The DevOps Terminal Pack

18 cheatsheets — AWS, Kubernetes, Docker, Jenkins, bash, Python, Go and the terminal

Karandeep Singh

karandeepsingh.ca

Contents

bash	Bash Basics Cheatsheet
bash	Bash CLI One-Liners Cheatsheet
bash	Bash sed & awk Cheatsheet
containers	Docker Cheatsheet
containers	Kubernetes Cheatsheet
cost	Low-Cost Cloud Stack Cheatsheet
cost	MSK vs Kinesis Cheatsheet (Cost & When to Choose)
devops	Ansible Cheatsheet
devops	AWS CloudFormation Cheatsheet
devops	Jenkins Cheatsheet
languages	Go (Golang) Cheatsheet
python	Python Basics & CLI Cheatsheet
python	Python Boto3 Cheatsheet
python	Python Threading & Concurrency Cheatsheet
terminal	Amazon Linux 2 Terminal Cheatsheet
terminal	Atuin Cheatsheet
terminal	tmux Cheatsheet
terminal	Ubuntu Terminal Cheatsheet

Bash Basics Cheatsheet

I write Bash almost every day (deploy scripts, CI glue, one-off automation) and I still forget the exact parameter expansion syntax more often than I'd like to admit. This is the page I keep open: the Bash 5.x bits I actually use, condensed. For text-wrangling inside scripts, see my Bash sed & awk Cheatsheet.

Shebang & Safe Defaults

> ***Docs:*** *The Set Builtin, GNU Bash manual*

Start every script with this:

> Expand your knowledge with **Ansible Cheatsheet**

```
#!/usr/bin/env bash
set -euo pipefail
IFS=$'\n\t'
```

Option	Effect
`set -e`	Exit immediately on any command failure
`set -u`	Error on unset variables (catches typos)
`set -o pipefail`	A pipeline fails if *any* stage fails, not just the last
`set -x`	Print each command before running (debug mode)

```
bash -x script.sh          # debug an existing script without editing it
set +e                     # temporarily allow failures
some_flaky_command || true # tolerate a single failing command under set
```

Variables & Quoting

Docs: *Quoting, GNU Bash manual*

> *Deepen your understanding in* ***Ansible Cheatsheet***

```
name="karandeep"           # no spaces around =
readonly REGION="ca-central-1"
export PATH="$PATH:/opt/bin"
unset name

echo "$name"               # double quotes expand variables; use by defau
echo '$name'               # single quotes: literal, no expansion
echo "${name}_backup"      # braces to delimit the variable name
```

warning

Always quote variable expansions: `rm -rf "$dir/"`. Left unquoted, an empty or space-containing `$dir` can wipe the wrong path. `shellcheck` will catch most of these; run it on every script.

Parameter Expansion

Docs: *Shell Parameter Expansion, GNU Bash manual*

> *Explore this further in **Jenkins Cheatsheet***

Expansion	Meaning
`${var:-default}`	Use `default` if `var` is unset or empty
`${var:=default}`	Assign `default` to `var` if unset or empty
`${var:?message}`	Exit with `message` if `var` is unset or empty
`${var:+alt}`	Use `alt` only if `var` IS set
`${#var}`	Length of `var`
`${var#pattern}`	Remove shortest match from the front
`${var##pattern}`	Remove longest match from the front
`${var%pattern}`	Remove shortest match from the end
`${var%%pattern}`	Remove longest match from the end
`${var/a/b}`	Replace first `a` with `b`
`${var//a/b}`	Replace all `a` with `b`
`${var^^}` / `${var,,}`	Uppercase / lowercase
`${var:2:5}`	Substring: offset 2, length 5

```
file="app.tar.gz"
echo "${file%%.*}"     # app          (strip longest from end-matching .*
echo "${file%.*}"      # app.tar      (strip shortest)
echo "${file#*.}"      # tar.gz
path="/var/log/nginx/access.log"
echo "${path##*/}"     # access.log   (basename)
echo "${path%/*}"      # /var/log/nginx (dirname)
env="${DEPLOY_ENV:?DEPLOY_ENV must be set}"
```

Conditionals & `[[ ]]` Test Operators

Docs: *Bash Conditional Expressions, GNU Bash manual*

```
if [[ -f "$config" && "$env" == "prod" ]]; then
  echo "prod config found"
elif [[ "$env" =~ ^(dev|stage)$ ]]; then
  echo "non-prod"
else
  echo "unknown env" >&2
  exit 1
fi
```

Operator	True if…
`-f file`	Regular file exists
`-d dir`	Directory exists
`-e path`	Path exists (any type)
`-r` / `-w` / `-x`	Readable / writable / executable
`-s file`	File exists and is non-empty
`-z str`	String is empty
`-n str`	String is non-empty
`str1 == str2`	Strings equal (glob match if unquoted RHS)
`str =~ regex`	Regex match (capture groups in `BASH_REMATCH`)
`n1 -eq n2`	Integers equal (`-ne -lt -le -gt -ge`)
`f1 -nt f2`	`f1` newer than `f2`

Use `[[ ]]` over `[ ]`: no word splitting, and it supports `&&`, `||`, `=~`, and glob matching.

> Discover related concepts in **Go (Golang) Cheatsheet**

Loops

> ***Docs:*** *Looping Constructs, GNU Bash manual*

> *Uncover more details in **Bash sed & awk Cheatsheet***

```
for f in /etc/*.conf; do echo "$f"; done          # glob loop
for i in {1..5}; do echo "attempt $i"; done       # brace range
for ((i = 0; i < 10; i++)); do echo "$i"; done    # C-style

while read -r line; do                            # read a file safely
  echo "line: $line"
done < servers.txt

until curl -sf http://localhost:8080/health; do   # wait for a service
  sleep 2
done

while true; do date; sleep 5; done                # poll loop
break; continue                                   # work as expected
```

Functions

> ***Docs:*** *Shell Functions, GNU Bash manual*

```
deploy() {
  local env="$1"                 # local keeps vars scoped
  local version="${2:-latest}"   # default for optional args
  echo "deploying $version to $env"
  return 0                       # numeric status only; echo for data
}

deploy prod v1.2.3
result="$(deploy stage)"         # capture output via command substituti
```

`$1..$9 ${10}` positional args, `$#` count, `$@` all args (quote it: `"$@"`), `$0` script name, `$FUNCNAME` current function.

> Journey deeper into this topic with **AWS CloudFormation Cheatsheet**

Arrays & Associative Arrays

Docs: *Arrays, GNU Bash manual*

> *Enrich your learning with **Bash sed & awk Cheatsheet***

```
hosts=("web1" "web2" "db1")
hosts+=("cache1")                # append
echo "${hosts[0]}"               # first element
echo "${hosts[@]}"               # all elements
echo "${#hosts[@]}"              # length
echo "${hosts[@]:1:2}"           # slice: web2 db1
for h in "${hosts[@]}"; do ssh "$h" uptime; done

declare -A ports=([http]=80 [https]=443 [ssh]=22)
ports[dns]=53
echo "${ports[https]}"           # 443
for k in "${!ports[@]}"; do echo "$k -> ${ports[$k]}"; done

mapfile -t lines < servers.txt   # file into array, one line per element
```

Exit Codes

Docs: *Exit Status, GNU Bash manual*

> *Gain comprehensive insights from **Kubernetes Cheatsheet***

```
somecommand
echo "$?"                       # status of last command: 0 = success

grep -q "ERROR" app.log && echo "errors found"
systemctl is-active nginx || systemctl start nginx
command -v jq >/dev/null || { echo "jq required" >&2; exit 127; }
```

Code	Convention
`0`	Success
`1`	General error
`2`	Misuse of builtin / bad usage
`126`	Found but not executable
`127`	Command not found
`130`	Killed by Ctrl-C (SIGINT)

Traps

> ***Docs:*** *Bourne Shell Builtins (trap), GNU Bash manual*

```
tmpdir="$(mktemp -d)"
cleanup() { rm -rf "$tmpdir"; }
trap cleanup EXIT                          # runs on any exit, even set -

trap 'echo "failed at line $LINENO" >&2' ERR
trap 'echo "interrupted"; exit 130' INT TERM
```

`EXIT` is the workhorse: temp files, lock files, and spawned background jobs all get cleaned up no matter how the script dies.

> Master this concept through **Bash CLI One-Liners Cheatsheet**

Here-Docs & Here-Strings

Docs: *Redirections, GNU Bash manual*

> *Delve into specifics at **Bash sed & awk Cheatsheet***

```
cat <<EOF > /etc/myapp/config.yaml
env: $DEPLOY_ENV
region: ca-central-1
EOF

cat <<'EOF'                          # quoted delimiter: NO expansion
literal $HOME stays as-is
EOF

sudo tee /etc/motd <<EOF >/dev/null
Managed by automation. Do not edit.
EOF

grep "prod" <<< "$config_text"    # here-string: feed a variable to stdir
```

Command Substitution & Arithmetic

> **Docs:** *Shell Arithmetic, GNU Bash manual*

```
now="$(date -u +%Y-%m-%dT%H:%M:%SZ)"       # prefer $() over backticks
files="$(find /var/log -name '*.gz' | wc -l)"

(( count = files + 1 ))                     # arithmetic context
echo $(( 7 / 2 ))                           # 3 (integer division only)
(( count > 100 )) && echo "log buildup"
((i++)); ((total += batch))
echo $(( RANDOM % 6 + 1 ))                  # dice roll

awk 'BEGIN { printf "%.2f\n", 7/2 }'        # need floats? use awk or bc
```

Everything here compounds with the interactive tricks in the Bash CLI One-Liners Cheatsheet. The loop you prototype at the prompt becomes the script you commit.

> Deepen your understanding in **Ansible Cheatsheet**

0

Related Cheatsheets

- Bash sed & awk: substitution, address ranges, capture groups, and awk field handling for when plain Bash string manipulation runs out of road
- Bash CLI One-Liners: find/xargs, grep, jq, rsync, and process-triage pipelines worth running interactively before they ever become scripts
- Go (Golang): a different escape hatch. When a script needs real types, tests, and a single static binary, port it to Go

More Bash pages live in the Bash group, and everything else is in the cheatsheet library.

bash

Bash CLI One-Liners Cheatsheet

These are the one-liners I actually type on production boxes. Muscle-memory commands for finding files, chasing disk usage, poking APIs, and triaging misbehaving processes. When a one-liner turns into a script, the Bash Basics Cheatsheet covers making it robust.

find + xargs

Docs: *find(1) man page*

> *Expand your knowledge with **Bash Basics Cheatsheet***

```
find /var/log -name '*.gz' -mtime +30 -delete        # old compressed log
find . -type f -size +100M                           # large files
find . -name '*.sh' -exec chmod +x {} +              # batch chmod
find . -type f -name '*.yaml' | xargs grep -l 'image:'
find . -name '*.log' -print0 | xargs -0 rm --        # -print0/-0: safe w
find . -type d -empty -delete                        # remove empty dirs
find /etc -newer /etc/hostname -type f               # changed since that
ls *.png | xargs -P4 -I{} convert {} {}.webp         # -P4: 4 parallel jo
```

grep

Docs: *grep(1) man page*

> *Deepen your understanding in **Bash Basics Cheatsheet***

```
grep -r 'TODO' src/                    # recursive
grep -rn --include='*.tf' 'aws_s3' .   # line numbers, filter by glob
grep -E 'error|fatal|panic' app.log    # extended regex, alternation
grep -o 'req_id=[a-f0-9]*' app.log     # print only the match
grep -c '500' access.log               # count matching lines
grep -v '^#' config | grep -v '^$'     # strip comments and blanks
grep -B2 -A5 'Traceback' app.log       # context: 2 before, 5 after
grep -i -l 'password' -r . 2>/dev/null # files containing, case-insensiti
```

sort / uniq / wc Pipelines

Docs: *sort(1) man page*

```
sort file | uniq -c | sort -rn | head           # frequency count, desce
sort -t: -k3 -n /etc/passwd                     # numeric sort on field
sort -u hosts.txt                               # sort + dedupe in one
sort -h                                         # human sizes: 1K < 2M <
uniq -d sorted.txt                              # only duplicated lines
wc -l < file                                    # line count, no filenam
comm -23 <(sort a.txt) <(sort b.txt)            # lines in a but not b
awk '{print $1}' access.log | sort | uniq -c | sort -rn | head   # top IF
```

For heavier field surgery in these pipelines, see the Bash sed & awk Cheatsheet.

Explore this further in **Python Basics & CLI Cheatsheet**

tar / gzip / zstd

Docs: *tar(1) man page*

> *Discover related concepts in **tmux Cheatsheet***

```
tar czf app.tar.gz app/                # create gzip archive
tar xzf app.tar.gz -C /opt/           # extract to a directory
tar tzf app.tar.gz                    # list contents without extracting
tar czf - /data | ssh host 'cat > /backup/data.tar.gz'   # archive over s
tar --exclude='.git' --exclude='node_modules' -czf src.tar.gz .
tar caf app.tar.zst app/              # -a: pick compressor by extension
zstd -T0 bigfile                      # zstd on all cores; much faster th
gzip -dk file.gz                      # decompress, keep original
zcat access.log.gz | grep ' 500 '     # search inside without extracting
```

curl

Docs: *curl manpage on curl.se*

> *Uncover more details in **Bash Basics Cheatsheet***

```
curl -I https://example.com                    # headers only
curl -sfSL https://example.com/health          # silent, fail on HTTP
curl -o out.bin -L https://example.com/dl      # follow redirects, sav
curl -w '%{http_code} %{time_total}s\n' -o /dev/null -s https://api.examp
curl -X POST https://api.example.com/v1/deploys \
  -H 'Content-Type: application/json' \
  -H "Authorization: Bearer $TOKEN" \
  -d '{"service": "web", "version": "1.4.2"}'
curl --retry 5 --retry-delay 2 --retry-all-errors -sf https://flaky.examp
curl -u user:pass ftp://host/file              # basic auth
curl -x http://proxy:3128 https://example.com  # via proxy
```

jq Basics

Docs: *jq Manual on jqlang.org*

> *Journey deeper into this topic with **Python Basics & CLI Cheatsheet***

```
curl -s https://api.example.com/pods | jq .              # pretty-print
jq '.items[].metadata.name' pods.json                    # extract a fiel
jq -r '.token' resp.json                                 # -r: raw, no qu
jq '.items | length' pods.json                           # count
jq '.[] | select(.status == "failed") | .id' jobs.json   # filter
jq '.[] | {name: .metadata.name, ip: .status.podIP}'     # reshape
jq -r '.[] | [.name, .cpu] | @tsv' nodes.json            # TSV for awk/cc
jq -n --arg env "$ENV" '{environment: $env}'             # build JSON frc
jq 'keys' obj.json                                       # what fields ex
```

Disk & Process Triage

Docs: *ps(1) man page*

> *Enrich your learning with **Python Threading & Concurrency Cheatsheet***

```
df -h                                    # filesystem usage
du -sh /var/* | sort -rh | head          # what's eating this disk
du -xh --max-depth=2 / | sort -rh | head -20   # -x: don't cross filesyst
ncdu /var                                # interactive, if installed

ps aux --sort=-%mem | head               # top memory hogs
ps aux --sort=-%cpu | head               # top CPU hogs
ps -ef --forest                          # process tree
top -o %MEM                              # top sorted by memory (htop if yc
lsof -p 1234                             # files open by PID
lsof +D /var/log                         # who has files open under a dir
lsof -i :8080                            # what's listening on 8080
kill -TERM 1234 && sleep 5 && kill -0 1234 && kill -KILL 1234
```

tip

“Disk full” but `du` doesn’t add up? A deleted file is still held open by a process. `lsof +L1` lists them. Restart the offender (usually a log-hoarding daemon) and the space comes back.

Network

Docs: *ss(8) man page*

> *Gain comprehensive insights from **Docker Cheatsheet***

```
ss -tlnp                                # listening TCP sockets + owning p
ss -s                                   # socket summary (state counts)
dig +short example.com                  # just the A record
dig example.com MX                      # specific record type
dig @1.1.1.1 example.com                # query a specific resolver
dig -x 93.184.216.34                    # reverse lookup
nc -zv host 5432                        # is that port open?
nc -l 9000                              # quick listener for testing
timeout 3 bash -c '</dev/tcp/host/443' && echo open   # no nc? pure bash
curl -sv telnet://host:6379 </dev/null # handshake check via curl
ip -br addr; ip route                   # interfaces and routing at a glar
```

ssh / scp / rsync

> **Docs:** *rsync(1) man page*

```
ssh -J bastion.example.com user@10.0.2.15        # jump host
ssh -L 5432:db.internal:5432 bastion             # local port forward
ssh host 'sudo journalctl -u nginx -n 50'        # run remote command
ssh-copy-id user@host                            # install your key

scp file.tar.gz user@host:/tmp/                  # simple copy
scp -r user@host:/var/log/app ./logs/            # recursive pull

rsync -avz --progress ./site/ user@host:/var/www/site/   # sync (mind the
rsync -avz --delete src/ dst/                    # exact mirror; deletes
rsync -avzn --delete src/ dst/                   # -n: ALWAYS dry-run fir
rsync -avz -e 'ssh -J bastion' data/ user@internal:/data/
```

Trailing slash on the source means “contents of”; no slash copies the directory itself.

> Master this concept through **Bash Basics Cheatsheet**

History Tricks

Docs: *History Expansion, GNU Bash manual*

> *Delve into specifics at **Bash Basics Cheatsheet***

Trick	Effect
`!!`	Re-run the last command (`sudo !!` is the classic)
`!$`	Last argument of the previous command
`!*`	All arguments of the previous command
`!grep`	Re-run the most recent command starting with `grep`
`^old^new`	Re-run last command with first `old` replaced by `new`
`Ctrl-r`	Reverse-search history incrementally (press again to cycle)
`Alt-.`	Insert last argument, keep pressing to go further back
`$_`	Last argument, as a variable in scripts

```
mkdir -p /opt/app/releases/v2 && cd !$    # !$ expands to the new path
history | awk '{print $2}' | sort | uniq -c | sort -rn | head   # your tc
```

Job Control

> ***Docs:*** *Job Control, GNU Bash manual*

```
long_build.sh &                # run in background
jobs -l                        # list jobs with PIDs
fg %1                          # bring job 1 to foreground
bg %1                          # resume a Ctrl-Z'd job in background
kill %2                        # kill by job number
Ctrl-Z                         # suspend current foreground job

nohup ./migrate.sh > migrate.log 2>&1 &   # survive hangup on logout
./worker.sh & disown           # background + detach from the shell
wait                           # block until all background jobs finish
wait -n                        # until any one finishes (bash 4.3+)
```

For anything long-running over ssh, prefer `tmux` over `nohup`; you get your session back, output included.

> Deepen your understanding in **Bash Basics Cheatsheet**

0

Related Cheatsheets

- Bash Basics: `set -euo pipefail`, parameter expansion, traps, and arrays, everything needed to turn these one-liners into scripts that survive production
- Bash sed & awk: heavier text processing, meaning in-place edits, column sums, extracting text between markers, and the GNU vs BSD gotchas
- tmux: session, window, and pane keybindings plus scripted layouts, so long-running pipelines outlive a dropped SSH connection

There's more in the Bash group, and the whole cheatsheet library is one level up.

bash

Bash sed & awk Cheatsheet

Half of DevOps is reshaping text: log files, CSVs, config files, command output. `sed` and `awk` are the two tools I reach for constantly, and somehow I still re-derive the syntax from scratch every time. So I wrote it down. If your shell fundamentals are rusty, start with the Bash Basics Cheatsheet first.

sed: Substitution

> ***Docs:*** *GNU sed manual*
>
> > *Discover related concepts in **Bash Basics Cheatsheet***
> >
> > *Explore this further in **Bash Basics Cheatsheet***
> >
> > *Expand your knowledge with **Bash Basics Cheatsheet***

```
sed 's/foo/bar/' file             # replace first foo per line
sed 's/foo/bar/g' file            # replace ALL per line
sed 's/foo/bar/2' file            # only the 2nd occurrence per line
sed 's/foo/bar/gi' file           # global + case-insensitive
sed 's|/var/log|/mnt/log|g' file # any delimiter, handy for paths
sed -E 's/[0-9]+/N/g' file        # -E: extended regex (no backslash escap
sed -n 's/foo/bar/p' file         # print only lines where a change happer
```

sed: Deletion & Insertion

Docs: *sed(1) man page*

> *Deepen your understanding in **AWS CloudFormation Cheatsheet***

```
sed '/^#/d' config.conf          # delete comment lines
sed '/^$/d' file                 # delete blank lines
sed '3d' file                    # delete line 3
sed '2,5d' file                  # delete lines 2-5
sed '$d' file                    # delete last line

sed '3i\inserted before line 3' file
sed '3a\appended after line 3'  file
sed '/^\[nginx\]/a\worker_processes auto;' app.ini   # append after a mat
sed '/pattern/c\replacement line' file               # change whole line
```

sed: In-Place Editing (GNU vs BSD Gotcha)

Docs: *GNU sed manual*

> *Discover related concepts in **Bash Basics Cheatsheet***

> *Explore this further in **Bash Basics Cheatsheet***

> *Expand your knowledge with **Bash Basics Cheatsheet***

```
sed -i 's/foo/bar/g' file        # GNU sed (Linux): edit in place
sed -i.bak 's/foo/bar/g' file    # in place + keep file.bak backup
```

warning

On macOS (BSD sed), `-i` **requires** an argument: `sed -i '' 's/foo/bar/g' file`. GNU's `sed -i 's/...'` fails on BSD with "invalid command code", and BSD's `sed -i ''` breaks GNU. For cross-platform scripts, use `sed -i.bak ... && rm file.bak`, or install `gsed` via Homebrew.

sed: Addresses & Ranges

Docs: *GNU sed manual*

> *Discover related concepts in **Bash Basics Cheatsheet***

> *Explore this further in **Bash Basics Cheatsheet***

> *Expand your knowledge with **Bash Basics Cheatsheet***

Address	Applies command to…
`5`	Line 5
`$`	Last line
`2,10`	Lines 2 through 10
`/regex/`	Lines matching regex
`/start/,/end/`	From a `start` match to the next `end` match
`0~3`	Every 3rd line (GNU)
`5,$`	Line 5 to end of file
`/regex/!`	Lines NOT matching

```
sed -n '10,20p' file                    # print lines 10-20
sed -n '/BEGIN CERT/,/END CERT/p' pem   # extract a certificate block
sed '/^#/!s/foo/bar/' file              # substitute only on non-comments
sed '1,/^$/d' mail.txt                  # delete headers (up to first bla
```

sed: Capture Groups & Backreferences

Docs: *GNU sed manual*

> *Discover related concepts in **Bash Basics Cheatsheet***

> *Explore this further in **Bash Basics Cheatsheet***

> *Expand your knowledge with **Bash Basics Cheatsheet***

```
sed -E 's/(user)-([0-9]+)/\2-\1/' file        # swap: user-42 -> 42-user
sed -E 's/^([a-z]+)=(.*)$/export \1="\2"/'    # key=val -> export key="va
sed -E 's/(error)/[\1]/gi' app.log            # wrap matches; \1 refers b
sed -E 's/([0-9]{4})-([0-9]{2})-([0-9]{2})/\3\/\2\/\1/'  # date reshuffle
sed 's/.*"id": *"\([^"]*\)".*/\1/' resp.json  # basic regex needs \( \)
```

`&` in the replacement means “the whole match”: `sed -E 's/[0-9]+/<&>/g'`.

