes (also written K8s) is a system that runs and manages your containers for you across many machines - starting them, restarting crashed ones, and scaling them up or down automatically.

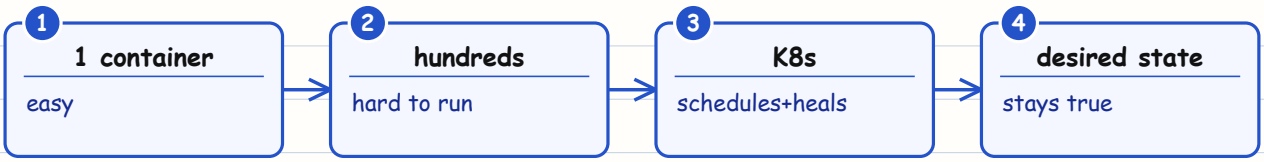
* The Problem It Solves

- One container is easy. Hundreds across many machines need scheduling, healing, networking and safe releases.
- Kubernetes continuously reconciles the cluster toward the desired state you declare in YAML.

* What You Get

- Self-healing, rolling updates/rollbacks, service discovery, load balancing, autoscaling and secrets.
- It is declarative: you describe the result; controllers make it happen and keep it true.

Why Kubernetes



REAL-WORLD EXAMPLE

```
# desired state in one command
kubectl apply -f shop-api.yaml
kubectl get deploy,pods,svc
# controllers keep actual state == desired state
```

COMMON MISTAKE
Reaching for Kubernetes for a single small app. It adds real operational complexity. Use it when you genuinely need scale, resilience and many services - not for a hello-world site.

REVISION SNAPSHOT

ASK YOURSELF	What does "declarative reconciliation" mean in Kubernetes?
DO THIS	Name three problems Kubernetes solves that a single Docker host cannot.

2. Containers Recap

GrowthSchool

IN SIMPLE TERMS
A container is your app packaged with everything it needs to run; an image is the saved template you start containers from. Kubernetes simply runs these containers at scale.

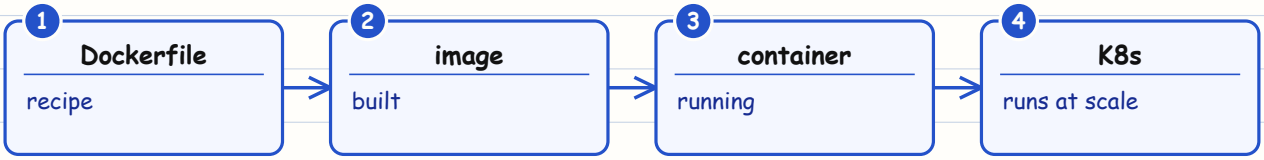
* Image vs Container

- An image is an immutable package (code + runtime + deps). A container is a running instance of it.
- Kubernetes schedules containers (inside Pods); it does not replace Docker knowledge, it builds on it.

* Why It Matters Here

- The same image runs on a laptop, CI and the cluster. Rollbacks are just running an older image tag.
- Never store durable data only in a container filesystem - use persistent storage.

Container to cluster



REAL-WORLD EXAMPLE

```
docker build -t shop-api:1.0 .
docker run -p 8080:8080 shop-api:1.0
# then Kubernetes runs the same image at scale
kubectl create deploy shop-api --image=shop-api:1.0
```

COMMON MISTAKE
Using the latest tag for cluster workloads. It is mutable and ambiguous, making rollbacks and debugging painful. Pin immutable tags like shop-api:1.4.2 or a digest.

REVISION SNAPSHOT

ASK YOURSELF	Why must container images be immutable and versioned for Kubernetes?
DO THIS	Build an image, run it locally, then create a Deployment from it.

3. Cluster Architecture

GrowthSchool

IN SIMPLE TERMS
A Kubernetes cluster is a group of machines working as one: a "control plane" that makes decisions and "worker nodes" that actually run your apps.

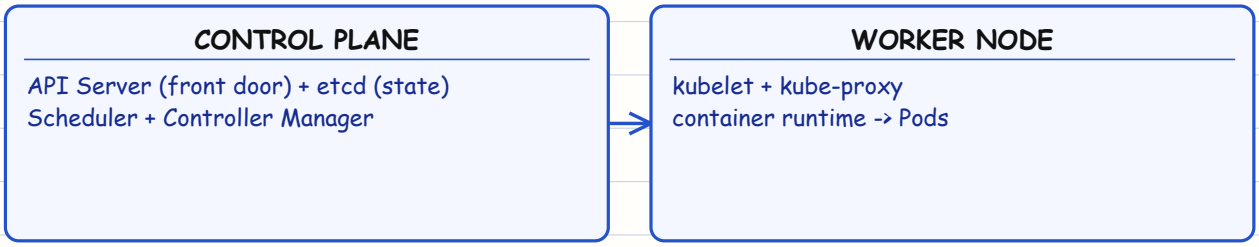
* Two Halves

- The control plane makes decisions; worker nodes run your Pods. Together they form a cluster.
- You talk to the cluster via the API Server using kubectl or CI/CD tooling.

* Desired vs Actual

- You submit desired state; controllers observe actual state and act to close the gap, forever.

Cluster: Control Plane (decides) + Worker Nodes (run)



REAL-WORLD EXAMPLE

```
kubectl cluster-info
kubectl get nodes -o wide
kubectl get componentstatuses   # legacy health view
kubectl get pods -n kube-system
```

REVISION SNAPSHOT

ASK YOURSELF	Which half decides and which half runs your workloads?
DO THIS	List your nodes and the system Pods running in kube-system.

4. Control Plane in Depth

GrowthSchool

IN SIMPLE TERMS
The control plane is the brain of the cluster - the components that decide what should run where and keep everything in the state you asked for.

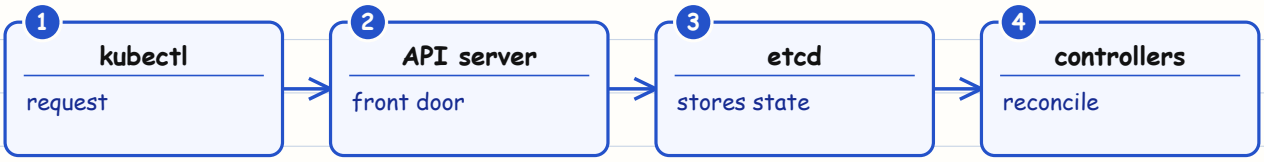
* The Components

- API Server: the front door and only component that talks to etcd. etcd: the key-value store of cluster state.
- Scheduler: picks a node for each Pod. Controller Manager: runs reconciliation loops for objects.

* Cloud Controller

- The cloud controller manager integrates load balancers, nodes and volumes with a cloud provider.
- The control plane does not run your business Pods; it coordinates and decides.

kubectl to running Pod



REAL-WORLD EXAMPLE

```
kubectl get pods -n kube-system
kubectl -n kube-system get pod -l component=kube-apiserver
kubectl describe pod -n kube-system <scheduler-pod>
# etcd holds ALL cluster state - back it up!
```

COMMON MISTAKE
Ignoring etcd backups. etcd is the single source of truth; losing it loses the whole cluster state. On self-managed clusters, snapshot etcd regularly and test restores.

REVISION SNAPSHOT

ASK YOURSELF	What is the only component that reads/writes etcd directly?
DO THIS	Identify the control-plane Pods and explain each one's job.

5. Worker Nodes in Depth

GrowthSchool

IN SIMPLE TERMS
A worker node is a machine that actually runs your application containers, with small agents that take orders from the control plane.

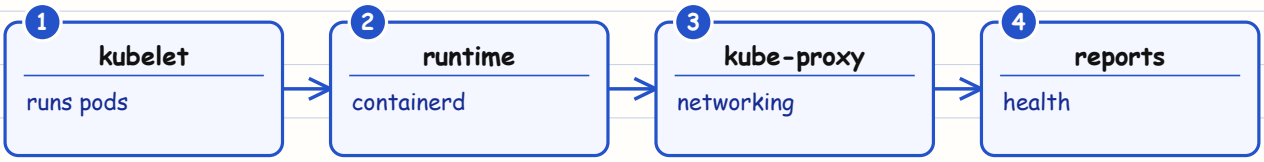
* Node Agents

- kubelet runs on each node, creates Pods assigned to it, and reports health back to the API Server.
- kube-proxy programs network rules so Service traffic reaches the right Pods.

* The Runtime

- A CRI runtime (containerd or CRI-O) actually pulls images and runs containers.
- Unhealthy nodes stop sending heartbeats; the control plane reschedules their Pods when possible.

Inside a worker node



REAL-WORLD EXAMPLE

```
kubectl get nodes
kubectl describe node <node>      # capacity, conditions
kubectl top nodes                 # needs metrics-server
kubectl get pods -o wide          # which node each Pod is on
```

COMMON MISTAKE
Treating nodes as pets. A node can die at any time; design workloads (via controllers) so Pods are recreated elsewhere automatically. Do not store state on the node.

REVISION SNAPSHOT

ASK YOURSELF	What do kubelet and kube-proxy each do on a worker node?
DO THIS	Describe a node and read its capacity, conditions and allocatable resources.

6. kubectl Basics

GrowthSchool

IN SIMPLE TERMS
kubectl is the command-line tool you use to talk to a Kubernetes cluster - to create, view, and change things.

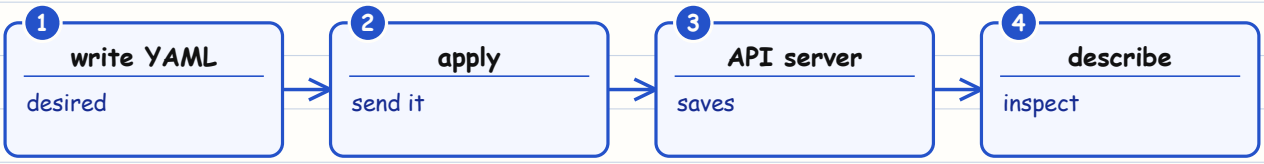
* The Universal Tool

- kubectl talks to the API Server. get lists, describe explains, logs reads output, apply creates/updates.
- Use -n for namespace, -o wide/yaml/json for output, and --dry-run=client to preview YAML.

* Imperative vs Declarative

- Imperative (create, run) is fast for learning; declarative (apply -f) is how you run production.
- Generate YAML with --dry-run=client -o yaml, then edit and apply it.

The kubectl loop



REAL-WORLD EXAMPLE

```
kubectl get pods -A
kubectl create deploy web --image=nginx --dry-run=client -o yaml > web.yaml
kubectl apply -f web.yaml
kubectl describe deploy web
kubectl explain deployment.spec.strategy
```

COMMON MISTAKE
Living on imperative commands in production. They are not reproducible or reviewable. Capture YAML in Git and apply it, so every change has history and can be rolled back.

REVISION SNAPSHOT

ASK YOURSELF	How do you generate starter YAML without creating the object?
DO THIS	Create a deployment YAML with --dry-run, edit it, then apply it.

7. YAML, Namespaces & Metadata

GrowthSchool

IN SIMPLE TERMS
A YAML manifest is a text file where you describe what you want Kubernetes to create; a namespace is a folder-like way to group and separate resources.

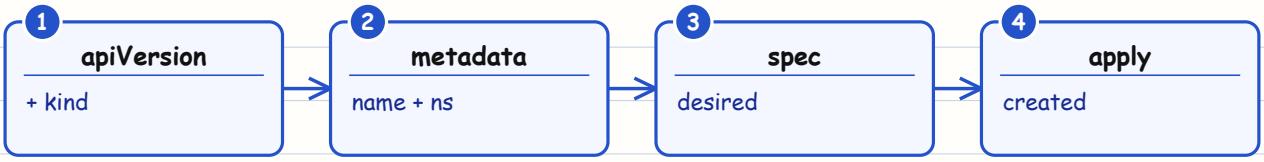
* Manifest Anatomy

- apiVersion selects the API; kind is the object type; metadata identifies it; spec is desired state.
- status is written by Kubernetes - never use it to configure an app.

* Namespaces

- Namespaces separate environments, teams and system components within one cluster.
- A namespace is not a hard security boundary alone - pair it with RBAC, quotas and NetworkPolicy.

Manifest anatomy



REAL-WORLD EXAMPLE

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: shop-api
  namespace: shop
spec:
  replicas: 3
kubectl create namespace shop
```

COMMON MISTAKE
Dumping everything into the default namespace. It gets messy and risky fast. Create purposeful namespaces (shop, monitoring) and set quotas per namespace.

REVISION SNAPSHOT

ASK YOURSELF	What are the four top-level fields of almost every manifest?
DO THIS	Create a namespace and deploy a labelled object into it.

8. Labels, Selectors & Annotations

GrowthSchool

IN SIMPLE TERMS
Labels are simple name tags you stick on objects, and selectors find objects by those tags. This tagging is how Kubernetes links things together.

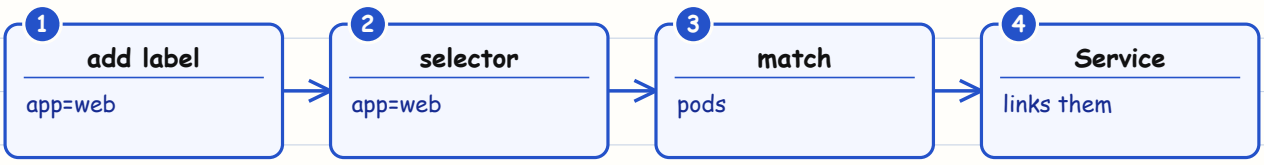
* Labels & Selectors

- Labels are key/value tags (app=shop-api, tier=backend). Selectors query them.
- Services and Deployments find their Pods through label selectors - this is the glue of Kubernetes.

* Annotations

- Annotations hold non-identifying metadata for tools (e.g. ingress config, checksums).
- Use labels for selection/grouping; use annotations for machine-readable extras.

Labels link things



REAL-WORLD EXAMPLE

```
kubectl label pod shop-api-x tier=backend
kubectl get pods -l app=shop-api,tier=backend
kubectl get pods -l "env in (prod,staging)"
kubectl annotate deploy shop-api note="owned by payments team"
```

COMMON MISTAKE
Inconsistent labels (App vs app, api vs shop-api). Selectors then silently match nothing. Agree a label convention and apply it everywhere.

REVISION SNAPSHOT

ASK YOURSELF	How do Services connect to the right Pods?
DO THIS	Select Pods by two labels at once, then by a set-based selector.

9. Pods

GrowthSchool

IN SIMPLE TERMS
A Pod is the smallest thing Kubernetes runs - a wrapper around one (or a few) containers that share a network address and storage.

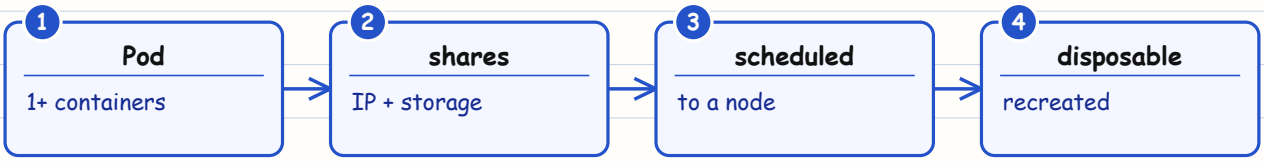
* The Smallest Unit

- A Pod wraps one or more containers that share an IP, localhost and volumes.
- Usually one app container per Pod; Pods are the unit Kubernetes schedules and heals.

* Pods Are Disposable

- Never depend on a Pod name or IP for stable traffic - they change as Pods are recreated.
- A controller (Deployment) should own production Pods so failures are recreated automatically.

A Pod



REAL-WORLD EXAMPLE

```
kubectl run demo --image=nginx
kubectl get pods -o wide
kubectl describe pod demo
kubectl delete pod demo    # a bare Pod is NOT recreated
```

COMMON MISTAKE
Running production apps as bare Pods. If the node dies, the Pod is gone for good. Always use a controller like Deployment so Pods self-heal.

REVISION SNAPSHOT

ASK YOURSELF	Why should you never rely on a Pod's IP for client traffic?
DO THIS	Create a bare Pod, delete it, and confirm nothing recreates it.

10. Multi-Container Pods

GrowthSchool

IN SIMPLE TERMS
A multi-container Pod holds a main app container plus small helper containers that support it and share the same Pod.

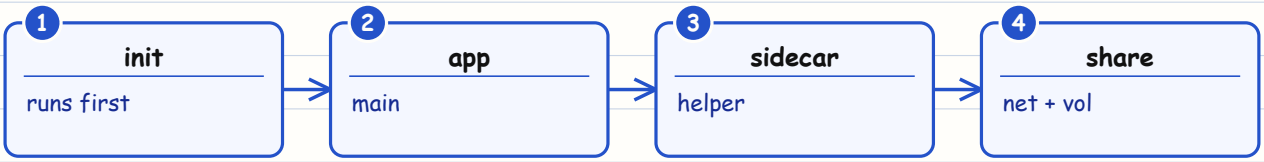
* Patterns

- Sidecar: a helper beside the app (log shipper, proxy). Init container: runs to completion first.
- Ambassador: a local proxy for external services. All containers in a Pod share network + volumes.

* Do Not Overuse

- Do not co-locate unrelated microservices in one Pod - they cannot scale or release independently.
- Use separate Deployments + a Service for communication between distinct apps.

Multi-container Pod



REAL-WORLD EXAMPLE

```
# init container waits for the DB, then the app starts
initContainers:
- name: wait-db
  image: busybox
  command: ["sh","-c","until nc -z db 5432; do sleep 2; done"]
```

COMMON MISTAKE
Cramming app + database + cron into one Pod. You lose independent scaling, restarts and clean logs. One primary concern per Pod; add sidecars only for a clear supporting purpose.

REVISION SNAPSHOT

ASK YOURSELF	When is a sidecar appropriate vs a separate Deployment?
DO THIS	Add an init container that waits for a dependency before the app starts.

11. ReplicaSet & Deployment

GrowthSchool

IN SIMPLE TERMS
A Deployment is a manager that keeps the right number of identical Pods running and updates them safely; a ReplicaSet is the piece that maintains the count.

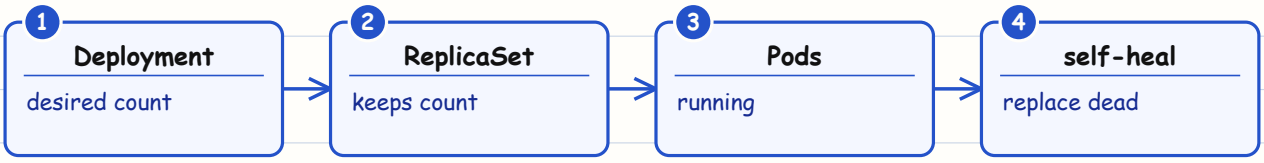
* ReplicaSet

- Keeps a specified number of matching Pod replicas alive; recreates any that die.
- You rarely create these directly - Deployments manage them for you.

* Deployment

- Declaratively manages stateless apps: rolling updates, pause/resume and revision history for rollback.
- Use it for web APIs, frontends and workers whose replicas are interchangeable.

Deployment manages Pods



REAL-WORLD EXAMPLE

```
kubectl create deploy shop-api --image=shop-api:1.0 --replicas=3
kubectl get rs,deploy
kubectl scale deploy/shop-api --replicas=5
kubectl rollout history deploy/shop-api
```

COMMON MISTAKE
Editing Pods created by a Deployment directly. The controller reverts your change on the next reconcile. Change the Deployment template instead - it owns the Pods.

REVISION SNAPSHOT

ASK YOURSELF	What does a Deployment add on top of a ReplicaSet?
DO THIS	Create a 3-replica Deployment, scale it, and view its rollout history.

12. Deployment YAML Deep Dive

GrowthSchool

IN SIMPLE TERMS
The Deployment YAML is the detailed recipe telling Kubernetes which image to run, how many copies, and how to update them.

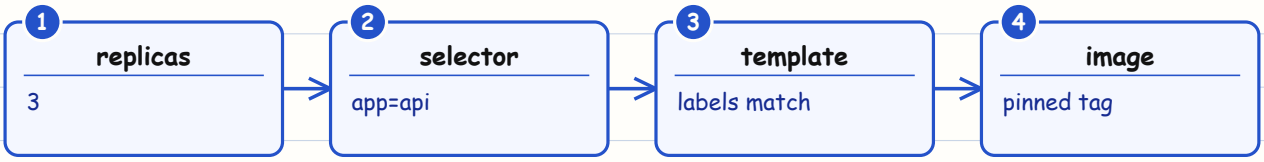
* Selectors Must Match

- spec.selector.matchLabels must equal template.metadata.labels or the Deployment cannot own its Pods.
- Set replicas for a baseline; let HPA adjust when metrics require.

* Safe Defaults

- Use immutable image tags, resource requests/limits, readiness probes and a rolling update strategy.
- Configure imagePullPolicy and pull secrets correctly for private images.

Deployment spec parts



REAL-WORLD EXAMPLE

```
spec:
  replicas: 3
  selector:
    matchLabels: { app: shop-api }
  template:
    metadata: { labels: { app: shop-api } }
    spec:
      containers: [{ name: api, image: shop-api:1.4.2 }]
```

COMMON MISTAKE
A selector that does not match the template labels. The Deployment then adopts no Pods (or the
API rejects it). Keep selector.matchLabels and template labels identical.

REVISION SNAPSHOT

ASK YOURSELF	Why must selector.matchLabels match the template labels exactly?
DO THIS	Write a Deployment spec and deliberately mismatch labels to see the error.

13. Rolling Update & Rollback

GrowthSchool

IN SIMPLE TERMS
A rolling update replaces old Pods with new ones gradually so users see no downtime; a rollback returns to the previous working version.

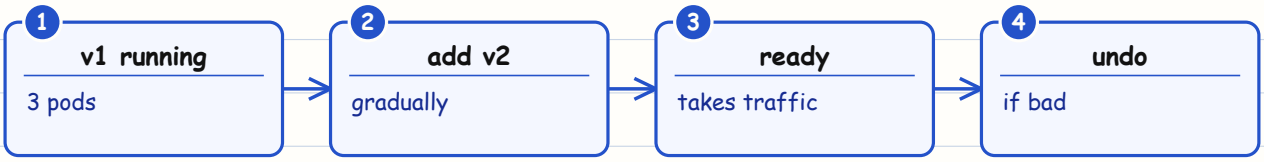
* Rolling Update

- New Pods are added and old ones removed gradually; readiness gates when new Pods take traffic.
- maxSurge is extra temporary capacity; maxUnavailable is how many may be down during the update.

* Rollback

- If a release is unhealthy, inspect events + logs, then roll back to a known-good revision fast.
- Rollback is quick only if the old image and config still exist.

Rolling update



REAL-WORLD EXAMPLE

```
kubectl set image deploy/shop-api api=shop-api:1.5.0
kubectl rollout status deploy/shop-api
kubectl rollout undo deploy/shop-api
kubectl rollout undo deploy/shop-api --to-revision=3
```

COMMON MISTAKE
Declaring a release "done" before rollout status completes. Watch it finish and check readiness and error rates before moving on - a half-rolled release can serve broken Pods.

REVISION SNAPSHOT

ASK YOURSELF	What do maxSurge and maxUnavailable control?
DO THIS	Roll a Deployment to a new tag, watch status, then roll it back.

14. Pod Lifecycle & Phases

GrowthSchool

IN SIMPLE TERMS
A Pod lifecycle is the set of stages a Pod passes through - from waiting to be placed, to running, to finishing or failing.

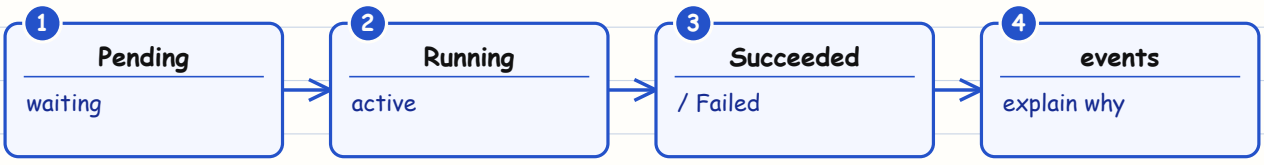
* Phases

- Pending: accepted, waiting to schedule or pull. Running: bound to a node, containers running.
- Succeeded/Failed: containers finished. Unknown: the node cannot report status.

* Restarts

- restartPolicy (Always for Deployments) plus backoff governs container restarts.
- CrashLoopBackOff is Kubernetes backing off between repeated crash restarts - read the logs.

Pod phases



REAL-WORLD EXAMPLE

```
kubectl get pods
kubectl describe pod <pod>     # Events explain Pending
kubectl logs <pod> --previous  # last crashed container
kubectl get events --sort-by=.lastTimestamp
```

COMMON MISTAKE
Deleting a CrashLoopBackOff Pod over and over. The controller recreates the same broken config.
Read logs + events and fix the root cause instead.

REVISION SNAPSHOT

ASK YOURSELF	What does CrashLoopBackOff actually indicate?
DO THIS	Force a Pending Pod (huge request) and diagnose it from Events.

15. Probes

GrowthSchool

IN SIMPLE TERMS
A probe is a small health check Kubernetes runs on your container to decide if it is alive and ready to receive traffic.

* Three Probes

- Liveness: is it alive? Failure restarts the container. Readiness: can it serve now? Failure removes it from endpoints.
- Startup: protects slow-booting apps so liveness/readiness only begin after startup succeeds.

* Use Them Right

- Readiness prevents traffic hitting unready Pods (during warm-up, dependency waits or draining).
- A too-aggressive liveness probe causes needless restart loops - give apps time to start.

Three Probes Guard Availability

REAL-WORLD EXAMPLE

```
readinessProbe:
  httpGet: { path: /ready, port: 8080 }
  initialDelaySeconds: 5
livenessProbe:
  httpGet: { path: /health, port: 8080 }
  periodSeconds: 10
```

COMMON MISTAKE
Pointing liveness at a deep dependency (e.g. the database). If the DB blips, Kubernetes kills healthy app Pods. Keep liveness shallow; use readiness for dependency checks.

REVISION SNAPSHOT

ASK YOURSELF	What is the difference in effect between liveness and readiness failure?
DO THIS	Add readiness + liveness probes to the Shop API and watch endpoints update.

16. Resources, Limits & QoS

GrowthSchool

IN SIMPLE TERMS
Requests are the resources a Pod is guaranteed; limits are the most it may use. QoS is how Kubernetes ranks Pods when resources run short.

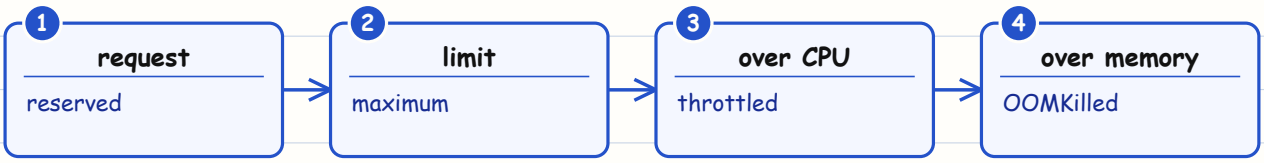
* Requests vs Limits

- A request is what the scheduler reserves; a limit caps usage. CPU over-limit throttles; memory over-limit OOM-kills.
- Requests decide placement; without them, a busy Pod can land on an already-crowded node.

* QoS Classes

- Guaranteed (requests==limits) > Burstable (some set) > BestEffort (none). Higher QoS survives pressure better.

Requests vs limits



REAL-WORLD EXAMPLE

