

Coding & Scripting for SRE / Platform — EknathaLabs Interview Guide

Platform, SRE & DevOps interview guide

Questions	200 questions · spoken answers included
Online (always latest)	interviewhub.eknathalabs.com/coding_scripting_interview_guide.html
Printed	2026-10-21

Free to use and share for personal study. Answers are written for interview practice, so check command flags and versions against the official documentation before running anything in production.

Q1 **Write a Bash script that exits on any error, unset variable, or failed pipe.**
Why does each flag matter?

Bash / Shell

 Spoken answer · ~83 s

I start production scripts with `set -euo pipefail`, and each flag closes a different class of silent failure. The `e` flag, `errexit`, makes the script stop when a command returns a non-zero status, so a failed step like a copy or a download doesn't let later steps run on bad assumptions. It has caveats: it doesn't trigger inside conditions such as an `if` test or on the left side of an `and-or` list, so I still check critical commands explicitly. The `u` flag, `nounset`, treats a reference to an unset variable as an error, so a typo like a misspelled variable name, or a missing argument, fails immediately instead of expanding to an empty string, which is how scripts end up running `rm -rf` on an empty prefix. The `pipefail` option makes a pipeline return the status of the last command that failed, rather than only the last command's status; without it, a failing producer in a pipe such as a `curl` piped to `grep` is hidden. I also set a safe IFS, quote variables, and add a trap on `ERR` to print the line number. For optional variables I use the default-value syntax, so `nounset` stays useful.

Likely follow-ups

- Where does `set -e` not take effect?
- How do you handle an optional variable under `nounset`?

```
#!/usr/bin/env bash
```

```
set -Eeuo pipefail
```

```
IFS=$'\n\t'
```

This is the standard "strict mode" preamble for any production Bash script. `-e` exits on the first command that returns non-zero, `-u` treats unset variables as an error (catches typos in var names), and `-o pipefail` makes a pipeline fail if *any* stage fails, not just the last. `-E` ensures an `ERR` trap is inherited by functions/subshells. Without `pipefail`, `cmd_that_fails | grep x` returns `grep`'s exit code and silently masks the real failure.

Gotcha: `set -e` does NOT trigger inside `if` conditions or commands joined by `&&||`. Code like `grep x file` in an `if` won't abort the script even with `-e`.

 Official docs

- [GNU Bash reference manual](#)

 Spoken answer · ~83 s

Filenames can contain spaces, tabs, newlines, and leading dashes, so any approach that relies on word splitting breaks. The classic mistake is for `f` in `$(ls)`, which splits on whitespace and also mangles special characters. Safer options: use shell globs directly, for `f` in `./*.log`, quoted as double-quoted variable expansions inside the loop body, which handles spaces and even newlines correctly because the glob expands to separate words without splitting. Prefixing with `./` avoids names starting with a dash being read as options. When I need `find`, I use `find` with `-print0` and pipe to a while loop that does `IFS= read -r -d '' file`, so the delimiter is a null byte, the only character that can't appear in a filename. Alternatively, `find -exec` with a plus sign, or `xargs -0`, passes names safely as arguments. For arrays, `mapfile` with `-d ''` reads null-delimited input. Always quote expansions, use `read -r` to preserve backslashes, and set `IFS` empty on the read to preserve leading and trailing whitespace. Add `shopt -s nullglob` so an unmatched glob expands to nothing instead of the literal pattern. Test with awkward names like one with a space, a newline, and a leading dash.

Likely follow-ups

- Why is null the right delimiter for filenames?
- What does `nullglob` change?

```
# BAD: breaks on spaces/newlines/globs
```

```
for f in $(ls); do rm "$f"; done
```

```
# GOOD: null-delimited, handles every filename
```

```
find . -type f -print0 | while IFS= read -r -d '' f; do
```

```
    printf '%s\n' "$f"
```

```
done
```

Never parse `ls` and never leave variables unquoted in word-splitting contexts. The robust pattern is `find -print0` piped to `read -d ''`, which separates entries by NUL bytes — the only character that can't appear in a filename. `IFS=` prevents trimming, `-r` stops backslash mangling.