> Uncover more details in **Bash Basics Cheatsheet**

awk: Fields & Built-in Variables

> ***Docs:** GNU gawk manual*

> *Gain comprehensive insights from **Bash Basics Cheatsheet***

awk splits each line into fields: `$1`, `$2`, … `$NF`. `$0` is the whole line.

> Journey deeper into this topic with **Bash Basics Cheatsheet**

Variable	Meaning
`NR`	Current line (record) number
`NF`	Number of fields on this line
`FS`	Input field separator (default: runs of whitespace)
`OFS`	Output field separator (default: single space)
`RS` / `ORS`	Input / output record separator
`FILENAME`	Current input file
`FNR`	Line number within the current file

```
awk '{print $1, $3}' access.log         # fields 1 and 3
awk '{print $NF}' file                  # last field
awk '{print $(NF-1)}' file              # second-to-last
awk -F: '{print $1}' /etc/passwd        # -F sets FS: usernames
awk -F',' -v OFS='\t' '{$1=$1; print}' data.csv   # CSV -> TSV
awk 'NR==5' file                        # print line 5
awk 'NR>1' data.csv                     # skip the header row
```

awk: Patterns & Actions

> ***Docs:*** *gawk(1) man page*

The model is `pattern { action }`. The action runs only on matching lines, and a pattern alone means `{ print }`.

> Enrich your learning with **Bash Basics Cheatsheet**

```
awk '/ERROR/' app.log                       # grep-like
awk '$3 > 500' access.log                   # numeric comparison on a fie
awk '$1 == "nginx" && $4 ~ /5[0-9][0-9]/'   # combine conditions, regex c
awk 'NF == 0 {next} {print $1}' file        # skip blank lines
awk 'length($0) > 120' script.sh            # lines longer than 120 chars
awk '!seen[$0]++' file                      # dedupe, preserving order
```

awk: BEGIN / END

Docs: *GNU gawk manual*

> *Gain comprehensive insights from **Bash Basics Cheatsheet***

```
awk 'BEGIN {FS=","; print "host,status"} $2=="down" {print $1","$2}' host
awk '{total += $5} END {print "bytes:", total}' access.log
awk 'END {print NR}' file                   # line count, like wc -l
awk 'BEGIN {for (i=1; i<=5; i++) print "srv" i}'   # no input file needed
```

awk: Associative Arrays

Docs: *GNU gawk manual*

> *Gain comprehensive insights from **Bash Basics Cheatsheet***

```
# count requests per IP, sorted by count
awk '{count[$1]++} END {for (ip in count) print count[ip], ip}' access.lc

# sum bytes per status code
awk '{bytes[$9] += $10} END {for (s in bytes) print s, bytes[s]}' access.

# join two files on field 1 (like a lookup table)
awk 'NR==FNR {owner[$1]=$2; next} {print $0, owner[$1]}' owners.txt serve
```

The `NR==FNR` idiom is worth memorizing: true only while awk reads the *first* file, so you load it into an array, then enrich the second.

> Master this concept through **Bash Basics Cheatsheet**

awk: printf

> ***Docs:*** *GNU gawk manual*
>
> > *Gain comprehensive insights from **Bash Basics Cheatsheet***

```
awk '{printf "%-20s %8d\n", $1, $2}' file      # left-pad name, right-pac
awk '{printf "%.2f%%\n", $1/$2*100}' file      # floats; awk does real ma
awk 'BEGIN {printf "%s\t%s\n", "col1", "col2"}'
```

`%s` string, `%d` int, `%f` float, `%x` hex, `%-10s` left-justified width 10. `\n` is not automatic, so add it.

> Delve into specifics at **Bash Basics Cheatsheet**

Common One-Liners

Docs: *GNU gawk manual*

> *Gain comprehensive insights from **Bash Basics Cheatsheet***

Task	Command
Sum column 3	`awk '{s+=$3} END {print s}' file`
Average column 2	`awk '{s+=$2} END {print s/NR}' file`
Dedupe keeping order	`awk '!seen[$0]++' file`
Extract between markers	`sed -n '/START/,/END/p' file`
Pick CSV columns 1 & 4	`awk -F',' '{print $1","$4}' file.csv`
Strip trailing whitespace	`sed -i 's/[ \t]*$//' file`
Number all lines	`awk '{print NR": "$0}' file`
Print file between line 100-110	`sed -n '100,110p' file`
Comment out a config line	`sed -i '/^listen 80/s/^/#/' nginx.conf`
Top talkers in a log	`awk '{print $1}' log | sort | uniq -c | sort -rn | head`

More pipeline building blocks (`sort` , `uniq` , `xargs` , `jq`) live in the Bash CLI One-Liners Cheatsheet.

> Deepen your understanding in **AWS CloudFormation Cheatsheet**

0

When to Use What

Tool	Reach for it when…
`grep`	You only need to *find* lines (fastest, simplest)
`cut`	Fixed single-char delimiter, simple column extraction
`sed`	Line-oriented *editing*: substitute, delete, insert, in-place
`awk`	Fields, arithmetic, aggregation, conditions across columns
Python	Nested data, JSON, anything a one-liner can't express cleanly

Rule of thumb: `grep` to find, `sed` to change, `awk` to compute. If your awk program grows past a few lines, it's a script. Commit it with a shebang and tests.

> Deepen your understanding in **AWS CloudFormation Cheatsheet**

1

Related Cheatsheets

> Bash Basics: quoting, conditionals, loops, and here-docs, the scripting frame these sed and awk snippets slot into

- Bash CLI One-Liners: the rest of the pipeline toolkit (find, `grep -o`, `sort | uniq -c`, jq, and curl recipes)
- Ubuntu Terminal: apt, systemd, journalctl, and ufw, which is where the logs and config files you'll run these one-liners against actually live

The rest of the shell pages are in the Bash group; the full cheatsheet library has everything else.

containers

Docker Cheatsheet

I use Docker more than any other tool and still end up looking up the same flags every week, so I keep this page open in a tab. Just the commands I actually run, nothing padded out. If you run containers in production, pair this with my Kubernetes Cheatsheet, because most of these images end up in a cluster eventually.

Image Lifecycle

> ***Docs:*** *docker image commands (Docker Docs)*
>
> > *Expand your knowledge with **Bash CLI One-Liners Cheatsheet***

```
docker build -t myapp:1.0 .              # Build from Dockerfile in curr
docker build -t myapp:1.0 -f prod.Dockerfile .   # Custom Dockerfile name
docker build --no-cache -t myapp:1.0 .    # Ignore build cache
docker tag myapp:1.0 registry.example.com/team/myapp:1.0
docker push registry.example.com/team/myapp:1.0
docker pull nginx:1.27-alpine
docker images                             # List local images
docker image inspect myapp:1.0            # Full metadata (JSON)
docker history myapp:1.0                  # Layer-by-layer sizes
docker save myapp:1.0 -o myapp.tar        # Export image to tarball
docker load -i myapp.tar                  # Import from tarball
```

Command	What it does
`docker build -t name:tag .`	Build an image from a Dockerfile
`docker tag src tgt`	Add another name/tag to an image
`docker push name:tag`	Upload to a registry
`docker pull name:tag`	Download from a registry
`docker rmi name:tag`	Delete a local image
`docker login registry.example.com`	Authenticate to a registry

Running Containers

> ***Docs:*** *docker container run reference on Docker Docs*

```
docker run -d --name web -p 8080:80 nginx:1.27-alpine
docker run --rm -it ubuntu:24.04 bash     # Throwaway interactive contair
docker run -d --restart unless-stopped -e TZ=America/Edmonton myapp:1.0
```

Run Flags I Use Constantly

Flag	Meaning	Example
`-d`	Detached (background)	`docker run -d nginx`
`-p`	Publish port host:container	`-p 8080:80`
`-v`	Mount volume or bind mount	`-v pgdata:/var/lib/postgresql/data`
`-e`	Set environment variable	`-e POSTGRES_PASSWORD=secret`
`--env-file`	Load env vars from file	`--env-file .env`
`--network`	Attach to a network	`--network backend`
`--rm`	Auto-remove on exit	`docker run --rm alpine echo hi`
`--restart`	Restart policy	`--restart unless-stopped`
`-it`	Interactive TTY	`-it ubuntu bash`
`--name`	Name the container	`--name web`
`--memory` / `--cpus`	Resource limits	`--memory 512m --cpus 1.5`

Restart policies: `no` (default), `on-failure[:max-retries]`, `always`, `unless-stopped`. I use `unless-stopped` on standalone hosts so containers survive reboots but respect a manual `docker stop`.

> Deepen your understanding in **Kubernetes Cheatsheet**

Exec, Logs, Inspect, Stats

Docs: *docker container commands (Docker Docs)*

> *Explore this further in **Atuin Cheatsheet***

```
docker ps                                 # Running containers
docker ps -a                              # Include stopped ones
docker exec -it web sh                    # Shell into a running containe
docker exec web env                       # One-off command, no TTY
docker logs web                           # Dump logs
docker logs -f --tail 100 web             # Follow, last 100 lines
docker logs --since 15m web               # Logs from the last 15 minutes
docker inspect web                        # Full JSON: mounts, IP, env, s
docker inspect -f '{{ .NetworkSettings.IPAddress }}' web
docker stats                              # Live CPU/mem/net per containe
docker top web                            # Processes inside the containe
docker cp web:/etc/nginx/nginx.conf ./    # Copy files out (or in)
docker stop web && docker rm web          # Stop then remove
docker kill web                           # SIGKILL immediately
```

Dockerfile Essentials

Docs: *Dockerfile reference on Docker Docs*

Instruction	Purpose
`FROM`	Base image (first non-comment line; one per stage)
`WORKDIR`	Set working directory (creates it if missing)
`COPY`	Copy files from build context into image
`RUN`	Execute a command at build time (new layer)
`ENV`	Set persistent environment variable
`ARG`	Build-time variable (not in final image env)
`EXPOSE`	Document the listening port (metadata only)
`USER`	Run as a non-root user (drop root)
`ENTRYPOINT`	Fixed executable
`CMD`	Default args (overridable at `docker run`)
`HEALTHCHECK`	Command Docker runs to test container health

Multi-Stage Build (Go example)

```
# Stage 1: build
FROM golang:1.24-alpine AS builder
WORKDIR /src
COPY go.mod go.sum ./
RUN go mod download
COPY . .
RUN CGO_ENABLED=0 go build -ldflags="-s -w" -o /app ./cmd/server

# Stage 2: minimal runtime
FROM alpine:3.21
RUN adduser -D -u 10001 appuser
COPY --from=builder /app /app
USER appuser
EXPOSE 8080
ENTRYPOINT ["/app"]
```

The final image here is a few MB instead of ~800MB, since the entire Go toolchain stays behind in the builder stage. If you write Go, my Go (Golang) Cheatsheet covers the build flags used above.

.dockerignore

Keep the build context small. Docker uploads everything in the context to the daemon before building.

> Discover related concepts in **Go (Golang) Cheatsheet**

```
.git
node_modules
tmp/
*.log
.env
Dockerfile
docker-compose*.yml
```

Volumes & Networks

Docs: *Docker volumes docs*

```
docker volume create pgdata
docker volume ls
docker volume inspect pgdata              # Shows host mountpoint
docker volume rm pgdata

docker network create backend             # User-defined bridge (has DNS)
docker network ls
docker network inspect backend
docker network connect backend web       # Attach a running container
docker network disconnect backend web
```

Containers on the same user-defined network resolve each other by container name, so `db:5432` just works. The default `bridge` network does **not** do name resolution; always create your own.

> Uncover more details in **Kubernetes Cheatsheet**

```
# Bind mount (host path) vs named volume
docker run -v $(pwd)/conf:/etc/nginx/conf.d:ro nginx   # bind, read-only
docker run -v pgdata:/var/lib/postgresql/data postgres # named volume
```

Docker Compose

Docs: *Docker Compose docs*

> *Journey deeper into this topic with **Kubernetes Cheatsheet***

```
docker compose up -d                    # Start stack in background
docker compose up -d --build            # Rebuild images first
docker compose down                     # Stop and remove containers/ne
docker compose down -v                  # ...and volumes (destructive!)
docker compose ps                       # Status of services
docker compose logs -f api              # Follow one service's logs
docker compose exec api sh             # Shell into a service
docker compose restart worker          # Restart one service
docker compose pull                     # Refresh images
```

Sample compose.yaml

```yaml
services:
  api:
    build: .
    ports:
      - "8080:8080"
    environment:
      DATABASE_URL: postgres://app:secret@db:5432/app
    depends_on:
      db:
        condition: service_healthy
    restart: unless-stopped

  db:
    image: postgres:16-alpine
    environment:
      POSTGRES_USER: app
      POSTGRES_PASSWORD: secret
      POSTGRES_DB: app
    volumes:
      - pgdata:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U app"]
      interval: 5s
      timeout: 3s
      retries: 5

volumes:
  pgdata:
```

Cleanup

Docs: *docker system prune (Docker Docs)*

> *Enrich your learning with **Bash Basics Cheatsheet***

```
docker system df                          # What's eating disk
docker system prune                       # Stopped containers, dangling
docker system prune -a --volumes          # Everything unused, incl. volu
docker image prune                        # Dangling images only
docker image prune -a                     # All images not used by a cont
docker container prune                    # All stopped containers
docker volume prune                       # Unused volumes
docker builder prune                      # Build cache
```

warning

`docker system prune -a --volumes` deletes every image not attached to a running container and every unused volume, including database data. On a shared host, run `docker system df` first and prune selectively.

Debugging Tips

Docs: *docker CLI reference on Docker Docs*

> *Gain comprehensive insights from **Kubernetes Cheatsheet***

```
docker logs --details web                       # Extra log metadata
docker inspect -f '{{ .State.ExitCode }}' web   # Why did it die?
docker inspect -f '{{ json .State.Health }}' web | jq   # Healthcheck his
docker events --filter container=web            # Live daemon events
docker run --rm -it --entrypoint sh myapp:1.0   # Bypass a crashing er
docker exec -it web sh -c 'wget -qO- localhost:8080/healthz'   # Test frc
docker run --rm --network container:web nicolaka/netshoot      # Netshoot
docker diff web                                 # Files changed since
```

> **Exit code 137** = OOM-killed (or SIGKILL). Check `--memory` limits and `dmesg`.

> **Exit code 126/127** = command not executable / not found, usually a bad `ENTRYPOINT` or a missing shell in a distroless image.

> Container exits immediately? The main process finished. Run with `-it ... sh` or check that your process runs in the foreground (no daemon mode inside containers).

> Can't reach a service? Confirm `-p` mapping with `docker port web`, and remember published ports bind to the host; container-to-container traffic uses the internal port on a shared network.

Related Cheatsheets

> Kubernetes is the next step for these images: kubectl verbs, Deployment and Service YAML skeletons, rollouts, and crash-loop triage

> Jenkins covers the declarative pipelines that build, tag, and push these images, including docker agents and credential binding

> Ubuntu Terminal handles the host side: systemd units, journal logs, disk cleanup, and the ufw firewall in front of your containers

The rest of my container notes are in the Containers group, and everything else lives in the cheatsheet library.

Kubernetes Cheatsheet

I spend a good chunk of every week inside kubectl and still blank on flag combinations at the worst moments, so this page is my working memory. Commands, YAML skeletons, and the debugging moves I actually use on real clusters. Everything here assumes your images are already built and pushed; my Docker Cheatsheet covers that half of the workflow.

kubectl Basics

> ***Docs:*** *kubectl reference on kubernetes.io*
>
> > *Expand your knowledge with **Python Basics & CLI Cheatsheet***

```
kubectl get pods                          # Pods in current namespace
kubectl get pods -n kube-system           # Specific namespace
kubectl get pods -A                       # All namespaces
kubectl get pods -o wide                  # Node, IP, readiness detail
kubectl get deploy web -o yaml            # Full live manifest
kubectl describe pod web-7d4b9c6f5-x2k8j  # Events, conditions, mounts
kubectl logs web-7d4b9c6f5-x2k8j          # Container logs
kubectl logs -f deploy/web                # Follow logs via the Deploymer
kubectl logs web-... -c sidecar --previous   # Prior crashed container
kubectl exec -it web-... -- sh            # Shell into a pod
kubectl apply -f manifest.yaml            # Create/update declaratively
kubectl apply -f k8s/ -R                  # Whole directory, recursive
kubectl delete -f manifest.yaml
kubectl delete pod web-... --grace-period=0 --force   # Last resort
kubectl diff -f manifest.yaml             # Preview what apply would char
```

Flag	Meaning
`-n <ns>`	Target a namespace
`-A`	All namespaces
`-o wide`	Extra columns (node, IP)
`-o yaml` / `-o json`	Full object output
`-o jsonpath='{...}'`	Extract specific fields
`-l app=web`	Filter by label selector
`--watch` / `-w`	Stream changes live
`--dry-run=client -o yaml`	Generate YAML without creating

Context & Namespace Switching

> ***Docs:*** *Kubernetes docs on configuring access to multiple clusters*

```
kubectl config get-contexts                 # List clusters/contexts
kubectl config current-context
kubectl config use-context prod-cluster
kubectl config set-context --current --namespace=payments   # Default ns
```

I alias these constantly. `kubectx` and `kubens` do the same job with less typing if you can install them.

> Deepen your understanding in **Bash Basics Cheatsheet**

Core Resource YAML Skeletons

Docs: *Kubernetes Deployment docs*

> *Explore this further in **Python Basics & CLI Cheatsheet***

Deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
  labels:
    app: web
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: web
          image: registry.example.com/team/web:1.4.2
          ports:
            - containerPort: 8080
          envFrom:
            - configMapRef:
                name: web-config
            - secretRef:
                name: web-secrets
          resources:
            requests:
              cpu: 100m
              memory: 128Mi
            limits:
              memory: 256Mi
          readinessProbe:
            httpGet:
              path: /healthz
              port: 8080
            initialDelaySeconds: 5
            periodSeconds: 10
```

Service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: web
spec:
  type: ClusterIP        # NodePort / LoadBalancer for external
  selector:
    app: web
  ports:
    - port: 80
      targetPort: 8080
```

ConfigMap & Secret

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: web-config
data:
  LOG_LEVEL: info
  FEATURE_FLAGS: "beta-ui,new-checkout"
---
apiVersion: v1
kind: Secret
metadata:
  name: web-secrets
type: Opaque
stringData:            # stringData = plain text; k8s base64-encodes it
  DATABASE_URL: postgres://app:secret@db:5432/app
```

```
kubectl create secret generic web-secrets \
  --from-literal=DATABASE_URL=postgres://... --dry-run=client -o yaml
```

Ingress

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: web
  annotations:
    cert-manager.io/cluster-issuer: letsencrypt-prod
spec:
  ingressClassName: nginx
  tls:
    - hosts: [app.example.com]
      secretName: web-tls
  rules:
    - host: app.example.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: web
                port:
                  number: 80
```

Rollouts

Docs: *Kubernetes docs on updating and rolling back Deployments*

> *Discover related concepts in **Docker Cheatsheet***

```
kubectl rollout status deploy/web         # Block until rollout completes
kubectl rollout history deploy/web        # Revision list
kubectl rollout history deploy/web --revision=3   # Details of one revisi
kubectl rollout undo deploy/web           # Roll back to previous
kubectl rollout undo deploy/web --to-revision=2
kubectl rollout restart deploy/web        # Rolling restart (picks up cor
kubectl set image deploy/web web=registry.example.com/team/web:1.4.3
```

Scaling & Autoscaling

> **Docs:** *Horizontal Pod Autoscaling on kubernetes.io*

```
kubectl scale deploy/web --replicas=5
kubectl autoscale deploy/web --min=3 --max=10 --cpu-percent=70
kubectl get hpa
kubectl top pods                          # Needs metrics-server
kubectl top nodes
```

`kubectl autoscale` generates an `autoscaling/v2` HorizontalPodAutoscaler behind the scenes. Export it with `kubectl get hpa web -o yaml` if you want to commit it declaratively (that is where memory and custom-metric targets go too).

> Uncover more details in **Low-Cost Cloud Stack Cheatsheet**

Port-Forward

> **Docs:** *Kubernetes port-forwarding docs*

> *Journey deeper into this topic with **Amazon Linux 2 Terminal Cheatsheet***

```
kubectl port-forward svc/web 8080:80              # localhost:8080 → serv
kubectl port-forward deploy/web 8080:8080
kubectl port-forward pod/db-0 5432:5432 -n data   # Handy for DB clients
```

Debugging

Docs: *Monitoring, logging, and debugging on kubernetes.io*

> *Enrich your learning with **Docker Cheatsheet***

```
kubectl get events --sort-by=.lastTimestamp       # Recent cluster events
kubectl get events --field-selector involvedObject.name=web-7d4b9c6f5-x2k
kubectl describe pod web-...                      # Events section at the
kubectl debug -it web-... --image=busybox --target=web   # Ephemeral cont
kubectl debug node/worker-3 -it --image=ubuntu    # Debug pod on a node
kubectl get pod web-... -o jsonpath='{.status.containerStatuses[0].lastSt
```

CrashLoopBackOff Triage

1. `kubectl logs web-... --previous`. The crash reason is almost always here.
2. `kubectl describe pod web-...`, then check Events for OOMKilled, failed probes, image pull errors.
3. Exit code 137 → OOMKilled: raise memory limits or fix the leak.
4. Failing liveness probe → the app is slow to start; raise `initialDelaySeconds` or switch to a startup probe.

5. Still stuck? Override the entrypoint to keep the pod alive and poke around: `kubectl debug` with an ephemeral container, or set `command: ["sleep", "3600"]` in a copy of the pod.

tip

`ImagePullBackOff` is nearly always one of three things: a typo in the image tag, a missing `imagePullSecrets` entry for a private registry, or the image genuinely never got pushed. Check `kubectl describe pod`; the event text tells you which.

Labels & Selectors

Docs: *Kubernetes labels and selectors docs*

```
kubectl get pods -l app=web                          # Equality selector
kubectl get pods -l 'env in (staging,prod)'          # Set-based selector
kubectl get pods -l app=web,tier!=cache
kubectl label pod web-... debug=true                 # Add a label
kubectl label pod web-... debug-                     # Remove it
kubectl get pods --show-labels
```

Services, Deployments, HPAs, and NetworkPolicies all match pods by labels. A Service with `selector: app: web` routes to every ready pod carrying that label, no matter what created it.