```
resources:
  requests: { cpu: 500m, memory: 768Mi }
  limits:   { cpu: "1",  memory: 1Gi }
kubectl top pods    # needs metrics-server
```

COMMON MISTAKE
Setting a memory limit below real usage, causing constant OOMKills and restarts. Measure normal
and peak usage first, then set requests near normal and limits with headroom.

REVISION SNAPSHOT

ASK YOURSELF	What happens when a Pod exceeds its CPU limit vs its memory limit?
DO THIS	Set requests/limits on the Shop API and read its QoS class.

17. Autoscaling

GrowthSchool

IN SIMPLE TERMS
Autoscaling means Kubernetes automatically adds or removes Pods (and even machines) as demand rises and falls.

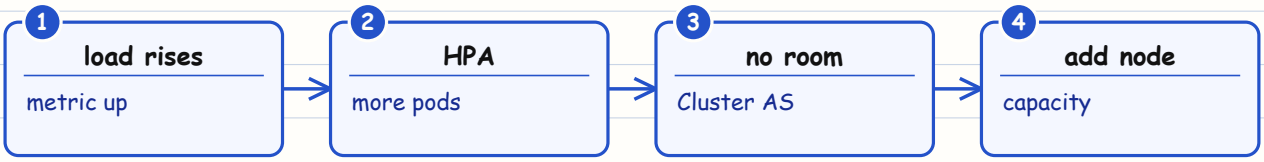
* Horizontal Pod Autoscaler

- HPA changes replica count from CPU/memory/custom metrics; it needs metrics-server for basic metrics.
- It works best when replicas are stateless and each has a meaningful resource request.

* Cluster Autoscaler

- Adds nodes when Pods cannot schedule and removes underused nodes when safe.
- It cannot fix impossible requests or bad node selectors - only capacity shortages.

Two kinds of scaling



REAL-WORLD EXAMPLE

```
kubectl autoscale deploy/shop-api --cpu-percent=60 --min=3 --max=20
kubectl get hpa
kubectl describe hpa shop-api
# HPA scales Pods; Cluster Autoscaler scales Nodes
```

COMMON MISTAKE
Expecting HPA to help a Pod with no resource requests. With no request, CPU utilisation is undefined and HPA cannot decide. Always set requests before enabling HPA.

REVISION SNAPSHOT

ASK YOURSELF	What does HPA scale vs what does the Cluster Autoscaler scale?
DO THIS	Add an HPA to the Shop API and load-test to watch it scale.

18. Services

GrowthSchool

IN SIMPLE TERMS
A Service gives a stable address and name to a group of Pods, so other apps can reach them even as Pods come and go.

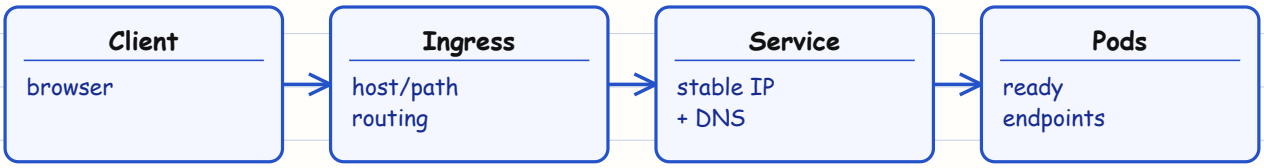
* Why Services

- Pods are ephemeral; a Service gives a stable virtual IP and DNS name to a changing set of Pods.
- A label selector picks the target Pods; traffic is load-balanced across ready endpoints.

* Service Types

- ClusterIP (internal, default), NodePort (a port on every node), LoadBalancer (cloud LB),
- ExternalName (DNS alias). Most in-cluster traffic uses ClusterIP.

Traffic Path: Ingress -> Service -> ready Pods



REAL-WORLD EXAMPLE

```
kubectl expose deploy/shop-api --port=80 --target-port=8080
kubectl get svc,endpointslices
kubectl port-forward svc/shop-api 8080:80
curl http://localhost:8080/health
```

COMMON MISTAKE
Hard-coding a Pod IP in a client instead of using the Service name. Pod IPs change; the Service name/IP is stable. Always talk to Services.

REVISION SNAPSHOT

ASK YOURSELF	What stable things does a Service provide over raw Pods?
DO THIS	Expose the Shop API with a Service and reach it via port-forward.

19. DNS & Service Discovery

GrowthSchool

IN SIMPLE TERMS
Cluster DNS lets Pods find each other by an easy name instead of by ever-changing IP addresses.

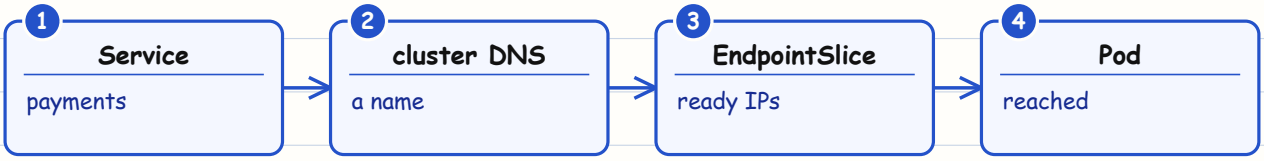
* Cluster DNS

- A Service payments in namespace shop resolves as payments, payments.shop, or payments.shop.svc.cluster.local.
- Apps should use these DNS names, never Pod IPs.

* Endpoints

- A selector builds EndpointSlices of ready Pod IPs. No endpoints usually means a bad selector or failed readiness.
- A headless Service (clusterIP: None) returns Pod addresses directly - common with StatefulSets.

Service discovery



REAL-WORLD EXAMPLE

```
kubectl get svc,endpoints,endpointslices
kubectl run netcheck --rm -it --image=busybox -- sh
# inside: nslookup shop-api.shop.svc.cluster.local
# inside: wget -qO- http://shop-api.shop/health
```

COMMON MISTAKE
Debugging "connection refused" without checking endpoints. If a Service has no endpoints, the selector or readiness is wrong - fix that before blaming the network.

REVISION SNAPSHOT

ASK YOURSELF	What does an empty EndpointSlice usually tell you?
DO THIS	From a debug Pod, resolve a Service by DNS and curl it.

20. Ingress & Gateway

GrowthSchool

IN SIMPLE TERMS
An Ingress is a smart HTTP entry point that routes web requests from the outside world to the right Service by hostname and URL path.

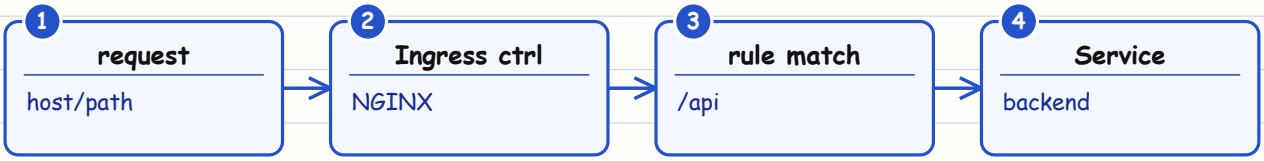
* Ingress

- Ingress routes external HTTP/HTTPS by host and path to Services. It needs an Ingress Controller (NGINX, Traefik).
- Example: shop.example.com/api -> shop-api Service; / -> frontend Service.

* TLS & Gateway API

- Terminate HTTPS at the Ingress with an automatic certificate manager; renew certs automatically.
- Gateway API is the newer, more expressive successor - learn it after mastering Ingress.

Ingress routing



REAL-WORLD EXAMPLE

```
rules:
- host: shop.example.com
  http:
    paths:
    - path: /api
      backend: { service: { name: shop-api, port: { number: 80 } } }
```

COMMON MISTAKE
Creating an Ingress resource with no Ingress Controller installed. Nothing routes traffic and you get confusing 404s. Install/confirm a controller first.

REVISION SNAPSHOT

ASK YOURSELF	What extra component must exist for Ingress to work?
DO THIS	Route two paths to two Services with a single Ingress resource.

21. ConfigMaps & Secrets

GrowthSchool

IN SIMPLE TERMS
A ConfigMap stores plain settings and a Secret stores sensitive values, so configuration lives outside your container image.

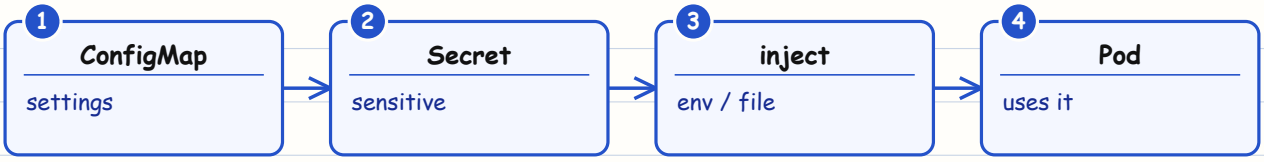
* ConfigMap

- Stores non-sensitive config: URLs, flags, log levels, config files. Inject as env vars or mounted files.
- Reload behaviour depends on the app - env changes usually need a Pod restart.

* Secret

- Stores sensitive values. base64 is encoding, not encryption. Protect with RBAC + encryption at rest.
- Avoid printing secrets in logs; prefer an external secrets manager for production.

Config vs Secret



REAL-WORLD EXAMPLE

```
kubectl create configmap shop-config --from-literal=MODE=production
kubectl create secret generic db-creds --from-literal=password=CHANGE-ME
# mount as env: envFrom: [{ configMapRef: { name: shop-config } }]
```

COMMON MISTAKE
Treating a Secret as secure just because it is base64. Anyone with get access can decode it. Lock down RBAC, enable encryption at rest, and keep Secrets out of Git.

REVISION SNAPSHOT

ASK YOURSELF	Why is base64 in a Secret not a security control?
DO THIS	Inject a ConfigMap as env and a Secret as a mounted file, then verify.

22. Volumes, PV, PVC & StorageClass

GrowthSchool

IN SIMPLE TERMS
A Volume gives a Pod storage; a PersistentVolumeClaim requests durable storage that survives Pod restarts, provisioned through a StorageClass.

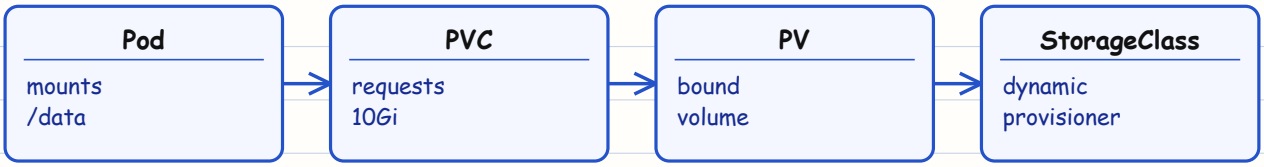
* The Chain

- A Volume attaches storage to a Pod. A PersistentVolume (PV) is a cluster storage resource.
- A PersistentVolumeClaim (PVC) requests storage; a StorageClass provisions PVs dynamically.

* Design Choice

- Keep state out of the container layer. Pick a StorageClass by latency, durability and access mode.
- ReadWriteOnce mounts to one node; some classes support ReadWriteMany.

Storage: Pod -> PVC -> PV (via StorageClass)



REAL-WORLD EXAMPLE

```
kubectl get sc,pv,pvc
# PVC requests 10Gi; a StorageClass provisions a PV
kubectl describe pvc shop-data
# mount: volumeMounts + volumes.persistentVolumeClaim
```

COMMON MISTAKE
Storing database files in the Pod filesystem. On reschedule the data is gone. Bind a PVC so data survives Pod restarts and rescheduling.

REVISION SNAPSHOT

ASK YOURSELF	What is the difference between a PV and a PVC?
DO THIS	Create a PVC, mount it in a Pod, recreate the Pod, and verify data persists.

23. StatefulSets

GrowthSchool

IN SIMPLE TERMS
A StatefulSet runs apps that need a stable identity and their own storage, like databases, giving each Pod a fixed name and disk.

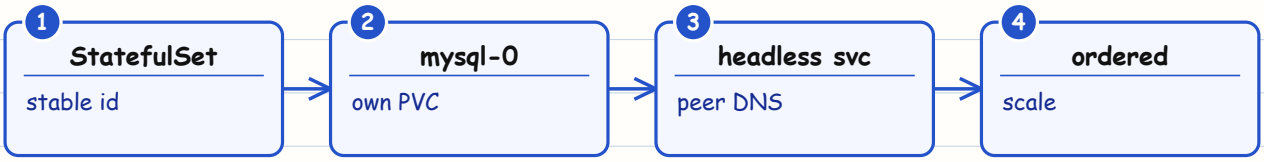
* When It Fits

- For databases, queues and clustered systems needing stable identities (mysql-0, mysql-1) and per-Pod storage.
- Each replica can get its own PVC; scaling and deletion follow a controlled order.

* Reality Check

- Kubernetes runs a database but does not remove its operational burden: backups, replication, upgrades, restore tests.
- For many teams, a managed database is the better default unless you have strong reasons to self-host.

StatefulSet identity



REAL-WORLD EXAMPLE

```
kubectl get statefulset,pvc
# pair with a headless Service for stable peer DNS:
# clusterIP: None -> mysql-0.mysql, mysql-1.mysql
kubectl delete pod mysql-0   # recreated with same name + PVC
```

COMMON MISTAKE
Using a Deployment for a stateful database expecting stable names/storage. Deployments give interchangeable Pods. Use a StatefulSet + headless Service for stable identity.

REVISION SNAPSHOT

ASK YOURSELF	What does a StatefulSet guarantee that a Deployment does not?
DO THIS	Deploy a small StatefulSet and observe stable Pod names and PVCs.

24. DaemonSets

GrowthSchool

IN SIMPLE TERMS
A DaemonSet makes sure one copy of a Pod runs on every node - handy for agents like log or monitoring collectors.

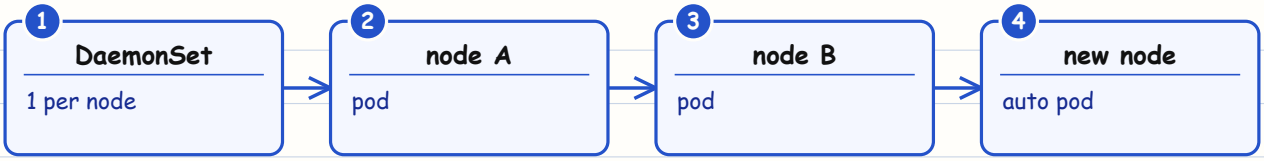
* One Pod Per Node

- A DaemonSet runs a copy of a Pod on every (or selected) node - ideal for node-level agents.
- Used for log collectors, monitoring agents and CNI/network components.

* Scheduling

- As nodes join, the DaemonSet adds a Pod; as nodes leave, its Pod is removed.
- Use node selectors/tolerations to target specific nodes (e.g. only GPU nodes).

DaemonSet coverage



REAL-WORLD EXAMPLE

```
kubectl get daemonset -A
# e.g. a log shipper on every node
kubectl describe ds -n logging fluent-bit
kubectl get pods -o wide -l app=fluent-bit
```

COMMON MISTAKE
Using a Deployment with nodeAffinity to fake "one per node". A DaemonSet is the correct, self-
maintaining primitive for per-node agents.

REVISION SNAPSHOT

ASK YOURSELF	When do you choose a DaemonSet over a Deployment?
DO THIS	Identify the DaemonSets in kube-system and explain what each does.

25. Jobs & CronJobs

GrowthSchool

IN SIMPLE TERMS
A Job runs a task once until it finishes; a CronJob runs Jobs automatically on a schedule.

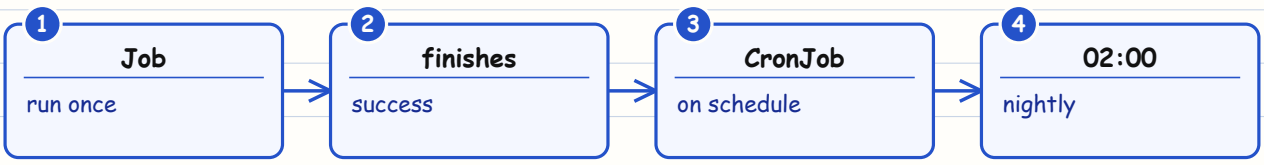
* Job

- Runs one-off work to successful completion: a migration, report or batch import.
- Use backoffLimit and activeDeadlineSeconds so failures do not retry forever.

* CronJob

- Creates Jobs on a schedule (cron syntax) - great for nightly exports and cleanup.
- Set concurrencyPolicy to avoid overlapping runs when a previous Job is slow.

Jobs and CronJobs



REAL-WORLD EXAMPLE

```
schedule: "0 2 * * *"          # daily 02:00
concurrencyPolicy: Forbid
successfulJobsHistoryLimit: 3
kubectl create job --from=cronjob/backup manual-backup
```

COMMON MISTAKE
Letting a CronJob overlap itself because a run is slow, doubling load or corrupting data. Set concurrencyPolicy: Forbid (or Replace) deliberately.

REVISION SNAPSHOT

ASK YOURSELF	How do you stop CronJob runs from overlapping?
DO THIS	Schedule a daily backup Job and trigger a manual run from it.

26. RBAC & ServiceAccounts

GrowthSchool

IN SIMPLE TERMS
RBAC controls who is allowed to do what in the cluster; a ServiceAccount is the identity an app uses to talk to the Kubernetes API.

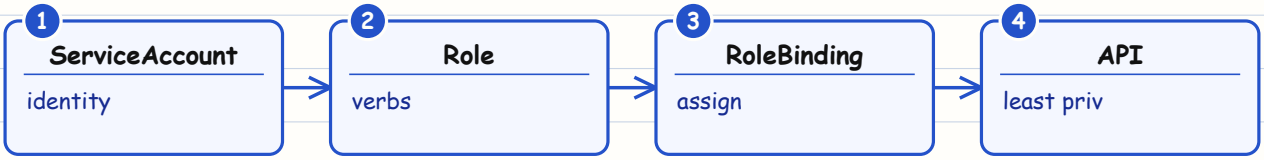
* Least Privilege

- A Role grants verbs on namespaced resources; a ClusterRole covers cluster-wide ones.
- A RoleBinding/ClusterRoleBinding assigns those to a user, group or ServiceAccount.

* Pod Identity

- Give each app its own ServiceAccount; disable token automount if it needs no API access.
- Never grant applications cluster-admin. Start read-only on exactly the resources needed.

Least-privilege RBAC



REAL-WORLD EXAMPLE

```
kubectl create serviceaccount shop-api -n shop
kubectl create role pod-reader --verb=get,list --resource=pods -n shop
kubectl create rolebinding shop-api-read \
  --role=pod-reader --serviceaccount=shop:shop-api -n shop
kubectl auth can-i list pods --as=system:serviceaccount:shop:shop-api -n shop
```

COMMON MISTAKE
Binding cluster-admin to an app "to make it work". A compromise then owns the cluster. Grant the
minimum verbs/resources/namespace and expand only when proven necessary.

REVISION SNAPSHOT

ASK YOURSELF	How do you test what a ServiceAccount is allowed to do?
DO THIS	Create an SA + Role that can only list Pods in one namespace and verify.

27. Network Policies

GrowthSchool

IN SIMPLE TERMS
A NetworkPolicy is a firewall for Pods - it decides which Pods are allowed to talk to which.

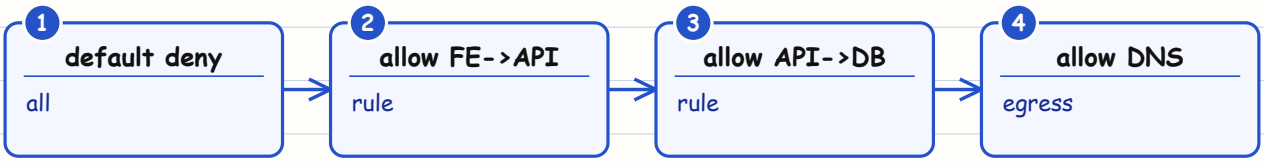
* Default Is Open

- Without policies, most CNIs allow all Pod-to-Pod traffic. A NetworkPolicy defines allowed ingress/egress.
- Enforcement needs a CNI that supports NetworkPolicy (Calico, Cilium, etc.).

* Zero-Trust Pattern

- Start with default-deny in a namespace, then allow only: frontend->api, api->db, and DNS egress.
- Select traffic by labels, not IPs - a missing DNS rule can break all name lookups.

Zero-trust network



REAL-WORLD EXAMPLE

```
# default deny all ingress in a namespace
spec:
  podSelector: {}
  policyTypes: [Ingress]
# then add the smallest allow rules needed
```

COMMON MISTAKE
Applying a default-deny egress policy but forgetting to allow DNS (port 53). Every outbound hostname lookup then fails. Always allow DNS egress explicitly.

REVISION SNAPSHOT

ASK YOURSELF	What is the safe order: deny first or allow first?
DO THIS	Apply default-deny, then add one rule allowing frontend to reach the API.

28. Pod Security & Image Safety

GrowthSchool

IN SIMPLE TERMS
Pod security and image safety are the settings and habits that stop containers running as root or shipping with known vulnerabilities.

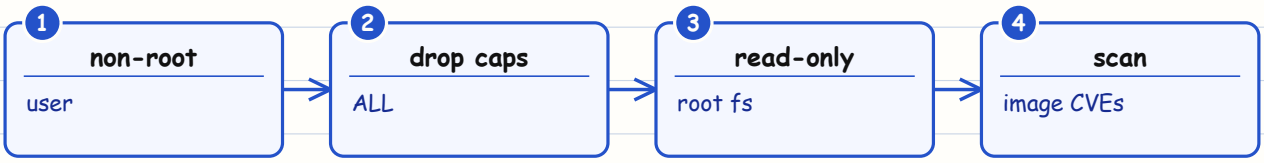
* Secure Runtime

- Run as non-root, disallow privilege escalation, drop Linux capabilities, use a read-only root FS.
- Avoid privileged containers and hostPath unless formally reviewed.

* Secure Supply Chain

- Use minimal trusted base images, scan for CVEs, sign images, and pin tags/digests.
- Never bake secrets or cloud keys into an image - anyone with the image can extract them.

Harden the Pod



REAL-WORLD EXAMPLE

```
securityContext:
  runAsNonRoot: true
  allowPrivilegeEscalation: false
  capabilities: { drop: ["ALL"] }
  readOnlyRootFilesystem: true
```

COMMON MISTAKE
Running every container as root by default. If the app is exploited, root in the container is a much bigger problem. Set runAsNonRoot and drop capabilities.

REVISION SNAPSHOT

ASK YOURSELF	Name four securityContext settings that harden a Pod.
DO THIS	Add a hardened securityContext to the Shop API and confirm it still runs.

29. Observability

GrowthSchool

IN SIMPLE TERMS
Observability means being able to see what your system is doing through its logs, metrics, and traces.

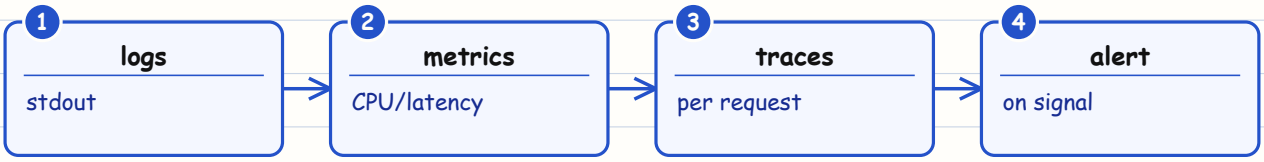
* Logs, Metrics, Traces

- Logs: write to stdout/stderr; a node collector ships them centrally. Use structured JSON + request IDs.
- Metrics: CPU, latency, error rate, saturation over time (Prometheus-style). Traces: one request across services.

* The Point

- You cannot operate what you cannot see. Every production change needs an observable success signal.

Three signals



REAL-WORLD EXAMPLE

```
kubectl logs -f deploy/shop-api
kubectl top pods    # needs metrics-server
kubectl get events --sort-by=.lastTimestamp
# dashboards + alerts on rate, errors, latency (RED method)
```

COMMON MISTAKE
Logging to a file inside the container. On restart the logs vanish and no collector sees them. Log to stdout/stderr so the platform captures everything centrally.

REVISION SNAPSHOT

ASK YOURSELF	What are the three pillars of observability?
DO THIS	Tail a Deployment's logs and read cluster Events during a rollout.

30. Troubleshooting Playbook

GrowthSchool

IN SIMPLE TERMS
Troubleshooting is the step-by-step way to find why a Pod is not working, using describe, logs, and events.

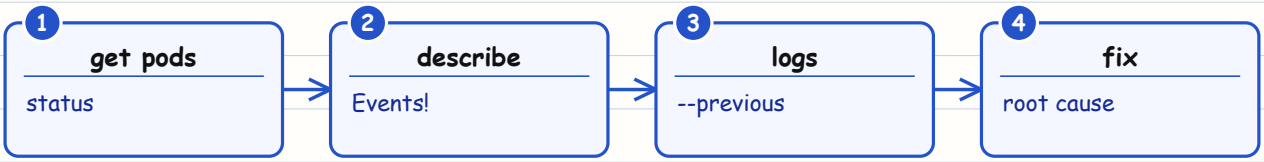
* Symptom -> Cause

- CrashLoopBackOff: read logs (+ --previous), check command/env/missing Secret/exit code/liveness.
- Pending: describe events for insufficient CPU/mem, taints, node selector, PVC binding or quota.

* More

- ImagePullBackOff: wrong image/tag, unreachable registry, missing imagePullSecrets.
- OOMKilled: memory limit too low or a leak - check usage and raise limits or fix the leak.

Debug a Pod



REAL-WORLD EXAMPLE

```
kubectl get pods -A
kubectl describe pod <pod>       # Events first!
kubectl logs <pod> --previous
kubectl get events --sort-by=.lastTimestamp
kubectl top pod <pod>
```

COMMON MISTAKE
Skipping kubectl describe and its Events. The answer to Pending, ImagePull and scheduling problems is almost always right there in the Events list.

REVISION SNAPSHOT

ASK YOURSELF	What is the first command you run for a misbehaving Pod?
DO THIS	Reproduce ImagePullBackOff (bad tag) and diagnose it from Events.

31. GitOps & Release Workflow

GrowthSchool

IN SIMPLE TERMS
GitOps means your cluster's desired state lives in Git, and an automated tool keeps the real cluster matching it.

* The GitOps Idea

- Store reviewed manifests (or Helm/Kustomize) in Git. CI builds + scans an image; CD reconciles the cluster.
- Git is the single source of truth; avoid imperative production changes that cannot be reproduced.

* Release Checklist

- Tests pass; image scanned; requests/limits + probes set; dashboards/alerts ready; rollback validated.
- Use progressive delivery (canary, blue/green) when the change is risky.

GitOps Release Flow (Shop API)

REAL-WORLD EXAMPLE

```
git commit -m "deploy shop-api:1.5.0"   # change manifest
# CI builds+scans image, CD applies desired state
kubectl rollout status deploy/shop-api
kubectl rollout undo deploy/shop-api     # fast rollback
```

COMMON MISTAKE
Hand-editing production with kubectl edit under pressure. It drifts from Git and cannot be reproduced. Change Git; let CD reconcile - even during incidents where possible.

REVISION SNAPSHOT

ASK YOURSELF	Why is Git the source of truth in GitOps?
DO THIS	Describe the path from a merged PR to a running change in the cluster.

32. Capstone: Deploy the Shop API

GrowthSchool

IN SIMPLE TERMS
This capstone ties everything together by deploying the full Shop API on Kubernetes, end to end.

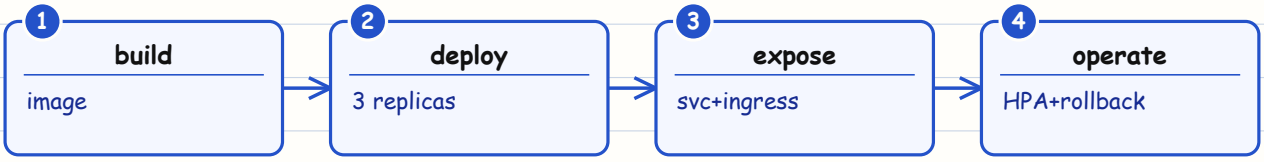
* Build & Deploy

- Containerise the API with /health and /ready endpoints and an immutable tag.
- Deploy 3 replicas with requests/limits, readiness/liveness probes and a rolling strategy.

* Expose & Secure

- Add a ClusterIP Service + Ingress host. Put config in a ConfigMap and the DB password in a Secret.
- Add an HPA on CPU and a default-deny NetworkPolicy allowing only required traffic.

Capstone steps



REAL-WORLD EXAMPLE