Always quote: `"$f"`, not `$f`. Unquoted expansion re-splits on `IFS` and expands globs — the cause of countless "rm deleted the wrong thing" incidents.

 Official docs

- [GNU Bash reference manual](#)

Q3 **Write a function with a trap that cleans up a temp directory even if the script is interrupted.**

Bash / Shell

 Spoken answer · ~83 s

I define a cleanup function and register it with trap on EXIT, plus signals, so the temp directory is removed however the script ends. The pattern: create the directory with `mktemp -d`, which makes a unique, private directory and avoids predictable names; store it in a variable; define cleanup that removes it with `rm -rf`, guarded by a check that the variable is non-empty and points to the directory I created, so a bug can't delete something else; then `trap cleanup EXIT`. The EXIT trap fires on normal completion, on errors under `set -e`, and on exit calls, which covers most cases. For interrupts, I add trap with INT and TERM that call exit with a conventional status, such as 130 for Ctrl-C, because exit then triggers the EXIT trap, avoiding duplicated cleanup logic. Note that SIGKILL can't be trapped, so a `kill -9` will leave the directory behind; for that, I accept the limitation or use a startup sweep of stale directories. Keep the cleanup function idempotent and quiet, preserve the original exit code by capturing it at the beginning of the handler, and don't rely on traps in subshells, because they are not inherited unless set again.

Likely follow-ups

- Why does trapping EXIT cover most cases?
- What can never be trapped?

```
tmpdir=$(mktemp -d)
```

```
cleanup() { rm -rf "$tmpdir"; }
```

```
trap cleanup EXIT INT TERM
```

```
# ... work in "$tmpdir" ...
```

A `trap ... EXIT` runs the cleanup on normal exit, errors (with `-e`), and signals — so the temp dir never leaks. Adding INT/TERM makes Ctrl-C and `kill` clean up too. Use `mktemp -d` rather than a hard-coded path: it creates a unique directory with safe permissions and avoids races/symlink attacks in `/tmp`.

Why SREs care: Scripts that leak temp files or half-written state on interruption are a classic source of "disk full" pages and corrupted runs. Idempotent cleanup is table stakes.

 Start here (official docs)

- [Python documentation](#)

Spoken answer · ~82 s

For simple scripts I use the `getopts` builtin, which handles short options. You pass an option string like `e colon v`, where a colon after a letter means that option takes an argument. A while loop runs `getopts`, and a case statement maps each option: `e` stores the `OPTARG` as the environment, `v` sets a verbose flag, and a final pattern handles unknown options and missing arguments, printing usage and exiting non-zero. After the loop I run `shift` of `OPTIND` minus one to leave positional arguments. I set defaults before parsing, validate required options afterwards, and give a help flag that prints usage. Limitations: `getopts` doesn't support long options like `double-dash env`. For those, I write a while loop over the arguments with a case statement matching both forms, shifting as appropriate, and supporting equals syntax and double dash to end option parsing. The external `getopt` utility handles long options but has portability differences between GNU and BSD versions. For anything complicated, with subcommands or validation, it's usually time to move to Python with `argparse`. Good scripts also validate inputs, such as allowing only `dev`, `staging`, or `prod`, and fail with a clear message and usage text.

Likely follow-ups

- Why does `getopts` not support long options?
- How do you end option parsing with a double dash?

```
verbose=0; env=""

while getopts ":e:v" opt; do

  case "$opt" in

    e) env="$OPTARG" ;;

    v) verbose=1 ;;

    \?) echo "bad flag: -$OPTARG" >&2; exit 2 ;;

    :) echo "-$OPTARG needs an arg" >&2; exit 2 ;;

  esac

done; shift $((OPTIND-1))
```

`getopts` is the built-in for short flags. A colon after a letter (`e:`) means it takes an argument exposed as `$OPTARG`; a leading colon enables silent error handling so you can print your own messages. After the loop, `shift $((OPTIND-1))` drops the parsed flags so `@` holds the remaining positional args.