Resource Requests & Limits

Docs: *Kubernetes docs on resource management for Pods and containers*

Setting	Effect
`requests.cpu` / `requests.memory`	What the scheduler reserves; pod won't schedule without capacity
`limits.cpu`	Throttled above this (no kill)
`limits.memory`	OOM-killed above this (exit 137)

My defaults: always set requests (scheduling accuracy), set a memory limit (protect neighbours), and think twice before CPU limits, because throttling shows up as mystery latency.

> Master this concept through **Docker Cheatsheet**

Handy One-Liners

> ***Docs:*** *kubectl quick reference on kubernetes.io*

Task	Command
Generate Deployment YAML	`kubectl create deploy web --image=nginx --dry-run=client -o yaml`
Throwaway debug pod	`kubectl run tmp --rm -it --image=busybox -- sh`
Decode a secret	`kubectl get secret web-secrets -o jsonpath='{.data.DATABASE_URL}'`
Pods on one node	`kubectl get pods -A --field-selector spec.nodeName=worker-3`
Restart counts	`kubectl get pods -o custom-columns=NAME:.metadata.name,RESTARTS:.`
All images running in cluster	`kubectl get pods -A -o jsonpath='{range .items[*]}{.spec.containe`
Drain a node	`kubectl drain worker-3 --ignore-daemonsets --delete-emptydir-data`
Uncordon it after	`kubectl uncordon worker-3`
Delete all Evicted pods	`kubectl delete pods -A --field-selector status.phase=Failed`
Watch a rollout live	`kubectl get pods -l app=web -w`

Most of my controllers and operators are written in Go. The client-go ecosystem is a big reason I keep my Go (Golang) Cheatsheet nearby when working against the Kubernetes API.

> Delve into specifics at **Bash CLI One-Liners Cheatsheet**

Related Cheatsheets

- Docker: multi-stage builds, image tagging, volumes, and compose. Every Pod spec on this page starts there
- Go (Golang): the language kubectl and most operators are written in. Toolchain, goroutines, channels, and table-driven tests
- AWS CloudFormation: templates, change sets, and drift detection for the VPCs and cluster infrastructure underneath these workloads

More container pages sit in the Containers group; the full index is the cheatsheet library.

cost

Low-Cost Cloud Stack Cheatsheet

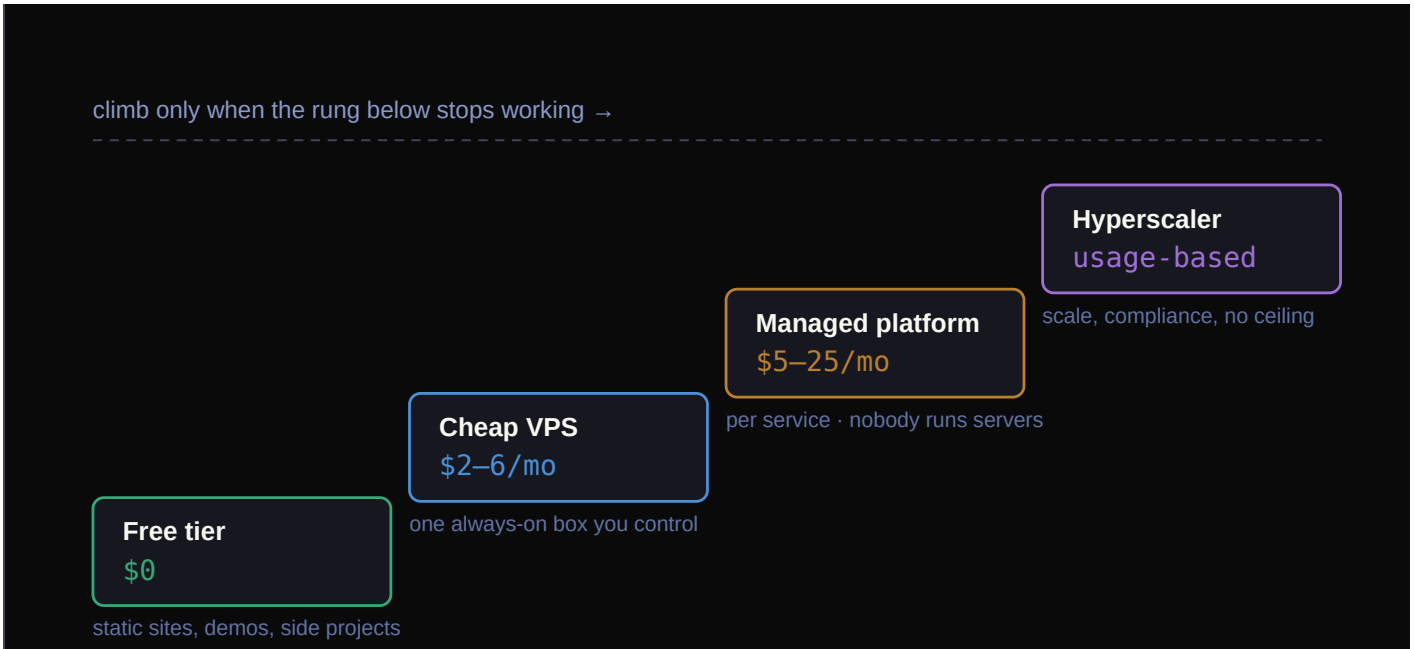
When a project is pre-revenue, every dollar spent on infrastructure is a dollar not spent on the product. The good news: managed and serverless platforms now give you production-grade hosting, databases, and CI/CD on free tiers, and low-cost VPS providers give you a full server for the price of a coffee. This is the reference I reach for to keep cloud spend near **$0** until usage justifies more. For the AWS-native path once you outgrow these, see my CloudFormation cheatsheet and boto3 cheatsheet.

> ***Note:*** *Prices below are ballparks, last re-checked* ***August 2026****, and they change often — sometimes without announcement, as the Oracle box further down shows. Treat them as relative comparisons, not exact quotes, and confirm on each vendor's pricing page before you commit to anything.*

Golden rules

- **Start on free tiers + serverless** (scale-to-zero). Bills rise only as usage does.
- **Managed platforms over your own servers** until you genuinely can't.
- **One tool per job.** No duplicate SaaS paying for the same capability twice.
- **A cheap VPS often beats a hyperscaler** for fixed, predictable workloads.
- **Watch data egress.** It is the silent line item, cheap or unmetered on OVH/Hetzner, expensive on the big clouds.
- **Move to AWS/GCP/Azure only when scale, control, or compliance demands it,** not before.

the cost ladder



Ranges, not quotes — the tables below carry the per-provider detail. The rungs are deliberately not to scale: what matters is that each step buys you something specific, and that ***you should only climb when the step below actually stops working.*** *Most projects never need the top rung.*

Managed services cost comparison

Layer	Service	Free tier	Paid entry	Pick it when
Frontend	Netlify	100 GB bandwidth, 300 build min/mo	~$19/mo	React/Vue/Astro sites + serverless functions
Frontend	Vercel	Hobby free, 100 GB	~$20/mo	Next.js, edge functions
Frontend	Cloudflare Pages	Unlimited requests, free	Workers ~$5/mo	Cheapest at scale, static + edge
Frontend	GitHub Pages	Free static hosting	Free	Pure static, no SSR/functions
API / backend	Railway	Usage credit	~$5/mo + usage	APIs, workers, cron, quick DB add-ons
API / backend	Render	Free web service (sleeps)	~$7/mo	Always-on APIs + background workers
API / backend	Fly.io	Small free allowance	Usage-based	Containers close to users, global
Database	Neon	Serverless Postgres, scales to zero	~$19/mo	Postgres; cheapest when idle
Database	Supabase	500 MB Postgres + auth + storage	~$25/mo	Postgres **plus** auth, storage, realtime
Database	Xata	Free tier (Postgres-based)	Usage-based	Tables + built-in search

Layer	Service	Free tier	Paid entry	Pick it when
Database	CockroachDB	Free tier; distributed SQL (Postgres-compatible)	Usage-based	Postgres wire-compatible with HA / horizontal scale
Database	Turso	SQLite/libSQL, generous free tier	~$5/mo+	Edge/low-latency reads, many small DBs
Database	PlanetScale	Serverless MySQL/Postgres (no free tier)	~$39/mo	Branching + serverless scale
Database	MongoDB Atlas	Free 512 MB (M0)	~$9/mo (Flex)	NoSQL / document model
Database	Upstash	Serverless Redis/Postgres, pay-per-request	Pay-per-use	Cache, queues, serverless Redis
Auth / BaaS	Supabase Auth	Free (in the plan above)	Included	Bundled with your database
Auth / BaaS	Firebase	Spark plan free	Blaze pay-as-you-go	Google BaaS, realtime, mobile
Auth / BaaS	Clerk	Free to ~10k monthly users	~$25/mo	Drop-in auth with prebuilt UI

Low-cost cloud & VPS providers

When you want a real server (fixed price, full control) rather than a managed platform, these are the cost leaders:

Explore this further in **MSK vs Kinesis Cheatsheet (Cost & When to Choose)**

Provider	Cheapest entry	Free tier / credit	Best for
Oracle Cloud	Pay-as-you-go	**Always-free: 2 ARM cores + 12 GB RAM** (halved in 2026 — see below), 2 AMD micro VMs, 200 GB block	Free always-on compute, with a caveat
Hetzner	~€4/mo VPS (2 vCPU, 4 GB)	None	Cheapest reliable VPS + dedicated (EU)
OVHcloud	~$4–6/mo VPS	Trial	EU cloud, VPS, cheap dedicated, DDoS protection
Contabo	~$5/mo VPS (big RAM/disk)	None	Most RAM & storage per dollar
DigitalOcean	$4–6/mo droplet	$200 credit (60 days)	Simple UX, managed DB, App Platform
Vultr	~$2.50–6/mo	Trial credit	Many regions, cheap instances
Linode (Akamai)	$5/mo (1 GB)	Trial credit	Straightforward VPS
Scaleway	~€4/mo (ARM)	Trial	EU cloud, cheap ARM instances
Netcup	~€3/mo VPS	None	Rock-bottom EU VPS pricing

Provider	Cheapest entry	Free tier / credit	Best for
Kamatera	~$4/mo (configurable)	30-day trial	Flexible specs, hourly billing, global
IONOS	~$2/mo VPS (intro)	Trial	Cheapest intro VPS (EU/US)
UpCloud	~$5/mo	Trial credit	Fast MaxIOPS storage, EU/global
AWS	Pay-as-you-go	12-month free tier	Full control, scale, compliance
Google Cloud	Pay-as-you-go	$300 credit + free tier	Data / ML workloads
Azure	Pay-as-you-go	$200 credit + free tier	Microsoft ecosystem

Package configuration

Configuration oracle-always-free

Enforced from **18 August 2026**: the Always Free Ampere A1 pool drops to **2 OCPU / 12 GB** per tenancy.

Anything above the new ceiling is **terminated automatically** — no notice, no email. If you are running a 4-core Always Free box, resize it before that date.

Do you accept?

< Yes > < No >

oracle

Oracle halved its Always Free ARM allowance in 2026, and it is enforced from 18 August 2026.

The Ampere A1 pool went from **4 OCPU / 24 GB** to **2 OCPU / 12 GB** — a tenancy-wide ceiling of 1,500 OCPU-hours and 9,000 GB-hours a month. Anything above the new limit is **terminated automatically**, and Oracle shipped the change with no blog post and no customer email; people found out when instances stopped. The 2 AMD micro VMs and the 200 GB of block storage are unchanged.

If you are running a 4-core Always Free box, resize it *now*. And take the general lesson: a free tier is a loan, not a guarantee — never put anything on one you would be sorry to have terminated with no notice.

Reported by InfoQ and Linuxiac; check Oracle's Free Tier docs for the current numbers before you build on them.

Canadian options (data residency)

If you need data to stay in Canada, you have two routes. First, the big clouds and several budget hosts already run **Canadian regions**, so you get residency without a Canadian vendor:

> **AWS** — Montreal (`ca-central-1`) and **Calgary (`ca-west-1`)**
> **Azure** — Canada Central (Toronto), Canada East (Quebec City)
> **Google Cloud** — Montreal (`northamerica-northeast1`), Toronto (`northamerica-northeast2`)
> **OVHcloud** — Beauharnois, Quebec
> **Vultr / DigitalOcean** — Toronto

Second, Canadian-owned hosts (data stays in Canada, local billing/support):

Provider	Cheapest entry (CA$)	Data centers	Notes
Web Hosting Canada (whc.ca)	~CA$10/mo VPS	Montreal, Beauharnois	Canadian-owned, data in Canada
Canadian Web Hosting	~CA$15/mo cloud/VPS	Vancouver, Toronto	Canadian-owned, SOC 2
HostPapa	~CA$20/mo VPS	Toronto	Canadian-owned, small-business focus
Aptum	Custom (enterprise)	Toronto, Montreal	Managed hybrid/multi-cloud
Cloud.ca	Custom (enterprise)	Canada	Data-sovereign cloud

For most projects the cheapest Canadian-residency option is simply an **AWS**/**OVH Canadian region** or a **Toronto Vultr/DigitalOcean droplet**; the Canadian-owned hosts are worth it when local ownership, support, or procurement matters.

> Discover related concepts in **Jenkins Cheatsheet**

Quick picks

The “just tell me what to use” version:

> Uncover more details in **Amazon Linux 2 Terminal Cheatsheet**

> **Marketing / static site** → Astro on **Netlify** or **Cloudflare Pages** (free)

> **App frontend** → Next.js on **Vercel**

> **API / backend** → **Railway** (or Render)

> **Database** → **Neon** (Postgres), or **Supabase** if you also want auth + storage

> **Auth** → **Supabase Auth** or **Clerk**

> **Cheap always-on server** → a **Hetzner / OVH** VPS, or **Oracle Cloud**'s free tier if you can live inside 2 ARM cores and the terms-change risk above

> **Canadian data residency** → an AWS/Azure/GCP **Canadian region**, or a Canadian-owned host (below)

Total monthly cost: day one vs traction

Stage	Setup	Est. cost
Day 1 (pre-traffic)	Netlify + Neon + Railway free tiers	**~$0**
Always-on server	Hetzner/OVH VPS, or Oracle free tier (2 ARM cores)	**~$0–6/mo**
Early traction	Paid managed tiers as you cross free limits	**~$25–75/mo**
Scaling	AWS/GCP/Azure for scale & compliance	Usage-based

Cost traps to avoid

> Running an **always-on VM for a static site** (a CDN host does it for free).

> **Over-provisioning a managed database** when serverless scale-to-zero fits.

> **Defaulting to a hyperscaler** when a $5 VPS would do — hyperscaler egress, NAT, and per-GB storage fees add up fast.

> **Ignoring data-egress fees** — cheap/unmetered on OVH and Hetzner, expensive on AWS/GCP/Azure.

> Jumping to **Kubernetes or a full AWS build too early** — complexity you don't need yet.

> Buying **annual plans** before you've validated you'll use the service.

When to move to AWS / GCP / Azure

Graduate off the free tiers and cheap VPS boxes when one of these is true:

- **Cost crossover** — your managed/VPS bill exceeds equivalent hyperscaler cost at your usage.
- **Control** — you need VPCs, private networking, or specific managed services the cheaper providers can't give.
- **Compliance** — SOC 2 / PCI / data-residency requirements a budget host won't meet.
- **Scale** — traffic or data volume outgrows the platform's ceilings.

At that point, keep the same discipline: infrastructure as code (CloudFormation or Terraform), and only provision what the numbers justify.

The whole point: spend on **product**, not idle servers, until the revenue is there to fund them.

cost

MSK vs Kinesis Cheatsheet (Cost & When to Choose)

Both **Amazon MSK** (managed Apache Kafka) and **Amazon Kinesis Data Streams** move high-throughput event streams. They overlap heavily, but they price very differently and pull you in different directions on lock-in and ops. This is the quick comparison, with the cost angle up front.

> ***Note:*** *Prices are early-2026 ballparks and change often. Treat them as relative comparisons, confirm on the AWS pricing pages.*

What each one is

> **Amazon MSK** — real, open-source **Apache Kafka**, managed by AWS. Full Kafka API, ecosystem, and portability. Comes as **Provisioned** (you size brokers) or **Serverless** (no brokers to size).

> **Kinesis Data Streams** — AWS's **native** streaming service. Proprietary API, deeply integrated with Lambda/Firehose/Flink. Comes as **Provisioned** (you set shards) or **On-demand** (auto-scales).

Cost comparison (the headline)

rates

Every figure below is **us-east-1**, taken from AWS's public Price List API. Rates differ by Region and do change — check the Kinesis and MSK pricing pages before you budget against them. A month here is 730 hours.

Service	Pricing model	Idle / floor cost	Scales by	Cheapest when
Kinesis (On-demand)	Per GB in + per-stream-hour	**Very low** (near-zero idle)	Automatic	Low, spiky, or unknown volume
Kinesis (Provisioned)	Per **shard-hour** (~$0.015) + PUT units	**~$11/shard/mo** (1 shard min)	Shards you set	Steady, known moderate volume
MSK Serverless	Per **cluster-hour** (~$0.75) + throughput + storage	**~$540/mo** baseline	Automatic	Want Kafka + no broker mgmt, higher volume
MSK Provisioned	Per **broker-hour** + storage + transfer	**~$70–300+/mo** (min brokers)	Brokers / partitions	Very high sustained throughput

The one-line cost takeaway: Kinesis (especially **on-demand or a single provisioned shard**) is far cheaper at low or spiky volume; MSK carries a real **always-on floor**. At very high *sustained* throughput, MSK Provisioned's per-GB economics can win.

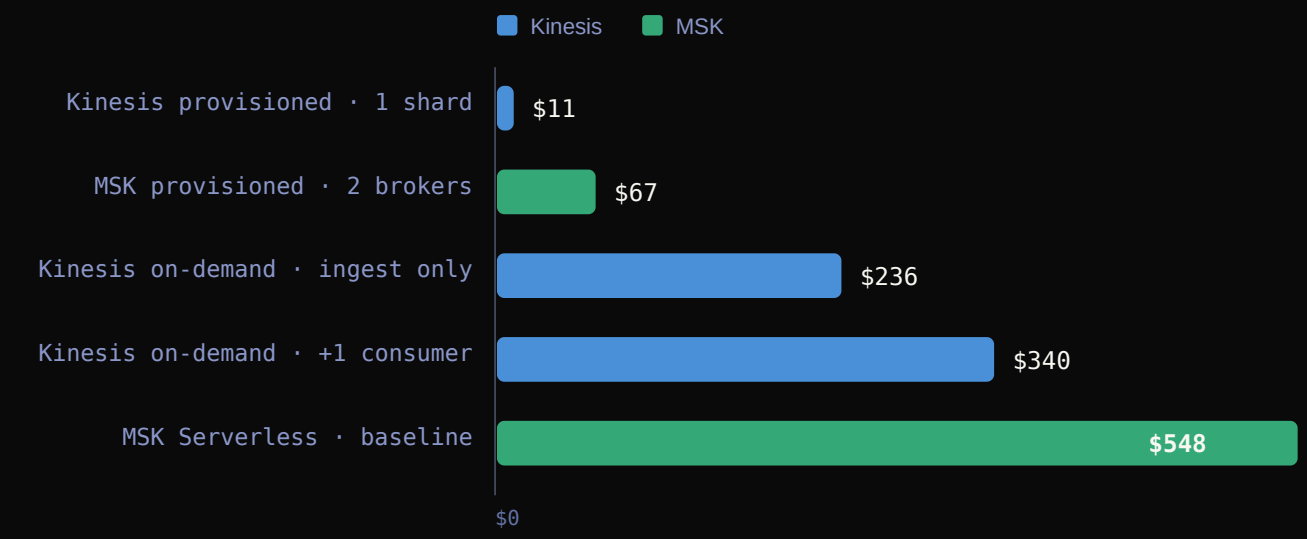
Worked example: a ~1 MB/s stream

> **Kinesis Provisioned:** 1 shard handles 1 MB/s in → **~$11/mo** + small PUT-unit cost. Cheapest.

> **Kinesis On-demand:** 1 MB/s sustained ≈ 2.6 TB/mo. Watch which rate you apply — **ingest is $0.08/GB, double the $0.04/GB read rate**, and it is the ingest one that matters here: 2,592 GB × $0.08 = **$207**, plus $0.04/stream-hour (**$29**), plus another **$104** if one consumer reads it all back. Call it **$235–340/mo** against ~$11 provisioned. On-demand is for *spiky*, not sustained — and that 20–30× gap is why.

> **MSK:** overkill at this size, you're paying the broker/cluster floor regardless.

```
1 MB/s sustained · us-east-1 · 730h month
```



*One provisioned shard covers 1 MB/s for **$11 a month**. The same load on Kinesis on-demand is **20–30× more**, because on-demand charges $0.08/GB to write — that is the number people get wrong. MSK is not competing at this size; you are paying its floor either way.*

*Lesson: **on-demand for spiky, provisioned for sustained** — within Kinesis that choice alone can be a 10x cost swing.*

> *Deepen your understanding in **Low-Cost Cloud Stack Cheatsheet***

Feature comparison

Dimension	MSK (Kafka)	Kinesis Data Streams
API / protocol	Apache Kafka (open, portable)	AWS proprietary SDK/API
Ecosystem	Full Kafka (Connect, Streams, MirrorMaker, 3rd-party)	AWS-native (Lambda, Firehose, Flink, KCL)
Portability / lock-in	Portable, multi-cloud, no lock-in	Locked to AWS
Scaling unit	Partitions / brokers	Shards (or auto in on-demand)
Ordering	Per-partition	Per-shard
Max record size	Configurable (default ~1 MB, can raise)	**1 MB hard limit**
Retention	Configurable, up to unlimited (tiered storage)	Default 24 h, up to 365 days
Ops overhead	Higher (Kafka concepts, even when managed)	Lower (esp. on-demand)
Lambda trigger	Yes (event source)	Yes (native)

Choosing like a senior DevOps engineer

The default

Default to Kinesis Data Streams (on-demand) for greenfield AWS work: lowest ops, near-zero idle cost, native Lambda/Firehose/Flink integration, automatic scaling. Move to **MSK** only when a concrete need pulls you there. Don't adopt Kafka for its own sake, it's a standing operational commitment even when "managed."