```
kubectl apply -f shop-api/       # deploy, svc, ingress, hpa
kubectl rollout status deploy/shop-api
kubectl get hpa,ingress,networkpolicy -n shop
kubectl port-forward svc/shop-api 8080:80
```

STUDENT LAB
Deliverables: 3 healthy replicas, working Service + Ingress, ConfigMap + Secret in use, an HPA, a NetworkPolicy, and a practised bad-release-then-rollback in a safe namespace.

REVISION SNAPSHOT

ASK YOURSELF	Does your capstone cover deploy, expose, scale, secure and rollback?
DO THIS	Complete every deliverable, then deliberately break a release and recover it.

33. kubectl Cheat Sheet

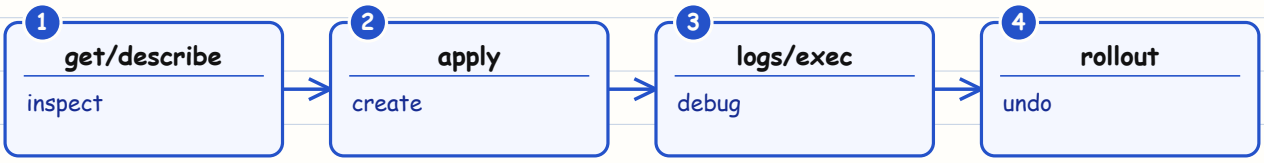
GrowthSchool

IN SIMPLE TERMS
This is a quick-reference list of the kubectl commands you will reach for every day.

* Daily Drivers

- get/describe/logs/exec, apply -f, rollout status/undo, scale, port-forward, top, get events.

Everyday kubectl



DEBUG LOOP
get -> describe (Events!) -> logs (--previous) -> exec -> events. This sequence solves the vast majority of Pod problems.

ONE-PAGE COMMAND REFERENCE

```
kubectl get pods -A              kubectl apply -f app.yaml
kubectl describe pod <p>         kubectl delete -f app.yaml
kubectl logs <p> -c <c>          kubectl rollout status deploy/x
kubectl logs <p> --previous      kubectl rollout undo deploy/x
kubectl exec -it <p> -- sh       kubectl scale deploy/x --replicas=5
kubectl get svc,endpointslices   kubectl port-forward svc/x 8080:80
kubectl top pods/nodes           kubectl get events --sort-by=.lastTimestamp
kubectl explain <kind>.spec      kubectl auth can-i <verb> <res>
```

COMMON MISTAKE
Forgetting -n / -A and looking in the wrong namespace, then concluding "it is not there". Always confirm the namespace; use -A to search everywhere.

REVISION SNAPSHOT

ASK YOURSELF	What is the standard get -> describe -> logs debug loop?
DO THIS	Recall ten kubectl commands from memory grouped by purpose.

34. Interview & Revision Notes

GrowthSchool

> **IN SIMPLE TERMS**
> This page gathers the key ideas and common interview questions so you can revise quickly.

* Explain Clearly

- Pod vs Deployment vs Service vs Ingress; PV vs PVC; ConfigMap vs Secret; requests vs limits.
- Why readiness matters, why Pod IPs are unstable, and how reconciliation works.

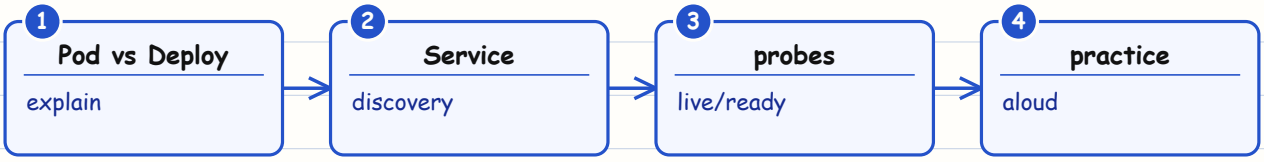
* Common Questions

- What happens when a Pod dies? How does a Service find Pods? How do you debug CrashLoopBackOff?
- How does HPA decide to scale? What is the difference between liveness and readiness?

* Your Next Step

- Build the capstone on kind/minikube, read Events daily, and learn by diagnosing real failures.

Revise out loud



> **REAL-WORLD EXAMPLE**

```
# 30-second self test
kubectl get deploy,rs,pods    # who owns whom?
kubectl describe pod <p>      # can you read the Events?
# Pod vs Deployment? PV vs PVC? liveness vs readiness?
```

> **REVISION SNAPSHOT**
>
> **ASK YOURSELF** Can you explain Pod vs Deployment vs Service in 30 seconds?
>
> **DO THIS** Answer every "Common Question" above out loud without notes.

GrowthSchool

TERRAFORM

COMPLETE NOTES

WRITE - PLAN - APPLY

Infrastructure as Code: providers, state, modules and CI/CD

growthschool.cc | @growthschool.cc

Learning Roadmap

GrowthSchool

Learn Terraform in order. Every page has explanation, real HCL, a diagram or example, common mistakes, a student lab and a revision snapshot. We provision infra for a "Shop API" throughout. Provider-specific blocks are clearly labelled.

HOW TO USE THESE NOTES

Use a free-tier or local provider to practice safely. Always read the plan before you apply. Never edit the state file by hand. Use pages 33-37 for fast revision before interviews.

1. Why Infrastructure as Code

GrowthSchool

> **IN SIMPLE TERMS**
> Infrastructure as Code (IaC) means describing your servers and cloud resources in text files instead of clicking in a console, so they are repeatable, reviewable, and version-controlled.

* The Problem With Clicking

- Hand-configured servers drift, are undocumented, and cannot be reproduced reliably.
- "It works because someone clicked something months ago" is not an operations strategy.

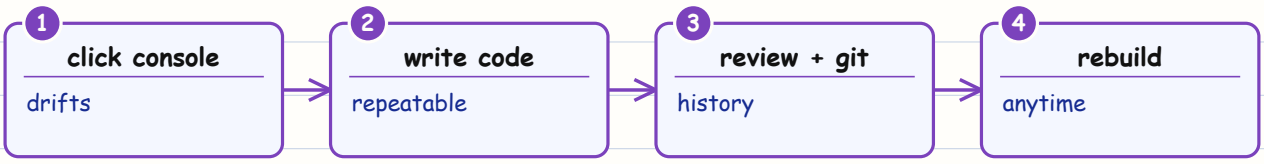
* IaC to the Rescue

- Describe infrastructure in version-controlled code; recreate identical environments on demand.
- You get review, history, repeatability and automation - the same discipline as application code.

* Why Terraform

- Terraform is declarative and cloud-agnostic: one workflow across many providers via plugins.

Why IaC



> **REAL-WORLD EXAMPLE**

```
# infra as code, reviewed and versioned in git
# describe WHAT you want; Terraform figures out HOW
resource "aws_s3_bucket" "assets" {
  bucket = "shop-api-assets"
}
```

> **COMMON MISTAKE**
> Mixing manual console changes with Terraform-managed resources. Terraform then fights your clicks on the next apply. Manage a resource in one place: code, not the console.

> **REVISION SNAPSHOT**
>
> **ASK YOURSELF** What concrete benefits does IaC add over clicking in a console?
>
> **DO THIS** List three problems with manual infra that Terraform solves.

2. How Terraform Works

GrowthSchool

IN SIMPLE TERMS
Terraform reads your desired setup, compares it to what already exists, and then creates or changes only what is needed to match - the same run twice does nothing extra.

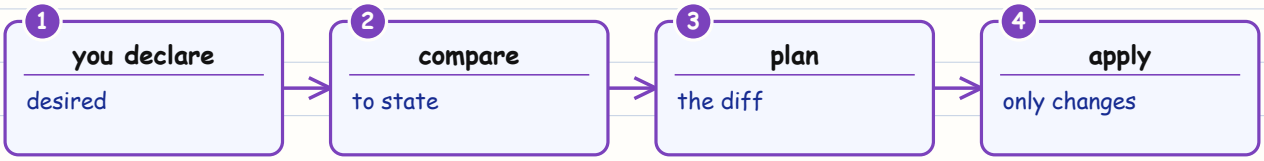
* Declarative & Idempotent

- You declare the desired end state; Terraform computes the changes needed to reach it.
- Running apply repeatedly with no config change makes no changes - it is idempotent.

* Providers & Graph

- Providers are plugins that talk to APIs (AWS, GCP, Azure, Kubernetes, GitHub, and many more).
- Terraform builds a dependency graph and creates/updates resources in the correct order.

How Terraform works



REAL-WORLD EXAMPLE

```
terraform {
  required_providers {
    aws = { source = "hashicorp/aws", version = "~> 5.0" }
  }
}
provider "aws" { region = "us-east-1" }   # provider-specific
```

REVISION SNAPSHOT

ASK YOURSELF	What does "declarative and idempotent" mean for Terraform?
DO THIS	Explain how Terraform decides the order to create resources.

3. Install, Providers & First Config

GrowthSchool

IN SIMPLE TERMS
Getting started means installing the Terraform tool, declaring which providers (plugins) you need, and running init to download them.

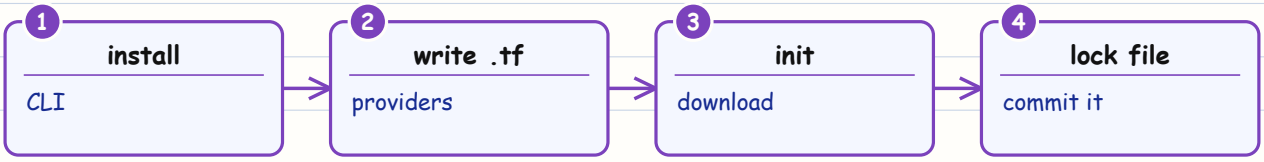
* Get Set Up

- Install the terraform CLI; create a folder with a .tf file; declare required providers.
- terraform init downloads providers into .terraform and writes a lock file.

* The Lock File

- .terraform.lock.hcl pins provider versions for reproducible builds - commit it to Git.

First config



REAL-WORLD EXAMPLE

```
terraform -version
mkdir shop-infra && cd shop-infra
# main.tf declares provider + resources
terraform init
terraform providers
```

COMMON MISTAKE
Not committing .terraform.lock.hcl. Teammates and CI then resolve different provider versions, causing "works on my machine" plan differences. Commit the lock file.

REVISION SNAPSHOT

ASK YOURSELF	What does terraform init do, and what should you commit from it?
DO THIS	Initialise a new config and inspect the generated lock file.

4. Core Workflow

GrowthSchool

IN SIMPLE TERMS
The core workflow is the four commands you repeat: init (set up), plan (preview changes), apply (make them real), and destroy (tear down).

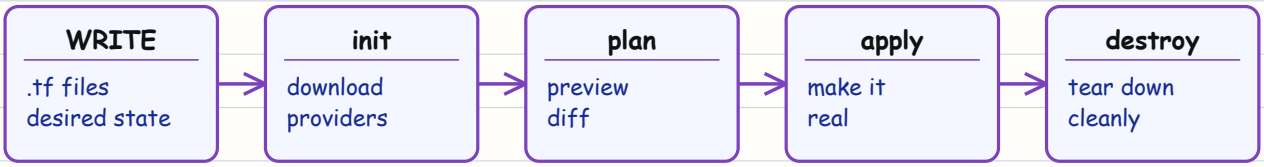
* init -> plan -> apply

- init sets up providers/backend; plan previews changes; apply makes them real; destroy tears down.
- plan is your safety net: always read it before applying, especially in production.

* Reading a Plan

- + create, ~ update in place, -/+ replace (destroy then create), - destroy. Watch for surprise replaces.

Terraform Core Workflow



REAL-WORLD EXAMPLE

```
terraform init
terraform fmt && terraform validate
terraform plan -out=tfplan
terraform apply tfplan
terraform destroy      # careful in shared envs
```

COMMON MISTAKE
Running apply without reading the plan. A small config change can trigger a destructive replace of a database or disk. Always review the +/~/-/+ symbols first.

REVISION SNAPSHOT

ASK YOURSELF	What do +, ~ and -/+ mean in a Terraform plan?
DO THIS	Apply a change with a saved plan file and read every action symbol.

5. HCL Syntax & Blocks

GrowthSchool

IN SIMPLE TERMS
HCL is Terraform's configuration language, built from simple blocks like resource, variable, and output.

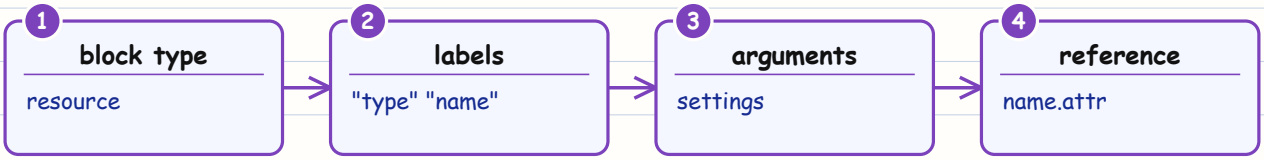
* The Building Blocks

- HCL is built from blocks: block_type "label" "name" { arguments and nested blocks }.
- Resources, variables, outputs, providers, modules and locals are all blocks.

* Reference Style

- Reference values as type.name.attribute, e.g. aws_s3_bucket.assets.arn.
- Comments use # or //; strings support ${...} interpolation.

HCL blocks



REAL-WORLD EXAMPLE

```
resource "aws_instance" "api" {   # block_type "type" "name"
  ami           = var.ami_id
  instance_type = "t3.micro"
  tags = { Name = "shop-api" }
}
# reference: aws_instance.api.public_ip
```

COMMON MISTAKE
Confusing the resource TYPE with its local NAME. In aws_instance.api, aws_instance is the type (fixed by the provider) and api is your chosen name used only for references in this config.

STUDENT LAB
Write one resource block, then in an output reference its attribute using type.name.attribute. Run terraform validate to confirm the reference resolves correctly.

REVISION SNAPSHOT

ASK YOURSELF	What is the structure of an HCL block and a resource reference?
DO THIS	Write a resource block and reference one of its attributes elsewhere.

6. Providers in Depth

GrowthSchool

IN SIMPLE TERMS
A provider is a plugin that lets Terraform talk to a specific platform (AWS, Azure, GCP, Kubernetes, and many more).

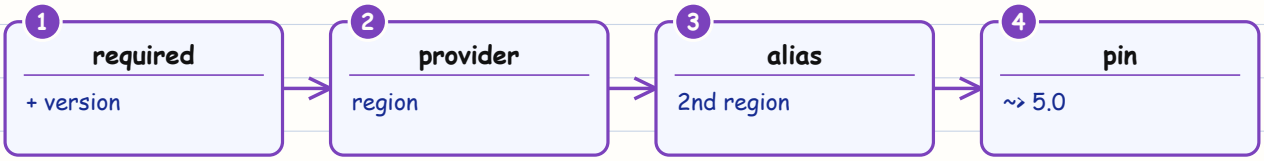
* Configure & Pin

- Declare providers in required_providers and pin versions with constraints like ~> 5.0.
- Configure provider settings (region, credentials source) in a provider block.

* Multiple Providers

- Use alias to configure the same provider more than once, e.g. two regions, and select with provider =.

Providers



REAL-WORLD EXAMPLE

```
provider "aws" {
  region = "us-east-1"
}
provider "aws" {
  alias  = "west"
  region = "us-west-2"
}
# resource ... { provider = aws.west }   # provider-specific
```

COMMON MISTAKE
Leaving version constraints open. A new major provider version can change behaviour and break plans. Pin with a sensible constraint (e.g. ~> 5.0) and upgrade deliberately.

REVISION SNAPSHOT

ASK YOURSELF	How do you configure the same provider for two regions?
DO THIS	Pin a provider version and add a second aliased provider.

7. Resources

GrowthSchool

IN SIMPLE TERMS
A resource is a single piece of infrastructure Terraform manages, like a server, bucket, or database.

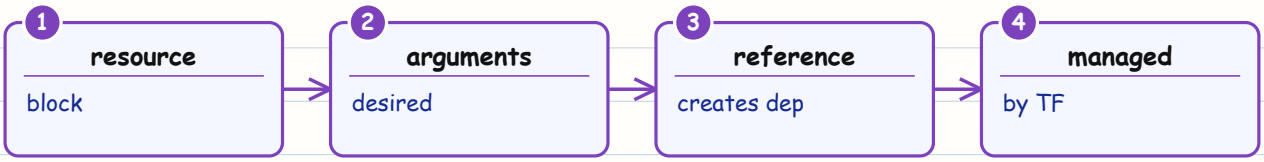
* The Core Object

- A resource block manages one infrastructure object. Terraform creates, updates and deletes it.
- Arguments set desired config; exported attributes (like id, arn) can be referenced by others.

* Implicit Dependencies

- Referencing one resource's attribute in another creates an ordering dependency automatically.
- This implicit graph is why you rarely need explicit depends_on.

A resource



REAL-WORLD EXAMPLE

```
resource "aws_s3_bucket" "assets" {   # provider-specific
  bucket = "shop-api-assets-123"
}
resource "aws_s3_bucket_versioning" "v" {
  bucket = aws_s3_bucket.assets.id   # implicit dependency
  versioning_configuration { status = "Enabled" }
}
```

REVISION SNAPSHOT

ASK YOURSELF	How does Terraform infer the order to create two related resources?
DO THIS	Create two resources where one references the other's attribute.

8. Data Sources

GrowthSchool

IN SIMPLE TERMS
A data source reads information about existing infrastructure without managing it - for example, looking up the latest image ID.

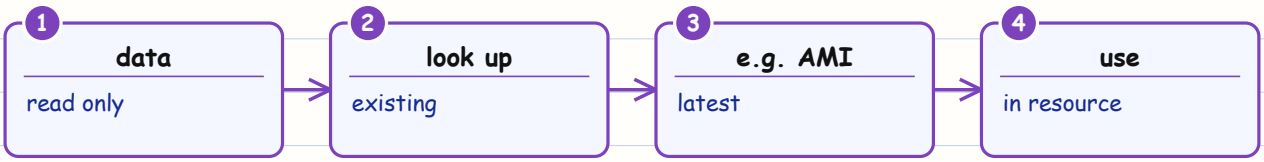
* Read, Do Not Manage

- A data source reads existing infrastructure or provider data without managing its lifecycle.
- Use it to look up an existing VPC, the latest AMI, or your account/region details.

* Why It Matters

- It lets your config depend on things Terraform did not create, keeping values dynamic not hard-coded.

Data source



REAL-WORLD EXAMPLE

```
data "aws_ami" "ubuntu" {          # provider-specific
  most_recent = true
  owners      = ["099720109477"]
  filter { name = "name"
           values = ["ubuntu/*-24.04-amd64-server-*"] }
}
# use: ami = data.aws_ami.ubuntu.id
```

COMMON MISTAKE
Hard-coding an AMI or resource ID that changes over time. Look it up with a data source so the config stays correct as the environment evolves.

REVISION SNAPSHOT

ASK YOURSELF	What is the difference between a resource and a data source?
DO THIS	Use a data source to fetch a value and feed it into a resource.

9. Input Variables

GrowthSchool

IN SIMPLE TERMS
An input variable is a parameter that makes your configuration reusable, so the same code can build dev, staging, and prod.

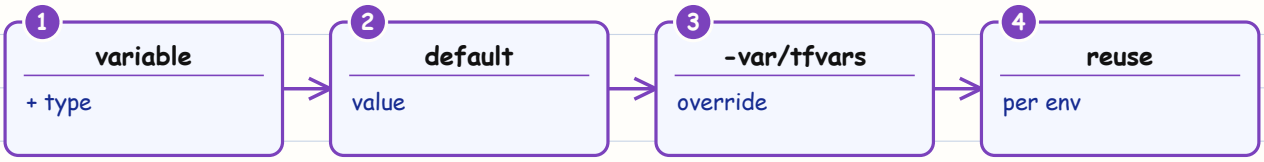
* Parameterise Config

- variable blocks make configs reusable. Set type, description, default and validation.
- Provide values via -var, .tfvars files, or TF_VAR_ environment variables.

* Good Hygiene

- Always type your variables and add descriptions; mark secrets sensitive = true.

Input variables



REAL-WORLD EXAMPLE

```
variable "instance_type" {
  type        = string
  default     = "t3.micro"
  description = "EC2 size for the Shop API"
}
# terraform apply -var="instance_type=t3.small"
# or a prod.tfvars file: instance_type = "t3.large"
```

COMMON MISTAKE
Committing a secrets.tfvars with real credentials. Keep secret tfvars out of Git (.gitignore) and inject them via environment or a secrets manager instead.

REVISION SNAPSHOT

ASK YOURSELF	Name three ways to supply a value for a variable.
DO THIS	Add a typed, described variable and override it with a .tfvars file.

10. Locals

GrowthSchool

IN SIMPLE TERMS
A local is a named value you compute once and reuse throughout a configuration, keeping it tidy (DRY).

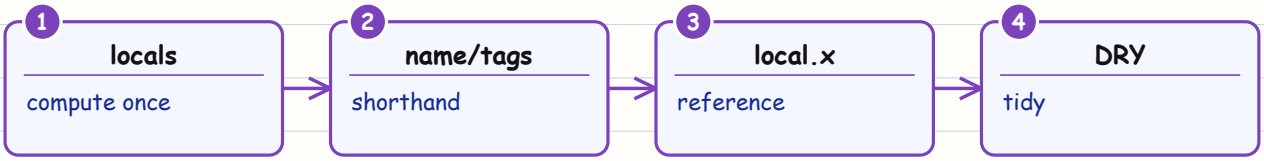
* Named Expressions

- locals define reusable computed values within a module - like constants or shorthands.
- Great for common tags, name prefixes, or expressions used in many places.

* DRY Config

- Change a local once and every reference updates. Reference as local.name.

Locals



REAL-WORLD EXAMPLE

```
locals {
  project = "shop-api"
  common_tags = {
    Project = local.project
    Managed = "terraform"
  }
}
# tags = local.common_tags
```

COMMON MISTAKE
Overusing locals for values that should be input variables. Locals are internal computations; variables are the external interface. Keep the distinction clear.

REVISION SNAPSHOT

ASK YOURSELF	When do you use a local vs an input variable?
DO THIS	Define a common_tags local and apply it to two resources.

11. Outputs

GrowthSchool

IN SIMPLE TERMS
An output exposes a useful value after apply - like a server's IP or URL - and passes values out of a module.

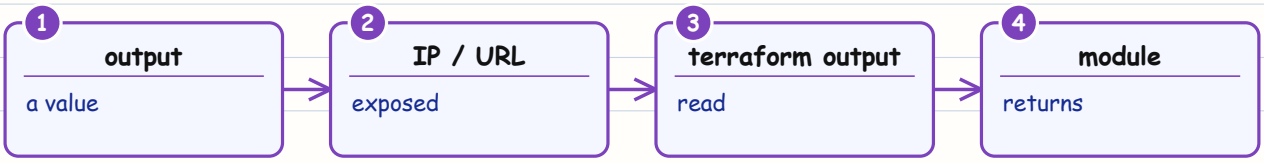
* Expose Results

- output blocks surface useful values after apply: an IP, a URL, an ARN, a database endpoint.
- Outputs are also how a child module returns values to its caller.

* Consume Outputs

- terraform output prints them; other tooling can read them as JSON for downstream steps.

Outputs



REAL-WORLD EXAMPLE

```
output "api_url" {
  value       = "https://${aws_lb.api.dns_name}"
  description = "Public URL of the Shop API"
}
terraform output api_url
terraform output -json | jq .api_url.value
```

COMMON MISTAKE
Printing a secret via an output without sensitive = true. It then appears in plan/apply logs and CI output. Mark secret outputs sensitive so Terraform redacts them.

STUDENT LAB
Add two outputs: one plain (a URL) and one sensitive (a token). Run terraform output and confirm
the sensitive one is redacted unless you request it explicitly.

REVISION SNAPSHOT

ASK YOURSELF	What are the two main uses of outputs?
DO THIS	Expose an API URL as an output and read it with terraform output.

12. State: What & Why

GrowthSchool

IN SIMPLE TERMS
State is the file where Terraform records what it created, so it knows what already exists and what to change next time.

* The Source of Truth

- Terraform records what it manages in a state file mapping config to real resource IDs.
- plan compares config vs state; state lets Terraform know what to update or destroy.

* Handle With Care

- State can contain sensitive values. Never edit it by hand; use terraform state commands.

Terraform reconciles: config vs state vs reality

REAL-WORLD EXAMPLE

```
terraform show           # human-readable state
terraform state list     # managed resources
terraform state show aws_s3_bucket.assets
# NEVER hand-edit terraform.tfstate
```

COMMON MISTAKE
Hand-editing terraform.tfstate to "fix" something. One wrong edit corrupts state and Terraform
may try to destroy/recreate real resources. Use state subcommands instead.

REVISION SNAPSHOT

ASK YOURSELF	Why does Terraform need a state file at all?
DO THIS	List managed resources and inspect one with terraform state show.

13. Remote State & Locking

GrowthSchool

IN SIMPLE TERMS
Remote state stores that file centrally so a whole team can share it safely; locking stops two people applying at the same time.

* Why Remote

- Local state does not work for teams: it is not shared, not backed up, and easy to lose.
- A remote backend stores state centrally, encrypted, and shared across the team and CI.

* Locking

- A lock prevents two people applying at once, which would corrupt state. Most backends lock automatically.

REMOTE BACKEND	STATE LOCKING
state stored centrally (S3, GCS, Azure Blob, Terraform Cloud)	a lock stops two applies at once (DynamoDB, backend-native lock)
shared safely across the team	prevents state corruption
encrypted at rest	force-unlock only when truly stuck

REAL-WORLD EXAMPLE

```
terraform {
  backend "s3" {                  # provider-specific
    bucket         = "shop-tfstate"
    key            = "prod/terraform.tfstate"
    region         = "us-east-1"
    dynamodb_table = "tf-locks"   # state locking
  }
}
```

COMMON MISTAKE
Keeping state on a laptop for a team project. Someone deletes the folder, and now Terraform has no idea what exists. Use a remote backend with locking from day one on shared infra.

REVISION SNAPSHOT

ASK YOURSELF	What two problems does a remote backend with locking solve?
DO THIS	Describe how you would migrate local state to a remote backend.

14. State Safety & Secrets

GrowthSchool

IN SIMPLE TERMS
State safety matters because state can contain secrets in plain text, so you must protect it and keep it out of Git.

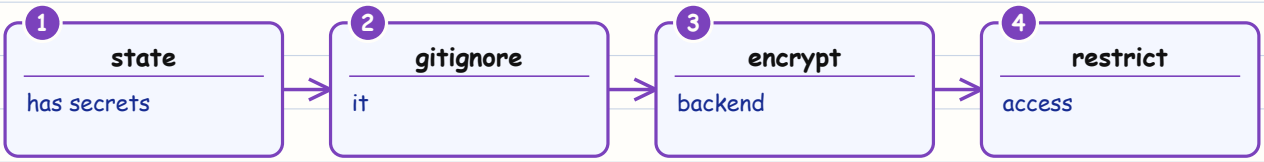
* Secrets Live in State

- Resource attributes (DB passwords, keys) are stored in plaintext inside state.
- Therefore protect state like a secret: encrypt at rest, restrict access, never commit it.

* Safe Practices

- Add terraform.tfstate* to .gitignore; use a backend with encryption + access control + versioning.
- Limit who can read state; prefer generating secrets outside Terraform where practical.

State is sensitive



REAL-WORLD EXAMPLE