For long flags (`--env`), `getopts` won't help — use a manual while `[[ $# -gt 0 ]]; case "$1" in --env) ... ;; esac; shift` loop, or switch to Python for anything complex.

Official docs

- [GNU Bash reference manual](#)

Q5 Write a one-liner to find the top 5 IPs by request count in an Nginx access log.

Bash / Shell

Spoken answer · ~86 s

The idiom is a pipeline: extract the field, sort, count duplicates, sort by count, and take the top five. For a default Nginx combined log, the client IP is the first field, so I use `awk` with `print` dollar one to pull it out, pipe to `sort` so identical IPs are adjacent, then `uniq -c` to count each group, then `sort -rn` to order by count descending, and finally `head -5`. Sorting before `uniq` is essential because `uniq` only collapses adjacent duplicates. An `awk`-only version avoids the first sort by using an associative array, incrementing a counter per IP and printing at the end, which is faster on large files because it doesn't sort every line; you then sort the much smaller summary. Operational notes: if there's a load balancer in front, the real client address may be in `X-Forwarded-For`, so the first field would just be the proxy; adjust the field or the log format. For rotated and compressed logs, use `zcat` or `zgrep`. For very large files, consider limiting by time window first, such as with a `grep` on the date, and for repeated analysis, use a proper log tool. Also be aware that in security investigations, counting by IP alone can hide distributed sources.

Likely follow-ups

- Why must you sort before `uniq`?
- What is the `awk` associative-array alternative?

```
awk '{print $1}' access.log | sort | uniq -c | sort -rn | head -5
```

with %:

```
awk '{c[$1]++} END{for(i in c) print c[i],i}' access.log | sort -rn | head -5
```

The classic pipeline: `awk` extracts the first field (client IP), `sort | uniq -c` counts duplicates (`uniq` needs sorted input), and `sort -rn | head` ranks descending. The pure-`awk` version uses an associative array to count in a single pass — faster on large logs because it avoids the first sort.

Interview favorite: Knowing `uniq -c` requires pre-sorted input, and that an `awk` hash map does it in one pass, signals real command-line fluency.

Official docs

- [NGINX documentation](#)

 Spoken answer · ~82 s

Bash itself can open TCP connections through the special `/dev/tcp` pseudo-device, a feature of bash, not a real file. The command is to try redirecting to `/dev/tcp/host/port`, for example running `bash -c` with echo of nothing redirected to `/dev/tcp/10.0.0.5/443`; if the connection succeeds the exit status is zero, and if it fails or is refused the status is non-zero. Wrapping it in the timeout command, such as `timeout 3`, prevents hanging on filtered ports where packets are dropped. It's handy in minimal containers that lack nc, telnet, or curl. Other options without those tools: curl or wget if present, with a connect timeout; Python's socket module with a short timeout, since most hosts have Python; openssl s_client for TLS ports, which also reveals certificate and handshake problems; and ss or netstat on the target side to confirm the service is listening. Caveats: `/dev/tcp` is bash-specific, not available in dash or sh, and some distributions compile bash without it; success only proves the TCP handshake, not that the application works, so follow up with an application-level check; and firewalls or security groups can make a port look closed or filtered differently, so interpret timeouts versus refusals carefully.

Likely follow-ups

- What does a timeout versus a refusal indicate?
- Why isn't a successful connect proof the app is healthy?

```
# Bash built-in /dev/tcp pseudo-device

if timeout 3 bash -c 'echo > /dev/tcp/db.internal/5432' 2>/dev/null; then
    echo "port open"
else
    echo "port closed/unreachable"
fi
```

Bash has a built-in `/dev/tcp/HOST/PORT` pseudo-device — opening it attempts a TCP connection, so you can health-check ports on minimal containers that lack nc, telnet, or curl. Wrapping it in timeout prevents hanging on a filtered/dropped port.

Container-friendly: Distroless and Alpine images often strip networking tools. `/dev/tcp` needs nothing but bash, making it ideal for readiness probes and debug.