Questions to ask before you pick

Each "yes" is a reason to leave the default and consider MSK:

1. **Do we already run Kafka** (apps, skills, tooling, a migration)? → MSK
2. **Do we need the Kafka ecosystem** (Connect connectors, Kafka Streams, MirrorMaker)? → MSK
3. **Do we need portability / multi-cloud / no lock-in?** → MSK
4. **Are records ever larger than 1 MB**, or do we need very long / unlimited retention? → MSK
5. **Is throughput very high and sustained** (24/7, many MB/s)? → MSK Provisioned can win on unit cost
6. **Is volume low, spiky, or unknown?** → Kinesis on-demand
7. **Are consumers mostly AWS-native** (Lambda, Firehose, Flink)? → Kinesis
8. **How much ops do we want to own, and how mature is on-call?** → less / smaller team → Kinesis (or MSK Serverless)
9. **How budget-sensitive are we at low volume?** → Kinesis (MSK has a real floor)

If the answers mostly point at AWS-native, low-ops, cost-sensitive, stay on **Kinesis**. It takes a couple of firm "yes → MSK" answers to justify Kafka.

Monolith vs microservices

> **Monolith:** you usually don't need a streaming backbone at all, reach for **SQS/SNS/EventBridge** first. If you genuinely need a durable, replayable stream (analytics ingest, event sourcing), **Kinesis** is the low-friction pick; you don't yet have the many independent consumers that justify Kafka.

> **Microservices:** many independent producers/consumers, event-driven choreography, per-consumer replay, this is where a durable log earns its keep. **Kinesis** fits AWS-native microservices well; **MSK/Kafka** grows compelling as the estate scales: many teams, a shared event backbone, rich Connect integrations to external systems. Kafka is the classic "central nervous system" for a large microservice org.

Server/containers vs Lambda consumers

> **Lambda / serverless consumers → Kinesis.** Native event source, no VPC wiring, scales and bills per use; on-demand + Lambda is the lowest-friction combo.

> **Long-running server/container consumers (EC2/ECS/EKS) running Kafka clients, Kafka Streams, Flink, or Spark → MSK.** Standard Kafka clients, consumer groups, and stateful stream processors feel native there.

> **Rule:** serverless/event-driven consumers → **Kinesis**; long-running stateful stream processors on servers → **MSK**.

First, is streaming even the right tool?

A senior instinct is *not* reaching for Kafka/Kinesis when something simpler fits:

- Just decouple + retry work → **SQS**
- Fan-out pub/sub → **SNS**
- Route/filter events across services → **EventBridge**
- Ordered, replayable, high-throughput log with multiple independent consumers / analytics → **Kinesis or MSK**

Don’t pay for a streaming platform when a queue solves it.

> Discover related concepts in **Docker Cheatsheet**

Quick decision

- **Greenfield, AWS-native, cost- and ops-sensitive?** → **Kinesis** (on-demand for spiky, provisioned for steady).
- **Already Kafka, need its ecosystem/portability, >1 MB records, or huge sustained throughput?** → **MSK** (Serverless to skip broker ops, Provisioned for max scale).

References and Further Reading

- Amazon Web Services. Amazon MSK pricing.
- Amazon Web Services. Amazon Kinesis Data Streams pricing.
- Amazon Web Services. Amazon MSK: provisioned vs serverless.
- Amazon Web Services. Kinesis Data Streams capacity modes.

devops

Ansible Cheatsheet

Ansible is what I grab the moment a server count goes above one. These notes are the stuff I actually type: inventory patterns, the module table I scan for every task, and the Vault commands I keep mixing up. I usually run these playbooks from a Jenkins stage against hosts provisioned by CloudFormation.

Inventory

> ***Docs:*** *How to build your inventory, Ansible docs*

```
# inventory.ini (INI format)
[web]
web1.example.com
web2.example.com ansible_host=10.0.1.12

[db]
db1.example.com ansible_user=postgres

[prod:children]
web
db

[prod:vars]
ansible_user=deploy
env_name=prod
```

```
# inventory.yaml (YAML format)
all:
  children:
    web:
      hosts:
        web1.example.com:
        web2.example.com: { ansible_host: 10.0.1.12 }
    db:
      hosts:
        db1.example.com:
      vars: { ansible_user: postgres }
```

Per-host/per-group variable files auto-load from `group_vars/all.yaml`, `group_vars/<group>.yaml` (or a `group_vars/<group>/` directory, vault files included), and `host_vars/<host>.yaml` next to the inventory or playbook.

> Expand your knowledge with **Atuin Cheatsheet**

```
ansible-inventory -i inventory.yaml --graph     # visualise groups
```

Ad-hoc Commands

> ***Docs:*** *Introduction to ad hoc commands, Ansible docs*

```
ansible all  -i inventory.yaml -m ping                          # connectivi
ansible web  -a 'uptime'                                        # default mc
ansible db   -b -m service -a 'name=postgresql state=restarted'   # -b =
ansible all  -b -m apt -a 'name=htop state=present update_cache=yes'
ansible 'web1*' -m shell -a 'ss -tlnp | grep 443'               # shell wher
ansible all  --limit 'prod:!db' -m ping                         # pattern: p
```

Useful flags: `-f 20` (forks), `-K` (ask sudo password), `-u deploy` (remote user), `--one-line`.

> Deepen your understanding in **Docker Cheatsheet**

Playbook Anatomy

> ***Docs:*** *Ansible playbooks, Ansible docs*
>
> > *Explore this further in **AWS CloudFormation Cheatsheet***

```yaml
---
- name: Configure web tier
  hosts: web
  become: true
  vars_files:
    - vars/common.yaml
  tasks:
    - name: Install nginx
      ansible.builtin.apt:
        name: nginx
        state: present
        update_cache: true
      tags: [packages]

    - name: Deploy config
      ansible.builtin.template:
        src: nginx.conf.j2
        dest: /etc/nginx/nginx.conf
      notify: Restart nginx
      tags: [config]

  handlers:
    - name: Restart nginx
      ansible.builtin.service:
        name: nginx
        state: restarted
```

```
ansible-playbook -i inventory.yaml site.yaml
ansible-playbook site.yaml --limit web --tags config --check --diff
ansible-playbook site.yaml --syntax-check --list-tasks --list-hosts
```

Common Modules

> **Docs:** *ansible.builtin collection index, Ansible docs*

Module	Purpose	Key params
`apt` / `yum` / `dnf`	Package management (`package` = distro-agnostic)	`name` , `state` , `update_cache`
`copy`	Push static file/content	`src` , `dest` , `mode` , `content`
`template`	Render Jinja2 to remote	`src` (.j2), `dest` , `validate`
`file`	Files, dirs, links, perms	`path` , `state` (touch/directory/link/absent), `mode` , `owner`
`service` / `systemd`	Manage services	`name` , `state` , `enabled` , `daemon_reload`
`user`	Manage users	`name` , `groups` , `shell` , `state`
`lineinfile`	Edit single line idempotently	`path` , `regexp` , `line` , `state`
`git`	Clone/update repos	`repo` , `dest` , `version`
`uri`	HTTP calls / health checks	`url` , `method` , `status_code` , `body_format`
`command`	Run command (no shell)	`cmd` , `creates` , `chdir`
`shell`	Run through /bin/sh	needed for pipes, redirects, `$VAR`

`command` vs `shell` : prefer `command` (safer, no shell injection); use `shell` only when you need pipes/globs/env expansion. Neither is idempotent by default; guard with `creates:` , `changed_when:` , or better, find a real module.

Discover related concepts in **AWS CloudFormation Cheatsheet**

Variables & Precedence

Docs: *Using variables, Ansible docs*

Low → high (abridged to the ones that matter day-to-day): role `defaults/main.yaml` → `group_vars/all` → `group_vars/<group>` → `host_vars/<host>` → play `vars:` / `vars_files:` → task `vars:` → `set_fact` /registered vars → extra vars `-e key=value` (always wins).

Uncover more details in **Bash Basics Cheatsheet**

```
- name: Register output for later
  ansible.builtin.command: git rev-parse --short HEAD
  register: git_sha
  changed_when: false
# use later as: {{ git_sha.stdout }}
```

Facts

Docs: *Discovering variables: facts and magic variables, Ansible docs*

```
ansible web1 -m setup                              # everything
ansible web1 -m setup -a 'filter=ansible_mem*'     # filtered
```

Common ones: `ansible_hostname`, `ansible_distribution`, `ansible_default_ipv4.address`, `ansible_processor_vcpus`, `ansible_memtotal_mb`. Set `gather_facts: false` on plays that don't need them; big speedup on large inventories.

> Journey deeper into this topic with **AWS CloudFormation Cheatsheet**

Loops & Conditionals

> ***Docs:*** *Loops, Ansible docs*

```
- name: Install packages
  ansible.builtin.apt:
    name: "{{ item }}"
    state: present
  loop: [nginx, certbot, jq]

- name: Create users with attributes
  ansible.builtin.user:
    name: "{{ item.name }}"
    groups: "{{ item.groups }}"
  loop:
    - { name: alice, groups: sudo }
    - { name: bob,   groups: docker }

- name: Only on Debian prod hosts   # list items are AND-ed
  ansible.builtin.apt:
    name: unattended-upgrades
  when:
    - ansible_os_family == 'Debian'
    - env_name == 'prod'
```

Retry pattern: `until: result.status == 200` with `retries: 10` and `delay: 5` on a registered task.

> Enrich your learning with **Bash Basics Cheatsheet**

Handlers & notify

> ***Docs:*** *Handlers: running operations on change, Ansible docs*

Handlers run once at the end of the play, only if something notified them, in the order they are defined (not notification order).

```
tasks:
  - name: Update nginx config
    ansible.builtin.template:
      src: nginx.conf.j2
      dest: /etc/nginx/nginx.conf
    notify: Reload nginx

handlers:
  - name: Reload nginx
    ansible.builtin.service:
      name: nginx
      state: reloaded
```

Force handlers to fire mid-play with a `ansible.builtin.meta: flush_handlers` task.

> Gain comprehensive insights from **AWS CloudFormation Cheatsheet**

Roles Layout

> ***Docs:*** *Roles, Ansible docs*

```
roles/nginx/
├── defaults/main.yaml     # overridable defaults (lowest precedence)
├── vars/main.yaml         # role constants (high precedence)
├── tasks/main.yaml        # entry point
├── handlers/main.yaml
├── templates/ + files/    # nginx.conf.j2, static assets
└── meta/main.yaml         # dependencies, galaxy info
```

```
ansible-galaxy init roles/myrole             # scaffold the layout above
ansible-galaxy install -r requirements.yaml  # pull external roles/colle
```

Apply in a play with `roles: [common, nginx]`, or `role: nginx` plus inline `vars:` to override defaults.

> Master this concept through **Bash CLI One-Liners Cheatsheet**

Vault

> ***Docs:*** *Protecting sensitive data with Ansible Vault, Ansible docs*

```
ansible-vault create  group_vars/prod/vault.yaml     # new encrypted file
ansible-vault edit    group_vars/prod/vault.yaml
ansible-vault encrypt secrets.yaml
ansible-vault decrypt secrets.yaml
ansible-vault encrypt_string 's3cret' --name db_password   # inline singl

ansible-playbook site.yaml --ask-vault-pass
ansible-playbook site.yaml --vault-password-file ~/.vault_pass   # chmod
```

Convention: keep secrets in `vault.yaml` prefixed `vault_`, reference them from a plaintext `vars.yaml` (`db_password: "{{ vault_db_password }}"`) so `grep` still finds variable names.

> Delve into specifics at **AWS CloudFormation Cheatsheet**

Tags, Check Mode & Diff

> ***Docs:*** *Tags, Ansible docs*

```
ansible-playbook site.yaml --tags 'config,certs'
ansible-playbook site.yaml --skip-tags packages
ansible-playbook site.yaml --check --diff        # dry run + show file ch
```

`--check` predicts changes without applying; `--diff` shows template/lineinfile deltas. Tasks that must always/never run: `tags: always` / `tags: never`.

> Deepen your understanding in **Docker Cheatsheet**

0

tip

Make `--check --diff` against prod part of your review flow. A playbook that errors in check mode (usually a `command` task whose output later tasks depend on) is telling you where idempotency is broken. Fix it with `check_mode: false` on the task or a proper module.

ansible.cfg Essentials

> ***Docs:*** *Ansible configuration settings, Ansible docs*

> *Deepen your understanding in **Docker Cheatsheet***

1

```
# ansible.cfg (project root wins over ~/.ansible.cfg)
[defaults]
inventory = inventory/inventory.yaml
remote_user = deploy
forks = 20
host_key_checking = False        # convenient for cattle; understand the
stdout_callback = yaml

[privilege_escalation]
become = True

[ssh_connection]
pipelining = True                # big speedup
ssh_args = -o ControlMaster=auto -o ControlPersist=60s
```

Jinja2 Templating Basics

> ***Docs:** Templating (Jinja2), Ansible docs*

```
# templates/nginx.conf.j2
worker_processes {{ ansible_processor_vcpus }};
events { worker_connections {{ nginx_worker_connections | default(1024) }
{% for site in sites %}
server {
  listen 80;
  server_name {{ site.domain }};
{% if site.redirect_https | default(false) %}
  return 301 https://{{ site.domain }}$request_uri;
{% endif %}
}
{% endfor %}
```

Filters I use constantly: `default('x')`, `join(',')`, `to_yaml` / `to_nice_json`, `regex_replace('a', 'b')`, `b64encode`, `ternary('yes', 'no')`. Test any expression fast with `ansible localhost -m debug -a "msg={{ expr }}"`.

> Deepen your understanding in **Docker Cheatsheet**

2

Related Cheatsheets

- AWS CloudFormation: intrinsic functions, change sets, and drift detection to provision the infrastructure these playbooks configure
- Jenkins: wire `ansible-playbook` into declarative pipelines with credential binding, parameters, and parallel stages
- Bash Basics: quoting, exit codes, and error handling for every `command` / `shell` task and dynamic-inventory script

Everything else is in the DevOps Tooling group, or start from the full cheatsheet library.

devops

AWS CloudFormation Cheatsheet

CloudFormation is still my default for AWS-native infrastructure. No state file to babysit, and rollbacks come free. This is the reference I wish I had when I started: template anatomy, the intrinsic functions I actually use, and the CLI commands for the full stack lifecycle. For imperative AWS scripting I reach for boto3 instead, and config management after provisioning goes to Ansible.

Template Anatomy

> ***Docs:*** *Template anatomy, CloudFormation User Guide*

```yaml
AWSTemplateFormatVersion: '2010-09-09'   # only valid value
Description: App network layer

Parameters:        # runtime inputs
  EnvName:
    Type: String
    Default: dev
    AllowedValues: [dev, staging, prod]
  VpcCidr:
    Type: String
    Default: 10.0.0.0/16
    AllowedPattern: ^\d+\.\d+\.\d+\.\d+/\d+$

Mappings:          # static lookup tables
  RegionAmi:
    us-east-1: { Ami: ami-0abc1234 }
    ca-central-1: { Ami: ami-0def5678 }

Conditions:        # booleans evaluated at deploy time
  IsProd: !Equals [!Ref EnvName, prod]

Resources:         # the only REQUIRED section
  MyVpc:
    Type: AWS::EC2::VPC
    Properties:
      CidrBlock: !Ref VpcCidr

Outputs:           # values to display/export
  VpcId:
    Value: !Ref MyVpc
    Export:
      Name: !Sub '${EnvName}-vpc-id'
```

Other top-level sections: `Metadata`, `Rules` (parameter validation), `Transform` (`AWS::Serverless-2016-10-31` for SAM, `AWS::LanguageExtensions` for `Fn::ForEach`).

> Expand your knowledge with **Ansible Cheatsheet**

Intrinsic Functions

> **Docs:** *Intrinsic function reference, CloudFormation User Guide*

Function	Short form	What it does
`Ref`	`!Ref`	Parameter value or resource's default ID (e.g. VPC ID, bucket name)
`Fn::GetAtt`	`!GetAtt Res.Attr`	Named attribute of a resource (`!GetAtt Db.Endpoint.Addr`
`Fn::Sub`	`!Sub '${Var}-x'`	String interpolation of params, resources, pseudo params
`Fn::Join`	`!Join [':', [a, b]]`	Join list with delimiter
`Fn::If`	`!If [Cond, valTrue, valFalse]`	Conditional value (condition fro `Conditions`)
`Fn::ImportValue`	`!ImportValue name`	Read another stack's export
`Fn::Select`	`!Select [0, !GetAZs '']`	Pick item from a list (`!GetAZs` region's AZs)
`Fn::Split`	`!Split [',', !Ref Csv]`	String to list
`Fn::FindInMap`	`!FindInMap [Map, Key, SubKey]`	Lookup in `Mappings`

`!Sub` beats `!Join` for readability in almost every case:

> Deepen your understanding in **Ansible Cheatsheet**

```
BucketArn: !Sub 'arn:${AWS::Partition}:s3:::${MyBucket}/*'
```

Sample Template: S3 + IAM

Docs: *AWS::S3::Bucket, CloudFormation User Guide*

> *Explore this further in **Docker Cheatsheet***

```yaml
AWSTemplateFormatVersion: '2010-09-09'
Description: Versioned artifact bucket with a scoped writer role

Parameters:
  EnvName: { Type: String, Default: dev }

Conditions:
  IsProd: !Equals [!Ref EnvName, prod]

Resources:
  ArtifactBucket:
    Type: AWS::S3::Bucket
    DeletionPolicy: Retain
    UpdateReplacePolicy: Retain
    Properties:
      BucketName: !Sub 'artifacts-${EnvName}-${AWS::AccountId}'
      VersioningConfiguration:
        Status: !If [IsProd, Enabled, Suspended]
      PublicAccessBlockConfiguration:
        BlockPublicAcls: true
        RestrictPublicBuckets: true

  WriterRole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - { Effect: Allow, Principal: { Service: ec2.amazonaws.com }, A
      Policies:
        - PolicyName: write-artifacts
          PolicyDocument:
            Version: '2012-10-17'
            Statement:
              - Effect: Allow
                Action: ['s3:PutObject', 's3:GetObject']
                Resource: !Sub '${ArtifactBucket.Arn}/*'

Outputs:
  BucketName:
    Value: !Ref ArtifactBucket
    Export:
```

```
        Name: !Sub '${EnvName}-artifact-bucket'
```

CLI: Deploy & Stack Lifecycle

Docs: *cloudformation, AWS CLI Command Reference*

> *Discover related concepts in **Python Basics & CLI Cheatsheet***

```
# One command, idempotent - creates or updates (my default)
aws cloudformation deploy \
  --stack-name app-dev \
  --template-file template.yaml \
  --parameter-overrides EnvName=dev VpcCidr=10.1.0.0/16 \
  --capabilities CAPABILITY_NAMED_IAM \
  --no-fail-on-empty-changeset

# Explicit create / update
aws cloudformation create-stack --stack-name app-dev \
  --template-body file://template.yaml \
  --parameters ParameterKey=EnvName,ParameterValue=dev \
  --capabilities CAPABILITY_IAM
aws cloudformation update-stack --stack-name app-dev \
  --template-body file://template.yaml \
  --parameters ParameterKey=EnvName,UsePreviousValue=true

# Wait + inspect
aws cloudformation wait stack-create-complete --stack-name app-dev
aws cloudformation describe-stacks --stack-name app-dev --query 'Stacks[0
aws cloudformation describe-stack-events --stack-name app-dev \
  --query 'StackEvents[?ResourceStatus==`CREATE_FAILED`]'

# Delete
aws cloudformation delete-stack --stack-name app-dev
```

Change Sets

> **Docs:** *Updating stacks using change sets, CloudFormation User Guide*

Preview exactly what an update will touch before executing. Mandatory habit for prod:

```
aws cloudformation create-change-set \
  --stack-name app-prod --change-set-name add-alarms \
  --template-body file://template.yaml \
  --capabilities CAPABILITY_NAMED_IAM

aws cloudformation describe-change-set \
  --stack-name app-prod --change-set-name add-alarms \
  --query 'Changes[].ResourceChange.{Action:Action,Resource:LogicalResour

aws cloudformation execute-change-set \
  --stack-name app-prod --change-set-name add-alarms
```

`Replacement: True` means the resource gets destroyed and recreated. Check this column every time.

> Uncover more details in **Bash sed & awk Cheatsheet**

Drift Detection

> **Docs:** *Detecting stack drift, CloudFormation User Guide*

> *Journey deeper into this topic with **Kubernetes Cheatsheet***

```
aws cloudformation detect-stack-drift --stack-name app-prod
aws cloudformation describe-stack-drift-detection-status --stack-drift-de
aws cloudformation describe-stack-resource-drifts --stack-name app-prod \
  --stack-resource-drift-status-filters MODIFIED DELETED
```

Stack Policies & Termination Protection

> ***Docs:*** *Stack policies, CloudFormation User Guide*

```
# Termination protection - flip on for anything prod
aws cloudformation update-termination-protection \
  --stack-name app-prod --enable-termination-protection
```

```
{
  "Statement": [
    { "Effect": "Allow", "Action": "Update:*", "Principal": "*", "Resourc
    { "Effect": "Deny",  "Action": ["Update:Replace", "Update:Delete"],
      "Principal": "*", "Resource": "LogicalResourceId/ProdDatabase" }
  ]
}
```

```
aws cloudformation set-stack-policy --stack-name app-prod \
  --stack-policy-body file://policy.json
```

Also set `DeletionPolicy: Retain` (and `UpdateReplacePolicy: Retain`) on stateful resources: databases, buckets, EFS.

> Enrich your learning with **Low-Cost Cloud Stack Cheatsheet**

Nested Stacks & Exports

> ***Docs:*** *Working with nested stacks, CloudFormation User Guide*

```
Resources:
  NetworkStack:
    Type: AWS::CloudFormation::Stack
    Properties:
      TemplateURL: https://s3.amazonaws.com/my-templates/network.yaml
      Parameters:
        EnvName: !Ref EnvName

  App:
    Type: AWS::EC2::Instance
    Properties:
      SubnetId: !GetAtt NetworkStack.Outputs.PublicSubnetId
```

Cross-stack (not nested): export in one stack's `Outputs`, then `!ImportValue prod-vpc-id` elsewhere. You cannot delete or change an export while another stack imports it. Nested stacks avoid that coupling; exports make it explicit.

> Gain comprehensive insights from **Python Basics & CLI Cheatsheet**

Pseudo Parameters

> ***Docs:*** *Pseudo parameters reference, CloudFormation User Guide*

> *Master this concept through **Jenkins Cheatsheet***

Parameter	Example value
`AWS::AccountId`	`123456789012`
`AWS::Region`	`ca-central-1`
`AWS::StackName`	`app-prod`
`AWS::Partition`	`aws` (or `aws-cn`, `aws-us-gov`)
`AWS::NoValue`	Removes a property when used with `!If`

Validation & Linting

> ***Docs:*** *validate-template, AWS CLI Command Reference*

```
aws cloudformation validate-template --template-body file://template.yaml

pip install cfn-lint
cfn-lint templates/**/*.yaml           # catches type errors, bad refs, r
```

`validate-template` only checks syntax; `cfn-lint` checks semantics against the real resource specs. Run both in CI.

> Delve into specifics at **Bash Basics Cheatsheet**

Rollback Behavior

Docs: *Continue rolling back an update, CloudFormation User Guide*

> *Deepen your understanding in **Ansible Cheatsheet***

0

> Create fails → `ROLLBACK_COMPLETE` : stack is dead, delete it and recreate.