```
# .gitignore
terraform.tfstate
terraform.tfstate.backup
.terraform/
*.tfvars   # if they contain secrets
```

COMMON MISTAKE
Committing terraform.tfstate to Git. It often contains secrets in plaintext and is now in history forever. gitignore it, rotate any exposed secret, and move to a secure backend.

REVISION SNAPSHOT

ASK YOURSELF	Why must state be treated as sensitive data?
DO THIS	Confirm your .gitignore excludes state and secret tfvars files.

15. Expressions & Operators

GrowthSchool

IN SIMPLE TERMS
Expressions are how you compute values in HCL using operators, references, and types like lists and maps.

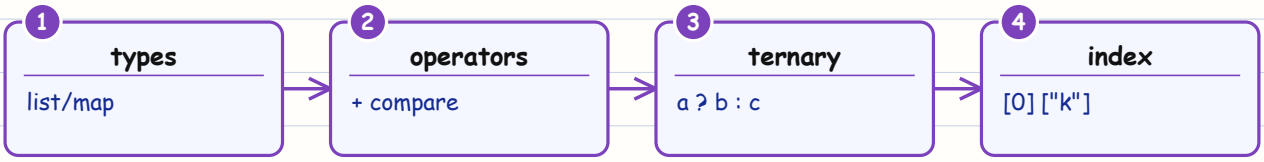
* Work With Values

- HCL supports arithmetic, comparison and logical operators, plus string interpolation "${...}".
- Types include string, number, bool, list, map, set, object and tuple.

* Reference & Index

- Access list items with [0], map values with ["key"], and object attributes with .attr.

Expressions



REAL-WORLD EXAMPLE

```
locals {
  replicas = var.env == "prod" ? 5 : 1
  name     = "${local.project}-${var.env}"
  first_az = var.azs[0]
  port_map = { http = 80, https = 443 }
}
```

COMMON MISTAKE
Mixing types in a comparison or indexing a list that might be empty. Terraform errors at plan time; use length() checks and consistent types to keep expressions robust.

STUDENT LAB
In terraform console, build a map and index it, index a list, and evaluate a ternary. Predict each result before pressing enter to test your mental model of HCL types.

REVISION SNAPSHOT

ASK YOURSELF	Which types and operators does HCL provide?
DO THIS	Write a ternary that sets replicas based on the environment.

16. Built-in Functions

GrowthSchool

IN SIMPLE TERMS
Built-in functions are ready-made helpers (like join, merge, and length) for working with strings and collections.

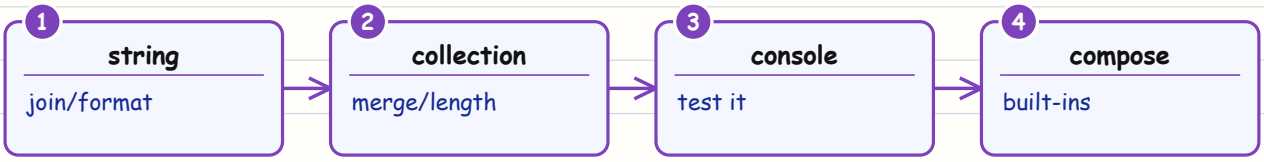
* Common Helpers

- String: upper, lower, format, join, split, replace. Collection: length, merge, concat, lookup, keys.
- There is no user-defined functions - compose the built-ins instead.

* Try It Live

- terraform console is a REPL to test expressions and functions before using them in config.

Functions



REAL-WORLD EXAMPLE

```
terraform console
> merge({a=1}, {b=2})
> join("-", ["shop","api","prod"])
> lookup({http=80,https=443}, "https", 0)
> length(["a","b","c"])
```

COMMON MISTAKE
Building fragile string concatenation when a function like format() or join() is clearer and safer. Explore terraform console to find the right built-in.

REVISION SNAPSHOT

ASK YOURSELF	How do you test an expression without running apply?
DO THIS	Use terraform console to try merge, join and lookup.

17. count

GrowthSchool

IN SIMPLE TERMS
count creates a fixed number of copies of a resource, or switches one on and off with a simple 0-or-1 toggle.

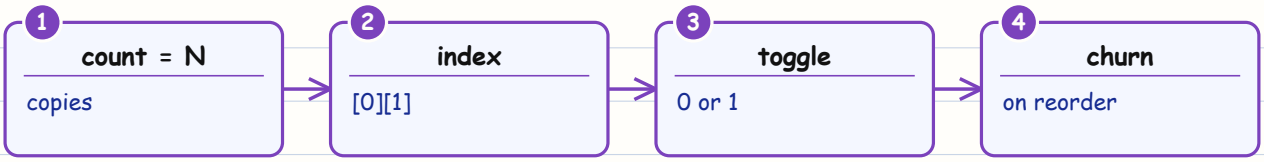
* Repeat a Resource

- count = N creates N instances of a resource, indexed by count.index (0-based).
- Good for identical copies; reference them as name[0], name[1], or name[*] for all.

* Toggle On/Off

- count = var.enabled ? 1 : 0 conditionally creates a resource - a common feature-flag pattern.

count



REAL-WORLD EXAMPLE

```
resource "aws_instance" "worker" {   # provider-specific
  count         = 3
  instance_type = "t3.micro"
  tags = { Name = "worker-${count.index}" }
}
# aws_instance.worker[*].id  -> all ids
```

COMMON MISTAKE
Using count for a set of items that can change order. Removing the middle item re-indexes and churns unrelated resources. Prefer for_each with stable keys for named sets.

REVISION SNAPSHOT

ASK YOURSELF	When does count cause unwanted resource churn?
DO THIS	Create 3 identical resources with count and reference them all.

18. for_each

GrowthSchool

IN SIMPLE TERMS
for_each creates one resource per item in a map or set, giving each a stable key so changes do not disturb the others.

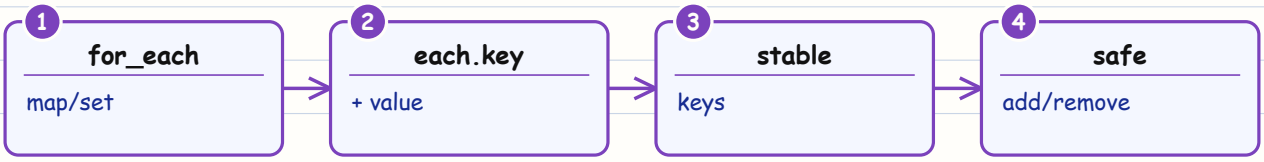
* Create Named Instances

- for_each iterates a map or set, creating one instance per key with each.key / each.value.
- Keys are stable, so adding/removing one item does not disturb the others.

* Prefer Over count

- For a set of distinct, named things (buckets, users, subnets), for_each is safer than count.

for_each



REAL-WORLD EXAMPLE

```
resource "aws_s3_bucket" "b" {        # provider-specific
  for_each = toset(["logs", "assets", "backups"])
  bucket   = "shop-${each.key}"
}
# aws_s3_bucket.b["assets"].arn
```

COMMON MISTAKE
Feeding for_each a value not known until apply (e.g. a generated id). for_each keys must be known at plan time. Use static keys or restructure the data.

REVISION SNAPSHOT

ASK YOURSELF	Why is for_each safer than count for named resources?
DO THIS	Create three buckets from a set and reference one by its key.

19. Conditionals & for Expressions

GrowthSchool

IN SIMPLE TERMS
Conditionals pick a value based on a test; for expressions transform or filter a whole list or map at once.

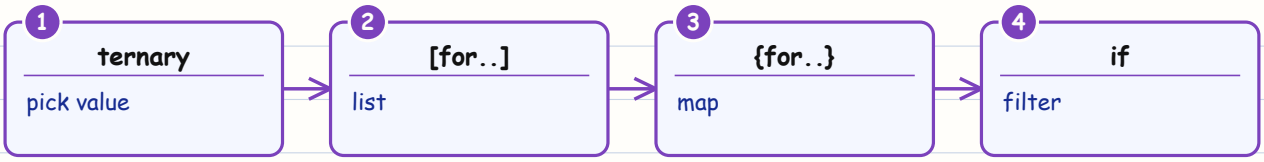
* Conditional Expression

- condition ? true_val : false_val chooses a value inline - great for env-based settings.

* for Expressions

- Transform collections: [for x in list : upper(x)] or {for k,v in map : k => v*2}.
- Filter with an if: [for x in list : x if x != ""].

for expressions



REAL-WORLD EXAMPLE

```
locals {
  tier      = var.env == "prod" ? "large" : "small"
  upper_azs = [for a in var.azs : upper(a)]
  enabled   = { for k,v in var.flags : k => v if v }
}
```

COMMON MISTAKE
Using [ ] for a for expression when you meant to build a map, or { } when you meant a list. Square brackets produce a list/tuple; braces with k => v produce a map/object. Match the shape.

STUDENT LAB
Write one list-comprehension and one map-comprehension over the same input, each with an if filter. Test both in terraform console and confirm the output shapes differ.

REVISION SNAPSHOT

ASK YOURSELF	How do you transform and filter a list in HCL?
DO THIS	Write a for expression that uppercases and filters a list.

20. dynamic Blocks

GrowthSchool

IN SIMPLE TERMS
A dynamic block generates repeated nested settings (like several firewall rules) from a list, instead of writing each by hand.

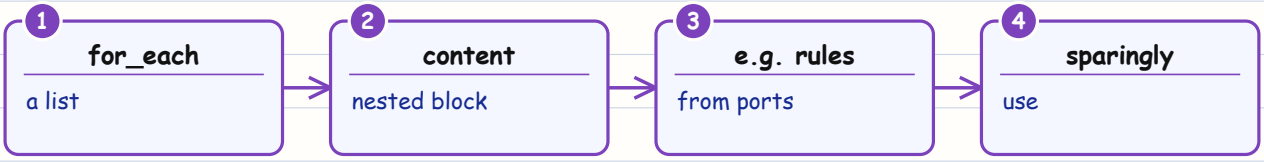
* Generate Nested Blocks

- A dynamic block builds repeated nested blocks (like ingress rules) from a collection.
- Use it when the number of nested blocks depends on a variable.

* Use Sparingly

- Dynamic blocks reduce readability. Reach for them only when static blocks truly cannot express it.

dynamic blocks



REAL-WORLD EXAMPLE

```
resource "aws_security_group" "api" {   # provider-specific
  dynamic "ingress" {
    for_each = var.allowed_ports
    content {
      from_port = ingress.value
      to_port   = ingress.value
      protocol  = "tcp"
    }
  }
}
```

COMMON MISTAKE
Wrapping everything in dynamic blocks until the config is unreadable. If the count is fixed, write the blocks out plainly - clarity beats cleverness.

REVISION SNAPSHOT

ASK YOURSELF	When is a dynamic block the right tool?
DO THIS	Generate security-group rules dynamically from a list of ports.

21. Modules: Basics

GrowthSchool

IN SIMPLE TERMS
A module is a reusable folder of Terraform files you can call with inputs and read outputs from, like a function for infrastructure.

* Reuse Infrastructure

- A module is a folder of .tf files you can call with inputs and read outputs from.
- The top-level config is the root module; it can call child modules to compose infrastructure.

* Call a Module

- Use a module block with source (local path or registry) and pass variables as arguments.

ROOT MODULE	CHILD MODULE
main.tf variables.tf outputs.tf backend.tf calls child modules one per environment	reusable component e.g. modules/network inputs (variables) + outputs versioned + shared

REAL-WORLD EXAMPLE

```
module "network" {
  source     = "./modules/network"
  cidr_block = "10.0.0.0/16"
  env        = var.env
}
# use: module.network.vpc_id
```

REVISION SNAPSHOT

ASK YOURSELF What is the difference between the root module and a child module?

DO THIS Extract a couple of resources into a local module and call it.

22. Module Structure & Registry

GrowthSchool

IN SIMPLE TERMS
A well-structured module has main, variables, and outputs files; the Registry lets you reuse community modules by version.

* Standard Layout

- A module contains main.tf, variables.tf and outputs.tf. Inputs = variables, results = outputs.
- Keep modules focused (one concern) and document their inputs/outputs.

* Sharing

- Pull reusable modules from the Terraform Registry with a versioned source; pin the version.

REAL-WORLD EXAMPLE

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "~> 5.0"          # pin the module version
  name    = "shop-vpc"
  cidr    = "10.0.0.0/16"
}
```

COMMON MISTAKE
Using a registry module without pinning version. A new release can change resources and trigger surprise replacements. Always set version and upgrade intentionally.

REVISION SNAPSHOT

ASK YOURSELF	What three files make up a well-structured module?
DO THIS	Consume a pinned registry module and read one of its outputs.

23. Project & Environment Layout

GrowthSchool

IN SIMPLE TERMS
A good project layout keeps each environment (dev, prod) separate with its own state, sharing logic through modules.

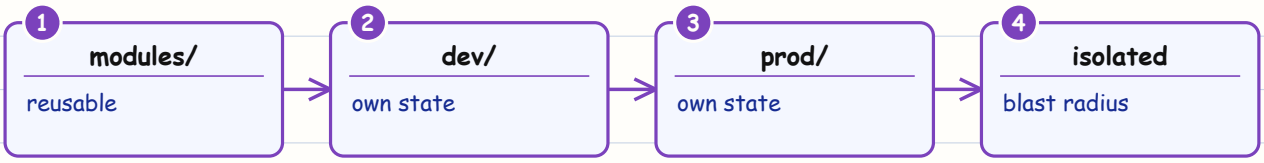
* Separate Environments

- Keep dev, staging and prod isolated with separate state so a mistake in dev cannot harm prod.
- Common pattern: environments/{dev,prod}/ each with its own backend, calling shared modules.

* DRY But Safe

- Share logic through modules; keep environment values in per-env tfvars. Avoid one giant state.

Environment layout



REAL-WORLD EXAMPLE

```
# layout
modules/           # reusable components
environments/dev/  main.tf  dev.tfvars  backend.tf
environments/prod/ main.tf  prod.tfvars backend.tf
# each env: terraform -chdir=environments/prod apply
```

COMMON MISTAKE
Putting all environments in one state. A single apply can then blast every environment at once. Isolate state per environment.

REVISION SNAPSHOT

ASK YOURSELF	Why keep separate state per environment?
DO THIS	Sketch a repo layout with shared modules and per-env configs.

24. Workspaces

GrowthSchool

IN SIMPLE TERMS
A workspace is a way to keep several separate copies of state for the same configuration, for lightweight variants.

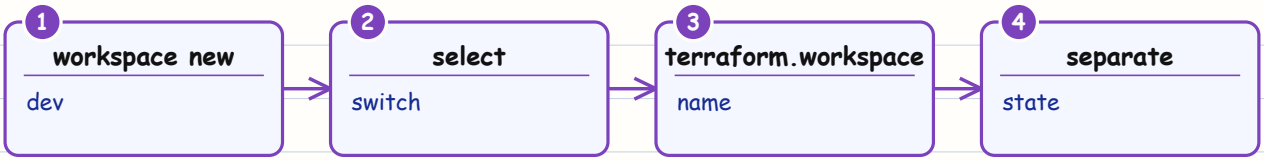
* What They Are

- CLI workspaces keep multiple state instances for the same config (terraform workspace ...).
- Reference the current one via terraform.workspace to vary names/sizes.

* Use With Care

- Workspaces suit lightweight variants; for strongly isolated prod, prefer separate configs/backends.

Workspaces



REAL-WORLD EXAMPLE

```
terraform workspace new dev
terraform workspace select dev
terraform workspace list
# locals { name = "shop-${terraform.workspace}" }
```

COMMON MISTAKE
Using a single set of workspaces as the only separation between dev and prod. They share config and backend; a blast-radius mistake is easy. Prefer separate configs for prod.

REVISION SNAPSHOT

ASK YOURSELF	When are workspaces appropriate vs separate configurations?
DO THIS	Create two workspaces and vary a resource name by workspace.

25. Dependencies & the Graph

GrowthSchool

IN SIMPLE TERMS
Terraform builds a dependency graph from how resources reference each other, so it creates things in the right order automatically.

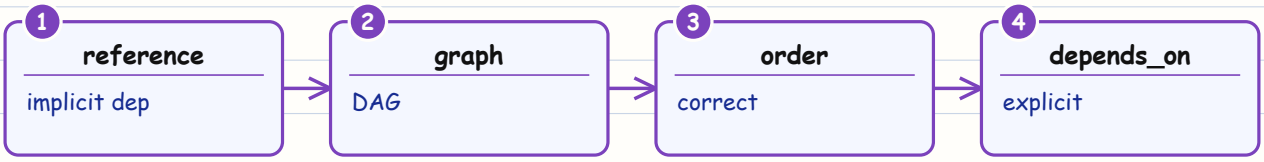
* Implicit First

- Referencing another resource's attribute creates an implicit, correctly-ordered dependency.
- Terraform builds a DAG and parallelises independent work automatically.

* Explicit When Needed

- depends_on forces ordering when there is a hidden dependency Terraform cannot infer from references.

Dependency graph



REAL-WORLD EXAMPLE

```
resource "aws_iam_role_policy_attachment" "a" {  # provider-specific
  # ...
  depends_on = [aws_iam_role.api]   # explicit ordering
}
terraform graph | dot -Tpng > graph.png   # visualise
```

COMMON MISTAKE
Sprinkling depends_on everywhere to "be safe". It hides real relationships and slows applies. Let references create dependencies; use depends_on only for genuine hidden ones.

REVISION SNAPSHOT

ASK YOURSELF	When do you need explicit depends_on?
DO THIS	Generate a dependency graph and identify one implicit edge.

26. Lifecycle Rules

GrowthSchool

IN SIMPLE TERMS
Lifecycle rules change how Terraform handles a resource on updates - for example, creating a replacement before destroying the old one.

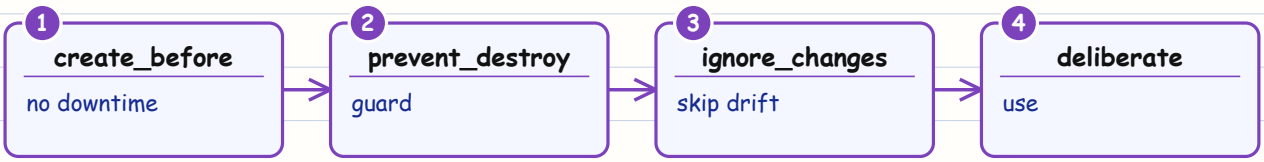
* Control Change Behaviour

- create_before_destroy avoids downtime by making the new resource before removing the old.
- prevent_destroy blocks accidental deletion; ignore_changes ignores drift on chosen attributes.

* Use Deliberately

- These change how apply behaves on sensitive resources like databases and load balancers.

Lifecycle rules



REAL-WORLD EXAMPLE

```
resource "aws_db_instance" "shop" {   # provider-specific
  # ...
  lifecycle {
    prevent_destroy = true
    ignore_changes  = [tags["LastSeen"]]
  }
}
```

COMMON MISTAKE
Leaving prevent_destroy off a production database. One accidental config change that forces replacement can then destroy it. Guard critical stateful resources.

REVISION SNAPSHOT

ASK YOURSELF	What does create_before_destroy prevent, and when use prevent_destroy?
DO THIS	Add prevent_destroy to a critical resource and try to destroy it.

27. Provisioners (Avoid When Possible)

GrowthSchool

IN SIMPLE TERMS
A provisioner runs a script during create or destroy; it is an escape hatch to avoid when a cleaner, declarative option exists.

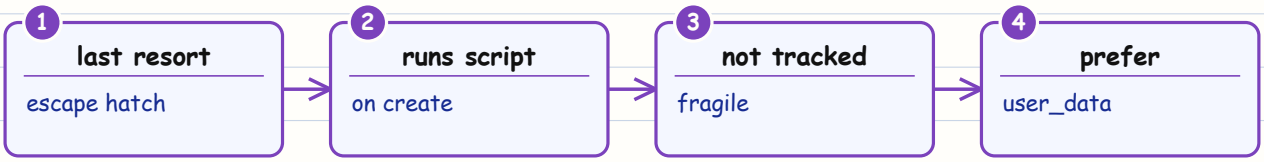
* What They Are

- Provisioners run scripts on create/destroy (local-exec, remote-exec). They are an escape hatch.
- They break the declarative model and are not tracked in state like resources are.

* Prefer Alternatives

- Use cloud-init/user-data, images baked with Packer, or config management instead.
- If you must use one, understand it may not re-run and can leave partial state.

Provisioners



REAL-WORLD EXAMPLE

```
resource "aws_instance" "api" {   # provider-specific
  # prefer user_data over provisioners:
  user_data = file("cloud-init.yaml")
  # last-resort escape hatch:
  # provisioner "local-exec" { command = "echo done" }
}
```

COMMON MISTAKE
Relying on remote-exec provisioners for configuration. They run once, are fragile, and are not reconciled. Bake images or use user_data / config management instead.

REVISION SNAPSHOT

ASK YOURSELF	Why are provisioners a last resort, and what replaces them?
DO THIS	Replace a hypothetical remote-exec setup step with user_data.

28. Import & State Commands

GrowthSchool

IN SIMPLE TERMS
Importing brings an already-existing resource under Terraform's management; state commands let you safely move or remove tracked resources.

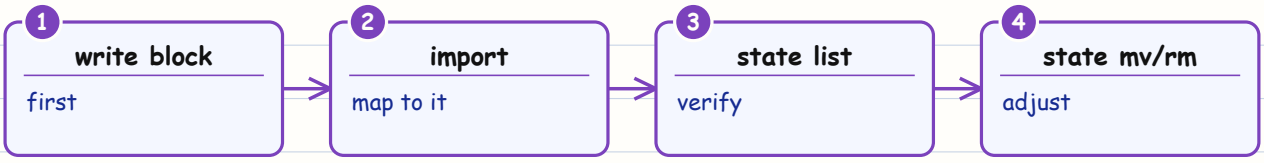
* Adopt Existing Resources

- terraform import brings a manually-created resource under management by mapping it to a config block.
- Newer Terraform also supports import blocks for a plan-reviewed import.

* State Surgery

- state mv renames/moves; state rm stops managing (without deleting); state show inspects.

Import existing



REAL-WORLD EXAMPLE

```
terraform import aws_s3_bucket.assets shop-api-assets-123
terraform state list
terraform state mv aws_s3_bucket.old aws_s3_bucket.assets
terraform state rm aws_s3_bucket.legacy   # stop managing
```

COMMON MISTAKE
Importing without first writing a matching resource block. Import maps to config; with no config, the next plan wants to destroy it. Write the block first, then import.

REVISION SNAPSHOT

ASK YOURSELF	What must exist in config before you terraform import a resource?
DO THIS	Import an existing resource and confirm plan shows no changes.

29. Terraform With a Cloud

GrowthSchool

IN SIMPLE TERMS
Using Terraform with a cloud means the same workflow applies everywhere - only the provider and resource names change per cloud.

* Provider-Specific Reality

- The workflow (init/plan/apply) is identical across clouds; only the provider + resources differ.
- Authenticate via the provider's recommended method (env vars, CLI profile, or CI OIDC role).

* Shop API Example (AWS)

- A minimal stack: an S3 bucket for assets and a security group - everything below is AWS-specific.

PROVIDER-SPECIFIC
Resource names, arguments and auth differ per cloud (AWS/GCP/Azure). The init -> plan -> apply workflow and HCL structure stay the same everywhere.

REAL-WORLD EXAMPLE

```
# ALL PROVIDER-SPECIFIC (AWS)
provider "aws" { region = "us-east-1" }
resource "aws_s3_bucket" "assets" { bucket = "shop-assets-123" }
resource "aws_security_group" "api" {
  name = "shop-api-sg"
  ingress { from_port=443 to_port=443 protocol="tcp"
            cidr_blocks=["0.0.0.0/0"] }
}
```

REVISION SNAPSHOT

ASK YOURSELF	What stays the same and what changes when you switch clouds?
DO THIS	Authenticate to a free-tier cloud and apply one tiny resource.

30. Secrets & Sensitive Values

GrowthSchool

IN SIMPLE TERMS
Sensitive values are secrets you mark so Terraform hides them in output; the real protection is securing the state and using a secrets store.

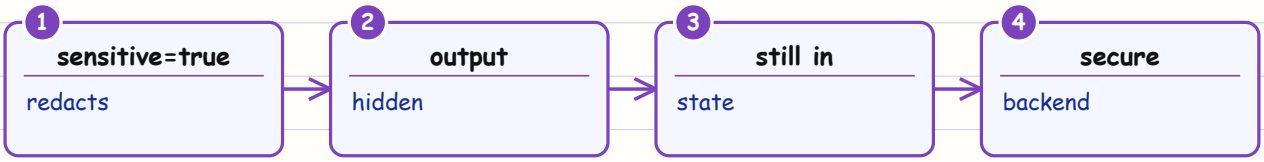
* Mark and Protect

- Set sensitive = true on variables/outputs so Terraform redacts them in plan/apply output.
- Redaction hides them in logs, but they still live in state - protect the backend.

* Source Secrets Well

- Inject via environment (TF_VAR_) or a secrets manager; do not commit secret tfvars.

Sensitive values



REAL-WORLD EXAMPLE

```
variable "db_password" {
  type      = string
  sensitive = true
}
output "db_endpoint" { value = aws_db_instance.shop.address }
# export TF_VAR_db_password=... (from a secrets manager)
```

COMMON MISTAKE
Assuming sensitive = true encrypts the value. It only hides it in CLI output; the value is still plaintext in state. Secure the backend and restrict access.

REVISION SNAPSHOT

ASK YOURSELF	What does sensitive = true actually do (and not do)?
DO THIS	Mark a variable sensitive and confirm it is redacted in plan output.

31. CI/CD & Plan Review

GrowthSchool

IN SIMPLE TERMS
A Terraform pipeline runs checks and a plan on each pull request for review, then applies the change automatically when it is merged.

* Automate Safely

- On a PR: run fmt + validate, then terraform plan and post the diff for human review.
- On merge to main: run apply (often with a required approval) using a saved plan.

* Guardrails

- Use remote state + locking, least-privilege CI credentials (OIDC), and policy checks (OPA/Sentinel).

Terraform CI/CD: plan on PR, apply on merge

REAL-WORLD EXAMPLE

```
# CI steps
terraform fmt -check
terraform validate
terraform plan -out=tfplan   # post summary to the PR
# on merge:
terraform apply tfplan
```

COMMON MISTAKE
Letting CI auto-apply on every push with no plan review. Infra changes deserve the same review as code. Plan on PR, apply on merge with approval.

REVISION SNAPSHOT

ASK YOURSELF	What runs on a PR vs on merge in a Terraform pipeline?
DO THIS	Describe a pipeline that plans on PR and applies on merge.

32. Drift Detection

GrowthSchool

IN SIMPLE TERMS
Drift is when the real infrastructure no longer matches your code (usually from manual changes); plan detects it so you can reconcile.

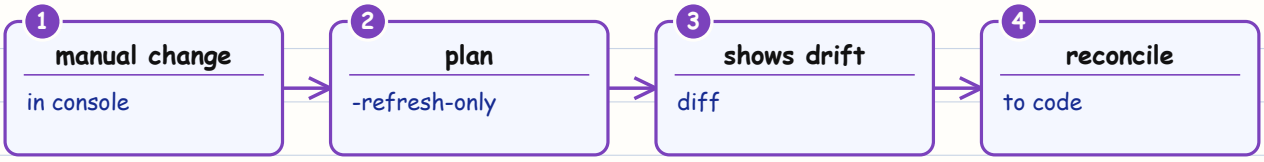
* What Is Drift

- Drift is when real infrastructure differs from state - usually from manual console changes.
- terraform plan detects drift by refreshing and comparing state to reality.

* Respond to Drift

- Decide: reconcile by applying config, or import/adjust config to match an intentional change.
- Prevent drift by managing resources only through Terraform.

Drift detection



REAL-WORLD EXAMPLE

```
terraform plan -refresh-only    # show drift without changes
terraform apply -refresh-only   # update state to match reality
terraform plan                  # then reconcile to config
```

COMMON MISTAKE
Ignoring drift until an apply suddenly wants to undo someone's emergency manual fix. Detect drift regularly and bring changes back into code promptly.

REVISION SNAPSHOT

ASK YOURSELF	What causes drift and how does plan reveal it?
DO THIS	Manually tag a resource in the console, then detect the drift with plan.

33. Troubleshooting

GrowthSchool

IN SIMPLE TERMS
Troubleshooting is the calm way to fix common Terraform issues like a stuck state lock or a value not known until apply.