 Official docs

- [GNU Bash reference manual](#)

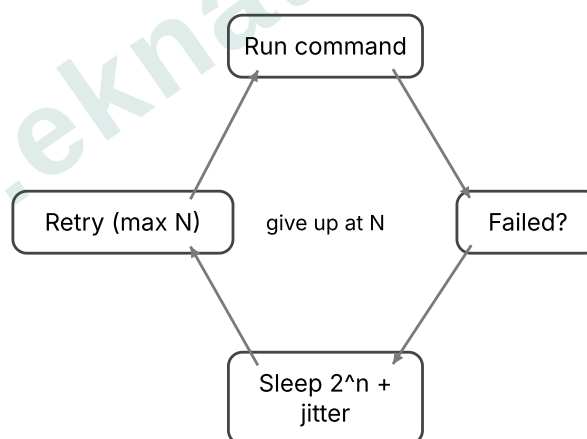
Spoken answer · ~84 s

A retry wrapper takes a maximum attempt count and a command, runs it, and on failure sleeps for an increasing delay before trying again. In Bash I write a function that loops from one to the maximum; it runs the command with the arguments passed through as quoted all-arguments, returns zero on success, and otherwise computes the delay, doubling each time from a base such as 1 second, capped at a ceiling like 30 seconds, plus random jitter from the RANDOM variable, so many clients don't retry in lockstep. After the final attempt it returns the last exit status, with a message to stderr noting the attempts. Design points: only retry errors that are transient, since retrying a bad credential or a 404 just wastes time; with curl, consider the retry options and treat 4xx differently from 5xx; make the operation idempotent, because a retry after a timeout may repeat a side effect; set an overall deadline and per-attempt timeout; log each attempt for debugging; and with `set -e`, put the command in an if condition so a failure doesn't abort the wrapper before it can retry. Exponential backoff with jitter reduces load on a struggling service, rather than adding to it.

Likely follow-ups

- Why add jitter to backoff?
- Why can't you blindly retry non-idempotent operations?

Retry with backoff



```
retry() {  
  
    local max=$1; shift; local n=1 delay=1  
  
    until "$@"; do  
  
        (( n++ > max )) && { echo "failed after $max" >&2; return 1; }  
  
        echo "attempt $n in ${delay}s..." >&2
```

```
sleep "$delay"; delay=$(( delay * 2 ))
```

```
done
```

```
}
```

```
# usage: retry 5 curl -fsS https://api.internal/health
```

This runs "\$@" (the command + args) until it succeeds or hits the max attempts, doubling the delay each time (exponential backoff). "\$@" preserves argument boundaries. For production you'd add jitter (delay + RANDOM%delay) so many clients don't retry in lockstep and create a thundering herd.

Cap the retries: Unbounded retries against a down dependency turn a blip into a self-sustaining overload. Always bound attempts and total time.

Official docs

- [AWS Builders' Library: Timeouts, retries and backoff with jitter](#)
- [GNU Bash reference manual](#)

Q8 What is the difference between \$@ and \$* and "\$@" and "\$*"? Bash / Shell

 Spoken answer · ~83 s

These four forms control how a script's positional parameters are expanded. Unquoted, dollar-at and dollar-star behave the same: all parameters are joined and then subject to word splitting and globbing, so an argument with a space becomes multiple words. Quoted, they differ. Double-quoted dollar-star expands to a single word, with all parameters joined by the first character of IFS, usually a space. So a script called with three arguments sees one string. Double-quoted dollar-at expands to a separate word for each parameter, preserving each argument exactly, including spaces and empty arguments. That is almost always what you want when forwarding arguments, for example calling another command with the double-quoted at sign so that an argument like 'my file.txt' stays intact. The star form is useful when you deliberately want one string, such as building a log message, or joining with a custom separator by setting IFS locally. The same applies to arrays: double-quoted array with at sign expands each element as a separate word, and with star joins them. The rule of thumb: quote everything, and use dollar-at for passing arguments along. Forgetting quotes is a frequent source of bugs and, with untrusted input, of security issues like argument injection.