> Update fails → automatic rollback to previous working state (`UPDATE_ROLLBACK_COMPLETE`); if that rollback fails, `continue-update-rollback` (optionally `--resources-to-skip`).

> `--disable-rollback` / `--on-failure DO_NOTHING` on create keeps failed resources up for debugging.

warning

A stack stuck in `ROLLBACK_COMPLETE` after a failed create can never be updated, only deleted. Delete it, fix the template, and create again. This surprises everyone exactly once.

Common Errors

Docs: *Troubleshooting CloudFormation, CloudFormation User Guide*

> *Deepen your understanding in **Ansible Cheatsheet***

1

Error	Cause	F
`No updates are to be performed`	Template unchanged	U
`Requires capabilities: [CAPABILITY_IAM]`	Template creates IAM resources	A
`Export ... cannot be deleted as it is in use`	Another stack imports the value	U
`Resource ... already exists`	Name collision with unmanaged resource	R (
`Template format error: Unresolved resource dependencies`	`!Ref` / `!GetAtt` to a typo'd logical ID	F
Stuck in `UPDATE_ROLLBACK_FAILED`	Rollback blocked (e.g. deleted resource)	
`Rate exceeded`	API throttling on big stacks	R
`Circular dependency between resources`	Mutual `!Ref`s (common with SGs)	S s

Related Cheatsheets

> Ansible: configuration management for the instances these stacks provision, with modules, handlers, roles, and Vault

- Python Boto3: the imperative side, with clients, paginators, waiters, and ClientError handling for anything a template can’t express
- Kubernetes: manifests, rollouts, and kubectl triage for the workloads that land on the clusters these stacks create

The rest of the DevOps Tooling group is one click away, or start from the top at the cheatsheet library.

devops

Jenkins Cheatsheet

I write Jenkinsfiles weekly and still forget the exact `when` syntax every single time, so this page stays open in a pinned tab. Everything below is copy-paste ready and tested against a recent Jenkins LTS. Most of my pipelines end up calling Ansible or building Docker images, so those two sheets pair well with this one.

Declarative Pipeline Skeleton

> ***Docs:*** *Pipeline syntax (Jenkins docs)*
>
> > *Expand your knowledge with **Kubernetes Cheatsheet***

```
pipeline {
    agent any
    options {
        timeout(time: 30, unit: 'MINUTES')
        buildDiscarder(logRotator(numToKeepStr: '20'))
        disableConcurrentBuilds()
        timestamps()
    }
    environment {
        APP_ENV = 'staging'
    }
    stages {
        stage('Build') {
            steps {
                sh 'make build'
            }
        }
        stage('Test') {
            steps {
                sh 'make test'
            }
        }
        stage('Deploy') {
            steps {
                sh './deploy.sh'
            }
        }
    }
    post {
        always  { cleanWs() }
        success { echo 'Build passed' }
        failure { mail to: 'team@example.com', subject: "FAILED: ${env.JC
    }
}
```

Scripted vs Declarative

> **Docs:** *Pipeline overview (Jenkins docs)*

	Declarative	Scripted
Syntax	`pipeline { ... }`	`node { ... }`
Structure	Opinionated, validated	Free-form Groovy
`post` / `when` / `options`	Built-in	DIY with try/catch
Best for	95% of jobs	Complex dynamic logic

Rule of thumb: start declarative, drop into a `script { }` block only when you need loops or dynamic stage generation.

> Deepen your understanding in **Bash CLI One-Liners Cheatsheet**

Environment & Credentials

> **Docs:** *Using credentials (Jenkins docs)*

```
environment {
    // From Jenkins credentials store (ID must exist)
    AWS_CREDS   = credentials('aws-prod')          // sets AWS_CREDS_USR
    API_TOKEN   = credentials('api-token-secret')  // secret text -> sing
}
```

```
// Scoped binding inside steps - preferred for secrets
steps {
    withCredentials([
        usernamePassword(credentialsId: 'nexus', usernameVariable: 'NEXUS
        string(credentialsId: 'slack-token', variable: 'SLACK_TOKEN'),
        file(credentialsId: 'kubeconfig-prod', variable: 'KUBECONFIG'),
        sshUserPrivateKey(credentialsId: 'deploy-key', keyFileVariable: '
    ]) {
        sh 'curl -u "$NEXUS_USER:$NEXUS_PASS" https://nexus.internal/uplc
    }
}
```

Always use single quotes in `sh` so secrets are expanded by the shell, not interpolated into the Groovy string (which leaks them to the build log).

> Explore this further in **tmux Cheatsheet**

Parameters

> ***Docs:*** *Pipeline syntax: parameters (Jenkins docs)*
>
> > *Discover related concepts in **Ansible Cheatsheet***

```
parameters {
    string(name: 'VERSION', defaultValue: '1.0.0', description: 'Release
    choice(name: 'ENV', choices: ['dev', 'staging', 'prod'], description:
    booleanParam(name: 'SKIP_TESTS', defaultValue: false)
    text(name: 'RELEASE_NOTES', defaultValue: '')
}
// Access: params.VERSION, params.ENV, params.SKIP_TESTS
```

when Conditions

Docs: *Pipeline syntax: when (Jenkins docs)*

> *Uncover more details in **MSK vs Kinesis Cheatsheet (Cost & When to Choose)***

```
stage('Deploy Prod') {
    when {
        allOf {
            branch 'main'
            environment name: 'APP_ENV', value: 'prod'
            expression { return params.ENV == 'prod' }
            not { changeRequest() }          // skip on PR builds
        }
    }
    steps { sh './deploy.sh prod' }
}
```

Condition	Matches when
`branch 'main'`	Branch name matches (multibranch)
`tag 'v*'`	Build is for a matching tag
`changeRequest()`	Build is a PR
`changeset '**/*.tf'`	Commit touched matching files
`buildingTag()`	Any tag build
`triggeredBy 'TimerTrigger'`	Started by cron
`anyOf { } / allOf { } / not { }`	Boolean combinators

Parallel Stages

Docs: *Pipeline syntax: parallel (Jenkins docs)*

> *Journey deeper into this topic with **Ansible Cheatsheet***

```
stage('Quality Gates') {
    parallel {
        stage('Lint')  { steps { sh 'make lint' } }
        stage('Unit')  { steps { sh 'make unit' } }
        stage('Integration') {
            agent { label 'docker' }
            steps { sh 'make integration' }
        }
    }
}
// failFast true  -> abort siblings on first failure (put inside options
```

Shared Libraries

Docs: *Extending with shared libraries (Jenkins docs)*

```
// Jenkinsfile - library configured globally as "mylib"
@Library('mylib@main') _

standardBuild(language: 'go', notify: '#builds')   // vars/standardBuild.
```

Repo layout:

Enrich your learning with **Docker Cheatsheet**

```
(root)
├── vars/standardBuild.groovy      # def call(Map config) { ... } - calla
├── src/org/acme/Utils.groovy      # classes, import as org.acme.Utils
└── resources/templates/pod.yaml   # libraryResource 'templates/pod.yaml'
```

Stash/Unstash & Artifacts

Docs: *Pipeline: Basic Steps reference (Jenkins docs)*

```
stage('Build')  { steps { sh 'make dist'; stash name: 'dist', includes: '
stage('Deploy') { agent { label 'deployer' }
                  steps { unstash 'dist'; sh './push.sh' } }

// Keep artifacts on the build page
archiveArtifacts artifacts: 'dist/*.tar.gz', fingerprint: true, allowEmpt
junit 'reports/**/*.xml'
```

Stash = pass files between stages/agents in the same build (temporary). Artifacts = keep files after the build (permanent, per retention policy).

Gain comprehensive insights from **AWS CloudFormation Cheatsheet**

Triggers

Docs: *Pipeline syntax: triggers (Jenkins docs)*

```
triggers {
    cron('H 2 * * 1-5')          // weeknights, H spreads load
    pollSCM('H/5 * * * *')       // poll every ~5 min
    upstream(upstreamProjects: 'lib-build', threshold: hudson.model.Resul
}
```

Webhook (preferred over polling): GitHub webhook to `https://jenkins.example.com/github-webhook/`, GitLab to `/project/<job>`. Generic jobs: enable “Trigger builds remotely” and hit `JENKINS_URL/job/NAME/build?token=TOKEN`.

Master this concept through **AWS CloudFormation Cheatsheet**

CLI & jenkins-cli.jar

Docs: *Jenkins CLI (Jenkins docs)*

```
curl -O https://jenkins.example.com/jnlpJars/jenkins-cli.jar   # from you
alias jcli='java -jar jenkins-cli.jar -s https://jenkins.example.com -aut

jcli list-jobs
jcli build my-job -p ENV=staging -f -v      # -f wait, -v stream console
jcli console my-job 42                      # console log of build 42
jcli reload-configuration
jcli safe-restart
jcli install-plugin blueocean -deploy
```

Blue Ocean lives at `/blue`. Best for visualising parallel stages and PR pipelines; classic UI is still where all the config lives.

> Delve into specifics at **Bash CLI One-Liners Cheatsheet**

Common Groovy Snippets

Docs: *Pipeline Utility Steps reference (Jenkins docs)*

> *Deepen your understanding in **Bash CLI One-Liners Cheatsheet***

0

```
// Capture command output
def sha = sh(script: 'git rev-parse --short HEAD', returnStdout: true).tr

// Exit code without failing the stage
def rc = sh(script: './smoke-test.sh', returnStatus: true)
if (rc != 0) { unstable('smoke tests failed') }

// JSON in/out (Pipeline Utility Steps plugin)
def cfg = readJSON file: 'config.json'
echo "replicas: ${cfg.deploy.replicas}"
writeJSON file: 'out.json', json: [version: sha, env: params.ENV]

// YAML too
def vals = readYaml file: 'values.yaml'

// Retry and timeout wrappers
retry(3) { sh './flaky-fetch.sh' }
timeout(time: 5, unit: 'MINUTES') { input message: 'Deploy to prod?' }
```

Agent Labels & Docker Agents

> ***Docs:*** *Using Docker with Pipeline (Jenkins docs)*

```
agent { label 'linux && docker' }        // label expression

agent {
    docker {
        image 'golang:1.22'
        args  '-v /var/cache/go:/go/pkg'
        reuseNode true
    }
}

agent {
    dockerfile { filename 'ci/Dockerfile'; label 'docker' }
}
```

`agent none` at the top + per-stage agents = no idle executor held between stages.

> Deepen your understanding in **Bash CLI One-Liners Cheatsheet**

1

> **tip**
>
> Use `H` instead of fixed minutes in cron triggers (`H 2 * * *` not `0 2 * * *`). Jenkins hashes the job name to pick a consistent minute, so a hundred nightly jobs don't all fire at 02:00 and flatten your agents.

Troubleshooting

Symptom	Likely cause	Fix
`No such DSL method 'x'`	Missing plugin or typo	Install plugin (e.g. Pipeline Utility Steps), check spelling
Build stuck “Waiting for next available executor”	No agent matches label	Check label expression, agent o status, executor count
`script not permitted to use method`	Groovy sandbox block	Approve in Manage Jenkins → I process Script Approval, or mov to a shared library
Secrets appear as `****` but wrong value	Groovy string interpolation of creds	Use single-quoted `sh` strings, l shell expand `$VAR`
`stash` not found in later stage	Different build or expired	Stash only lives within one build `archiveArtifacts` / `copyArti` across builds
Webhook fires but no build	Branch not discovered	Re-scan multibranch pipeline, ch branch filters
`Jenkinsfile not found`	Wrong script path or lightweight checkout issue	Fix “Script Path” in job config, ve branch
Docker agent: permission denied on socket	Jenkins user not in	Add user to `docker` group on a restart agent

Symptom	Likely cause	Fix
	docker group	
Workspace out of disk	No cleanup, fat workspaces	`cleanWs()` in `post { always` `buildDiscarder` in options

Related Cheatsheets

> Docker: image builds, registries, and the run flags behind every `agent { docker { ... } }` stage on this page

> Ansible: playbooks, inventories, and Vault, which is my usual deploy step invoked from a Jenkins stage

> Python Basics & CLI: argparse, venv, and `json.tool` for the helper scripts your pipelines shell out to

More of these live in the DevOps Tooling group, and the full cheatsheet library has everything else.

languages

Go (Golang) Cheatsheet

Docker, Kubernetes, Terraform, and half the CLIs I touch daily are written in Go, so it became my default for tooling too. These are the notes I wish I'd had when I started: toolchain commands, syntax I keep forgetting, and the concurrency patterns that actually come up in DevOps work.

Toolchain

> ***Docs:*** *go command on pkg.go.dev*

```
go mod init github.com/karan/myapp        # Start a module
go mod tidy                               # Add missing / drop unused dep
go get github.com/spf13/cobra@latest      # Add or upgrade a dependency
go run .                                  # Compile + run current package
go build -o bin/myapp ./cmd/myapp         # Build a binary
go test ./...                             # Test everything, recursively
go test -run TestParse -v ./internal/...  # One test, verbose
go test -race -cover ./...                # Race detector + coverage
go vet ./...                              # Static analysis
gofmt -l -w .                             # Format (or: go fmt ./...)
go install golang.org/x/tools/cmd/goimports@latest   # Install a tool to
GOOS=linux GOARCH=arm64 go build -o myapp-arm64 .    # Cross-compile
go build -ldflags="-s -w" .               # Strip symbols, smaller binary
```

Static cross-compiled binaries are why Go containers are tiny; the multi-stage build in my Docker Cheatsheet shows the payoff.

> Expand your knowledge with **Docker Cheatsheet**

Syntax Essentials

Docs: *The Go language specification at go.dev*

> *Deepen your understanding in **Python Basics & CLI Cheatsheet***

Types, Variables, Slices, Maps

```
var count int = 10          // explicit
name := "karan"             // inferred, function scope only
const MaxRetries = 3        // constants: untyped until used

nums := []int{1, 2, 3}                  // slice
nums = append(nums, 4)
sub := nums[1:3]                        // [2 3] - shares backing array

ports := map[string]int{"http": 80, "https": 443}
ports["ssh"] = 22
v, ok := ports["ftp"]                   // ok == false: key absent
delete(ports, "ssh")

for i, n := range nums { fmt.Println(i, n) }    // map range order is rar
```

Structs, Methods, Interfaces

```
type Server struct {
    Host string
    Port int
    tags []string          // lowercase = unexported
}

// Value receiver: gets a copy. Pointer receiver: can mutate.
func (s Server) Addr() string      { return fmt.Sprintf("%s:%d", s.Host,
func (s *Server) AddTag(t string)  { s.tags = append(s.tags, t) }

type Notifier interface {
    Notify(msg string) error
}

// Interfaces are satisfied implicitly - no "implements" keyword.
type SlackNotifier struct{ Webhook string }

func (s SlackNotifier) Notify(msg string) error { return post(s.Webhook,

func alert(n Notifier, msg string) error { return n.Notify(msg) }
```

Errors & Defer

```
f, err := os.Open("config.yaml")
if err != nil {
    return fmt.Errorf("opening config: %w", err)   // %w wraps for errors
}
defer f.Close()          // runs when the function returns, LIFO order

if errors.Is(err, os.ErrNotExist) { /* fall back to defaults */ }

var pathErr *os.PathError
if errors.As(err, &pathErr) { fmt.Println(pathErr.Path) }

var ErrNotFound = errors.New("not found")   // sentinel error
```

Goroutines & Channels

Docs: *Effective Go on go.dev*

> *Explore this further in **Kubernetes Cheatsheet***

```
go doWork()                          // fire-and-forget goroutine

ch := make(chan string)              // unbuffered: send blocks until recei
buf := make(chan int, 10)            // buffered: blocks only when full

ch <- "ping"; msg := <-ch            // send / receive
close(ch)                            // sender closes; receivers range safe

for msg := range ch { fmt.Println(msg) }   // loop until channel closed
```

select

```
select {
case msg := <-ch:
    fmt.Println("got", msg)
case <-time.After(2 * time.Second):
    fmt.Println("timeout")
case <-ctx.Done():
    return ctx.Err()
}
```

WaitGroup

```
var wg sync.WaitGroup
for _, host := range hosts {
    wg.Add(1)
    go func(h string) {
        defer wg.Done()
        checkHealth(h)
    }(host)
}
wg.Wait()
```

Context Cancelation

```
ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
defer cancel()

req, _ := http.NewRequestWithContext(ctx, http.MethodGet, url, nil)
resp, err := http.DefaultClient.Do(req)   // aborts when ctx expires

// In workers, select on ctx.Done() alongside your jobs channel and
// return ctx.Err() when it fires - same pattern as the select above.
```

tip

Always run `go test -race ./...` in CI. The race detector catches shared-memory bugs that pass every normal test run. The classic one is goroutines appending to a shared slice without a mutex.

Table-Driven Tests

Docs: *testing package on pkg.go.dev*

> *Discover related concepts in **Kubernetes Cheatsheet***

```
func TestParsePort(t *testing.T) {
    tests := []struct {
        name    string
        input   string
        want    int
        wantErr bool
    }{
        {"plain", "8080", 8080, false},
        {"with colon", ":443", 443, false},
        {"garbage", "abc", 0, true},
    }
    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            got, err := ParsePort(tt.input)
            if (err != nil) != tt.wantErr {
                t.Fatalf("err = %v, wantErr %v", err, tt.wantErr)
            }
            if got != tt.want {
                t.Errorf("got %d, want %d", got, tt.want)
            }
        })
    }
}
```

Generics (Quick Example)

Docs: *Generics tutorial on go.dev*

> *Uncover more details in **Docker Cheatsheet***

```go
func Map[T, U any](in []T, f func(T) U) []U {
    out := make([]U, len(in))
    for i, v := range in {
        out[i] = f(v)
    }
    return out
}

func Max[T cmp.Ordered](a, b T) T { if a > b { return a }; return b }

names := Map(servers, func(s Server) string { return s.Addr() })
```

Formatting Verbs

Docs: *fmt package on pkg.go.dev*

> *Journey deeper into this topic with **Docker Cheatsheet***

Verb	Meaning	Example output
`%v`	Default format	`{web 8080}`
`%+v`	Struct with field names	`{Host:web Port:8080}`
`%#v`	Go syntax representation	`main.Server{Host:"web", ...}`
`%T`	Type of the value	`main.Server`
`%d`	Integer (base 10)	`8080`
`%s`	String	`web`
`%q`	Quoted string	`"web"`
`%f` / `%.2f`	Float / 2 decimals	`3.14`
`%x`	Hex	`1f90`
`%w`	Wrap error (`fmt.Errorf` only)	-

Common Stdlib Snippets

> ***Docs:*** *Standard library on pkg.go.dev*

net/http Server

```
func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /healthz", func(w http.ResponseWriter, r *http.Re
        w.WriteHeader(http.StatusOK)
        w.Write([]byte("ok"))
    })
    srv := &http.Server{Addr: ":8080", Handler: mux, ReadTimeout: 5 * tim
    log.Fatal(srv.ListenAndServe())
}
```

JSON Marshal / Unmarshal

```
type Alert struct {
    Name     string    `json:"name"`
    Severity string    `json:"severity,omitempty"`
    FiredAt  time.Time `json:"fired_at"`
}

b, err := json.Marshal(alert)              // struct → []byte
err = json.Unmarshal(b, &alert)            // []byte → struct
err = json.NewDecoder(r.Body).Decode(&alert)   // straight from an io.Rea
```

os/exec

```
out, err := exec.Command("kubectl", "get", "pods", "-o", "json").Output()
if err != nil {
    var exitErr *exec.ExitError
    if errors.As(err, &exitErr) {
        log.Printf("exit %d: %s", exitErr.ExitCode(), exitErr.Stderr)
    }
}

cmd := exec.CommandContext(ctx, "sh", "-c", "long-running-task")   // kil
cmd.Stdout, cmd.Stderr = os.Stdout, os.Stderr
err = cmd.Run()
```

time

```
now := time.Now()
now.Format(time.RFC3339)                          // 2025-07-23T09:09:00Z
t, _ := time.Parse("2006-01-02", "2025-07-23")    // reference date layou
elapsed := time.Since(start)

ticker := time.NewTicker(30 * time.Second)   // defer ticker.Stop()
for range ticker.C { poll() }
```

Goroutines are a different world from Python's GIL-constrained threads. If you work in both languages, my Python Threading & Concurrency Cheatsheet makes a handy side-by-side.

> Enrich your learning with **Ansible Cheatsheet**

Related Cheatsheets

- Python Threading & Concurrency is the contrast case: the GIL, ThreadPoolExecutor, and asyncio next to goroutines and channels
- Docker has the multi-stage builds that shrink a Go binary into a scratch image, plus the run flags and compose reference
- Kubernetes is where Go tooling lives in production: kubectl reference, manifest skeletons, rollouts, and debugging

There's more in the Languages group, or start from the whole cheatsheet library.

Python Basics & CLI Cheatsheet

Python shows up in every DevOps job I've had, from quick log parsers to full AWS automation. These are the syntax bits and CLI commands I look up most often. For AWS scripting specifically, see my Python Boto3 Cheatsheet.