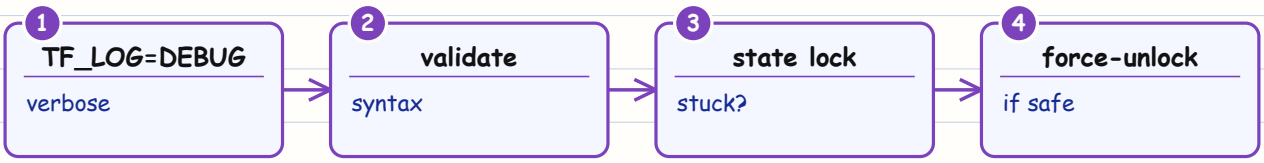
* Common Issues

- State lock stuck: a crashed apply left a lock; investigate, then force-unlock only if truly safe.
- for_each/count value not known at plan time: restructure to use plan-time-known keys.

* Debugging

- Set TF_LOG=DEBUG for verbose provider logs; use terraform validate and console to isolate issues.
- Read the error carefully - Terraform errors usually name the exact resource and attribute.

Troubleshoot



REAL-WORLD EXAMPLE

```
TF_LOG=DEBUG terraform apply
terraform force-unlock <LOCK_ID>   # only when safe
terraform validate
terraform state list | grep <name>
```

COMMON MISTAKE
Running force-unlock reflexively. If another apply is genuinely running, unlocking corrupts state. Confirm no apply is in progress first.

REVISION SNAPSHOT

ASK YOURSELF	When is terraform force-unlock safe to run?
DO THIS	Enable TF_LOG=DEBUG and read where a failing apply spends its time.

34. Best Practices

GrowthSchool

IN SIMPLE TERMS
Best practices are the proven habits - pinned versions, remote locked state, small blast radius, reviewed plans - that keep infra safe.

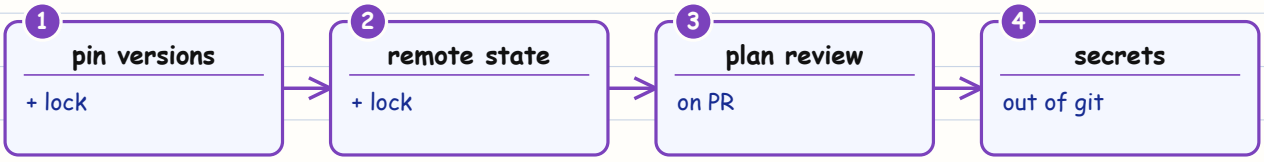
* Code & State

- Pin provider + module versions; commit the lock file; remote state with locking; small blast radius.
- fmt + validate in CI; plan on PR, apply on merge; least-privilege credentials.

* Design

- Use modules for reuse; for_each over count for named sets; keep secrets out of Git; guard stateful resources.

Best practices



CHECKLIST
pinned versions - committed lock file - remote state + locking - plan reviewed - secrets out of git - modules for reuse - prevent_destroy on critical data.

COMMON MISTAKE
Treating infra code more loosely than app code. It deploys real, costly, sometimes irreversible resources. Review, test and version it at least as carefully.

REVISION SNAPSHOT

ASK YOURSELF	Recite the Terraform best-practice checklist.
DO THIS	Audit one of your configs against every checklist item.

35. Terraform Cheat Sheet

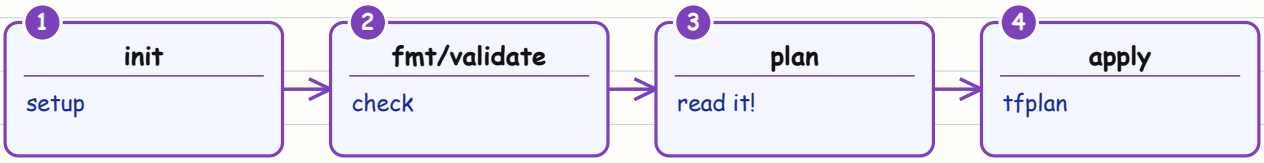
GrowthSchool

IN SIMPLE TERMS
This is a quick-reference list of the Terraform commands you will use every day.

* Daily Drivers

- init, fmt, validate, plan -out, apply, destroy, output, state list/show/mv/rm, import, console.

Everyday Terraform



SAFE LOOP
fmt -> validate -> plan (read it!) -> apply. Never skip reading the plan, and never hand-edit state. These two habits prevent most Terraform disasters.

ONE-PAGE COMMAND REFERENCE

```
terraform init            terraform state list
terraform fmt -recursive  terraform state show <addr>
terraform validate        terraform state mv A B
terraform plan -out=tf    terraform state rm <addr>
terraform apply tf        terraform import <addr> <id>
terraform destroy         terraform output -json
terraform console         terraform workspace new/select
terraform graph           TF_LOG=DEBUG terraform apply
```

COMMON MISTAKE
Skipping plan review to save time. The plan is the one moment to catch a destructive replace before it happens. Always read every +/~/-/+ line.

REVISION SNAPSHOT

ASK YOURSELF	What is the safe fmt -> validate -> plan -> apply loop?
DO THIS	Recall ten Terraform commands from memory grouped by purpose.

36. Capstone Challenge

GrowthSchool

IN SIMPLE TERMS
This capstone ties everything together by provisioning the Shop API's infrastructure the professional way.

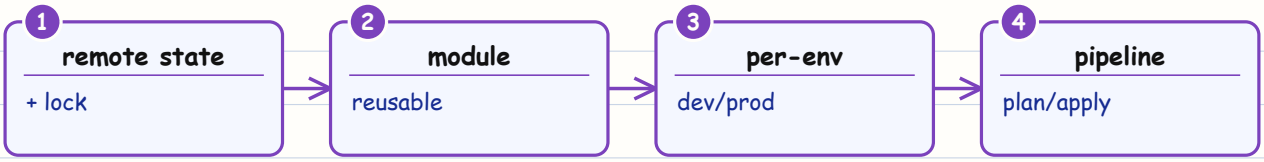
* Provision Shop API Infra

- Write a config with a remote backend + locking, typed variables, and a reusable network module.
- Provision a small stack (e.g. a bucket + security group) with common tags via a local.

* Operate It Properly

- Add outputs for key values, mark secrets sensitive, and separate dev/prod state.
- Wire a CI pipeline that plans on PR and applies on merge.

Capstone steps



REAL-WORLD EXAMPLE

```
terraform init
terraform workspace new dev   # or per-env config
terraform plan -out=tfplan
terraform apply tfplan
terraform output -json
```

STUDENT LAB
Deliverables: remote state + locking, a reusable module, per-env separation, sensitive-marked secrets, outputs for key values, and a plan-on-PR / apply-on-merge pipeline.

REVISION SNAPSHOT

ASK YOURSELF	Is your state remote+locked, modular, and environment-separated?
DO THIS	Complete every capstone deliverable and destroy cleanly at the end.

37. Interview & Revision Notes

GrowthSchool

IN SIMPLE TERMS
This page gathers the key ideas and common interview questions so you can revise quickly.

* Explain Clearly

- Declarative vs imperative; what state is and why remote+locking matters; count vs for_each.
- How plan/apply work, how drift happens, and why you never hand-edit state.

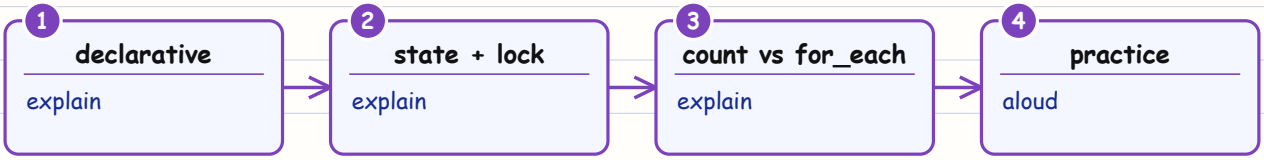
* Common Questions

- How do you manage secrets? How do you structure environments? What does a plan's -/+ mean?
- When do you use modules, data sources, lifecycle rules, and depends_on?

* Your Next Step

- Build the capstone against a free-tier cloud, break it deliberately, detect drift, and recover.

Revise out loud



REAL-WORLD EXAMPLE

```
# 30-second self test
terraform plan          # can you read every +/~/-/+ ?
terraform state list    # what is managed?
# count vs for_each? remote state + locking? drift?
```

REVISION SNAPSHOT

ASK YOURSELF	Can you explain state and remote locking in 30 seconds?
DO THIS	Answer every "Common Question" above out loud without notes.

GrowthSchool

AWS

COMPLETE NOTES

LEARN - BUILD - OPERATE

Core cloud services every DevOps engineer must know

growthschool.cc | @growthschool.cc

Learning Roadmap

GrowthSchool

Learn AWS for DevOps in order. Every page opens with a plain-English definition, then explains how it works with a simple numbered architecture diagram, real commands, common mistakes, a lab and a revision snapshot. We host an ecommerce "Shop API" on AWS throughout. Free-tier resources are enough to practise most of this.

HOW TO USE THESE NOTES

Create a free-tier account, set a billing alarm FIRST, and never commit access keys.
Every page has a diagram - trace the numbered steps 1 -> 2 -> 3 to see how it works.
Animated versions of the key flows are in output/animations (open the .gif files).

1. Why the Cloud & AWS

GrowthSchool

IN SIMPLE TERMS
AWS (Amazon Web Services) is a cloud provider: you rent computing, storage, and networking over the internet and pay only for what you use, instead of buying your own servers.

* Rent, Do Not Buy

- Instead of buying servers, you rent them by the second and give them back when done.
- You scale up in minutes for a sale and scale down at night - paying only for what you use.

* Shared Responsibility

- AWS secures the cloud itself; you secure what you put in it (data, access, config).
- Most incidents are customer misconfiguration, like an open S3 bucket - not AWS failing.

AWS SECURES (of the cloud)	YOU SECURE (in the cloud)
physical data centres	your data + who can access it
hardware + network	IAM users, roles, policies
the virtualization layer	security groups + encryption
managed service internals	patching what you install

REAL-WORLD EXAMPLE

```
aws sts get-caller-identity   # who am I / which account?
aws ec2 describe-regions --query "Regions[].RegionName"
```

REVISION SNAPSHOT

ASK YOURSELF	Under shared responsibility, who secures the data and access?
DO THIS	Run get-caller-identity and note your account id and identity.

2. Global Infrastructure

GrowthSchool

IN SIMPLE TERMS
AWS runs data centres worldwide grouped into Regions; each Region has several isolated Availability Zones, and edge locations sit close to users.

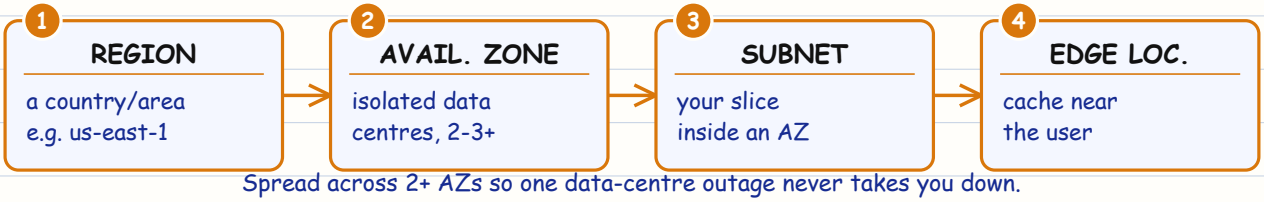
* Regions & AZs

- A Region is a location (like us-east-1); an Availability Zone is an isolated data centre in it.
- Running across 2+ AZs means one data-centre failure does not take your whole app down.

* Pick a Region

- Choose by closeness to users, data-residency laws, which services are offered, and price.

Zoom in: Region -> AZ -> Subnet -> Edge



Spread across 2+ AZs so one data-centre outage never takes you down.

REAL-WORLD EXAMPLE

```
aws ec2 describe-availability-zones --region us-east-1
# design rule: always spread across >= 2 AZs
```

REVISION SNAPSHOT

ASK YOURSELF	Why deploy across two Availability Zones?
DO THIS	List the AZs in your Region and say why you would use two.

3. Accounts, Billing & the CLI

GrowthSchool

IN SIMPLE TERMS
This covers setting up your AWS account safely, watching your bill, and the tools (console, CLI) you use to control everything.

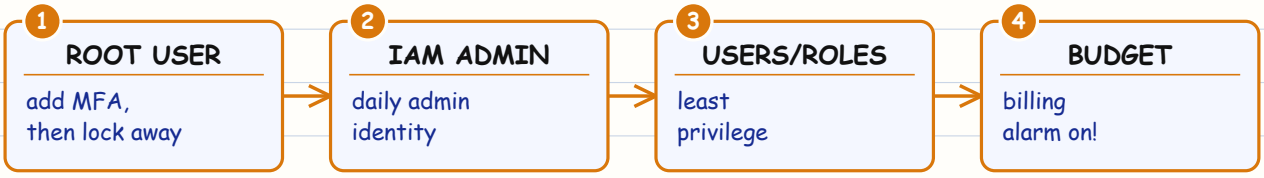
* Protect the Root User

- The root user can do anything - turn on MFA, then never use it for daily work.
- Make a separate IAM admin identity for yourself and lock the root credentials away.

* Control Cost From Day One

- Set a Budget and a billing alarm immediately so a mistake never becomes a shock bill.
- Tag resources (project, env, owner) so you can see and cut cost later.

Set up a new account safely (do this first)



REAL-WORLD EXAMPLE

```
aws configure          # key, secret, region, output
aws budgets describe-budgets --account-id <id>
# prefer SSO / roles over long-lived access keys
```

COMMON MISTAKE
Using the root user day to day. If those keys leak, the whole account is gone. MFA-protect root, make an admin user, and avoid long-lived keys.

REVISION SNAPSHOT

ASK YOURSELF	Why never use the root user for everyday tasks?
DO THIS	Configure the CLI, set a budget, and enable MFA on root.

4. IAM: Identities & Policies

GrowthSchool

IN SIMPLE TERMS
IAM (Identity and Access Management) controls who can do what in your account, using users, groups, roles, and permission policies.

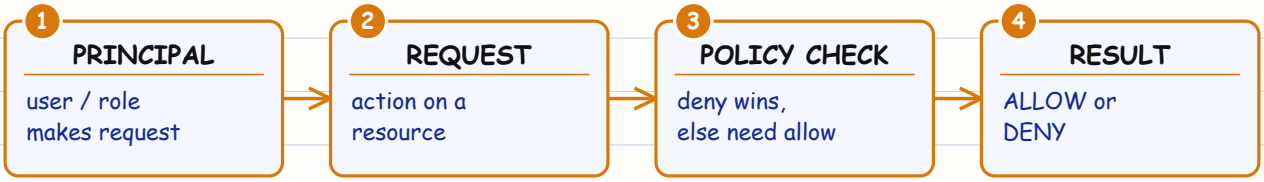
* Who Can Do What

- A user is a person/app, a group bundles users, and a role is temporary access that is assumed.
- A policy is a small JSON rule listing allowed (or denied) actions on which resources.

* How It Decides

- Everything is denied by default. You need an explicit Allow, and any explicit Deny always wins.

How an IAM permission decision is made



REAL-WORLD EXAMPLE

```
{ "Effect": "Allow",
  "Action": ["s3:GetObject"],
  "Resource": "arn:aws:s3:::shop-assets/*" }
```

COMMON MISTAKE
Attaching AdministratorAccess to make something work. Start with the few actions needed; broad permissions are the number-one cause of cloud breaches.

REVISION SNAPSHOT

ASK YOURSELF	If an Allow and a Deny both match, which wins?
DO THIS	Write a policy allowing read-only access to one S3 bucket.

5. IAM Best Practices

GrowthSchool

IN SIMPLE TERMS
IAM best practices are the safe habits - least privilege, roles instead of keys, and MFA - that keep your account from being hacked.

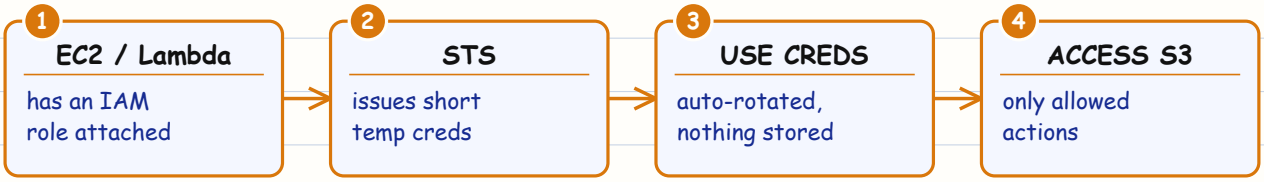
* Prefer Roles Over Keys

- Give EC2/Lambda a role instead of storing keys on the box - it gets short-lived auto-rotated creds.
- There is then nothing to leak, commit to Git, or rotate by hand.

* Least Privilege + MFA

- Grant only what is needed, require MFA for sensitive actions, and review unused permissions.

Roles give temporary keys - nothing to leak



REAL-WORLD EXAMPLE

```
aws iam create-role --role-name shop-api-role \
  --assume-role-policy-document file://trust.json
# app uses the role - no access keys anywhere
```

COMMON MISTAKE
Baking access keys into code or images. They leak via Git and logs. Use roles for compute and a secrets store for the rest.

REVISION SNAPSHOT

ASK YOURSELF	Why are roles safer than access keys for EC2 and Lambda?
DO THIS	Create a narrow role and attach it to a compute resource.

6. VPC Fundamentals

GrowthSchool

IN SIMPLE TERMS
A VPC (Virtual Private Cloud) is your own private network inside AWS where your servers and databases live.

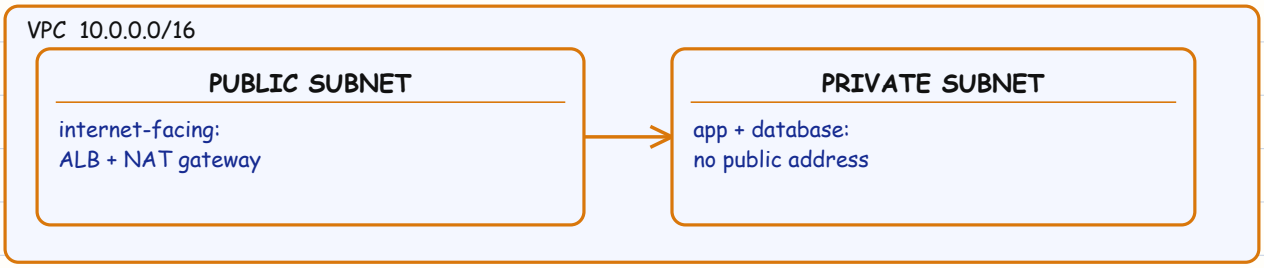
* Your Private Network

- A VPC is an isolated network you define with an address range like 10.0.0.0/16.
- Everything - servers, load balancers, databases - lives inside a VPC in one Region.

* Public vs Private

- Public subnets hold internet-facing bits (load balancer, NAT); private subnets hold app + DB.
- Keep databases in private subnets with no route straight to the internet.

A VPC: your private network, split into public + private subnets



REAL-WORLD EXAMPLE

```
aws ec2 create-vpc --cidr-block 10.0.0.0/16
aws ec2 create-subnet --vpc-id vpc-123 --cidr-block 10.0.1.0/24
```

REVISION SNAPSHOT

ASK YOURSELF What makes a subnet "public" versus "private"?

DO THIS Sketch a VPC with one public and one private subnet across two AZs.

7. Subnets, Routing, IGW & NAT

GrowthSchool

IN SIMPLE TERMS
Subnets split your VPC into sections; route tables, an internet gateway, and a NAT gateway decide what can reach the internet and how.

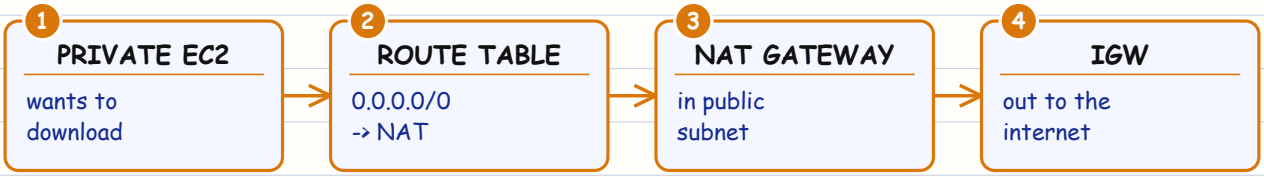
* Route Tables

- Each subnet follows a route table. Public subnets send internet traffic to an Internet Gateway.
- Private subnets send outbound traffic to a NAT gateway so they can reach out but not be reached.

* IGW vs NAT - the key difference

- IGW is TWO-WAY: a public-subnet EC2 with a public IP can be reached FROM the internet (inbound) and reach out.
- NAT is OUTBOUND-ONLY: a private-subnet EC2 can start connections out, but the internet cannot start one IN to it.

Outbound path from a private server (one-way)



INBOUND EXAMPLE: public vs private EC2
Public subnet EC2: a user opens http://<public-ip>; the request enters via the IGW and the web server answers - inbound works (if the security group allows port 80).
Private subnet EC2: that same request never arrives. NAT only passes REPLIES to connections the
instance itself started, so it drops unsolicited inbound. To expose it, put a load balancer in the public subnet and forward traffic to the private instance.

REAL-WORLD EXAMPLE

```
aws ec2 create-internet-gateway            # inbound + outbound
aws ec2 create-nat-gateway --subnet-id subnet-pub \
  --allocation-id eipalloc-1               # outbound only
```

REVISION SNAPSHOT

ASK YOURSELF	Can the internet start an inbound connection to a private-subnet EC2 behind a NAT?
DO THIS	Trace an inbound web request to a public EC2, then explain why it fails for a private one.

8. Security Groups vs NACLs

GrowthSchool

IN SIMPLE TERMS
Security Groups are per-server firewalls and NACLs are per-subnet firewalls; together they control which traffic is allowed.

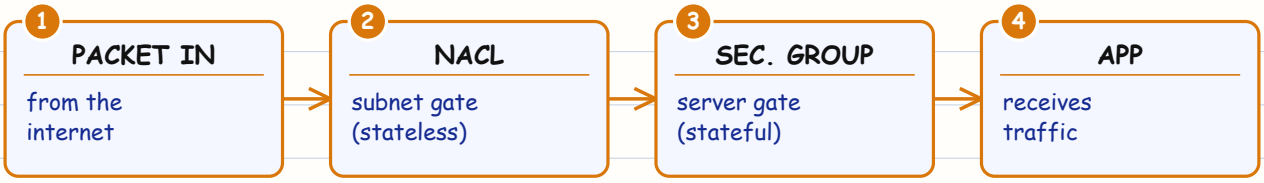
* Security Groups

- A firewall on the server. Stateful: allow traffic in and the reply is automatically allowed back.
- Allow rules only, and they can reference other groups (e.g. "allow from the load balancer").

* Network ACLs

- A firewall on the subnet. Stateless: you must allow both directions. Supports allow and deny.

A packet passes the subnet NACL, then the server SG



REAL-WORLD EXAMPLE

```
aws ec2 authorize-security-group-ingress --group-id sg-api \
  --protocol tcp --port 443 --source-group sg-alb
```

COMMON MISTAKE
Opening SSH (port 22) to 0.0.0.0/0. Bots attack it constantly. Restrict to your IP or use SSM Session Manager instead of open SSH.

REVISION SNAPSHOT

ASK YOURSELF	Why is a Security Group "stateful" but a NACL "stateless"?
DO THIS	Write an SG that only accepts traffic from the load balancer's SG.

9. EC2 Basics

GrowthSchool

IN SIMPLE TERMS
EC2 gives you virtual servers in the cloud that you can start, stop, and resize on demand.

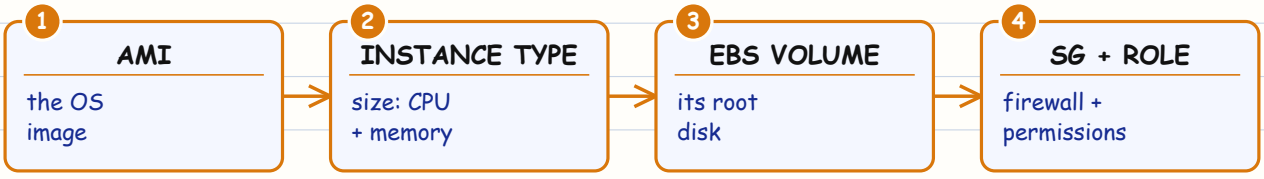
* Virtual Servers

- EC2 gives you virtual machines. You pick an image, a size, a subnet, a firewall and a role.
- It boots in your VPC and you can reach it, resize it, stop it, or throw it away.

* Stop vs Terminate

- Stop pauses compute billing and keeps the disk; terminate deletes the instance for good.

What makes up one EC2 instance



REAL-WORLD EXAMPLE

```
aws ec2 run-instances --image-id ami-123 --instance-type t3.micro \
  --subnet-id subnet-priv --security-group-ids sg-api --key-name shop
```

COMMON MISTAKE
Terminating when you meant to stop and losing the server. Stop pauses billing; terminate deletes - know which you want.

REVISION SNAPSHOT

ASK YOURSELF	What is the difference between stopping and terminating?
DO THIS	Launch a free-tier t3.micro, stop it, restart it, then terminate it.

10. EC2 Images, Types & User Data

GrowthSchool

IN SIMPLE TERMS
An AMI is the image an EC2 server boots from; instance types set its size, and user-data is a script that runs on first boot.

* AMI & Instance Type

- An AMI is the template a server boots from; bake "golden" AMIs for fast, consistent starts.
- Instance families fit jobs: t (burstable), m (general), c (compute), r (memory), g (GPU).

* User Data

- A first-boot script that installs and starts your app. Prefer baked images for anything complex.

EC2 first-boot sequence

REAL-WORLD EXAMPLE

```
#!/bin/bash   # passed as --user-data
yum install -y docker && systemctl enable --now docker
docker run -d -p 80:8080 <registry>/shop-api:1.0
```

REVISION SNAPSHOT

ASK YOURSELF	What does EC2 user-data do and when does it run?
DO THIS	Launch an instance with user-data that starts a web server.

11. EBS, EFS & Instance Store

GrowthSchool

IN SIMPLE TERMS
EBS is a virtual disk for one server, EFS is shared file storage for many, and instance store is fast but temporary scratch space.

* Three Storage Types

- EBS: a durable disk for one server, survives stop/start. EFS: shared files for many servers.
- Instance store: very fast but wiped when the instance stops - only for scratch data.

* Back It Up

- Snapshot EBS to S3 for backups. Snapshots are incremental, so they are cheap after the first.

Three storage types - pick by the job

REAL-WORLD EXAMPLE

```
aws ec2 create-volume --size 20 --availability-zone us-east-1a --volume-type gp3
aws ec2 create-snapshot --volume-id vol-1 --description "shop backup"
```

COMMON MISTAKE
Putting important data on instance store. It vanishes on stop or hardware failure. Use EBS or EFS for anything that must survive.

REVISION SNAPSHOT

ASK YOURSELF	When do you pick EBS vs EFS vs instance store?
DO THIS	Create a gp3 volume, attach it, and snapshot it.

12. Load Balancing (ELB)

GrowthSchool

IN SIMPLE TERMS
A load balancer spreads incoming traffic across many healthy servers so no single one is overwhelmed.

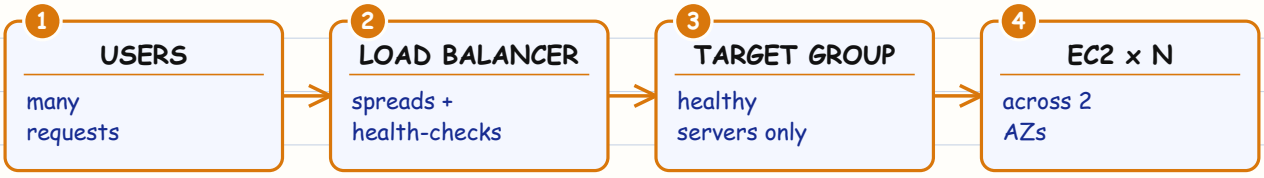
* Spread the Traffic

- A load balancer takes incoming requests and shares them across many healthy servers.
- ALB works at the web layer (routes by host/path); NLB works at the network layer (fast TCP).