Likely follow-ups

- When would you deliberately use double-quoted star?
- How does the same rule apply to arrays?

```
$* → all args as ONE word, split on IFS
```

`$@` → all args, split on IFS (usually same as above)

`"$*"` → ONE string: "arg1 arg2 arg3" (joined by first IFS char)

`"$@"` → EACH arg preserved as a separate quoted word ← almost always what you want

The only one that correctly preserves arguments containing spaces is `"$@"` — it expands to each argument individually quoted. Use it whenever you forward args to another command. `"$*"` is occasionally useful when you genuinely want one joined string (e.g. building a log message).

Rule of thumb: If you're passing arguments through, it's `"$@"` every time. Getting this wrong silently corrupts paths with spaces.

 **Start here (official docs)**

- [Python documentation](#)

Q9 How do you redirect stdout and stderr, and what does `2>&1` actually do?

Bash / Shell

 Spoken answer · ~83 s

Every process has file descriptors: 0 is standard input, 1 is standard output, and 2 is standard error. A redirection like `greater-than file` sends descriptor 1 to the file, and `2 greater-than` sends descriptor 2. The notation `2>&1` means 'make descriptor 2 point to wherever descriptor 1 currently points'. The ampersand tells the shell that the target is a descriptor, not a file named 1. Redirections are processed left to right, so order matters. Writing `greater-than file` then `2>&1` first points stdout at the file and then makes stderr follow it, so both go to the file. Writing `2>&1` first and then `greater-than file` sends stderr to the original stdout, usually the terminal, and only stdout to the file. That's the classic gotcha. In a pipeline, `2>&1` before the pipe lets stderr flow into the next command, since pipes carry only stdout by default. Bash also has the `ampersand-greater-than` shorthand to send both to a file, and `pipe-ampersand` to pipe both. To discard output, redirect to `/dev/null`. Separating the streams is useful: normal output to a file or pipeline, errors to a log. Remember that programs often buffer differently when output isn't a terminal, which affects ordering of interleaved output.

Likely follow-ups

- Why does the order of redirections matter?
- How do you pipe both stdout and stderr?

```
cmd > out.log 2> err.log      # separate files
```

```
cmd > all.log 2>&1             # both to one file (order matters!)
```

```
cmd &> all.log                 # bash shorthand for the above
```

```
cmd 2>&1 | tee log             # both into a pipe
```

2>&1 means "make file descriptor 2 (stderr) point to wherever fd 1 (stdout) currently points." Order is critical: > file 2>&1 works, but 2>&1 > file sends stderr to the *old* stdout (the terminal) because the redirect of stdout happens after stderr was already duplicated.

Common need: To pipe stderr through a filter, you must 2>&1 first — pipes only carry stdout by default.

 **Start here (official docs)**

- [Python documentation](#)

Q10 Write a script that processes a CSV and sums a numeric column with awk.

Bash / Shell

 Spoken answer · ~85 s

With awk, I treat the CSV as records split into fields, and keep a running total. For a simple file with no quoted commas, I set the field separator to a comma, skip the header row using NR greater than 1, add the chosen column, say dollar three, to a variable on each line, and print the sum in the END block. A typical command is awk with F comma, a condition for NR greater than one, an action adding dollar three to total, and an END print. I can extend this to group by another column using an associative array, such as totals indexed by dollar one, and print each key and value at the end, which is a quick pivot. Caveats: real CSV can have quoted fields containing commas or newlines, which a naive field split breaks on; for those, use a CSV-aware tool such as Python's csv module, csvkit, Miller, or gawk's FPAT. Validate that the field is numeric, skip blanks, and be careful with locale decimal separators. awk uses floating point, so for money, consider rounding with printf or use integer cents. Windows line endings can leave a carriage return on the last field and break arithmetic, so strip them first.

Likely follow-ups

- How would you group sums by another column?
- What breaks with quoted CSV fields?

```
# sum column 3 of a comma-separated file, skipping header
```

```
awk -F',' 'NR>1 { sum += $3 } END { printf "%.2f\n", sum }' data.csv
```

```
# group-by: sum col3 per category in col1
```

```
awk -F',' 'NR>1 { s[$1]+=$3 } END { for (k in s) print k, s[k] }' data.csv
```

-F', ' sets the field separator, NR>1 skips the header row, and \$3 is the third column. awk auto-initializes numeric vars to 0, so sum += \$3 just works. The group-by version uses an associative array keyed by the category column — awk's killer feature for ad-hoc log/CSV analysis.