Variables & Types

> **Docs:** *Built-in Types, Python 3 docs*
>
> > *Expand your knowledge with **Bash Basics Cheatsheet***

```
name = "karandeep"          # str
count = 42                  # int
ratio = 0.75                # float
enabled = True              # bool
nothing = None              # NoneType

# Multiple assignment and swap
a, b = 1, 2
a, b = b, a

# Type checks and conversion
isinstance(count, int)      # True
int("42"), str(42), float("0.5"), bool("")   # 42, '42', 0.5, False
```

Strings & f-strings

> **Docs:** *String methods, Python 3 docs*

> *Deepen your understanding in **Bash Basics Cheatsheet***

```
s = "prod-web-01"
s.upper(), s.lower(), s.title()
s.startswith("prod"), s.endswith("01")
s.split("-")                # ['prod', 'web', '01']
"-".join(["prod", "db", "02"])
s.replace("web", "app")
s.strip(), s.lstrip(), s.rstrip()
"web" in s                  # True

# f-strings (use these, always)
env, cpu = "prod", 87.5432
f"{env=}"                   # "env='prod'" - great for debugging
f"{cpu:.2f}%"               # "87.54%"
f"{count:>8}"               # right-align, width 8
f"{1024**3:,}"              # "1,073,741,824"
f"{0.875:.1%}"              # "87.5%"
```

Lists, Dicts, Sets & Tuples

Docs: *Data Structures in the Python tutorial*

> *Explore this further in **Bash sed & awk Cheatsheet***

```python
# list - ordered, mutable
hosts = ["web1", "web2"]
hosts.append("web3"); hosts.extend(["db1"]); hosts.insert(0, "lb1")
hosts.remove("web1"); last = hosts.pop()
hosts[0], hosts[-1], hosts[1:3], hosts[::-1]     # index, last, slice, re
sorted(hosts), len(hosts), "db1" in hosts

# tuple - ordered, immutable (good dict keys)
point = ("us-east-1", "a")

# dict - key/value
cfg = {"env": "prod", "replicas": 3}
cfg["env"]; cfg.get("region", "us-east-1")       # .get avoids KeyError
cfg["region"] = "eu-west-1"
cfg.keys(), cfg.values(), cfg.items()
merged = cfg | {"replicas": 5}                    # 3.9+ merge operator
cfg.setdefault("tags", []).append("blue")

# set - unique, unordered
seen = {"web1", "web2"}
seen.add("web3"); seen.discard("web1")
{"a", "b"} & {"b", "c"}     # intersection {'b'}
{"a", "b"} | {"b", "c"}     # union
{"a", "b"} - {"b"}          # difference
```

Comprehensions

Docs: *List comprehensions, Python 3 tutorial*

> *Discover related concepts in **Python Threading & Concurrency Cheatsheet***

```
squares = [x * x for x in range(10)]
evens   = [x for x in range(10) if x % 2 == 0]
by_name = {h: len(h) for h in hosts}                # dict comprehensior
unique  = {line.split()[0] for line in log_lines}   # set comprehension
gen     = (x * x for x in range(10**9))             # generator - lazy,
flat    = [x for row in matrix for x in row]        # nested flatten
```

Functions

Docs: *Defining functions, Python 3 tutorial*

> *Uncover more details in **AWS CloudFormation Cheatsheet***

```
def deploy(service: str, replicas: int = 1, *, dry_run: bool = False) ->
    """Args after * are keyword-only."""
    return f"deploy {service} x{replicas} dry_run={dry_run}"

deploy("api", replicas=3, dry_run=True)

def merge_tags(*args, **kwargs): ...     # variadic positional / keyword
run = lambda cmd: cmd.split()            # small throwaway functions

# Never use a mutable default
def add_host(host, hosts=None):
    hosts = hosts if hosts is not None else []
```

Classes (briefly)

Docs: *Python dataclasses docs*

> *Journey deeper into this topic with **Jenkins Cheatsheet***

```python
from dataclasses import dataclass, field

@dataclass
class Server:
    name: str
    env: str = "dev"
    tags: list[str] = field(default_factory=list)

    def fqdn(self) -> str:
        return f"{self.name}.{self.env}.example.com"

s = Server("web1", env="prod")
s.fqdn()                    # 'web1.prod.example.com'
```

Error Handling

Docs: *Errors and Exceptions, Python 3 tutorial*

> *Enrich your learning with **Python Boto3 Cheatsheet***

```python
try:
    result = 10 / int(value)
except (ValueError, ZeroDivisionError) as e:
    print(f"bad input: {e}")
except Exception:
    raise                    # re-raise unexpected errors - don't swallow
else:
    print("only runs if no exception")
finally:
    cleanup()                # always runs

# Raise your own, chain the cause
raise RuntimeError("deploy failed") from e

# Context managers handle cleanup for you
with open("app.log") as f, open("out.log", "w") as out:
    out.write(f.read())
```

File I/O & pathlib

Docs: *pathlib on docs.python.org*

> *Gain comprehensive insights from **Python Threading & Concurrency Cheatsheet***

```python
from pathlib import Path
import json

p = Path("/etc/app/config.json")
p.exists(), p.is_file(), p.is_dir()
p.name, p.stem, p.suffix, p.parent       # config.json, config, .json, /e
p.read_text(), p.write_text("data")
p.read_bytes()

cfg = json.loads(Path("config.json").read_text())
Path("out.json").write_text(json.dumps(cfg, indent=2))

# Build paths with / - no os.path.join
log_dir = Path.home() / "logs" / "app"
log_dir.mkdir(parents=True, exist_ok=True)

# Globbing
for f in Path("/var/log").glob("*.log"): ...
for f in Path(".").rglob("**/*.yaml"): ...       # recursive

# Stream large files line by line
with open("huge.log") as f:
    for line in f:
        if "ERROR" in line:
            print(line, end="")
```

Dates & Times

Docs: *Python datetime docs*

> *Master this concept through **Kubernetes Cheatsheet***

```python
from datetime import datetime, timedelta, timezone, date

now_utc = datetime.now(timezone.utc)          # always prefer aware datet
today   = date.today()

# Format / parse
now_utc.strftime("%Y-%m-%d %H:%M:%S")         # '2025-05-12 09:15:00'
now_utc.isoformat()                           # '2025-05-12T09:15:00+00:0
datetime.strptime("2025-05-12", "%Y-%m-%d")
datetime.fromisoformat("2025-05-12T09:15:00+00:00")

# Arithmetic
deadline = now_utc + timedelta(days=7, hours=3)
age = now_utc - datetime(2022, 7, 23, tzinfo=timezone.utc)
age.days, age.total_seconds()

# Epoch conversions (log timestamps, APIs)
now_utc.timestamp()                           # datetime -> epoch float
datetime.fromtimestamp(1752570000, tz=timezone.utc)

# Named zones without third-party libs (3.9+)
from zoneinfo import ZoneInfo
datetime.now(ZoneInfo("America/Edmonton"))
```

Sorting

Docs: *Sorting Techniques, Python 3 HOWTO*

> *Delve into specifics at* ***Ubuntu Terminal Cheatsheet***

```python
nums = [3, 1, 2]
sorted(nums)                          # new list; nums.sort() sorts in plac
sorted(nums, reverse=True)

# Sort by a key
hosts = ["web10", "web2", "db1"]
sorted(hosts, key=len)
sorted(hosts, key=str.lower)

# Sort dicts / tuples / objects
servers = [{"name": "web1", "cpu": 80}, {"name": "db1", "cpu": 45}]
sorted(servers, key=lambda s: s["cpu"], reverse=True)

from operator import itemgetter, attrgetter
sorted(servers, key=itemgetter("cpu"))
sorted(instances, key=attrgetter("launch_time"))

# Multi-key: env ascending, then cpu descending
sorted(servers, key=lambda s: (s["env"], -s["cpu"]))

# Top/bottom N without a full sort
import heapq
heapq.nlargest(3, servers, key=itemgetter("cpu"))

# Keep a list sorted while inserting
import bisect
bisect.insort(nums, 2)
```

Queues, Stacks & Heaps

> **Docs:** *collections.deque, Python 3 docs*

```
# deque - O(1) at both ends: stack AND queue
from collections import deque
q = deque(["job1", "job2"])
q.append("job3")           # enqueue right
q.popleft()                # dequeue left  -> FIFO queue
q.pop()                    # pop right     -> LIFO stack
tail = deque(open("app.log"), maxlen=10)   # like tail -n 10

# heapq - priority queue on a plain list
import heapq
pq = []
heapq.heappush(pq, (2, "deploy"))
heapq.heappush(pq, (1, "hotfix"))
heapq.heappop(pq)          # (1, 'hotfix') - smallest priority first

# queue.Queue - thread-safe, for producer/consumer pipelines
from queue import Queue
jobs = Queue()
jobs.put("build"); jobs.get(); jobs.task_done()
```

The thread-safe `queue.Queue` matters most once worker pools are involved. Full producer/consumer patterns are in the Python Threading & Concurrency Cheatsheet.

> Deepen your understanding in **Bash Basics Cheatsheet**

0

URLs & HTTP

> ***Docs:*** *urllib.request on docs.python.org*

```
# Parse and build URLs
from urllib.parse import urlparse, urlencode, quote, parse_qs
u = urlparse("https://api.example.com/v1/deploy?env=prod&tag=v2")
u.scheme, u.netloc, u.path       # 'https', 'api.example.com', '/v1/deplc
parse_qs(u.query)                # {'env': ['prod'], 'tag': ['v2']}
urlencode({"env": "prod", "app": "my api"})   # 'env=prod&app=my+api'
quote("path with spaces")                     # 'path%20with%20spaces'

# GET JSON - stdlib only, no dependencies
import json, urllib.request
with urllib.request.urlopen("https://api.github.com/zen", timeout=10) as
    body = r.read().decode()
    r.status, r.headers["Content-Type"]

# POST JSON
req = urllib.request.Request(
    "https://api.example.com/v1/deploy",
    data=json.dumps({"env": "prod"}).encode(),
    headers={"Content-Type": "application/json", "Authorization": "Bearer
    method="POST",
)
with urllib.request.urlopen(req, timeout=10) as r:
    result = json.load(r)
```

For anything beyond a quick script, just `pip install requests` and get sessions, retries, and connection pooling with far less ceremony.

> Deepen your understanding in **Bash Basics Cheatsheet**

1

CLI: Interpreter Flags

> ***Docs:*** *Command line and environment, Python 3 docs*

> *Deepen your understanding in **Bash Basics Cheatsheet***

2

Command	What it does
`python -c "print(1+1)"`	Run a one-liner
`python -i script.py`	Run script, then drop into REPL with its state
`python -m module`	Run a module as a script (uses your venv's version)
`python -u script.py`	Unbuffered stdout, needed in containers/CI logs
`python -X dev script.py`	Dev mode: extra warnings, asyncio debug
`python -V` / `python -VV`	Version / verbose version
`python -q`	REPL without the banner

CLI: Useful `-m` Modules

Docs: *The -m option, Python 3 docs*

> *Deepen your understanding in **Bash Basics Cheatsheet***

3

```
python -m http.server 8000              # instant file server in cwd
python -m json.tool < data.json         # pretty-print / validate JSON
python -m venv .venv                    # create virtualenv
python -m pip install requests          # pip tied to *this* interpreter
python -m site                          # show sys.path and site-packages
python -m timeit "'-'.join(map(str, range(100)))"
python -m calendar 2026
python -m zipfile -c out.zip src/       # create a zip
python -m tarfile -e release.tar.gz     # extract a tarball
python -m unittest discover             # run tests
python -m pdb script.py                 # step-through debugger
```

venv & pip

Docs: *Python venv docs*

> *Deepen your understanding in **Bash Basics Cheatsheet***

4

```
python -m venv .venv
source .venv/bin/activate               # Linux/macOS
.venv\Scripts\activate                  # Windows
deactivate

pip install requests boto3              # install
pip install -r requirements.txt         # from file
pip install -e .                        # editable local package
pip install 'urllib3<2'                 # version constraint
pip list --outdated
pip show requests                       # version, deps, location
pip freeze > requirements.txt           # pin exact versions
pip uninstall -y requests
```

tip

Always use `python -m pip` instead of bare `pip`. It guarantees you're installing into the interpreter you think you are, which saves you from the classic "installed it but import fails" mystery on hosts with multiple Pythons.

argparse Skeleton

Docs: *Python argparse docs*

> *Deepen your understanding in **Bash Basics Cheatsheet***

5

```
#!/usr/bin/env python3
import argparse

def main() -> None:
    parser = argparse.ArgumentParser(description="Deploy helper")
    parser.add_argument("service", help="service name")
    parser.add_argument("-e", "--env", default="dev", choices=["dev", "st
    parser.add_argument("-r", "--replicas", type=int, default=1)
    parser.add_argument("--dry-run", action="store_true")
    parser.add_argument("-v", "--verbose", action="count", default=0)
    args = parser.parse_args()

    print(f"deploying {args.service} to {args.env} x{args.replicas}")

if __name__ == "__main__":
    main()
```

```
./deploy.py api -e prod -r 3 --dry-run
./deploy.py --help                          # auto-generated help
```

Scripting Essentials

> ***Docs:*** *Python subprocess docs*

```
import sys, os, subprocess

sys.argv                                    # raw CLI args
sys.exit(1)                                 # non-zero = failure, scripts sh
os.environ.get("AWS_REGION", "us-east-1")

# Run shell commands - capture output, raise on failure
result = subprocess.run(
    ["git", "rev-parse", "--short", "HEAD"],
    capture_output=True, text=True, check=True,
)
sha = result.stdout.strip()
```

When a script outgrows a single loop and needs parallel downloads or API calls, jump to the Python Threading & Concurrency Cheatsheet.

> Deepen your understanding in **Bash Basics Cheatsheet**

6

Related Cheatsheets

> Python Threading & Concurrency: Thread, Lock, ThreadPoolExecutor, and asyncio, with a decision table for I/O-bound vs CPU-bound work

> Python Boto3: S3, EC2, Lambda, and DynamoDB automation built on this page's syntax (sessions, paginators, waiters)

> Bash Basics: the other scripting language on every box, with parameter expansion, tests, traps, and arrays

The rest of my Python notes live in the Python group, and everything else is in the cheatsheet library.

python

Python Boto3 Cheatsheet

Boto3 is how I automate everything in AWS that the console makes tedious: fleet-wide tag audits, S3 lifecycle scripts, Lambda deploys from CI. These are the calls I use constantly, plus the paginator/waiter/retry patterns that separate throwaway scripts from reliable ones. Brush up on the language itself in the Python Basics & CLI Cheatsheet if needed.

```
python -m pip install boto3
```

Session, Client, Resource

Docs: *Boto3 credentials guide*

> *Expand your knowledge with **Bash CLI One-Liners Cheatsheet***

```python
import boto3

# Client - low-level, 1:1 with the AWS API. Prefer this; it covers every
s3 = boto3.client("s3", region_name="us-east-1")

# Resource - higher-level OO wrapper (S3, EC2, DynamoDB, and a few others
# in maintenance mode - no new services get resources)
s3r = boto3.resource("s3")

# Session - explicit credentials/profile/region; use for multi-account sc
session = boto3.Session(profile_name="prod", region_name="eu-west-1")
ec2 = session.client("ec2")

# Cross-account via STS assume-role
sts = boto3.client("sts")
creds = sts.assume_role(
    RoleArn="arn:aws:iam::123456789012:role/Automation",
    RoleSessionName="audit",
)["Credentials"]
target = boto3.client(
    "s3",
    aws_access_key_id=creds["AccessKeyId"],
    aws_secret_access_key=creds["SecretAccessKey"],
    aws_session_token=creds["SessionToken"],
)
```

Credential Resolution Order

1. Parameters passed to `boto3.client()` / `Session()`
2. Environment: `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, `AWS_SESSION_TOKEN`
3. Shared credentials file `~/.aws/credentials` (profile via `AWS_PROFILE`)
4. Config file `~/.aws/config` (incl. SSO and assume-role profiles)
5. Container credentials (ECS task role)
6. EC2 instance profile (IMDS)

tip

Never hardcode keys in scripts. On EC2/ECS/Lambda, attach an IAM role and let the chain find it; locally, use `aws sso login` profiles. Your code stays identical everywhere; that's the whole point of the resolution chain.

S3

Docs: *Boto3 S3 API reference*

> *Deepen your understanding in **Low-Cost Cloud Stack Cheatsheet***

```
s3 = boto3.client("s3")

# Upload / download (managed transfers - multipart automatically for big
s3.upload_file("build.tar.gz", "my-bucket", "releases/build.tar.gz",
               ExtraArgs={"ServerSideEncryption": "aws:kms"})
s3.download_file("my-bucket", "releases/build.tar.gz", "/tmp/build.tar.gz

# Small objects in memory
s3.put_object(Bucket="my-bucket", Key="cfg.json", Body=b'{"env":"prod"}')
body = s3.get_object(Bucket="my-bucket", Key="cfg.json")["Body"].read()

# Presigned URL - temporary access without credentials
url = s3.generate_presigned_url(
    "get_object",
    Params={"Bucket": "my-bucket", "Key": "report.pdf"},
    ExpiresIn=3600,          # seconds
)
# For uploads: "put_object", or generate_presigned_post for browser forms

# List with pagination - list_objects_v2 caps at 1000, ALWAYS paginate
paginator = s3.get_paginator("list_objects_v2")
for page in paginator.paginate(Bucket="my-bucket", Prefix="logs/2026/"):
    for obj in page.get("Contents", []):
        print(obj["Key"], obj["Size"])

s3.delete_object(Bucket="my-bucket", Key="old.log")
s3.copy_object(Bucket="my-bucket", Key="dst",
               CopySource={"Bucket": "my-bucket", "Key": "src"})
```

EC2

Docs: *Boto3 EC2 API reference*

> *Explore this further in **Amazon Linux 2 Terminal Cheatsheet***

```python
ec2 = boto3.client("ec2")

# Describe with filters
resp = ec2.describe_instances(Filters=[
    {"Name": "tag:Env", "Values": ["prod"]},
    {"Name": "instance-state-name", "Values": ["running"]},
])
for reservation in resp["Reservations"]:
    for inst in reservation["Instances"]:
        name = next((t["Value"] for t in inst.get("Tags", [])
                     if t["Key"] == "Name"), "-")
        print(inst["InstanceId"], inst["InstanceType"],
              inst.get("PrivateIpAddress"), name)

# Start / stop / reboot / terminate
ec2.stop_instances(InstanceIds=["i-0abc123"])
ec2.start_instances(InstanceIds=["i-0abc123"])
ec2.terminate_instances(InstanceIds=["i-0abc123"], DryRun=True)  # permis

# Tags
ec2.create_tags(Resources=["i-0abc123"],
                Tags=[{"Key": "Owner", "Value": "platform-team"}])
ec2.delete_tags(Resources=["i-0abc123"], Tags=[{"Key": "Temp"}])
```

Lambda

Docs: *Boto3 Lambda API reference*

> *Discover related concepts in **MSK vs Kinesis Cheatsheet (Cost & When to Choose)***

```python
import json, base64
lam = boto3.client("lambda")

# Synchronous invoke
resp = lam.invoke(
    FunctionName="report-gen",
    InvocationType="RequestResponse",     # "Event" = async fire-and-forg
    Payload=json.dumps({"day": "2026-07-23"}),
    LogType="Tail",                       # last 4KB of logs in the respo
)
result = json.loads(resp["Payload"].read())
logs = base64.b64decode(resp["LogResult"]).decode()
if resp.get("FunctionError"):             # function raised - payload hol
    raise RuntimeError(result)

# Deploy new code from a zip
with open("function.zip", "rb") as f:
    lam.update_function_code(FunctionName="report-gen", ZipFile=f.read())
lam.get_waiter("function_updated_v2").wait(FunctionName="report-gen")

lam.update_function_configuration(FunctionName="report-gen",
                                  MemorySize=512, Timeout=30,
                                  Environment={"Variables": {"ENV": "proc
```

DynamoDB

Docs: *Boto3 DynamoDB API reference*

> *Uncover more details in* ***Python Basics & CLI Cheatsheet***

```
from boto3.dynamodb.conditions import Key, Attr

table = boto3.resource("dynamodb").Table("deployments")   # resource hanc

table.put_item(Item={
    "service": "api",              # partition key
    "deployed_at": "2026-07-23T10:00:00Z",   # sort key
    "version": "1.4.2",
    "healthy": True,
})

item = table.get_item(Key={"service": "api",
                           "deployed_at": "2026-07-23T10:00:00Z"}).get("I

# Query - partition key required, sort key optional
resp = table.query(
    KeyConditionExpression=Key("service").eq("api")
                           & Key("deployed_at").begins_with("2026-07"),
    FilterExpression=Attr("healthy").eq(True),
    ScanIndexForward=False,        # newest first
    Limit=10,
)
items = resp["Items"]

# Batch write
with table.batch_writer() as batch:
    for item in items:
        batch.put_item(Item=item)
```

SSM Parameter Store

Docs: *Boto3 SSM API reference*

> *Journey deeper into this topic with **Jenkins Cheatsheet***

```
ssm = boto3.client("ssm")

ssm.put_parameter(Name="/prod/api/db-url", Value="postgres://...",
                  Type="SecureString", Overwrite=True)

val = ssm.get_parameter(Name="/prod/api/db-url",
                        WithDecryption=True)["Parameter"]["Value"]

# All params under a path
paginator = ssm.get_paginator("get_parameters_by_path")
for page in paginator.paginate(Path="/prod/api/", Recursive=True,
                               WithDecryption=True):
    for p in page["Parameters"]:
        print(p["Name"], "=", p["Value"])
```

Error Handling with ClientError

Docs: *Boto3 error handling guide*

```
from botocore.exceptions import ClientError, NoCredentialsError

try:
    s3.head_object(Bucket="my-bucket", Key="maybe-missing.txt")
except ClientError as e:
    code = e.response["Error"]["Code"]
    if code == "404":                          # head_object returns 404,
        print("object does not exist")
    elif code in ("AccessDenied", "403"):
        print("check IAM policy")
    else:
        raise
except NoCredentialsError:
    print("no credentials found - check the resolution chain")
```

Common codes worth branching on: `NoSuchKey`, `NoSuchBucket`, `AccessDenied`, `Throttling`, `ThrottlingException`, `ResourceNotFoundException` (DynamoDB/Lambda), `ParameterNotFound` (SSM), `ConditionalCheckFailedException` (DynamoDB).

> Enrich your learning with **Python Basics & CLI Cheatsheet**

Paginators & Waiters

> ***Docs:*** *Boto3 paginators guide*
>
> > *Gain comprehensive insights from **AWS CloudFormation Cheatsheet***

```
# Any list/describe call with a NextToken has a paginator - never loop tc
ec2_pag = ec2.get_paginator("describe_instances")
ids = [i["InstanceId"]
       for page in ec2_pag.paginate()
       for r in page["Reservations"]
       for i in r["Instances"]]

# Waiters - poll until a state is reached, with sane backoff built in
ec2.get_waiter("instance_running").wait(
    InstanceIds=["i-0abc123"],
    WaiterConfig={"Delay": 15, "MaxAttempts": 40},
)
s3.get_waiter("object_exists").wait(Bucket="my-bucket", Key="flag.done")
```

Handy waiters	Service
`instance_running`, `instance_stopped`, `instance_terminated`	EC2
`object_exists`, `bucket_exists`	S3
`function_active_v2`, `function_updated_v2`	Lambda
`table_exists`, `table_not_exists`	DynamoDB

Retries & Client Config

> ***Docs:*** *Boto3 retries guide*

```
from botocore.config import Config

cfg = Config(
    region_name="us-east-1",
    retries={"max_attempts": 10, "mode": "adaptive"},  # adaptive throttl
    connect_timeout=5,
    read_timeout=60,
    max_pool_connections=50,       # raise when fanning out across threac
)
s3 = boto3.client("s3", config=cfg)
```

Retry modes: `legacy` (old default), `standard` (sane default, jittered backoff), `adaptive` (standard + client-side rate limiting, best for scripts hammering one API). Clients are thread-safe; share one client across a `ThreadPoolExecutor` fan-out (see the Python Threading & Concurrency Cheatsheet) but give each process its own.