* Health Checks

- It probes each target and stops sending traffic to unhealthy ones automatically.

A load balancer sends traffic only to healthy servers



REAL-WORLD EXAMPLE

```
aws elbv2 create-load-balancer --name shop-alb --type application \
  --subnets subnet-a subnet-b
# target group health check path: /health
```

COMMON MISTAKE
A wrong health-check path, so the balancer keeps sending users to broken servers. Point it at a real endpoint like /health.

REVISION SNAPSHOT

ASK YOURSELF	When would you choose an ALB versus an NLB?
DO THIS	Create an ALB with a target group whose health check hits /health.

13. Auto Scaling Groups

GrowthSchool

IN SIMPLE TERMS
An Auto Scaling Group automatically adds or removes servers to match demand and replaces any that become unhealthy.

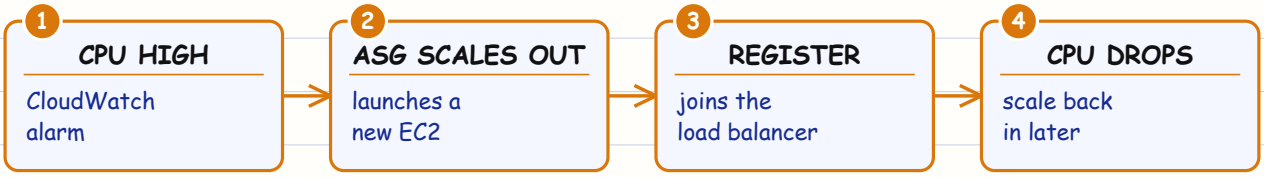
* Elastic Capacity

- An ASG keeps a target number of servers between a min and max, replacing unhealthy ones.
- It uses a launch template (image, size, firewall) and spreads servers across AZs.

* Scaling Policy

- Target tracking is simplest: "keep average CPU near 50%" and AWS adds/removes servers for you.

Auto Scaling reacts to load automatically



REAL-WORLD EXAMPLE

```
aws autoscaling create-auto-scaling-group --auto-scaling-group-name shop-asg \
  --launch-template LaunchTemplateName=shop-lt --min-size 2 --max-size 10 \
  --desired-capacity 2 --vpc-zone-identifier "subnet-a,subnet-b"
```

COMMON MISTAKE
Setting min equal to max, so it can never scale. Set a low min for normal load and a higher max for peaks, then let target tracking adjust.

REVISION SNAPSHOT

ASK YOURSELF	What three numbers define an ASG?
DO THIS	Create an ASG across two AZs with a CPU target-tracking policy.

14. S3 Object Storage

GrowthSchool

IN SIMPLE TERMS
S3 is object storage: it holds files (objects) in buckets with almost unlimited scale - great for assets, backups, and logs.

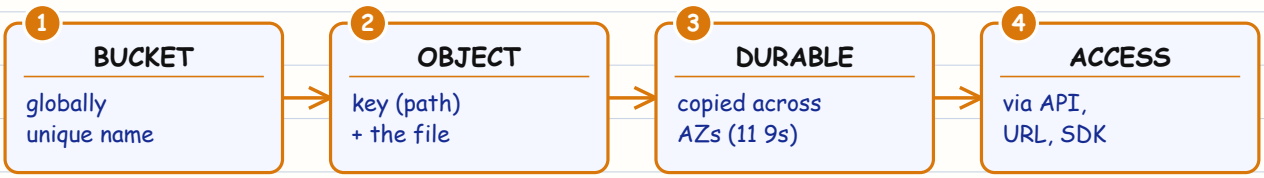
* Buckets & Objects

- S3 stores files (objects) in buckets. A bucket name is unique across all of AWS.
- Each object has a key (its path-like name), the data, and metadata. It is famously durable.

* Best For

- Static assets, backups, logs, and data lakes - not a live filesystem for random edits.

S3: put files (objects) into buckets



REAL-WORLD EXAMPLE

```
aws s3 mb s3://shop-assets-123
aws s3 cp ./logo.png s3://shop-assets-123/img/logo.png
aws s3 sync ./dist s3://shop-assets-123/site
```

COMMON MISTAKE
Trying to use S3 like a mounted disk for constant small writes. Upload/replace whole objects; for a real filesystem use EFS or EBS.

REVISION SNAPSHOT

ASK YOURSELF	Why must a bucket name be globally unique?
DO THIS	Create a bucket and sync a local folder into it.

15. S3 Classes & Lifecycle

GrowthSchool

IN SIMPLE TERMS
S3 storage classes offer cheaper tiers for less-used data; lifecycle rules move or delete objects automatically as they age.

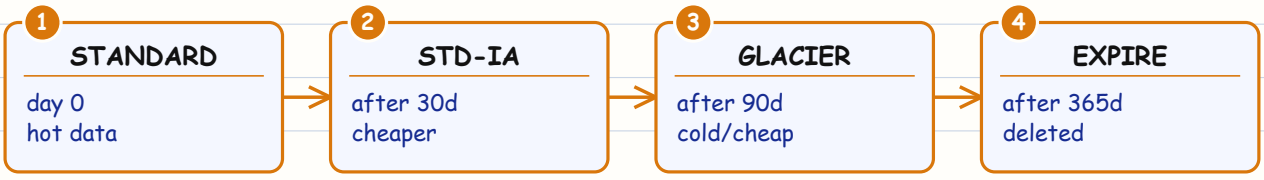
* Storage Classes

- Standard is for hot data; Standard-IA and One Zone-IA are cheaper for rarely used data.
- Glacier tiers are cheapest for cold archives; Intelligent-Tiering moves data for you.

* Lifecycle Rules

- Automatically move objects to cheaper tiers as they age, and delete them after a set time.

Lifecycle: data auto-moves to cheaper tiers over time



REAL-WORLD EXAMPLE

```
aws s3api put-bucket-lifecycle-configuration \
  --bucket shop-assets-123 --lifecycle-configuration file://lifecycle.json
```

COMMON MISTAKE
Keeping everything in Standard forever and overpaying for old logs and backups. Add lifecycle rules to tier or delete old data automatically.

REVISION SNAPSHOT

ASK YOURSELF	How do lifecycle rules cut S3 cost?
DO THIS	Add a rule that moves objects to Glacier after 90 days.

16. S3 Security

GrowthSchool

IN SIMPLE TERMS
S3 security is keeping buckets private by default, controlling access with policies, and encrypting the data.

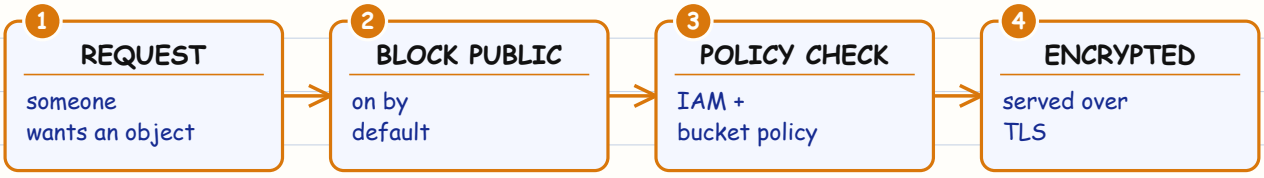
* Private By Default

- Keep Block Public Access ON unless you truly need public files. Default every bucket to private.
- Control access with IAM and bucket policies; share files with pre-signed URLs, not open buckets.

* Always Encrypt

- Turn on default encryption (SSE-S3 or SSE-KMS) and enforce HTTPS-only access.

Every S3 request is gated and encrypted



REAL-WORLD EXAMPLE

```
aws s3api put-public-access-block --bucket shop-assets-123 \
  --public-access-block-configuration BlockPublicAcls=true,\
IgnorePublicAcls=true,BlockPublicPolicy=true,RestrictPublicBuckets=true
```

COMMON MISTAKE
Making a bucket public just to share one file. Public buckets are a classic leak. Use a pre-signed URL or CloudFront instead.

REVISION SNAPSHOT

ASK YOURSELF	What is the safest default for Block Public Access?
DO THIS	Enable Block Public Access and default encryption on a bucket.

17. RDS Managed Databases

GrowthSchool

IN SIMPLE TERMS
RDS is managed relational databases (like PostgreSQL or MySQL) where AWS handles backups, patching, and failover for you.

* Managed Relational

- RDS runs engines like PostgreSQL and MySQL for you: patching, backups, and failover are handled.
- Aurora is AWS's faster MySQL/PostgreSQL-compatible engine.

* High Availability

- Multi-AZ keeps a standby copy that takes over automatically; read replicas scale reads out.

RDS Multi-AZ: automatic failover keeps the DB available

REAL-WORLD EXAMPLE

```
aws rds create-db-instance --db-instance-identifier shop-db \
  --engine postgres --db-instance-class db.t3.micro \
  --allocated-storage 20 --multi-az --no-publicly-accessible
```

COMMON MISTAKE
Making the database public or hard-coding its password. Keep it private and put the password in Secrets Manager.

REVISION SNAPSHOT

ASK YOURSELF	What does Multi-AZ give you that a read replica does not?
DO THIS	Launch a private Multi-AZ Postgres instance.

18. DynamoDB (NoSQL)

GrowthSchool

IN SIMPLE TERMS
DynamoDB is a fully managed NoSQL database that is fast and scales automatically, with no servers to run.

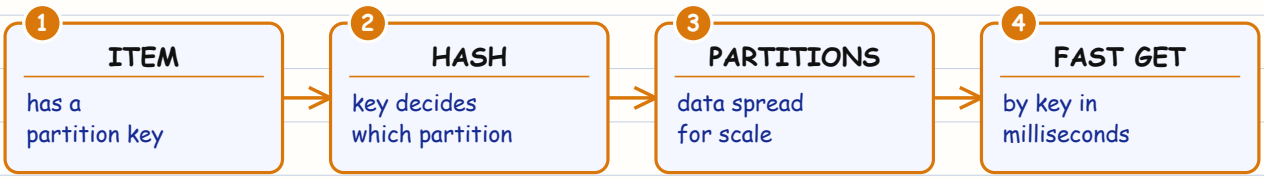
* Serverless Key-Value

- A fully managed NoSQL database with millisecond speed at any scale - no servers to run.
- You store items in tables and design around a partition key (plus an optional sort key).

* Capacity Modes

- On-demand auto-scales and bills per request; provisioned sets fixed capacity for steady load.

DynamoDB spreads items by their partition key



REAL-WORLD EXAMPLE

```
aws dynamodb create-table --table-name Carts \
  --attribute-definitions AttributeName=userId,AttributeType=S \
  --key-schema AttributeName=userId,KeyType=HASH --billing-mode PAY_PER_REQUEST
```

COMMON MISTAKE
Treating DynamoDB like SQL and scanning whole tables. That is slow and costly. Design your access patterns and query by the partition key.

REVISION SNAPSHOT

ASK YOURSELF	What is a partition key and why does it matter?
DO THIS	Create an on-demand table and get an item by its key.

19. Route 53 (DNS)

GrowthSchool

IN SIMPLE TERMS
Route 53 is AWS's DNS service: it turns your domain name into the address of your resources and can route users smartly.

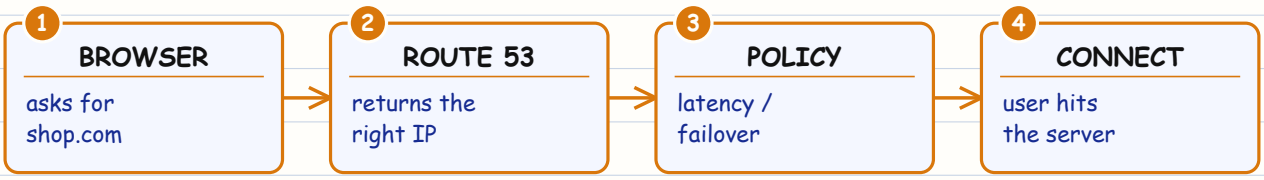
* Managed DNS

- Route 53 hosts your domain and answers "what is the address for shop.example.com?".
- An alias record points a name straight at AWS resources (ALB, CloudFront) for free.

* Smart Routing

- Routing policies can send users to the nearest, or fail over to a healthy backup automatically.

DNS: name in, best address out



REAL-WORLD EXAMPLE

```
aws route53 change-resource-record-sets --hosted-zone-id Z123 \
  --change-batch file://alias.json
dig +short shop.example.com
```

COMMON MISTAKE
Trying to use a CNAME at the domain apex (example.com). DNS forbids it - use a Route 53 alias record for the apex.

REVISION SNAPSHOT

ASK YOURSELF	Why prefer an alias record over a CNAME?
DO THIS	Create an alias record pointing your domain at an ALB.

20. CloudFront (CDN)

GrowthSchool

IN SIMPLE TERMS
CloudFront is a content delivery network (CDN) that caches your content at edge locations close to users for speed.

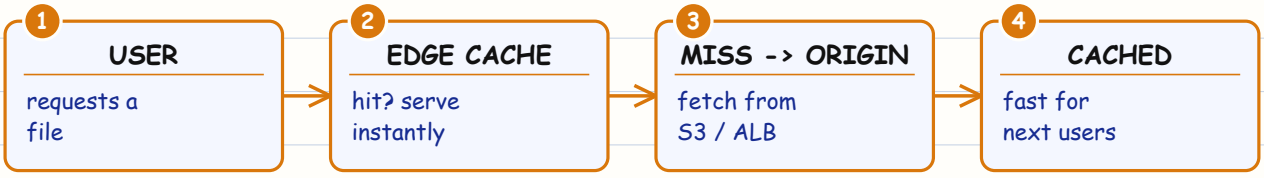
* Content Close to Users

- CloudFront caches your content at edge locations near users, so it loads fast and offloads origin.
- Origins can be S3 (static files) or an ALB/server (dynamic content).

* Secure Delivery

- Terminate HTTPS at the edge and use Origin Access Control so S3 is reachable only via CloudFront.

CloudFront serves from the nearest edge cache



REAL-WORLD EXAMPLE

```
aws cloudfront create-distribution --distribution-config file://dist.json
# after a deploy, refresh the cache:
aws cloudfront create-invalidation --distribution-id E123 --paths "/*"
```

COMMON MISTAKE
Forgetting to invalidate the cache after a release, so users see old files. Invalidate changed paths or use versioned filenames.

REVISION SNAPSHOT

ASK YOURSELF	What does CloudFront do on a cache hit versus a miss?
DO THIS	Serve an S3 site through CloudFront and invalidate after an update.

21. ECR Container Registry

GrowthSchool

IN SIMPLE TERMS
ECR is AWS's private registry for storing your Docker container images securely.

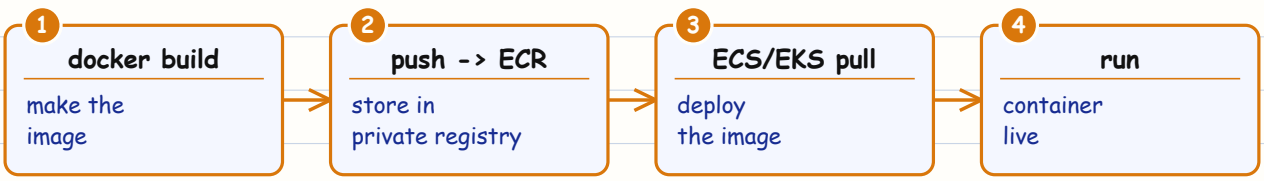
* Private Image Store

- ECR is AWS's private Docker registry, wired into IAM and ECS/EKS.
- You log in with a temporary token, then push and pull images like any registry.

* Keep It Tidy

- Turn on scan-on-push to catch vulnerabilities, and expire old images with a lifecycle policy.

ECR sits between build and deploy



REAL-WORLD EXAMPLE

```
aws ecr create-repository --repository-name shop-api
aws ecr get-login-password | docker login --username AWS \
  --password-stdin <acct>.dkr.ecr.us-east-1.amazonaws.com
```

COMMON MISTAKE
Never expiring old images, so the registry fills with junk layers and cost grows. Add a lifecycle policy to prune them.

REVISION SNAPSHOT

ASK YOURSELF	How do you authenticate Docker to ECR?
DO THIS	Create a repo, log in, and push a SHA-tagged image.

22. ECS & Fargate

GrowthSchool

IN SIMPLE TERMS
ECS runs your containers, and Fargate runs them without you managing any servers - you just say how much CPU and memory to use.

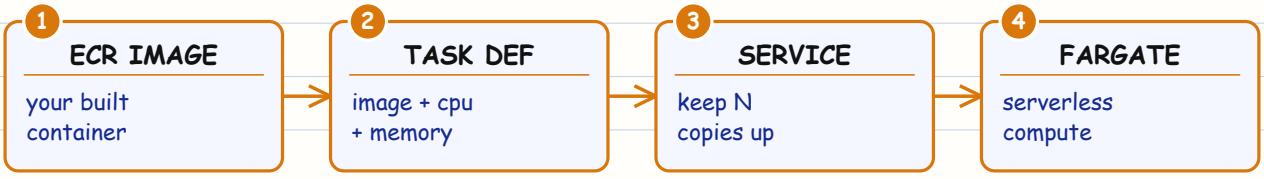
* Run Containers

- ECS runs your containers. A task definition says which image, how much CPU and memory, and ports.
- A service keeps a set number of tasks running behind a load balancer.

* Fargate

- Fargate runs the tasks with no servers to manage - you just pay for the CPU/memory each task uses.

ECS on Fargate: run containers, no servers to manage



REAL-WORLD EXAMPLE

```
aws ecs create-service --cluster shop --service-name shop-api \
  --task-definition shop-api --desired-count 3 --launch-type FARGATE
```

COMMON MISTAKE
Sizing task CPU/memory far above real usage and overpaying. Right-size from CloudWatch metrics.

REVISION SNAPSHOT

ASK YOURSELF	What is the difference between a task definition and a service?
DO THIS	Deploy the Shop API as a 3-task Fargate service.

23. EKS (Kubernetes)

GrowthSchool

IN SIMPLE TERMS
EKS is managed Kubernetes on AWS: AWS runs the Kubernetes control plane and you run your workloads on it.

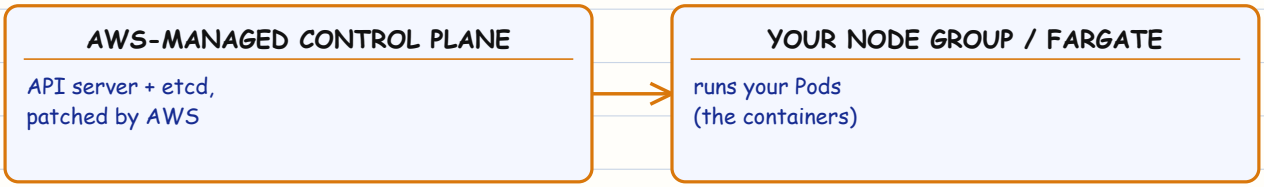
* Managed Kubernetes

- EKS runs the Kubernetes control plane for you; you run workloads on managed nodes or Fargate.
- Choose it when you want the Kubernetes ecosystem and portability rather than ECS's simplicity.

* Wire It Up

- Point kubectl at the cluster with the AWS CLI; use IAM Roles for Service Accounts for pod access.

EKS: AWS runs the control plane, you run the workloads



REAL-WORLD EXAMPLE

```
aws eks update-kubeconfig --name shop-cluster --region us-east-1
kubectl get nodes
kubectl apply -f shop-api.yaml
```

COMMON MISTAKE
Choosing EKS for a small app that ECS/Fargate would run more simply. Kubernetes adds overhead - pick it for a real reason.

REVISION SNAPSHOT

ASK YOURSELF	When would you choose EKS over ECS?
DO THIS	Point kubectl at an EKS cluster and list its nodes.

24. Lambda (Serverless)

GrowthSchool

IN SIMPLE TERMS
Lambda is serverless computing: you upload a function and AWS runs it on demand in response to events, billing per millisecond.

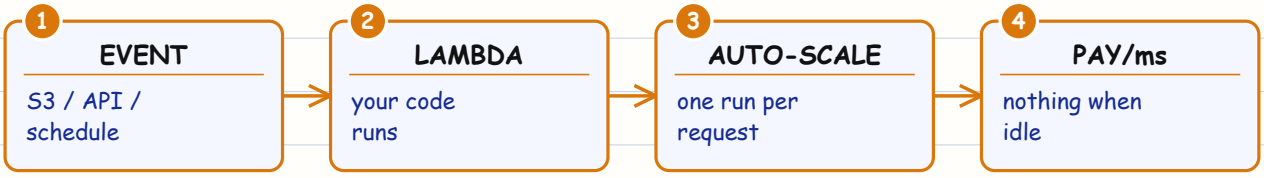
* Functions on Demand

- Upload code and AWS runs it in response to an event - no servers, billed per millisecond.
- It scales automatically per request and costs nothing while idle.

* Know the Limits

- Max 15-minute runtime and a cold-start delay on first call - great for event-driven work, not long jobs.

Serverless: an event triggers your function



REAL-WORLD EXAMPLE

```
aws lambda create-function --function-name resize-img \
  --runtime python3.12 --handler app.handler \
  --role arn:aws:iam::<acct>:role/lambda-role --zip-file fileb://fn.zip
```

COMMON MISTAKE
Using Lambda for an always-on or long job. The 15-minute limit and per-call model do not fit - use ECS/EC2 for sustained work.

REVISION SNAPSHOT

ASK YOURSELF	What work suits Lambda and what is the timeout?
DO THIS	Deploy a function triggered by an S3 upload.

25. API Gateway

GrowthSchool

IN SIMPLE TERMS
API Gateway is a managed front door for APIs that handles routing, security, and rate limiting to your backend.

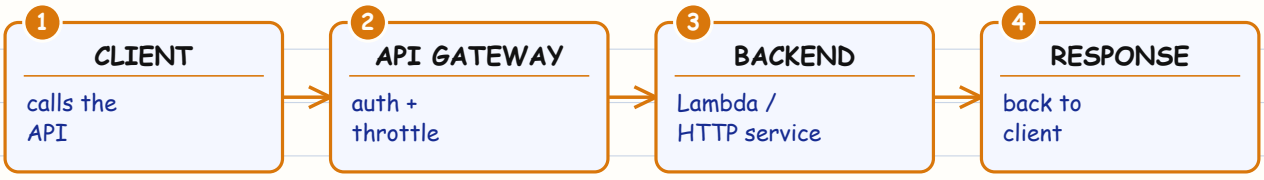
* Front Door for APIs

- API Gateway publishes your API and routes calls to Lambda or a backend service.
- It handles HTTPS, request checks, authorization, and rate limiting for you.

* Protect the Backend

- Add throttling and usage plans so one client cannot overwhelm your backend or your bill.

API Gateway is the guarded front door for APIs



REAL-WORLD EXAMPLE

```
aws apigatewayv2 create-api --name shop-api --protocol-type HTTP \
  --target arn:aws:lambda:us-east-1:<acct>:function:shop-fn
```

COMMON MISTAKE
Exposing a backend with no throttling or auth. One bad client can flood it and spike cost. Add rate limits and authorization.

REVISION SNAPSHOT

ASK YOURSELF	What does API Gateway handle so your backend does not?
DO THIS	Create an HTTP API that invokes a Lambda function.

26. SQS & SNS (Messaging)

GrowthSchool

IN SIMPLE TERMS
SQS is a message queue that buffers work between services; SNS is a publish/subscribe service that fans a message out to many receivers.

* SQS - Queues

- A queue holds messages so a fast producer and a slower consumer are decoupled and buffered.
- Standard is high-throughput; FIFO keeps strict order and exactly-once processing.

* SNS - Pub/Sub

- A topic pushes each message to many subscribers at once - queues, Lambdas, or email (fan-out).

SQS - QUEUE (pull)	SNS - TOPIC (push)
producer -> queue -> consumer	publisher -> topic -> many subs
buffers spikes of work	fan-out to queues, Lambda, email
one message, one worker	each subscriber gets a copy
standard or ordered FIFO	great for notifications

REAL-WORLD EXAMPLE

```
aws sqs create-queue --queue-name shop-orders
aws sns create-topic --name shop-events
aws sns subscribe --topic-arn <arn> --protocol sqs --notification-endpoint <queue-arn>
```

COMMON MISTAKE
Wiring services with direct synchronous calls everywhere, so one slow service stalls the rest. Use a queue to decouple and absorb spikes.

REVISION SNAPSHOT

ASK YOURSELF	What is the core difference between SQS and SNS?
DO THIS	Fan out one SNS message to two SQS queues.

27. CloudWatch (Observability)

GrowthSchool

IN SIMPLE TERMS
CloudWatch is AWS's monitoring service: it collects metrics and logs and raises alarms when something looks wrong.

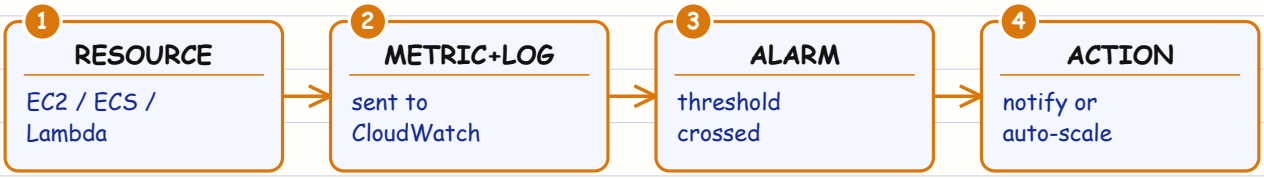
* Metrics & Logs

- CloudWatch collects numbers (CPU, latency) and log lines from AWS services and your apps.
- Dashboards show health at a glance; Logs Insights lets you query logs quickly.

* Alarms React

- An alarm watches a metric and, when a threshold is crossed, notifies you or triggers scaling.

CloudWatch watches, then reacts



REAL-WORLD EXAMPLE

```
aws logs tail /ecs/shop-api --follow
aws cloudwatch put-metric-alarm --alarm-name shop-cpu-high \
  --metric-name CPUUtilization --namespace AWS/ECS \
  --threshold 80 --comparison-operator GreaterThanThreshold \
  --evaluation-periods 5 --period 60
```

COMMON MISTAKE
Shipping with no alarms and hearing about outages from users. Alarm on the few signals that matter: errors, latency, and saturation.

REVISION SNAPSHOT

ASK YOURSELF	What are the three CloudWatch building blocks?
DO THIS	Tail an app log group and create a CPU-high alarm.

28. CloudTrail (Audit)

GrowthSchool

IN SIMPLE TERMS
CloudTrail records who did what in your AWS account, giving you an audit trail for security and troubleshooting.

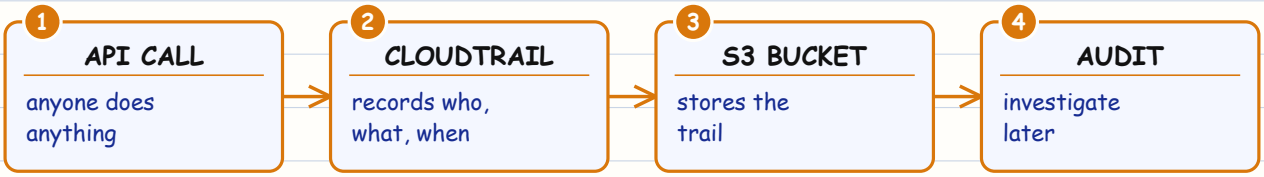
* Who Did What

- CloudTrail records every API action in your account: who, what, when, and from where.
- It is essential for security investigations, compliance, and tracking changes.

* Set It Up Right

- Enable a multi-Region trail that saves to a locked-down S3 bucket, and keep those logs immutable.

CloudTrail = the account's security camera



REAL-WORLD EXAMPLE

```
aws cloudtrail create-trail --name org-trail \
  --s3-bucket-name shop-cloudtrail-logs --is-multi-region-trail
aws cloudtrail lookup-events \
  --lookup-attributes AttributeKey=EventName,AttributeValue=RunInstances
```

COMMON MISTAKE
Turning CloudTrail on only after an incident, leaving no history to investigate. Enable it from day one and protect the log bucket.