Real CSVs are messy: Quoted fields containing commas break naive -F', '. For those, use a real parser (Python's csv module) rather than awk.

Official docs

- [GNU awk \(gawk\) manual](#)

Q11 How do you write an idempotent script that creates a config only if it doesn't exist?

Bash / Shell

Spoken answer · ~86 s

Idempotent means running the script repeatedly produces the same end state, with no errors or duplicated changes. For creating a config only if it doesn't exist, I test for existence first, with a file test, and create it only when absent. Better, I use an atomic approach: write the content to a temporary file in the same directory and then move it into place, so readers never see a half-written file. For more careful desired-state behaviour, generate the intended content, compare it with the existing file using cmp, and replace only when it differs, which also makes it self-correcting and lets you report whether anything changed. Other idempotent building blocks: mkdir -p, which succeeds if the directory exists; ln -sf for symlinks; install with a mode, owner, and group; sed or a grep check before appending a line, so a line isn't added twice; and checking whether a user, package, or service already exists before creating it. Watch for races between the check and the create, and use noclobber or set -o noclobber or a lock file with flock where concurrency is possible. Preserve file permissions and ownership. At scale, configuration management tools like Ansible or Terraform provide idempotency by design, so the script becomes a stopgap.

Likely follow-ups

- Why write to a temp file then move it?
- How do you avoid duplicate lines when appending to a file?

```
config=/etc/app/config.yml
```

```
if [[ ! -f "$config" ]]; then
```

```
  install -D -m 0640 -o app -g app /dev/stdin "$config" <<'EOF'
```

```
  key: value
```

```
EOF
```

```
echo "created $config"
```

```
else
```

```
echo "$config exists, skipping"
```

```
fi
```

Idempotency means running the script repeatedly yields the same end state without side effects. Guard creation with `[[ ! -f ]]`, and use `install` to atomically create the file with the right mode and ownership in one step (it also creates parent dirs with `-D`). A heredoc supplies the content.

Why it matters: Config-management and bootstrap scripts run over and over (cron, cloud-init, CI). Non-idempotent scripts duplicate entries, reset permissions, or clobber edits.

Official docs

- [RFC 9110: HTTP Semantics \(status codes, methods\)](#)
- [Idempotence in scripts \(Ansible concepts\)](#)

Q12 What's the difference between `[ ]`, `[[ ]]`, and `(( ))` in Bash?

Bash / Shell

Spoken answer · ~83 s

These are three different constructs. Single brackets are the test command, a POSIX-compatible builtin, essentially an alias for test. They're ordinary command arguments, so word splitting and globbing apply: you must quote variables, and comparisons use operators like equals for strings and `-eq`, `-lt` for integers; angle brackets need escaping or they're treated as redirections. Double brackets are a Bash, and Zsh and Ksh, keyword, parsed specially: no word splitting or pathname expansion on unquoted variables, so an empty variable doesn't break the expression; it supports pattern matching with a glob on the right of the double equals, regex matching with the equals-tilde operator, and the `and` and `or` operators inside, plus the lexicographic comparisons with angle brackets. It isn't portable to plain sh. Double parentheses are arithmetic evaluation: inside them you use C-style syntax, such as a variable greater than ten, increments, and assignments, without needing dollar signs; the exit status is zero if the result is non-zero, and the dollar-prefixed form returns the value for use in expansions. Rule of thumb: use double brackets for conditions in Bash scripts, double parentheses for numeric comparisons and math, and single brackets only when you need portability to sh.