> Master this concept through **Python Basics & CLI Cheatsheet**

Related Cheatsheets

- Python Threading & Concurrency: ThreadPoolExecutor patterns that turn slow serial AWS calls into fast parallel fan-outs
- AWS CloudFormation: declaring these resources in templates instead, with anatomy, intrinsic functions, and the change-set workflow
- Amazon Linux 2 Terminal: yum, amazon-linux-extras, IMDSv2, and SSM on the EC2 instances these scripts manage

More Python sheets are in the Python group; everything I've written up is in the cheatsheet library.

python

Python Threading & Concurrency Cheatsheet

Half my automation scripts start out sequential and end up needing to hit 200 hosts at once. That's when this sheet comes out. Python has four concurrency tools (threads, processes, executors, asyncio), and picking the wrong one for the workload is the classic mistake. If you're new to Python itself, start with the Python Basics & CLI Cheatsheet first.

The GIL in One Minute

> ***Docs:*** *Global interpreter lock in the Python glossary*
>
> > *Expand your knowledge with **Docker Cheatsheet***

> CPython's Global Interpreter Lock allows only **one thread to execute Python bytecode at a time**.

> Threads still help for **I/O-bound** work (HTTP calls, disk, DB, subprocess) because the GIL is released while waiting on I/O.

> Threads do **not** help CPU-bound work (parsing, hashing, compression). Use processes for that.

> C extensions (numpy, hashlib on large buffers) often release the GIL internally, so results vary. Measure.

Which One to Use When

> ***Docs:*** *Concurrent Execution, Python 3 docs*
>
> > *Deepen your understanding in **MSK vs Kinesis Cheatsheet (Cost & When to Choose)***

Workload	Best tool	Why
I/O-bound, few dozen tasks	`ThreadPoolExecutor`	Simple, GIL released during I/O
I/O-bound, thousands of tasks	`asyncio`	One thread, cheap coroutines
CPU-bound	`ProcessPoolExecutor` / `multiprocessing`	Real parallelism, sidesteps GIL
Mixed pipeline, decoupled stages	threads + `queue.Queue`	Producer/consumer backpressure
Fire-and-forget background work	`Thread(daemon=True)`	Dies with the main process

threading Module

> **Docs:** *Python threading docs*
>
> > *Explore this further in **Ansible Cheatsheet***

Thread

```
import threading

def worker(host: str) -> None:
    print(f"[{threading.current_thread().name}] pinging {host}")

t = threading.Thread(target=worker, args=("web1",), name="ping-1", daemor
t.start()
t.join(timeout=5)            # wait for it (with a timeout)
t.is_alive()                 # True if still running
```

Lock & RLock

```
lock = threading.Lock()
counter = 0

def increment() -> None:
    global counter
    with lock:               # always use `with` - releases on exception
        counter += 1

# RLock: same thread may re-acquire (needed when locked methods call eacl
rlock = threading.RLock()

class Registry:
    def __init__(self):
        self._lock = threading.RLock()
        self._data = {}

    def set(self, k, v):
        with self._lock:
            self._data[k] = v

    def set_many(self, items):
        with self._lock:          # re-enters fine because it's an RLock
            for k, v in items:
                self.set(k, v)
```

Event, Semaphore, Condition

```python
# Event - one-shot broadcast flag (shutdown signals, "ready" gates)
stop = threading.Event()

def poller():
    while not stop.is_set():
        check_health()
        stop.wait(timeout=5)  # sleep, but wake immediately on set()

stop.set()                    # tell all waiters to stop

# Semaphore - cap concurrency (e.g. max 5 simultaneous API calls)
sem = threading.BoundedSemaphore(5)

def call_api(url):
    with sem:
        return fetch(url)

# Condition - wait for a state change under a lock
cond = threading.Condition()
items = []

def consumer():
    with cond:
        cond.wait_for(lambda: len(items) > 0, timeout=10)
        item = items.pop()

def producer(x):
    with cond:
        items.append(x)
        cond.notify()         # notify_all() to wake every waiter
```

queue.Queue Producer/Consumer

> **Docs:** *Python queue docs*

```python
import queue, threading

q: queue.Queue = queue.Queue(maxsize=100)   # maxsize gives you backpress
SENTINEL = object()

def producer():
    for job in jobs:
        q.put(job)                # blocks when full
    q.put(SENTINEL)

def consumer():
    while True:
        job = q.get()             # blocks when empty; q.get(timeout=5) r
        if job is SENTINEL:
            q.put(SENTINEL)       # let sibling consumers see it too
            break
        try:
            process(job)
        finally:
            q.task_done()

threads = [threading.Thread(target=consumer, daemon=True) for _ in range(
[t.start() for t in threads]
producer()
q.join()                          # blocks until every task_done() is cal
```

`queue.LifoQueue` (stack) and `queue.PriorityQueue` (`(priority, item)` tuples) share the same API.

Discover related concepts in **Python Basics & CLI Cheatsheet**

concurrent.futures

Docs: *concurrent.futures on docs.python.org*

The interface I reach for 90% of the time. Same code works for threads and processes.

Uncover more details in [Jenkins Cheatsheet]

```
from concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor, a

urls = [f"https://host{i}/health" for i in range(50)]

# map - results in submission order
with ThreadPoolExecutor(max_workers=10) as pool:
    for status in pool.map(fetch, urls, timeout=30):
        print(status)

# submit + as_completed - results as they finish, per-task error handling
with ThreadPoolExecutor(max_workers=10) as pool:
    futures = {pool.submit(fetch, url): url for url in urls}
    for fut in as_completed(futures, timeout=60):
        url = futures[fut]
        try:
            print(url, fut.result())
        except Exception as e:
            print(f"{url} failed: {e}")

# CPU-bound? One-line switch:
with ProcessPoolExecutor(max_workers=4) as pool:
    checksums = list(pool.map(sha256_file, big_files))
```

warning

`pool.map` swallows nothing: the first task that raised will re-raise when you iterate its result, killing the loop. For fan-out where some tasks may fail (health checks across a fleet), use `submit` + `as_completed` and try/except each `fut.result()`.

multiprocessing Basics

Docs: *Python multiprocessing docs*

```
import multiprocessing as mp

def crunch(chunk):
    return sum(x * x for x in chunk)

if __name__ == "__main__":                # REQUIRED guard on every entry
    with mp.Pool(processes=mp.cpu_count()) as pool:
        results = pool.map(crunch, chunks)

    # Explicit Process + IPC
    q = mp.Queue()
    p = mp.Process(target=worker, args=(q,))
    p.start(); p.join()

    # Shared state (rarely needed - prefer passing data)
    counter = mp.Value("i", 0)
    with counter.get_lock():
        counter.value += 1
```

Notes: arguments and results must be picklable; each process has its own memory (no shared globals); default start method is `spawn` on macOS/Windows and since 3.14 on Linux too, so keep everything under the `__main__` guard.

Journey deeper into this topic with **Python Basics & CLI Cheatsheet**

asyncio Quick Reference

Docs: *Python asyncio docs*

> *Enrich your learning with **Amazon Linux 2 Terminal Cheatsheet***

```python
import asyncio
import aiohttp   # pip install aiohttp - requests is blocking, don't use

async def fetch(session: aiohttp.ClientSession, url: str) -> int:
    async with session.get(url) as resp:
        return resp.status

async def main() -> None:
    async with aiohttp.ClientSession() as session:
        # gather - run concurrently, keep order
        statuses = await asyncio.gather(
            *(fetch(session, u) for u in urls),
            return_exceptions=True,          # exceptions become results,
        )

        # Tasks - start now, await later
        task = asyncio.create_task(fetch(session, "https://x/health"))
        status = await task

        # Timeouts
        try:
            async with asyncio.timeout(5):           # 3.11+
                await fetch(session, slow_url)
        except TimeoutError:
            print("gave up")
        # pre-3.11 equivalent: await asyncio.wait_for(coro, timeout=5)

        # Structured concurrency (3.11+) - all-or-nothing task group
        async with asyncio.TaskGroup() as tg:
            for u in urls:
                tg.create_task(fetch(session, u))

asyncio.run(main())
```

```
# Other essentials
await asyncio.sleep(1)                          # never time.sleep() in asyr
sem = asyncio.Semaphore(10)                     # cap concurrent coroutines
async with sem: ...
await asyncio.to_thread(blocking_func, arg)  # run blocking code without
```

Common Pitfalls

> ***Docs:*** *Developing with asyncio, Python 3 docs*

Pitfall	Fix
`time.sleep()` inside async code	`await asyncio.sleep()`
Shared `counter += 1` across threads	Wrap in `Lock`; `+=` is not atomic
Forgetting `if __name__ == "__main__"` with multiprocessing	Spawn re-imports your module: infinite process bomb
Unbounded thread creation per request	Use an executor with `max_workers`
Daemon threads doing writes at shutdown	Non-daemon + `Event` for clean shutdown
Calling `fut.result()` with no timeout	Pass `timeout=` so hung tasks can't hang you

Go handles all this very differently, with goroutines that are cheap and CPU-parallel by default. See the Go (Golang) Cheatsheet for the comparison.

> Gain comprehensive insights from **Ansible Cheatsheet**

Related Cheatsheets

- Python Basics & CLI: comprehensions, pathlib, argparse, and the interpreter flags these concurrency examples assume
- Go (Golang): goroutines, channels, and select, which is what concurrency looks like without a GIL
- Python Boto3: where I use these pools most, for parallel S3 transfers and multi-region API sweeps

There's more in the Python group, and the full cheatsheet library has everything else.

terminal

Amazon Linux 2 Terminal Cheatsheet

A lot of my EC2 fleet still runs Amazon Linux 2, and it has just enough quirks (amazon-linux-extras, IMDSv2, SSM instead of SSH) that muscle memory from Ubuntu doesn't fully transfer. This is my working reference for AL2 boxes. If you're coming from Debian-land, my Ubuntu Terminal Cheatsheet is the apt-side mirror of this page.

yum - Package Management

> **Docs:** *Amazon Linux basics in the EC2 User Guide*

```
sudo yum update -y                    # update everything
sudo yum check-update                 # what would update (exit 100 = upda
sudo yum install -y htop              # install
sudo yum install -y https://example.com/pkg.rpm   # install from URL
sudo yum remove htop                  # remove
yum search nginx                      # search names/summaries
yum info docker                       # version, repo, description
yum list installed | grep kernel      # installed packages
sudo yum history                      # transaction history
sudo yum history undo 12              # roll back a transaction
sudo yum clean all                    # clear caches
repoquery -l htop                     # files a package installs (yum-util
```

Repo management with `yum-config-manager` (from `yum-utils`):

```
sudo yum install -y yum-utils
yum repolist                                # enabled repos
sudo yum-config-manager --enable epel
sudo yum-config-manager --disable epel
sudo yum-config-manager --add-repo https://repo.example.com/example.repo
```

Security-only patching:

> Expand your knowledge with **Ansible Cheatsheet**

```
sudo yum update --security -y        # only security-flagged updates
yum updateinfo list security         # list pending security advisories
yum updateinfo info ALAS2-2026-1234  # details on one advisory
```

amazon-linux-extras

> ***Docs:*** *Amazon Linux basics in the EC2 User Guide*

AL2's mechanism for newer software than core repos carry:

```
amazon-linux-extras list                    # topics and enabled state
sudo amazon-linux-extras enable docker      # enable a topic
sudo amazon-linux-extras install -y docker  # enable + install in one st
sudo amazon-linux-extras install -y epel    # EPEL, the sanctioned way c
sudo amazon-linux-extras install -y nginx1 postgresql14
```

Common topics: `docker`, `epel`, `nginx1`, `postgresql14`, `redis6`, `java-openjdk11`, `python3.8`, `kernel-5.10`.

> Deepen your understanding in **Bash Basics Cheatsheet**

```
# the usual docker-on-AL2 sequence
sudo amazon-linux-extras install -y docker
sudo systemctl enable --now docker
sudo usermod -aG docker ec2-user        # then re-login
```

systemd - Services

> **Docs:** *systemctl(1) on man7.org*

AL2 is systemd-based, so this is the same as any modern distro:

```
systemctl status amazon-ssm-agent
sudo systemctl restart docker
sudo systemctl enable --now nginx        # start now and on boot
systemctl list-units --type=service --state=failed
journalctl -u docker -f                  # follow a unit's logs
journalctl -u cloud-init --no-pager
sudo systemctl daemon-reload             # after editing unit files
```

Note: on AL2 much of syslog still lands in `/var/log/messages` (not `/var/log/syslog` as on Ubuntu).

> Explore this further in **Ubuntu Terminal Cheatsheet**

SSM Session Manager vs SSH

> **Docs:** *Session Manager in the AWS Systems Manager User Guide*

I default to SSM: no open port 22, no key distribution, everything audited in CloudTrail.

```
# from your workstation (needs session-manager-plugin + instance profile
aws ssm start-session --target i-0abc123def456

# port-forward through SSM (e.g. reach an RDS via the instance)
aws ssm start-session --target i-0abc123def456 \
  --document-name AWS-StartPortForwardingSessionToRemoteHost \
  --parameters '{"host":["db.internal"],"portNumber":["5432"],"localPortN

# on the instance: check the agent
sudo systemctl status amazon-ssm-agent
```

Requirements: the `amazon-ssm-agent` (preinstalled on AL2 AMIs), an instance profile with `AmazonSSMManagedInstanceCore`, and outbound HTTPS to SSM endpoints. Classic SSH still works when you need it:

> Discover related concepts in **Python Boto3 Cheatsheet**

```
ssh -i key.pem ec2-user@<public-ip>
```

ec2-user & sudo

> ***Docs:*** *Amazon Linux basics in the EC2 User Guide*

The default user is `ec2-user`, with passwordless sudo baked in via `/etc/sudoers.d/90-cloud-init-users`:

> Uncover more details in **Bash Basics Cheatsheet**

```
sudo -i                                 # root shell
sudo visudo -f /etc/sudoers.d/deploy # add rules safely, syntax-checked
sudo adduser deploy                     # note: adduser is a useradd alias h
sudo passwd deploy                      #   not Ubuntu's interactive wrapper
sudo usermod -aG wheel deploy           # wheel = the sudo group on AL2 (not
```

Instance Metadata (IMDSv2)

> **Docs:** *Instance metadata and user data in the EC2 User Guide*

Always use IMDSv2. It's session-token based, and many hardened AMIs disable IMDSv1 entirely:

```
TOKEN=$(curl -sX PUT "http://169.254.169.254/latest/api/token" \
  -H "X-aws-ec2-metadata-token-ttl-seconds: 21600")

curl -sH "X-aws-ec2-metadata-token: $TOKEN" http://169.254.169.254/latest
curl -sH "X-aws-ec2-metadata-token: $TOKEN" http://169.254.169.254/latest
curl -sH "X-aws-ec2-metadata-token: $TOKEN" http://169.254.169.254/latest
curl -sH "X-aws-ec2-metadata-token: $TOKEN" http://169.254.169.254/latest
curl -sH "X-aws-ec2-metadata-token: $TOKEN" http://169.254.169.254/latest
```

For anything beyond a quick lookup, script against AWS from the instance with the SDK; the instance role's credentials are picked up automatically. See my Python Boto3 Cheatsheet for the patterns.

> Journey deeper into this topic with **AWS CloudFormation Cheatsheet**

cloud-init - Debugging User Data

> **Docs:** *cloud-init documentation*

When an instance doesn't come up configured, this is where to look:

> Enrich your learning with **Low-Cost Cloud Stack Cheatsheet**

```
sudo cat /var/log/cloud-init-output.log   # stdout/stderr of your user-da
sudo less /var/log/cloud-init.log         # cloud-init's own detailed log
cloud-init status --long                  # done / running / error
sudo cat /var/lib/cloud/instance/user-data.txt   # what user data actuall
sudo cloud-init clean --logs && sudo reboot      # force user data to re-
```

CloudWatch Agent Basics

> **Docs:** *Install the CloudWatch agent in the CloudWatch User Guide*

> *Gain comprehensive insights from **Low-Cost Cloud Stack Cheatsheet***

```
sudo yum install -y amazon-cloudwatch-agent
# interactive config wizard → writes JSON config
sudo /opt/aws/amazon-cloudwatch-agent/bin/amazon-cloudwatch-agent-config-

# start with a local config file
sudo /opt/aws/amazon-cloudwatch-agent/bin/amazon-cloudwatch-agent-ctl \
  -a fetch-config -m ec2 -s \
  -c file:/opt/aws/amazon-cloudwatch-agent/etc/amazon-cloudwatch-agent.js

# or pull config from SSM Parameter Store (the fleet-friendly way)
sudo /opt/aws/amazon-cloudwatch-agent/bin/amazon-cloudwatch-agent-ctl \
  -a fetch-config -m ec2 -s -c ssm:AmazonCloudWatch-linux

/opt/aws/amazon-cloudwatch-agent/bin/amazon-cloudwatch-agent-ctl -a statu
sudo tail -f /opt/aws/amazon-cloudwatch-agent/logs/amazon-cloudwatch-ager
```

Timezone & Locale

Docs: *timedatectl(1) on man7.org*

> *Master this concept through **Python Basics & CLI Cheatsheet***

```
timedatectl                                  # current time, TZ, NTP syr
sudo timedatectl set-timezone America/Edmonton
timedatectl list-timezones | grep -i edmonton
localectl status
sudo localectl set-locale LANG=en_US.UTF-8
chronyc tracking                             # NTP sync details (Amazon
```

AL2 vs AL2023

Docs: *Comparing AL2 and AL2023 in the Amazon Linux 2023 User Guide*

> *Delve into specifics at **AWS CloudFormation Cheatsheet***

note

Amazon Linux 2023 replaces `yum` with `dnf` (mostly flag-compatible) and **drops** `amazon-linux-extras` **entirely**; packages live in the main repos. It also uses deterministic upgrades (versioned repositories) instead of continuous updates, and there is no EPEL for AL2023. AL2 hits end-of-life June 2026, so new builds should target AL2023; on it, read `yum` as `dnf` throughout this sheet.

Common Paths

Path	What lives there
`/etc/yum.repos.d/`	yum repo definitions
`/etc/system-release`	AL2 version string
`/var/log/cloud-init-output.log`	User-data script output
`/var/log/cloud-init.log`	cloud-init detailed log
`/var/lib/cloud/instance/`	Current instance's cloud-init state
`/var/log/messages`	General syslog (Ubuntu's `/var/log/syslog` equivalent)
`/var/log/secure`	Auth log (Ubuntu's `/var/log/auth.log` equivalent)
`/var/log/amazon/ssm/`	SSM agent logs
`/opt/aws/amazon-cloudwatch-agent/`	CloudWatch agent binaries, config, logs
`/etc/sudoers.d/90-cloud-init-users`	ec2-user's passwordless sudo grant
`/home/ec2-user/.ssh/authorized_keys`	Key pair injected at launch

Quick Reference

Task	Command
Patch security only	`sudo yum update --security -y`
Enable Docker	`sudo amazon-linux-extras install -y docker`
Shell in without SSH	`aws ssm start-session --target i-...`
Instance ID	IMDSv2 token curl (see above)
Debug user data	`sudo cat /var/log/cloud-init-output.log`
AL2 version	`cat /etc/system-release`

Related Cheatsheets

- Ubuntu Terminal: the apt/Debian side, with netplan, ufw, unattended upgrades, and `do-release-upgrade`
- Ansible: inventories, modules, and playbooks, so you stop configuring instances by hand
- tmux: persistent sessions for long yum updates and SSM/SSH work that has to survive a disconnect

There's more in the Terminal group, and the whole cheatsheet library if you want to browse.

terminal

Atuin Cheatsheet

Atuin replaces shell history with a SQLite database and puts a real search screen on `Ctrl-R`. These are the commands and keys I actually use. Everything checked against **Atuin 18.19.0**.

For what it is and why it is worth installing, see Stop Pressing the Up Arrow. For the panes you run it in, the tmux cheatsheet.

Install and wire up

```
curl --proto '=https' --tlsv1.2 -LsSf https://setup.atuin.sh | sh   # off
brew install atuin                                                  # mac
cargo install atuin --locked                                        # frc
paru -S atuin                                                       # Arc
```

```
echo 'eval "$(atuin init bash)"' >> ~/.bashrc
echo 'eval "$(atuin init zsh)"'  >> ~/.zshrc
echo 'atuin init fish | source'  >> ~/.config/fish/config.fish
```

Reload the shell — already-open tabs never read the new line:

```
source ~/.bashrc      # or: exec bash
source ~/.zshrc       # zsh
exec fish             # fish
```

`atuin init` **flag**	**Effect**
`--disable-ctrl-r`	Leave `Ctrl-R` bound to your shell's own search
`--disable-up-arrow`	Leave the up arrow alone
`--disable-ai`	Leave `?` alone

Shells: **bash**, **zsh**, **fish**, **nushell**, **xonsh**, **PowerShell**.

bash: no extra install needed

`atuin init bash` **bundles bash-preexec**. It uses `ble.sh` or your own `bash-preexec` if you already have one, otherwise it loads its own copy. Requires **bash >= 3.1**.

Env var	**Effect**
`ATUIN_NO_BUILTIN_PREEXEC=1`	Do not load the bundled bash-preexec (use your own)

Only if you want to manage it yourself:

> Expand your knowledge with **Ansible Cheatsheet**

```
# bash-preexec — a single file
curl -fsSL https://raw.githubusercontent.com/rcaloras/bash-preexec/master
  -o ~/.bash-preexec.sh
echo '[[ -f ~/.bash-preexec.sh ]] && source ~/.bash-preexec.sh' >> ~/.bas

# ble.sh — full line editor, also gives syntax highlighting and completic
curl -fsSL https://github.com/akinomyoga/ble.sh/releases/download/nightly
  -o /tmp/ble.tar.xz
tar xJf /tmp/ble.tar.xz -C ~/.local/share
echo 'source ~/.local/share/ble-nightly/ble.sh' >> ~/.bashrc
```

Import existing history

```
atuin import auto          # detect the current shell
atuin import bash
atuin import zsh
atuin import fish
atuin import nu
atuin import xonsh
atuin import powershell
```

Also accepts `zsh-hist-db`, `nu-hist-db`, `xonsh-sqlite`, `replxx`, `resh`.

Import **copies** from your old history file. It does not delete or rewrite it.

> Deepen your understanding in Docker Cheatsheet

Search screen keys

Key	Action
`Ctrl-R`	Open search — press again to **cycle filter mode**
`Up arrow`	Also opens search
`Enter`	Run the selected command
`Tab`	Put it on the prompt to **edit before running**
`Alt-1` … `Alt-9`	Jump to result 1–9
`Esc` / `Ctrl-C`	Cancel

Filter modes

What you are searching. Cycle with repeated `Ctrl-R`, or set `filter_mode` in config.

Discover related concepts in MSK vs Kinesis Cheatsheet (Cost & When to Choose)

Mode	Scope
`global`	Everything, every machine (default)
`host`	This machine only
`session`	This terminal session only
`directory`	Commands run in the current directory
`workspace`	The current git repo
`session-preload`	Session, preloaded

```
atuin search --filter-mode directory deploy
atuin search --filter-mode host kubectl
```

Search modes

How your typing is matched. Set `search_mode` in config.