REVISION SNAPSHOT

ASK YOURSELF	What question does CloudTrail answer that CloudWatch does not?
DO THIS	Enable a multi-Region trail and look up a recent API event.

29. SSM & Secrets Manager

GrowthSchool

IN SIMPLE TERMS
Systems Manager stores configuration and gives safe server access; Secrets Manager stores passwords and keys and can rotate them.

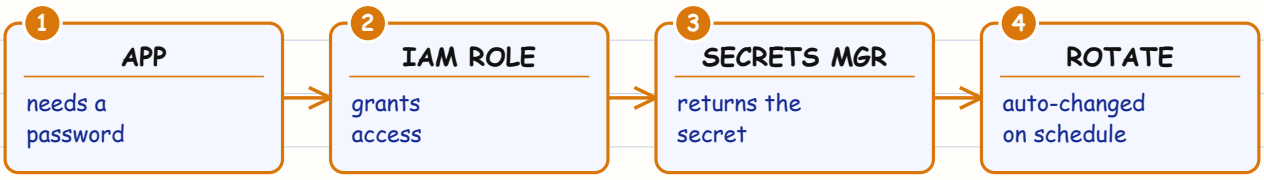
* Systems Manager

- Parameter Store keeps configuration (and secure strings); Session Manager gives shell access
- with no open SSH port and a full audit trail.

* Secrets Manager

- Stores passwords and keys, controls who can read them, and can rotate them automatically.

Apps fetch secrets at runtime - never bake them in



REAL-WORLD EXAMPLE

```
aws ssm put-parameter --name /shop/db-host --type String --value db.internal
aws ssm start-session --target i-123     # shell, no SSH port
aws secretsmanager get-secret-value --secret-id shop/db-password
```

COMMON MISTAKE
Passing secrets as plain env vars committed to a repo. Use Secrets Manager or secure parameters and fetch them at runtime with a role.

REVISION SNAPSHOT

ASK YOURSELF	When do you use Parameter Store vs Secrets Manager?
DO THIS	Store a config value and a secret, then read them via the CLI.

30. IaC: CloudFormation & Terraform

GrowthSchool

IN SIMPLE TERMS
This covers Infrastructure as Code on AWS - defining resources in CloudFormation (AWS-native) or Terraform instead of clicking.

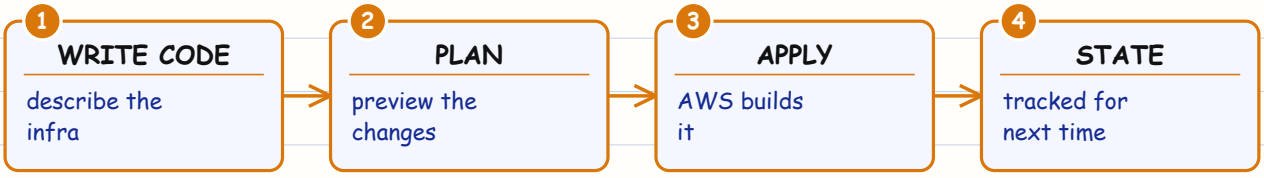
* Why IaC on AWS

- Describe infrastructure in files so environments are repeatable, reviewed, and version-controlled.
- CloudFormation is AWS-native; Terraform is multi-cloud (see the Terraform notes).

* Stacks & State

- Both track what they created (a stack or state file) and converge reality to your description.

Infrastructure as Code: describe, preview, apply



REAL-WORLD EXAMPLE

```
aws cloudformation deploy --template-file shop.yaml --stack-name shop-prod
aws cloudformation describe-stacks --stack-name shop-prod
# Terraform: terraform init && terraform apply
```

COMMON MISTAKE
Creating resources by hand next to IaC-managed ones. They drift and cause confusing changes. Manage each resource in code, in one place.

REVISION SNAPSHOT

ASK YOURSELF	What problem do CloudFormation and Terraform both solve?
DO THIS	Deploy a small stack and inspect its resources.

31. CI/CD on AWS

GrowthSchool

IN SIMPLE TERMS
CI/CD on AWS uses services like CodeBuild and CodePipeline to build, test, and deploy your app automatically.

* The Pipeline

- Source -> build+test (CodeBuild) -> push image (ECR) -> deploy (CodeDeploy/ECS), tied by CodePipeline.
- Each commit flows automatically toward production with checks along the way.

* Ship Safely

- Build one image tagged by commit and promote it everywhere; roll back automatically on failed health.

AWS CI/CD pipeline for the Shop API

REAL-WORLD EXAMPLE

```
# buildspec.yml (CodeBuild), simplified
phases:
  build:
    commands:
      - docker build -t $REPO:$CODEBUILD_RESOLVED_SOURCE_VERSION .
      - docker push $REPO:$CODEBUILD_RESOLVED_SOURCE_VERSION
```

REVISION SNAPSHOT

ASK YOURSELF	Why build one image and promote it, instead of rebuilding per environment?
DO THIS	Sketch a pipeline from commit to an ECS deployment.

32. Well-Architected Framework

GrowthSchool

IN SIMPLE TERMS
The Well-Architected Framework is AWS's checklist of six pillars for building secure, reliable, and cost-effective systems.

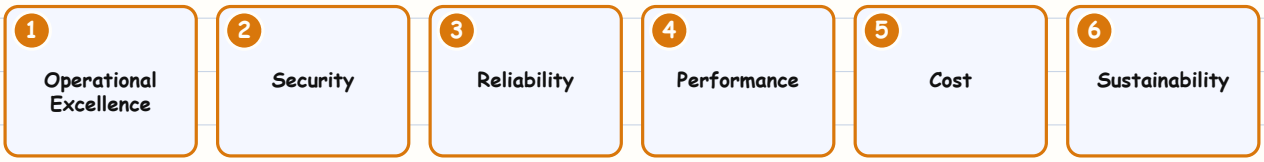
* Six Pillars

- Operational Excellence, Security, Reliability, Performance Efficiency, Cost Optimization, Sustainability.
- They are a lens for reviewing any workload for gaps and trade-offs.

* Apply It Often

- Automate ops, secure least-privilege, design for failure, measure performance, and right-size for cost.

The 6 pillars of the Well-Architected Framework



REAL-WORLD EXAMPLE

```
# quick self-review
# Multi-AZ? least-privilege IAM? encrypted? alarms + backups?
# tested restore? right-sized and tagged for cost?
```

COMMON MISTAKE
Treating Well-Architected as a one-time checkbox. Workloads change, so review the pillars regularly.

REVISION SNAPSHOT

ASK YOURSELF	Name the six pillars.
DO THIS	Review the Shop API against each pillar and note one gap.

33. Cost Optimization

GrowthSchool

IN SIMPLE TERMS
Cost optimization is choosing the right pricing models and turning off waste so you pay as little as possible.

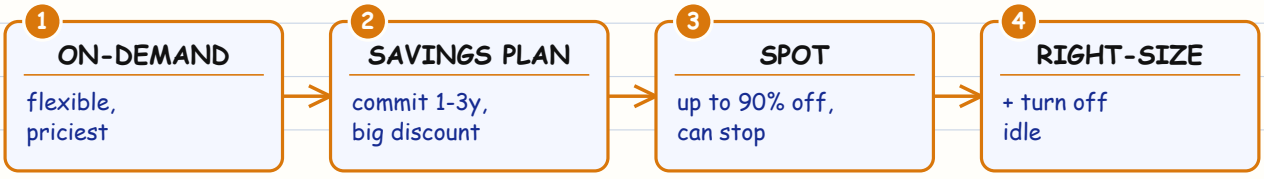
* Pricing Models

- On-Demand is flexible but priciest; Savings Plans/Reserved trade a commitment for big discounts;
- Spot is up to ~90% cheaper for work that can be interrupted.

* Cut Waste

- Right-size from metrics, switch off idle resources, use S3 lifecycle, and tag everything for visibility.

Pick the cheapest model that fits the workload



REAL-WORLD EXAMPLE

```
aws ce get-cost-and-usage --time-period Start=2025-01-01,End=2025-02-01 \
  --granularity MONTHLY --metrics UnblendedCost
```

COMMON MISTAKE
Leaving dev resources running 24/7. Schedule them off, right-size, and use Spot for interruptible work.

REVISION SNAPSHOT

ASK YOURSELF	When is Spot appropriate and when is it not?
DO THIS	Find your biggest service in Cost Explorer and one saving.

34. Shop API Reference Architecture

GrowthSchool

IN SIMPLE TERMS
This shows how all the services fit together into one complete, production-style architecture for the Shop API.

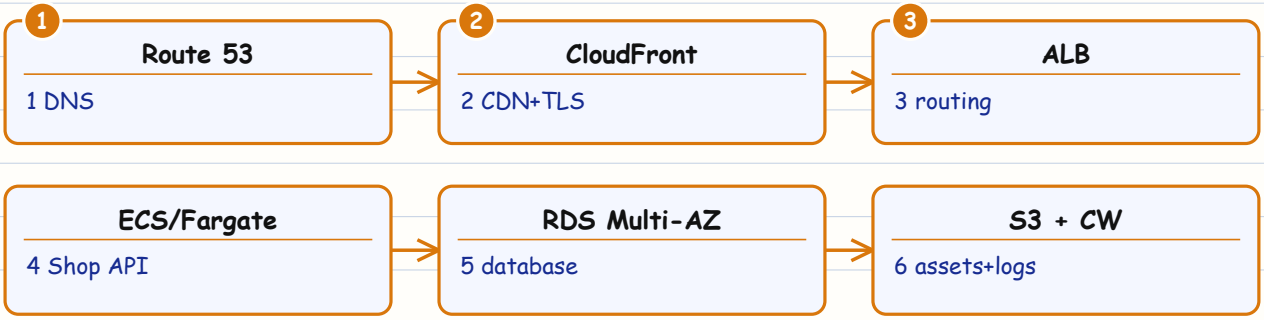
* End to End

- Route 53 -> CloudFront -> ALB -> ECS Fargate in private subnets across two AZs.
- RDS Postgres (Multi-AZ) for data, S3 for assets, Secrets Manager for the DB password.

* Operated Properly

- CloudWatch logs/alarms, CloudTrail audit, IAM roles (no keys), and CI/CD pushing tagged images.

Shop API on AWS - follow the request 1 -> 6



REAL-WORLD EXAMPLE

```
user -> Route53 -> CloudFront -> ALB -> Fargate -> RDS
assets: CloudFront -> S3 (OAC)   secrets: role -> Secrets Manager
```

REVISION SNAPSHOT

ASK YOURSELF	Trace a Shop API request from the user to the database.
DO THIS	Draw the full architecture and label each service's job.

35. Troubleshooting Playbook

GrowthSchool

IN SIMPLE TERMS
Troubleshooting is the calm, step-by-step way to fix common AWS problems like access denied or unreachable resources.

* Access & Network

- AccessDenied: a policy is missing an action or has an explicit deny - read the exact message.
- Cannot reach a server: check the SG, NACL, route table, and whether it is in a public subnet.

* Compute & Data

- No internet from a private server: missing NAT route. 5xx behind an ALB: unhealthy targets.
- RDS refused: security group, subnet routing, or wrong endpoint/credentials.

REAL-WORLD EXAMPLE

```
aws sts get-caller-identity
aws elbv2 describe-target-health --target-group-arn <arn>
aws logs tail /ecs/shop-api --follow
```

COMMON MISTAKE
Blaming "the network" before checking IAM and Security Groups - the usual culprits. Check permissions, SG rules, and routes first.

REVISION SNAPSHOT

ASK YOURSELF	First checks for AccessDenied vs a connectivity error?
DO THIS	Diagnose an unhealthy ALB target using target-health + logs.

36. AWS CLI Cheat Sheet

GrowthSchool

IN SIMPLE TERMS
This is a quick-reference list of the AWS CLI commands you will use every day.

* Daily Drivers

- sts, ec2, s3/s3api, iam, elbv2, ecs, rds, logs, cloudwatch, ssm, secretsmanager.
- Always confirm your --region and profile before deciding something "does not exist".

TRIAGE LOOP
get-caller-identity -> describe the resource -> check SG/IAM/route -> read logs.
This orients you on almost any AWS issue in under a minute.

ONE-PAGE COMMAND REFERENCE

```
aws sts get-caller-identity          aws s3 ls / cp / sync
aws ec2 describe-instances           aws ec2 run/stop/terminate-instances
aws elbv2 describe-target-health     aws ecs update-service --desired-count N
aws rds describe-db-instances        aws logs tail <group> --follow
aws cloudwatch put-metric-alarm      aws ssm start-session --target i-x
aws secretsmanager get-secret-value --secret-id <id>
```

REVISION SNAPSHOT

ASK YOURSELF	Which command confirms your identity, account, and Region?
DO THIS	Recall ten AWS CLI commands grouped by service.

37. Capstone Challenge

GrowthSchool

IN SIMPLE TERMS
This capstone ties everything together by building and operating the full Shop API stack on AWS.

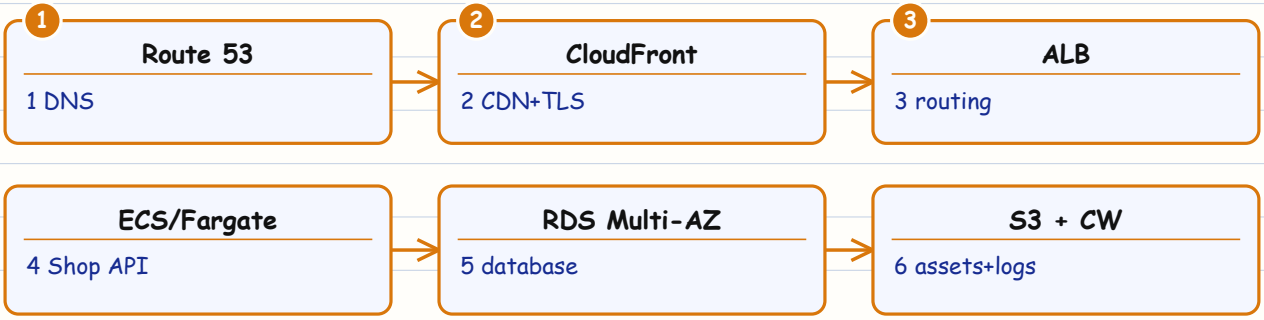
* Build the Shop API on AWS

- VPC with public+private subnets over 2 AZs, an ALB, and ECS Fargate running the Shop API.
- RDS Postgres (Multi-AZ, private) and S3 for assets, all defined in CloudFormation or Terraform.

* Operate It

- IAM roles (no keys), Secrets Manager for the DB password, CloudWatch alarms, CloudTrail on,
- and a pipeline that builds a tagged image to ECR and deploys it.

Shop API on AWS - follow the request 1 -> 6



STUDENT LAB
Deliverables: a reachable HTTPS endpoint, a private DB, least-privilege roles, alarms on high CPU/5xx, a billing budget, and infra fully in code. Tear it all down cleanly afterwards.

REVISION SNAPSHOT

ASK YOURSELF	Is your stack Multi-AZ, least-privilege, observable, and in code?
DO THIS	Complete every deliverable, then destroy the stack to avoid cost.

38. Interview & Revision Notes

GrowthSchool

IN SIMPLE TERMS
This page gathers the key ideas and common interview questions so you can revise quickly.

* Explain Clearly

- Region vs AZ; IAM user vs role; SG (stateful) vs NACL (stateless); public vs private subnet.
- S3 vs EBS vs EFS; RDS Multi-AZ vs read replica; ECS/Fargate vs EKS vs Lambda; SQS vs SNS.

* Common Questions

- How do you give an app permissions without keys? How do you make it highly available? How cut cost?
- Where do secrets live? What does the shared responsibility model mean in practice?

REAL-WORLD EXAMPLE

```
# 30-second self test
aws sts get-caller-identity   # identity + account
# SG vs NACL? role vs key? Multi-AZ vs read replica?
# S3 vs EBS vs EFS? ECS vs EKS vs Lambda? SQS vs SNS?
```

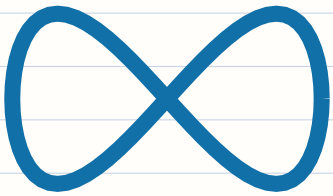
REVISION SNAPSHOT

ASK YOURSELF	Can you explain IAM roles vs access keys in 30 seconds?
DO THIS	Answer every "Common Question" above out loud without notes.

GrowthSchool

CI/CD END-TO-END

COMPLETE NOTES



COMMIT - BUILD - DEPLOY

From a git commit to a live app on AWS EKS - the GitOps way

growthschool.cc | @growthschool.cc

Learning Roadmap

GrowthSchool

A complete, interview-ready walkthrough of one real pipeline, using a school LMS app (ClassKira) as the example. Every page has a plain-English definition, a numbered diagram, real snippets, common mistakes, and a revision snapshot. Learn the 10-step flow so well you can draw it from memory in an interview.

HOW TO USE THESE NOTES

Learn the flow as a story: commit -> test -> build -> push -> update tag -> sync -> deploy.
If you can retell those 10 steps in order, you can pass almost any CI/CD interview.

1. What is CI/CD?

GrowthSchool

IN SIMPLE TERMS
CI/CD is an automated assembly line for software: every time you change the code it is tested, packaged, and shipped to users automatically - so releases are fast, safe, and repeatable.

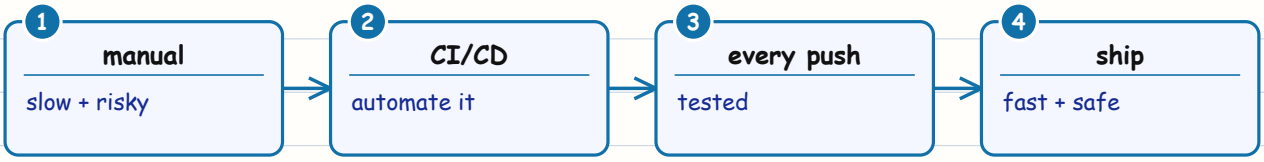
* The Idea

- Instead of manually testing and deploying (slow and error-prone), a pipeline does it for you.
- Every code change is automatically tested, packaged, and shipped - the same way every time.

* Why It Wins

- Faster releases, fewer bugs reaching users, and instant rollback when something is wrong.
- Humans review the change; the machine does the repetitive, risky steps.

Manual vs automated



REAL-WORLD EXAMPLE

```
# the whole promise of CI/CD, in one line:
git push   ->   tested, built, and deployed automatically
```

COMMON MISTAKE
Thinking CI/CD is "just a build script". It is a safety system: tests gate the build, and the deploy is reproducible. Skipping the gates defeats the point.

REVISION SNAPSHOT

ASK YOURSELF	What three things happen automatically on every code change?
DO THIS	Explain CI/CD to a friend in one sentence using the ClassKira app.

2. CI vs CD vs GitOps

GrowthSchool

IN SIMPLE TERMS
CI (Continuous Integration) = automatically build and test every change. CD (Continuous Delivery/Deployment) = automatically ship it. GitOps = Git is the single source of truth that a tool keeps your cluster in sync with.

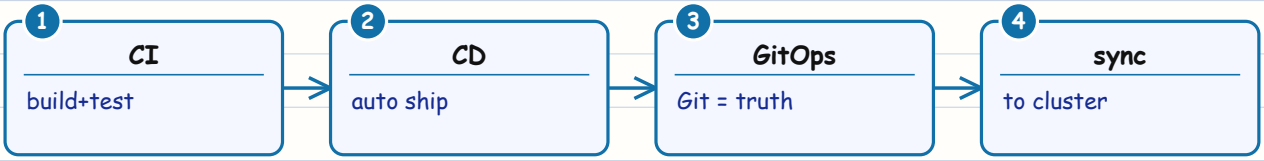
* The Three Terms

- CI = Continuous Integration: build + test every change quickly.
- CD = Continuous Delivery/Deployment: automatically ship the tested change to an environment.

* GitOps

- GitOps means the desired state lives in Git; a tool (Argo CD) makes the cluster match it.
- You never run kubectl by hand in production - you change Git, and the cluster follows.

CI -> CD -> GitOps



REAL-WORLD EXAMPLE

```
CI  : GitHub Actions runs tests + builds the image
CD  : the new image tag is shipped automatically
GitOps: Argo CD syncs the cluster to whatever Git says
```

COMMON MISTAKE
Confusing Continuous Delivery (auto to staging, human approves prod) with Continuous Deployment (auto all the way to prod). Know which one your team uses.

REVISION SNAPSHOT

ASK YOURSELF	In one line each: what is CI, CD, and GitOps?
DO THIS	Say which step Actions owns and which step Argo CD owns.

3. The ClassKira Pipeline

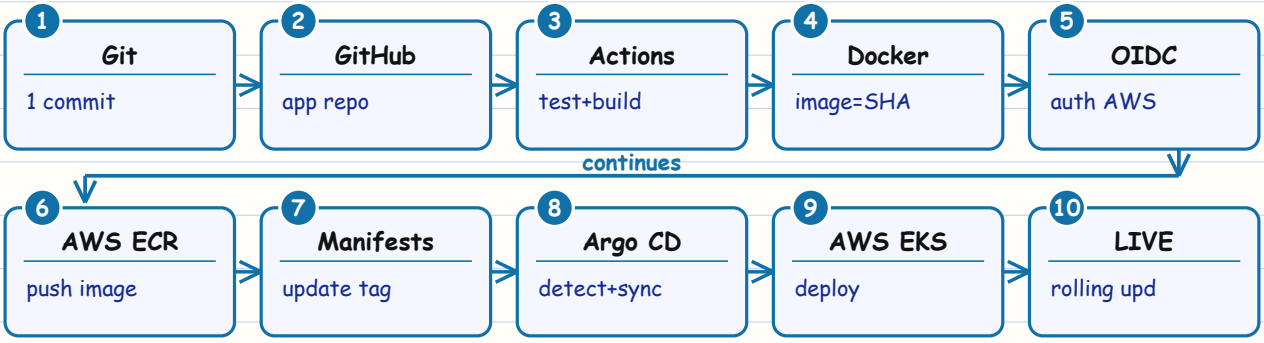
GrowthSchool

IN SIMPLE TERMS
This is the big picture of the whole ClassKira pipeline - one diagram showing how a single commit travels from a developer's laptop all the way to a live app.

* Follow One Commit

- ClassKira is our example school LMS. A developer fixes a bug and pushes once.
- That single push travels through ten automated steps and ends as a live update for students.

ClassKira LMS: one commit -> live in production (follow 1 -> 10)



REVISION SNAPSHOT

ASK YOURSELF	Can you name all ten steps from commit to live, in order?
DO THIS	Trace one ClassKira commit through the diagram out loud.

4. Step 1: Git - Commit

GrowthSchool

IN SIMPLE TERMS
Git is where a developer saves changes to the app code and the Kubernetes manifests (the YAML files that describe how the app should run).

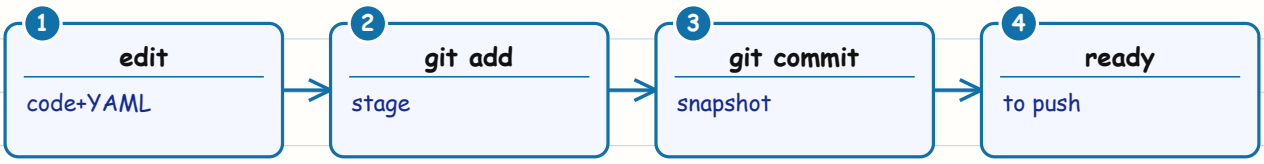
* What You Change

- You edit the app code (the bug fix) and commit it with a clear message.
- The Kubernetes manifests (which image to run) live in Git too - config as code.

* Good Habits

- Small, well-described commits make the pipeline history easy to read and to roll back.
- Use a feature branch, not main, so CI can check it before it merges.

Commit the change



REAL-WORLD EXAMPLE

```
git switch -c fix/login-timeout
git add .
git commit -m "fix(auth): extend login timeout"
git push -u origin fix/login-timeout
```

REVISION SNAPSHOT

ASK YOURSELF	What two kinds of files does GitOps keep in Git?
DO THIS	Make one small, well-named commit on a branch.

5. Step 2: GitHub - Repo & PRs

GrowthSchool

IN SIMPLE TERMS
GitHub is the shared online home for the repository, where teammates review changes through pull requests before they are merged into the main branch.

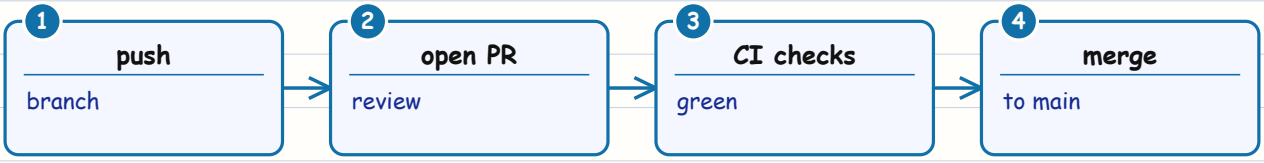
* The Shared Home

- GitHub stores the repository and opens your branch for review as a pull request (PR).
- Teammates comment; CI checks must pass before the PR can merge to main.

* The Trigger

- Opening or updating a PR (and merging to main) triggers the GitHub Actions pipeline.

GitHub pull request



REAL-WORLD EXAMPLE

```
gh pr create --base main --title "fix: login timeout" --fill
# CI runs automatically on the PR; merge when green
gh pr merge --squash
```

COMMON MISTAKE
Pushing straight to main with no PR or checks. One bad commit then ships to students. Protect main: require a PR plus passing CI.

REVISION SNAPSHOT

ASK YOURSELF	What event triggers the pipeline to run?
DO THIS	Open a PR and watch the CI checks appear on it.

6. App Repo vs Manifests Repo

GrowthSchool

IN SIMPLE TERMS
The GitOps pattern uses two repositories: an "app repo" (the source code) and a "manifests repo" (the Kubernetes YAML that says which image version to run).

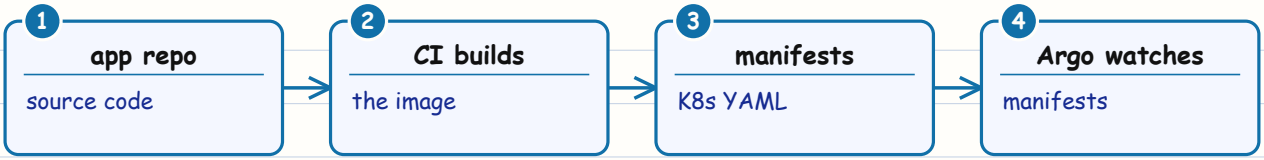
* Two Repositories

- App repo: the ClassKira source code and its Dockerfile. CI builds an image from it.
- Manifests repo: the Kubernetes YAML (Deployment, Service) that says which image tag to run.

* Why Separate

- Argo CD watches only the manifests repo, so a config change (image tag) is what triggers a deploy.
- This clean split is the heart of GitOps.

Two-repo GitOps



REAL-WORLD EXAMPLE

```
app-repo/       src/ ... Dockerfile
manifests-repo/ deployment.yaml   # image: <ECR>/classkira:<TAG>
# Argo CD is pointed at manifests-repo
```

REVISION SNAPSHOT

ASK YOURSELF Which repo does Argo CD watch, and why that one?

DO THIS Sketch the two repos and what each contains.

7. Step 3: GitHub Actions

GrowthSchool

IN SIMPLE TERMS
GitHub Actions is GitHub's built-in automation that runs your pipeline steps automatically whenever you push code or open a pull request.

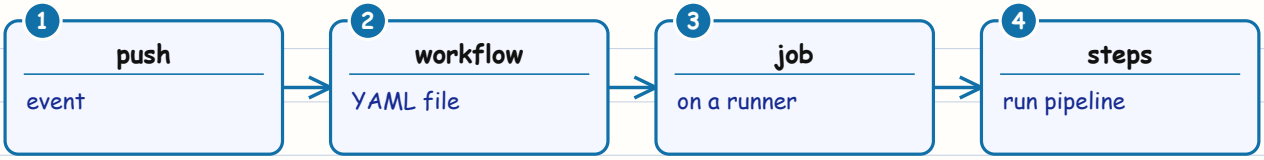
* The Automation Engine

- A workflow is a YAML file in .github/workflows. It lists jobs, each with steps.
- It runs on GitHub-hosted runners whenever the trigger event happens.

* Our Pipeline Job

- One job will: check out code, run tests, build the image, log in to AWS, and push to ECR.

GitHub Actions



REAL-WORLD EXAMPLE

```
name: ci-cd
on:
  push: { branches: [main] }
permissions:
  id-token: write   # needed for OIDC
  contents: write   # to commit the new tag
```

REVISION SNAPSHOT

ASK YOURSELF	Where do GitHub Actions workflows live in the repo?
DO THIS	Create a workflow that runs on every push to main.

8. Step 3: Test & Validate

GrowthSchool

IN SIMPLE TERMS
The CI stage runs your tests and checks (lint, unit tests) to prove a change is safe BEFORE anything is built or shipped.

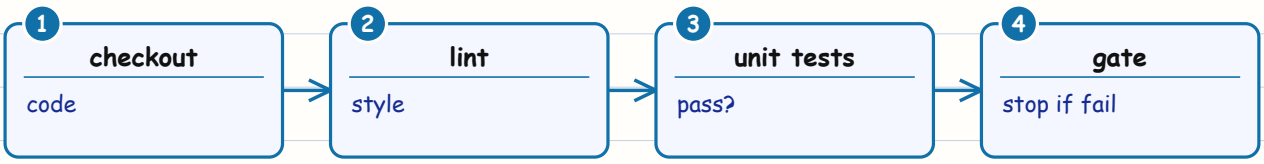
* Prove It Is Safe

- Before building anything, run lint and unit tests. If they fail, the pipeline stops here.
- This "quality gate" keeps broken code from ever becoming an image or a deployment.

* Validate Config Too

- Also validate the Kubernetes YAML (e.g. kubeconform) so a bad manifest cannot ship.

CI quality gate



REAL-WORLD EXAMPLE

```
- uses: actions/checkout@v4
- run: npm ci
- run: npm run lint
- run: npm test           # fails the job on any failing test
- run: kubeconform manifests/
```

COMMON MISTAKE
Building and pushing before tests pass. Always gate the build on green tests - otherwise you ship bugs fast, which is worse than shipping slow.

REVISION SNAPSHOT

ASK YOURSELF	What happens to the pipeline if a unit test fails?
DO THIS	Add a failing test and confirm the pipeline stops.

9. Step 4: Build the Docker Image

GrowthSchool

IN SIMPLE TERMS
A Docker image is the packaged app. CI builds it and labels it with the exact commit id (SHA), so every image is traceable back to its code.

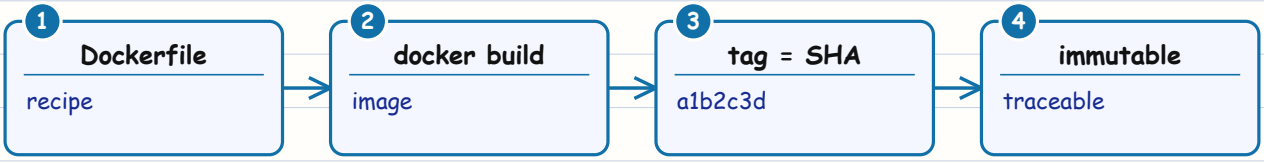
* Package the App

- The Dockerfile turns the tested code into an image. We tag it with the commit SHA.
- A SHA tag (e.g. classkira:a1b2c3d) is immutable and traceable to the exact code.

* Never "latest"

- Using latest hides which version is running and breaks rollbacks. Always use the SHA (or version).

Build the image



REAL-WORLD EXAMPLE

```
SHA=$(git rev-parse --short HEAD)
docker build -t classkira:$SHA .
# later tagged for ECR:  <acct>.dkr.ecr.<region>.amazonaws.com/classkira:$SHA
```

COMMON MISTAKE
Tagging the image latest and deploying that. Two deploys then run different code under the same name. Tag by commit SHA so every image is unique and traceable.

REVISION SNAPSHOT

ASK YOURSELF	Why tag the image with the commit SHA instead of latest?
DO THIS	Build an image tagged with your short commit SHA.

10. Step 5: GitHub OIDC to AWS

GrowthSchool

IN SIMPLE TERMS
OpenID Connect (OIDC) lets GitHub Actions log in to AWS with a short-lived token instead of stored access keys - keyless, and much safer.

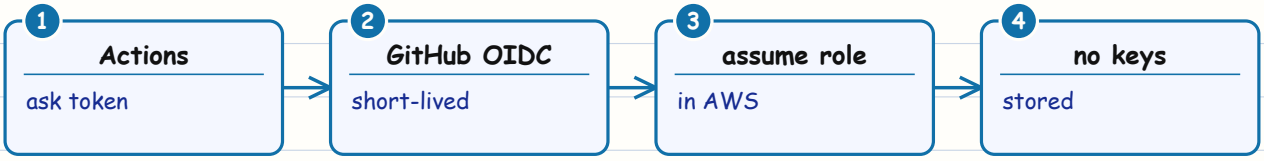
* Keyless Login

- Instead of storing AWS access keys in GitHub, Actions requests a short-lived OIDC token.
- AWS trusts GitHub as an identity provider and lets the workflow assume a scoped IAM role.

* Why It Matters

- No long-lived secrets to leak or rotate. The token lasts minutes and is scoped to this repo.
- This is the modern, secure way to connect CI to a cloud - and a favourite interview topic.

Keyless OIDC login



REAL-WORLD EXAMPLE

```
- uses: aws-actions/configure-aws-credentials@v4
  with:
    role-to-assume: arn:aws:iam::<acct>:role/classkira-ci
    aws-region: us-east-1
# no access keys anywhere - just a short-lived OIDC token
```

COMMON MISTAKE
Storing static AWS_ACCESS_KEY_ID / SECRET in GitHub Secrets. They leak and never expire.
Use OIDC + an IAM role trusting your repo instead.

REVISION SNAPSHOT

ASK YOURSELF	How does GitHub Actions log in to AWS without stored keys?
DO THIS	Explain OIDC keyless auth in two sentences.

11. Step 6: Push Image to AWS ECR

GrowthSchool

IN SIMPLE TERMS
AWS ECR is AWS's private image registry. After building, the pipeline pushes the tagged image there so the cluster can pull it later.

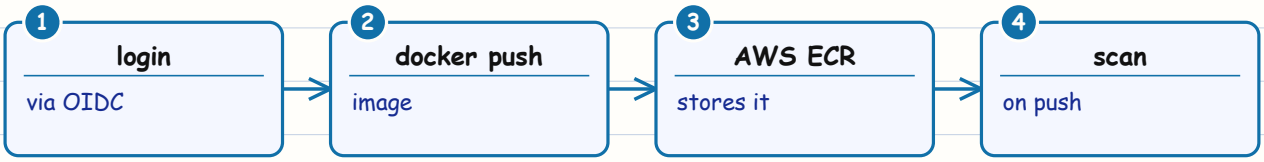
* Store the Image

- With the OIDC role assumed, the workflow logs in to ECR and pushes the SHA-tagged image.
- ECR is private; only the pipeline and the cluster (via IAM) can pull from it.

* Scan It

- Enable scan-on-push so vulnerabilities are caught before the image can ever be deployed.

Push to AWS ECR



REAL-WORLD EXAMPLE

```
aws ecr get-login-password | docker login --username AWS \
  --password-stdin <acct>.dkr.ecr.us-east-1.amazonaws.com
docker push <acct>.dkr.ecr.us-east-1.amazonaws.com/classkira:$SHA
```

REVISION SNAPSHOT

ASK YOURSELF	What just happened after a successful docker push to ECR?
DO THIS	Push a test image to a private ECR repo using OIDC.

12. Step 7: Update the Image Tag

GrowthSchool

IN SIMPLE TERMS
The pipeline automatically edits the Kubernetes manifest to point at the new image tag, and commits that change back to Git.

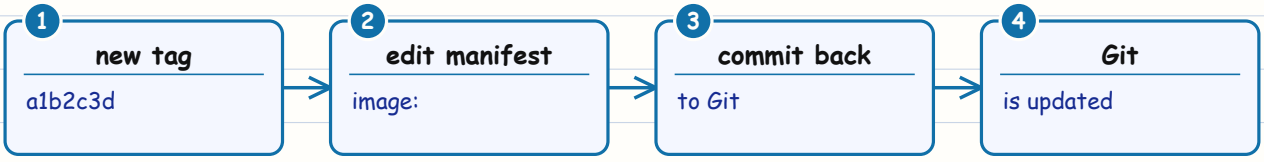
* Commit the New Tag

- The pipeline edits the manifest in the manifests repo to point at the new SHA tag,
- then commits and pushes that change back to Git. Git now holds the new desired state.

* This Is the Handoff

- CI is done. From here CD (Argo CD) takes over - it only reacts to the Git change.

Update image tag



REAL-WORLD EXAMPLE

```
# in the manifests repo, bump the tag and commit
sed -i "s|classkira:.*|classkira:$SHA|" deployment.yaml
git commit -am "deploy: classkira $SHA"
git push   # this commit is what triggers the deployment
```

COMMON MISTAKE
Having CI run kubectl apply directly to the cluster (push-based). That bypasses Git history and Argo CD. In GitOps, CI only updates Git; the cluster is changed by Argo CD.

REVISION SNAPSHOT

ASK YOURSELF	After CI pushes the tag commit, what part of the system reacts next?
DO THIS	Update an image tag in a manifest and commit it.

13. What is Argo CD?

GrowthSchool

IN SIMPLE TERMS
Argo CD is a GitOps tool that watches your manifests repo and continuously makes the cluster match exactly what Git says it should be.

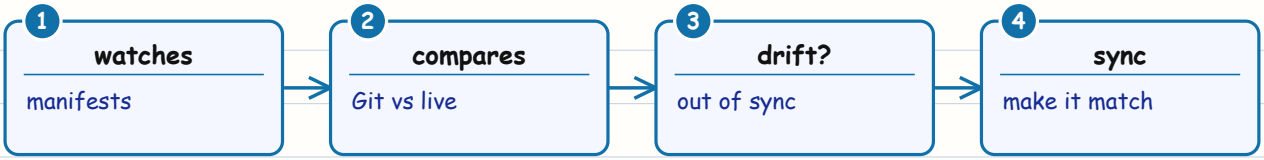
* The GitOps Deployer

- Argo CD runs inside the cluster and continuously watches the manifests repo.
- It compares what Git says (desired) with what is running (live) and reports "Synced" or "OutOfSync".

* Pull, Not Push

- The cluster pulls its own desired state from Git. Nothing outside needs cluster credentials.
- This is safer and fully auditable - every change is a Git commit.

What Argo CD does



REAL-WORLD EXAMPLE

```
argocd app get classkira
# Health: Healthy   Sync: Synced (or OutOfSync)
# source: manifests-repo   destination: EKS cluster
```

REVISION SNAPSHOT

ASK YOURSELF	What two states does Argo CD compare, and what does it watch?
DO THIS	Explain pull-based GitOps vs pushing with kubectl.

14. Step 8: Argo CD Detects the Chan

GrowthSchool

IN SIMPLE TERMS
When the image tag changes in Git, Argo CD notices the difference (drift) and syncs the new desired state to the cluster.

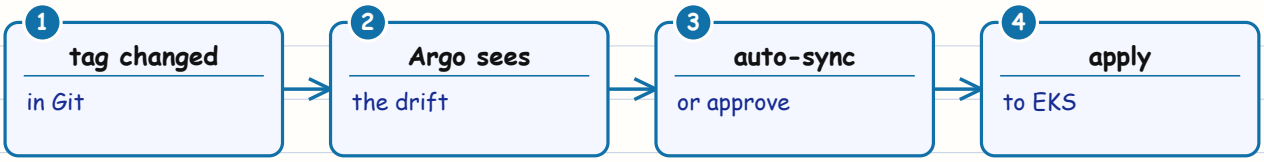
* Drift Detected

- The tag commit makes Git differ from the cluster, so Argo CD marks the app OutOfSync.
- With auto-sync on, it proceeds automatically; otherwise a human clicks "Sync" to approve.

* Controlled Release

- You can require manual sync for production to keep a human approval in the loop.

Argo detects change



REAL-WORLD EXAMPLE

```
# auto-sync policy (in the Argo CD Application)
syncPolicy:
  automated: { prune: true, selfHeal: true }
# or leave it manual and click Sync for prod
```

REVISION SNAPSHOT

ASK YOURSELF	Why does Argo CD suddenly show OutOfSync after the tag commit?
DO THIS	Toggle auto-sync vs manual sync and note the difference.

15. Step 9: Deploy to AWS EKS

GrowthSchool

IN SIMPLE TERMS
Deploying means Argo CD applies the updated manifest to AWS EKS, which then pulls the new image from AWS ECR.

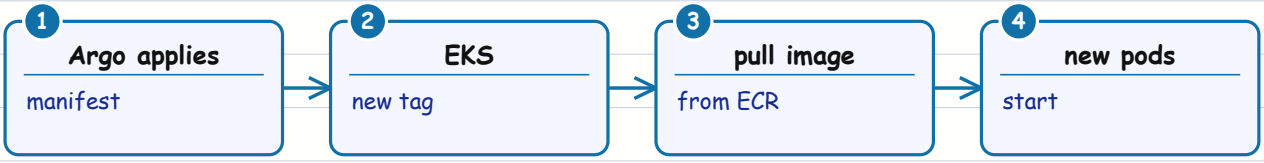
* Apply the Desired State

- Argo CD applies the updated Deployment to AWS EKS. EKS sees a new image tag.
- EKS pulls that exact image from AWS ECR (using the node/pod IAM permissions).

* New Pods Start

- Kubernetes creates new pods running the new image, keeping the old ones until the new are ready.

Deploy to EKS



REAL-WORLD EXAMPLE

```
kubectl get deploy classkira -o wide      # shows the new image tag
kubectl get pods -l app=classkira         # new pods appearing
kubectl describe pod <pod>                # events: pulling from ECR
```

COMMON MISTAKE
Forgetting the cluster needs permission to pull from a private ECR. If pods show ImagePullBackOff, fix the ECR pull permissions / IAM, not the image.

REVISION SNAPSHOT

ASK YOURSELF	Where does EKS pull the new image from, and how is it allowed to?
DO THIS	Watch new pods pull the image after a sync.

16. Step 10: Rolling Update - Live

GrowthSchool

IN SIMPLE TERMS
A rolling update replaces the old pods with new ones gradually, so users never see downtime while the new version goes live.

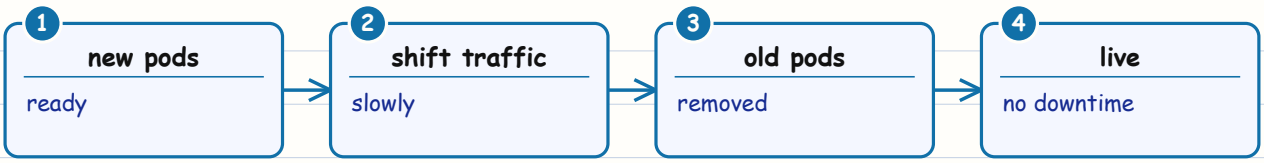
* Zero-Downtime Release

- Kubernetes brings up new pods, waits for readiness, shifts traffic, then removes old pods.
- Students keep using ClassKira the whole time - they just get the fixed version.

* Confirm Success

- Watch the rollout finish and check health/metrics before calling the release done.

Rolling update



REAL-WORLD EXAMPLE

```
kubectl rollout status deploy/classkira
# Waiting for rollout... deployment successfully rolled out
curl -f https://classkira.com/health
```

REVISION SNAPSHOT

ASK YOURSELF	How does a rolling update avoid downtime for users?
DO THIS	Watch a rollout complete and verify the health endpoint.

17. End-to-End Recap

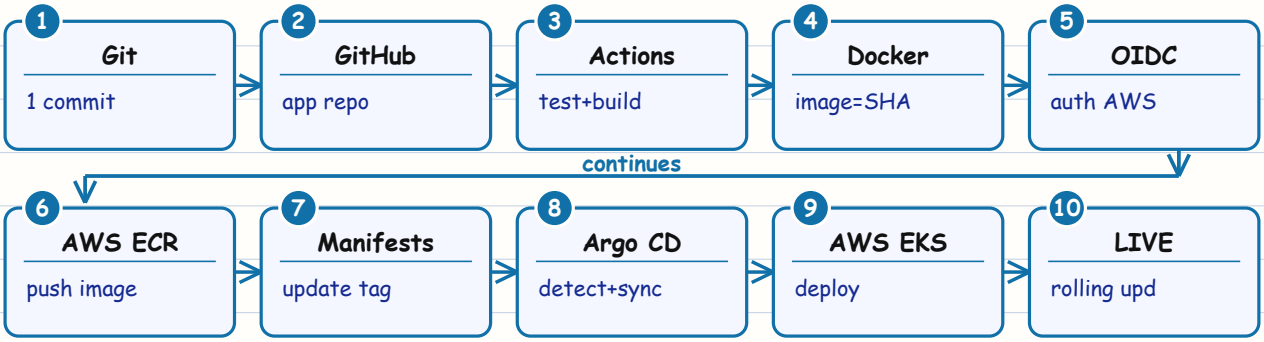
GrowthSchool

IN SIMPLE TERMS
This recaps the entire journey end to end in ten numbered steps, so you can retell the whole flow from memory.

* The Whole Story

- Commit -> push -> test -> build image (SHA) -> OIDC login -> push to ECR ->
- update tag in Git -> Argo CD syncs -> EKS pulls image -> rolling update -> live.

ClassKira LMS: one commit -> live in production (follow 1 -> 10)



REVISION SNAPSHOT

ASK YOURSELF	Retell all ten steps from memory, in order.
DO THIS	Draw the pipeline on paper without looking at the notes.

18. Rollback & Safety

GrowthSchool

IN SIMPLE TERMS
Rolling back is easy in GitOps: you revert the commit in Git, and Argo CD automatically returns the cluster to the previous version.

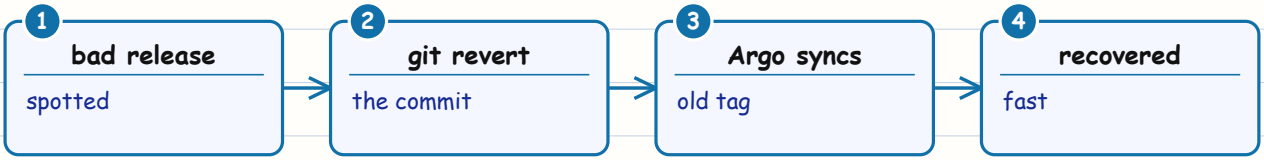
* Undo Is a Git Revert

- A bad release? Revert the tag commit in the manifests repo. Git now points at the old image.
- Argo CD syncs the cluster back automatically - no scary manual cluster surgery.

* Safety Nets

- Keep old images in ECR, protect main, and require review so bad tags rarely merge at all.

Instant rollback



REAL-WORLD EXAMPLE

```
git revert <bad-tag-commit>   # in the manifests repo
git push
# Argo CD detects the change and rolls back to the previous image
```

COMMON MISTAKE
Rolling back by editing the cluster directly. That drifts from Git; the next sync would undo your fix. Roll back IN GIT so Git stays the source of truth.

REVISION SNAPSHOT

ASK YOURSELF	What is the safest way to roll back in GitOps?
DO THIS	Revert a tag commit and watch Argo CD restore the old version.

19. Security in the Pipeline

GrowthSchool

IN SIMPLE TERMS
Pipeline security means no stored secrets (use OIDC), least-privilege permissions, and scanning images for vulnerabilities before they ship.

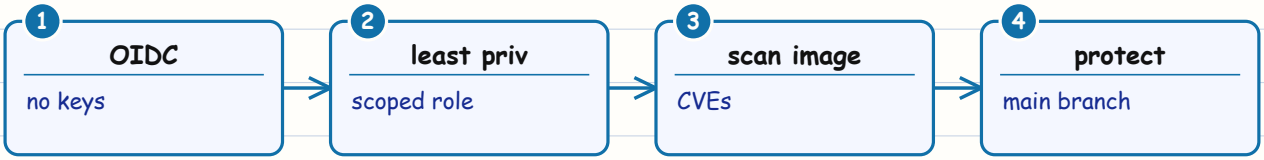
* No Stored Secrets

- Use GitHub OIDC (keyless) for AWS. Give the CI role least privilege - only ECR push it needs.
- Protect the main branch and require PR reviews so only approved code can ship.

* Trust the Image

- Scan images on push, pin base images, and run containers as non-root in the manifests.

Secure the pipeline



REAL-WORLD EXAMPLE

```
# IAM role trust: only this repo + branch can assume it
sub: repo:org/classkira:ref:refs/heads/main
# role policy: ecr:PutImage on the classkira repo only
```

COMMON MISTAKE
Giving the CI role broad admin "to make it work". A leaked pipeline then owns your account. Scope the OIDC role to exactly the ECR actions and repo it needs.

REVISION SNAPSHOT

ASK YOURSELF	Name three ways this pipeline stays secure.
DO THIS	Scope a CI IAM role to only ECR push for one repo.

20. Common Mistakes & Fixes

GrowthSchool

IN SIMPLE TERMS
These are the mistakes that break real pipelines - and the quick checks that fix each one.

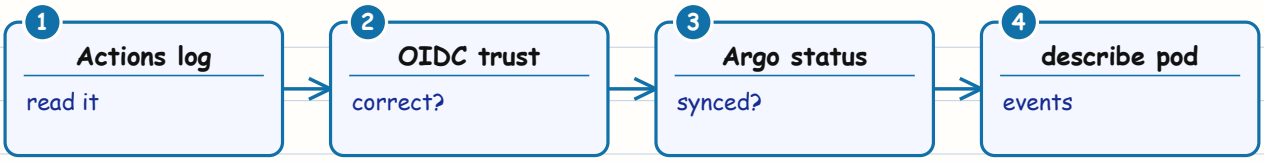
* CI Side

- Build before tests (ship bugs); using latest tag (no rollback); storing static AWS keys (leaks).
- OIDC "not authorized to assume role" usually means the IAM trust policy sub/branch is wrong.

* CD Side

- CI running kubectl directly (bypasses GitOps); ImagePullBackOff (ECR pull perms/wrong tag);
- Argo stuck OutOfSync (manual sync off, or a bad/typo'd manifest).

Debug the pipeline



REAL-WORLD EXAMPLE

```
# quick triage
gh run view --log            # read the Actions logs
argocd app get classkira     # sync + health status
kubectl describe pod <pod>   # Events explain pull failures
```

REVISION SNAPSHOT

ASK YOURSELF	What usually causes "not authorized to assume role" with OIDC?
DO THIS	Match each symptom above to its one-line fix.

21. CI/CD Cheat Sheet

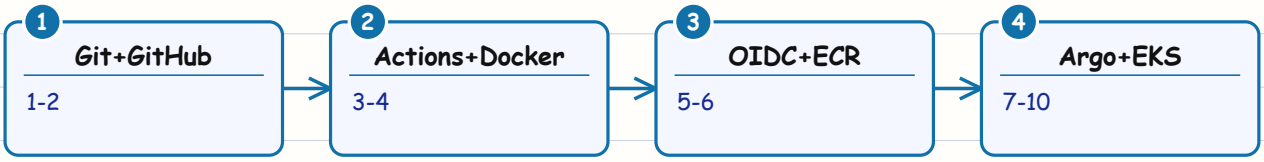
GrowthSchool

IN SIMPLE TERMS
A one-page quick reference of the tools, the commands, and the order of the whole pipeline.

* The Order (memorise this)

- 1 commit 2 push/PR 3 test+validate 4 build image (SHA) 5 OIDC login
- 6 push to ECR 7 update tag in Git 8 Argo detects 9 deploy to EKS 10 rolling update.

Remember the order



REMEMBER IT AS A STORY
Code it (Git+GitHub) -> prove it (test) -> pack it (Docker) ->
store it (ECR) -> point to it (tag in Git) -> ship it (Argo->EKS).

ONE-PAGE PIPELINE REFERENCE

```
git commit / git push              docker build -t app:$SHA .
gh pr create / gh pr merge         aws ecr ... / docker push
npm test / kubeconform             sed -i tag + git commit
configure-aws-credentials (OIDC)   argocd app get / sync
kubectl rollout status             git revert  (rollback)
```

REVISION SNAPSHOT

ASK YOURSELF	Can you list the 10 steps in order without looking?
DO THIS	Recite the pipeline as the 6-word story above.

22. Interview Questions & Answers

GrowthSchool

IN SIMPLE TERMS
The exact questions interviewers ask about CI/CD and GitOps, with crisp answers you can say out loud.

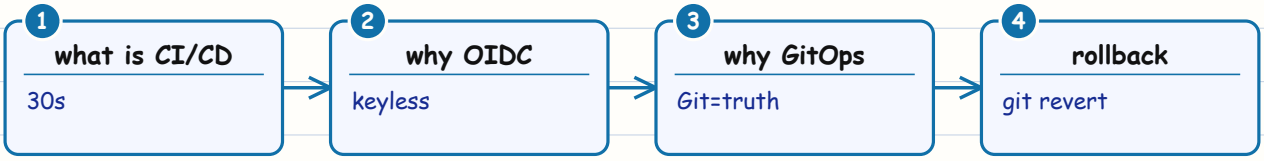
* Be Ready For

- Q: CI vs CD? A: CI builds+tests every change; CD ships it automatically.
- Q: Why OIDC over keys? A: short-lived, scoped tokens - no secrets to leak or rotate.

* The GitOps Ones

- Q: What triggers the deploy? A: a commit that changes the image tag in the manifests repo.
- Q: How do you roll back? A: git revert the tag commit; Argo CD syncs the cluster back.

Nail the interview



REAL-WORLD EXAMPLE

```
# 30-second whiteboard answer
commit -> Actions test+build -> OIDC -> ECR ->
tag commit -> Argo CD sync -> EKS rolling update -> live
```

REVISION SNAPSHOT

ASK YOURSELF	Answer all four questions above out loud in under a minute.
DO THIS	Whiteboard the flow while narrating each step.

23. Capstone: Build the Pipeline

GrowthSchool

IN SIMPLE TERMS
Build the whole ClassKira pipeline yourself, end to end, as your capstone project.

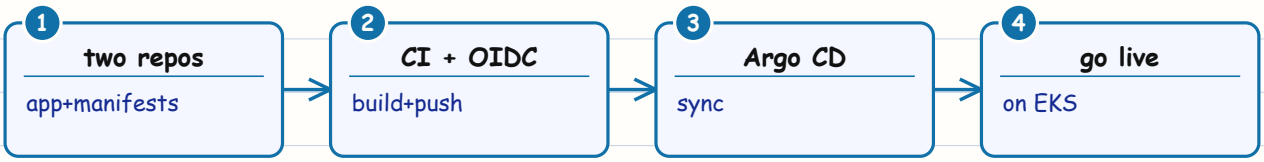
* Set It Up

- Create an app repo (with Dockerfile) and a manifests repo for ClassKira.
- Add a GitHub Actions workflow: test, build (SHA tag), OIDC login, push to ECR, commit the tag.

* Wire the Deploy

- Install Argo CD on an EKS cluster and point it at the manifests repo.
- Push a change and watch it flow all the way to a rolling update.

Capstone plan



STUDENT LAB
Deliverables: keyless OIDC (no stored keys), image tagged by SHA in ECR, an auto-committed tag
in the manifests repo, and an Argo CD app that syncs a rolling update to EKS end to end.

REVISION SNAPSHOT

ASK YOURSELF	Did one commit reach production with zero manual cluster commands?
DO THIS	Complete the capstone, then break a release and roll it back via git revert.