Mode	Behaviour
`fuzzy`	Letters in order, gaps allowed (default)
`prefix`	Must start with what you typed
`fulltext`	Match anywhere in the command
`daemon-fuzzy`	Fuzzy, served by the background daemon

`skim` was **removed** — still accepted, but only as an alias for `fuzzy`.

> Uncover more details in **Bash CLI One-Liners Cheatsheet**

```
atuin search --search-mode fuzzy "kgp"     # finds: kubectl get pods
```

Non-interactive search

Flag	Does
`--limit N`	Cap results
`--offset N`	Skip the first N
`-c` , `--cwd DIR`	Only commands run in DIR
`--exclude-cwd DIR`	Skip a directory
`-e` , `--exit N`	Only commands that exited N
`--exclude-exit N`	Skip commands that exited N
`--after "1 day ago"`	Only after this time
`-b` , `--before DATE`	Only before this time
`-i` , `--interactive`	Open the search UI

```
atuin search kubectl
atuin search --limit 5 docker
atuin search --cwd /srv/app terraform
atuin search --exit 0 make                # only what worked
atuin search --exclude-exit 0 make        # only what FAILED
atuin search --after "1 day ago" aws
atuin search --before "2026-08-01" psql
```

Sync and account

```
atuin register -u <username> -e <email>    # create account
atuin key                                  # PRINT ENCRYPTION KEY — save
atuin login -u <username>                  # other machines, asks for tha
atuin sync                                 # sync now
atuin sync -f                              # force full sync
atuin status                               # sync status
atuin logout
atuin account delete
```

History is encrypted client-side. **Lose the key and the history is unrecoverable** — the server only ever had ciphertext.

> Enrich your learning with [Bash CLI One-Liners Cheatsheet]

Package configuration

Configuration atuin-sync

`atuin key` prints the only copy of your encryption key. The sync server stores ciphertext and nothing else, so **it cannot reset or recover it for you**.

Save the key somewhere you will still have it after this laptop dies.

Do you accept?

< Yes > < No >

Stats

```
atuin stats                    # since the beginning
atuin stats last week
atuin stats --count 20         # top 20 instead of top 10
atuin stats --ngram-size 2     # count "git commit" separately from "git
```

config.toml

`~/.config/atuin/config.toml` — print the commented default with `atuin default-config`.

> Master this concept through **AWS CloudFormation Cheatsheet**

Option	Default	Notes
`filter_mode`	`global`	Set to `directory` or `workspace` — the setting that matters
`search_mode`	`fuzzy`	`prefix`, `fulltext`, `fuzzy`, `daemon-fuzzy`
`style`	`compact`	Or `full`
`inline_height`	`40`	`0` = full-screen takeover
`show_preview`	`true`	Preview pane for long commands
`enter_accept`	`true`	`false` = Enter puts it on the prompt instead of running
`keymap_mode`	`emacs`	Or `vim-normal` / `vim-insert`
`secrets_filter`	`true`	Keeps token-shaped strings out of the DB — leave on
`auto_sync`	`true`	
`sync_frequency`	`5m`	
`sync_address`	`https://api.atuin.sh`	Change for self-hosted
`workspaces`	`false`	Treat git repos as workspaces
`update_check`	`true`	

```
history_filter = ["^secret-cmd", "^curl .*token"]   # never record these
cwd_filter = ["^/very/secret/directory"]            # never record from t
```

Troubleshooting

```
atuin doctor      # run this first — catches missing shell init, bash-pre
atuin info        # where config and database live
atuin --version
```

`Failed to find $ATUIN_SESSION in the environment` — you are running `atuin search` outside an interactive shell (a script, cron, CI). The shell init normally exports it:

> Delve into specifics at **AWS CloudFormation Cheatsheet**

```
export ATUIN_SESSION=$(atuin uuid)
```

Other subcommands

```
atuin history list        # list history
atuin dotfiles            # sync aliases and env vars
atuin scripts             # save and run parameterised scripts
atuin kv                  # small synced key-value store
atuin wrapped             # year in review
atuin daemon              # background daemon (experimental)
atuin mcp                 # MCP server exposing history search to AI tool
atuin uuid                # generate a UUID
atuin default-config      # print the default config.toml
atuin gen-completions     # shell completions
```

Paths

What	Where
History database	`~/.local/share/atuin/history.db`
Config	`~/.config/atuin/config.toml`
Default sync server	`https://api.atuin.sh`

terminal

tmux Cheatsheet

Half my job happens inside tmux: long-running deploys that survive dropped SSH connections, split panes tailing logs on three servers at once, a scripted dev layout that comes up with one command. These are the commands and keybindings I actually use, nothing more. Pair it with my Bash CLI One-Liners Cheatsheet for what to run inside those panes.

All keybindings below assume the default prefix `Ctrl-b` (written `C-b`). Press the prefix, release, then press the key.

sessions › windows › panes

Three levels, and almost every tmux confusion is mixing them up. ***The session is the thing that survives*** *— it lives on the server, so closing your laptop or dropping SSH detaches you without killing anything. Windows are tabs inside it; panes are splits inside a window.*

Sessions

> ***Docs:*** *Getting Started on the tmux wiki*

```
tmux new -s deploy            # new named session
tmux new -s deploy -d         # new session, detached (great in scripts)
tmux ls                       # list sessions
tmux attach -t deploy         # attach to a session by name
tmux attach -d -t deploy      # attach and detach any other client
tmux kill-session -t deploy   # kill one session
tmux kill-server              # kill everything
tmux rename-session -t 0 work # rename a session
```

Keybinding	Action
`C-b d`	Detach from session (it keeps running)
`C-b s`	Interactive session picker
`C-b $`	Rename current session
`C-b (` / `C-b )`	Previous / next session

The whole point: detach with `C-b d`, close your laptop, SSH back in later, `tmux attach -t deploy`, and your job is still running.

> Expand your knowledge with **Python Boto3 Cheatsheet**

Windows

Docs: *tmux(1) on man.openbsd.org*

> *Discover related concepts in **Bash CLI One-Liners Cheatsheet***

Windows are like tabs inside a session.

> Deepen your understanding in **Python Basics & CLI Cheatsheet**

Keybinding	Action
`C-b c`	Create new window
`C-b ,`	Rename current window
`C-b n`	Next window
`C-b p`	Previous window
`C-b 0-9`	Jump to window by number
`C-b w`	Interactive window picker
`C-b &`	Kill current window (confirms)
`C-b .`	Move window to a new index
`C-b f`	Find window by name

Panes

> ***Docs:*** *tmux(1) on man.openbsd.org*
>
> > *Discover related concepts in **Bash CLI One-Liners Cheatsheet***

Panes split a window into multiple terminals.

Keybinding	Action
`C-b %`	Split vertically (side by side)
`C-b "`	Split horizontally (stacked)
`C-b o`	Cycle to next pane
`C-b ;`	Toggle to last-active pane
`C-b arrows`	Move to pane in that direction
`C-b z`	Zoom pane fullscreen (toggle)
`C-b x`	Kill current pane (confirms)
`C-b {` / `C-b }`	Swap pane left / right
`C-b !`	Break pane out into its own window
`C-b q`	Show pane numbers (press number to jump)
`C-b Space`	Cycle through preset layouts

`C-b z` is the one to memorize. Zoom a pane to read logs full-screen, hit it again to drop back to the split.

> Explore this further in **Atuin Cheatsheet**

Resizing & Layouts

> ***Docs:*** *tmux(1) on man.openbsd.org*

> *Discover related concepts in **Bash CLI One-Liners Cheatsheet***

Keybinding	Action
`C-b C-arrows`	Resize pane by 1 cell (hold prefix, tap arrow)
`C-b M-arrows`	Resize pane by 5 cells
`C-b M-1` … `M-5`	Apply preset layout (even-horizontal, even-vertical, main-horizontal, main-vertical, tiled)

```
tmux resize-pane -D 10          # resize down 10 cells (also -U, -L, -R)
tmux select-layout tiled        # apply a layout from the command line
```

Copy Mode & Scrollback

> ***Docs:*** *Getting Started on the tmux wiki*

Keybinding	Action
`C-b [`	Enter copy mode (scrollback)
`q`	Exit copy mode
`C-b ]`	Paste most recent buffer
`C-b PgUp`	Enter copy mode scrolled up one page

With vi keys enabled (see config below), inside copy mode:

> Uncover more details in **Atuin Cheatsheet**

Key	Action
`k` / `j`, `C-u` / `C-d`	Scroll up / down, half-page up / down
`g` / `G`	Top / bottom of scrollback
`/` then text	Search down (`?` searches up)
`n` / `N`	Next / previous search match
`Space`	Start selection
`Enter` (or `y`)	Copy selection and exit

```
tmux show-buffer              # print last copied buffer
tmux list-buffers             # all paste buffers
tmux save-buffer /tmp/out.txt # dump buffer to a file
```

Synchronize Panes

Docs: *tmux(1) on man.openbsd.org*

> *Discover related concepts in **Bash CLI One-Liners Cheatsheet***

Type into every pane in the window at once. Handy when you have a handful of servers open in splits and want to run the same command on all of them.

> Journey deeper into this topic with **Atuin Cheatsheet**

```
# toggle on/off (prompt shows "synchronize-panes on")
C-b : setw synchronize-panes
```

warning

Turn synchronize-panes **off** before doing anything destructive. It is easy to forget it is on and send `sudo systemctl restart` to four production hosts instead of one. Put the sync state in your status line so it is always visible.

~/.tmux.conf Essentials

Docs: tmux wiki

```
# Remap prefix from C-b to C-a (easier to reach)
unbind C-b
set -g prefix C-a
bind C-a send-prefix

# Mouse support: click panes, drag borders, wheel scroll
set -g mouse on

# Start windows and panes at 1, not 0
set -g base-index 1
setw -g pane-base-index 1

# vi keys in copy mode
setw -g mode-keys vi
bind -T copy-mode-vi v send -X begin-selection
bind -T copy-mode-vi y send -X copy-selection-and-cancel

# Bigger scrollback, faster escape, sane colors
set -g history-limit 50000
set -sg escape-time 10
set -g default-terminal "tmux-256color"

# Renumber windows when one closes
set -g renumber-windows on

# Intuitive split keys that keep the current path
bind | split-window -h -c "#{pane_current_path}"
bind - split-window -v -c "#{pane_current_path}"

# Status line: session name, window list, host + time
set -g status-style bg=black,fg=green
set -g status-left "[#S] "
set -g status-right "#h %H:%M"
```

Reload without restarting:

> Enrich your learning with **Go (Golang) Cheatsheet**

```
tmux source-file ~/.tmux.conf     # or: C-b : source-file ~/.tmux.conf
```

Scripting tmux

> ***Docs:*** *tmux(1) on man.openbsd.org*
>
> > *Discover related concepts in **Bash CLI One-Liners Cheatsheet***

`send-keys` and `split-window` let you build a whole dev environment in one script:

```
#!/usr/bin/env bash
SESSION="dev"

tmux new-session -d -s "$SESSION" -n editor
tmux send-keys -t "$SESSION:editor" 'cd ~/project && vim .' C-m

tmux split-window -h -t "$SESSION:editor"
tmux send-keys -t "$SESSION:editor.1" 'cd ~/project && hugo server -D' C-

tmux new-window -t "$SESSION" -n logs
tmux send-keys -t "$SESSION:logs" 'journalctl -f' C-m

tmux select-window -t "$SESSION:editor"
tmux attach -t "$SESSION"
```

Notes: `C-m` is Enter; targets are `session:window.pane`; `new-session -d` means the script works even when run outside tmux. Idempotent version: `tmux has-session -t dev 2>/dev/null || ./dev-env.sh`.

> Gain comprehensive insights from **Bash Basics Cheatsheet**

Nested tmux (SSH Into a Server Running tmux)

> ***Docs:*** *tmux(1) on man.openbsd.org*
>
> > *Discover related concepts in **Bash CLI One-Liners Cheatsheet***

When you run tmux locally and attach to a remote tmux over SSH, press the prefix **twice** to send it to the inner session: `C-b C-b c` creates a window on the remote. If you nest often, set the remote prefix to something else (e.g. `C-a`) so the two never collide.

> Master this concept through **Low-Cost Cloud Stack Cheatsheet**

Most-Used Commands

> ***Docs:*** *tmux(1) on man.openbsd.org*
>
> > *Discover related concepts in **Bash CLI One-Liners Cheatsheet***

Command	What it does
`tmux new -s name`	New named session
`tmux ls`	List sessions
`tmux attach -t name`	Attach to session
`tmux kill-session -t name`	Kill session
`C-b d`	Detach
`C-b c` / `C-b n` / `C-b p`	New / next / previous window
`C-b %` / `C-b "`	Vertical / horizontal split
`C-b z`	Zoom pane
`C-b [`	Copy mode / scrollback
`C-b : setw synchronize-panes`	Type into all panes

Most of this pays off on remote boxes. See my Ubuntu Terminal Cheatsheet for the server-side commands you'll be running inside those sessions.

> Delve into specifics at **Bash sed & awk Cheatsheet**

Related Cheatsheets

- Bash CLI One-Liners: the find/grep/jq/rsync pipelines I end up typing inside these panes
- Ubuntu Terminal: apt, systemd, and journalctl on the servers these sessions are attached to
- Amazon Linux 2 Terminal: the same idea for EC2, with yum, extras repos, SSM sessions, and instance-metadata queries

The rest of my terminal notes live in the Terminal group, and everything else is in the cheatsheet library.

terminal

Ubuntu Terminal Cheatsheet

Most of the fleet I manage runs Ubuntu LTS, and the same twenty commands cover 90% of the day-to-day: patching with apt, poking services with systemctl, reading journalctl, opening a port in ufw. This is that reference. I run nearly all of it inside a tmux session so a dropped SSH connection doesn't kill an upgrade (see my tmux Cheatsheet).

apt - Package Management

> ***Docs:*** *apt(8) on Ubuntu manpages*

```
sudo apt update                       # refresh package index (always firs
sudo apt upgrade                      # upgrade installed packages
sudo apt full-upgrade                 # upgrade, allowing removals (kernel
apt list --upgradable                 # what would upgrade
sudo apt install nginx                # install
sudo apt install nginx=1.24.0-2ubuntu7   # install a specific version
sudo apt remove nginx                 # remove package, keep config
sudo apt purge nginx                  # remove package AND config files
sudo apt autoremove --purge           # remove orphaned dependencies
sudo apt install -y --no-install-recommends jq   # lean install for serve
```

Inspecting packages:

```
apt show nginx                        # metadata: version, deps, descripti
apt-cache policy nginx                # installed vs candidate version, or
apt search "reverse proxy"            # search descriptions
apt list --installed | grep nginx     # what's installed
sudo apt-mark hold linux-image-generic   # pin a package (unhold to relea
```

Repositories:

```
sudo add-apt-repository ppa:ondrej/php    # add a PPA (runs apt update or
sudo add-apt-repository --remove ppa:ondrej/php
# modern keyring pattern for third-party repos:
curl -fsSL https://example.com/key.gpg | sudo gpg --dearmor -o /etc/apt/k
echo "deb [signed-by=/etc/apt/keyrings/example.gpg] https://repo.example.
  | sudo tee /etc/apt/sources.list.d/example.list
```

Unattended security updates:

> Expand your knowledge with **Ansible Cheatsheet**

```
sudo apt install unattended-upgrades
sudo dpkg-reconfigure --priority=low unattended-upgrades
# config: /etc/apt/apt.conf.d/50unattended-upgrades
sudo unattended-upgrade --dry-run -d      # test what it would do
```

dpkg Basics

Docs: *dpkg(1) on Ubuntu manpages*

> *Deepen your understanding in **Python Basics & CLI Cheatsheet***

```
sudo dpkg -i package.deb              # install a local .deb
sudo apt install ./package.deb        # better: resolves dependencies too
dpkg -l | grep nginx                  # list installed packages
dpkg -L nginx                         # files a package installed
dpkg -S /usr/sbin/nginx               # which package owns this file
sudo dpkg --configure -a              # fix interrupted installs
sudo apt --fix-broken install         # fix broken dependencies
```

snap Basics

Docs: *Snap documentation on snapcraft.io*

> *Explore this further in **Python Basics & CLI Cheatsheet***

```
snap list                             # installed snaps
snap find kubectl                     # search the store
sudo snap install kubectl --classic   # classic = no sandbox confinement
sudo snap refresh                     # update all snaps
sudo snap remove --purge kubectl      # remove including snapshots
snap info kubectl                     # channels and versions
```

systemd - Services

Docs: *systemctl(1) on man7.org*

> *Discover related concepts in **Docker Cheatsheet***

```
systemctl status nginx              # state, PID, recent log lines
sudo systemctl start nginx
sudo systemctl stop nginx
sudo systemctl restart nginx
sudo systemctl reload nginx         # reload config without dropping cor
sudo systemctl enable --now nginx   # start now AND on boot
sudo systemctl disable nginx
systemctl is-active nginx           # scripting-friendly: active/inactiv
systemctl list-units --type=service --state=failed
sudo systemctl daemon-reload        # after editing unit files
sudo systemctl edit nginx           # override unit via drop-in file
```

journalctl - Logs

Docs: *journalctl(1) on man7.org*

> *Uncover more details in **Amazon Linux 2 Terminal Cheatsheet***

```
journalctl -u nginx                 # logs for one unit
journalctl -u nginx -f              # follow (like tail -f)
journalctl -u nginx --since "1 hour ago"
journalctl -u nginx --since "2026-07-22 09:00" --until "2026-07-22 10:00"
journalctl -u nginx -n 100 --no-pager
journalctl -p err -b                # errors since last boot
journalctl --disk-usage             # how much space logs use
sudo journalctl --vacuum-size=500M  # trim the journal
```

ufw - Firewall

Docs: *UFW on the Ubuntu Community Help Wiki*

> *Journey deeper into this topic with **Docker Cheatsheet***

```
sudo ufw status verbose
sudo ufw allow OpenSSH                # ALWAYS before enabling, or you loc
sudo ufw enable
sudo ufw allow 80/tcp
sudo ufw allow 443/tcp
sudo ufw allow from 10.0.0.0/8 to any port 5432 proto tcp   # scoped rule
sudo ufw limit ssh                    # rate-limit brute force
sudo ufw status numbered
sudo ufw delete 3                     # delete rule by number
sudo ufw default deny incoming
sudo ufw default allow outgoing
```

warning

On a remote box, run `sudo ufw allow OpenSSH` **before** `sudo ufw enable`. The default incoming policy is deny, so enabling the firewall without an SSH rule cuts your own session's ability to reconnect.

Users & Permissions

> ***Docs:** Ubuntu Server documentation*

```
sudo adduser deploy                    # interactive: home dir, password, s
sudo usermod -aG sudo deploy           # add to sudo group (-a is critical!
sudo usermod -aG docker deploy         # docker without sudo (log out/in tc
groups deploy                          # check memberships
sudo deluser deploy --remove-home
sudo passwd -l deploy                  # lock an account
sudo visudo                            # edit sudoers safely (syntax-checke
# passwordless sudo for one command, via a drop-in:
echo 'deploy ALL=(ALL) NOPASSWD: /usr/bin/systemctl restart myapp' \
  | sudo tee /etc/sudoers.d/deploy-restart
```

chmod / chown recap:

> Enrich your learning with Low-Cost Cloud Stack Cheatsheet

Command	Effect
`chmod 644 file`	rw-r–r– (configs, regular files)
`chmod 755 dir`	rwxr-xr-x (directories, scripts)
`chmod 600 key.pem`	rw——- (private keys; SSH refuses looser)
`chmod +x script.sh`	Add execute bit
`chmod -R g+w shared/`	Recursive group write
`chown deploy:deploy file`	Change owner and group
`chown -R www-data: /var/www`	Recursive; empty group = user’s group

Networking

Docs: *Netplan documentation*

```
ip a                                # interfaces and addresses
ip r                                # routing table (default gateway)
ss -tulpn                           # listening sockets + owning process
ss -tn state established            # current TCP connections
resolvectl status                   # DNS config (systemd-resolved)
ping -c 4 1.1.1.1
curl -sI https://karandeepsingh.ca | head -5
```

Netplan (Ubuntu's network config since 18.04): YAML in `/etc/netplan/*.yaml`, applied with `sudo netplan apply`. On cloud instances the file is usually generated by cloud-init, so override via `/etc/cloud/cloud.cfg.d/` rather than editing the generated file, or your changes vanish on reboot. `sudo netplan try` applies with an auto-rollback timer, which is the safe option over SSH.

> Gain comprehensive insights from **Kubernetes Cheatsheet**

Release & Version Info

Docs: *Ubuntu Server docs on ubuntu.com*

> *Master this concept through **Python Basics & CLI Cheatsheet***

```
lsb_release -a                          # Ubuntu version and codename
cat /etc/os-release                     # same info, script-friendly
uname -r                                # kernel version
sudo do-release-upgrade                 # upgrade to next LTS (take a snapsh
sudo do-release-upgrade -d              # upgrade to development release
# /etc/update-manager/release-upgrades: Prompt=lts|normal|never
```

update-alternatives

> ***Docs:*** *update-alternatives(1) on Ubuntu manpages*

Manage multiple versions of the same tool:

> Delve into specifics at **AWS CloudFormation Cheatsheet**

```
update-alternatives --list editor
sudo update-alternatives --config editor      # interactive pick
sudo update-alternatives --install /usr/bin/python3 python3 /usr/bin/pyth
sudo update-alternatives --set editor /usr/bin/vim.basic
```

Common Paths

Path	What lives there
`/etc/apt/sources.list.d/`	Third-party apt repo definitions
`/etc/apt/apt.conf.d/`	apt behaviour incl. unattended-upgrades config
`/etc/apt/keyrings/`	Repo signing keys (modern location)
`/etc/netplan/`	Network configuration YAML
`/etc/systemd/system/`	Custom and overridden unit files
`/var/log/syslog`	General system log
`/var/log/auth.log`	Logins, sudo usage, SSH attempts
`/var/log/unattended-upgrades/`	Automatic update history
`/var/log/nginx/`	nginx access and error logs
`/etc/sudoers.d/`	Drop-in sudo rules (use with visudo)
`/etc/cron.d/` , `/etc/cron.daily/`	System cron jobs

If your servers are EC2 instances running Amazon Linux instead, most of this maps over with yum in place of apt. The differences are in my Amazon Linux 2 Terminal Cheatsheet.

> Deepen your understanding in **Python Basics & CLI Cheatsheet**

0

Related Cheatsheets

> Amazon Linux 2 Terminal: the yum/EC2 counterpart, covering extras repos, SSM Session Manager, IMDSv2, and cloud-init logs

> tmux: how I keep `do-release-upgrade` and other long jobs alive through dropped SSH sessions

> Jenkins: for when you stop running these commands by hand and codify them as pipeline stages

More terminal sheets in the Terminal group; the full cheatsheet library has everything else.