Likely follow-ups

- Why is `[[ ]]` safer with unquoted variables?
- How do you do regex matching in Bash?

```
[ "$a" = "$b" ]      # POSIX test, needs quoting, string =
```

```
[[ $a == "$b" ]]     # bash test: safer, supports == pattern, && ||
```

```
[[ $s =~ ^v[0-9]+$ ]] # regex match
```

```
(( x > 5 && y < 10 )) # arithmetic context, C-style
```

[] is the old POSIX test command — portable but fragile (word-splitting, needs careful quoting). [[]] is the bash keyword: no word-splitting inside, supports == globbing, =~ regex, and &&|||. (()) is arithmetic evaluation with C-like operators and no \$ needed on variables.

Default to [[]] for tests and (()) for numbers in bash scripts. Reserve [] for strict POSIX sh compatibility.

Official docs

- [GNU Bash reference manual](#)

Q13 How do you tail logs from multiple files and act on a matching pattern in real time?

Bash / Shell

Spoken answer · ~88 s

To follow multiple logs and react to patterns, I use tail -F, capital F, which follows by name and survives log rotation, and pass it multiple files; it prints headers showing which file each line came from. I pipe the output into a loop or into awk or grep. With grep, add line buffering, since otherwise output is buffered when writing to a pipe and the reaction arrives late; grep --line-buffered works. A robust pattern is tail -F files, piped to a while loop with IFS= read -r line, and inside a case statement or a regex test that matches, say, ERROR or a particular status code, then triggers an action such as sending an alert, incrementing a counter, or calling a webhook. Considerations: rate-limit actions so a flood of matches doesn't spam; deduplicate or aggregate over a time window; handle the subshell scope issue where variables set inside a piped while loop disappear, using process substitution instead; and make the script supervised by systemd so it restarts. For production, though, a purpose-built pipeline is better: a log shipper such as Fluent Bit or Vector with rules, or alert rules in the logging stack, since ad hoc tails don't handle backpressure, restarts, or multiple hosts. Use journalctl -f for systemd journals.

Likely follow-ups

- Why use tail -F instead of tail -f?
- Why does grep need --line-buffered in a pipeline?

```
tail -F /var/log/app/*.log | while read -r line; do
```

```
if [[ "$line" == *"FATAL"* ]]; then
```

```
echo "$line" | mail -s "FATAL on $(hostname)" oncall@co.com
```

```
fi
```

```
done
```

`tail -F` (capital F) follows files by name and survives log rotation (reopens when the file is recreated), unlike `-f` which follows the inode and goes silent after rotation. Piping into a `while read` loop lets you pattern-match each line and trigger an action — a poor-man's alerting hook.

Don't build real monitoring this way: It's fine for quick debugging, but for production use a log shipper (Promtail/Fluent Bit) + alerting. A shell loop has no buffering, dedup, or backpressure handling.

 **Start here (official docs)**

- [Python documentation](#)

Q14 **Write a script to check disk usage and alert if any mount exceeds 85%.**

Bash / Shell

 **Spoken answer** · ~87 s

I'd use `df` to read usage per filesystem, parse it, and compare against the 85 percent threshold. Use `df` with the `-P` flag for stable POSIX output, so long device names don't wrap across lines, and exclude pseudo filesystems such as `tmpfs` and `devtmpfs` with `-x` options. Read each line in a loop, or use `awk` to strip the percent sign from the usage column and check whether the number is greater than or equal to 85, printing the mount point and percentage. If any exceed the threshold, send an alert, via email, a webhook to Slack or PagerDuty, or write to `syslog` with `logger`, and return a non-zero exit code so a scheduler or monitoring agent can detect it. Don't forget inodes: a filesystem can be full of tiny files while `df` shows free blocks, so check `df -i` as well. Make the threshold configurable via an argument or an environment variable, avoid re-alerting every minute by tracking state or using a cooldown, and run it from `cron` or a `systemd` timer. In practice, I'd prefer a monitoring system with node exporter and alert rules, including predictive alerts such as disk full in four hours using linear prediction, which are far more useful than a static threshold.

Likely follow-ups

- Why check inodes as well as blocks?
- Why is a predictive alert better than a static threshold?

```
threshold=85
```

```
df -P | awk -v t="$threshold" 'NR>1 {
```

```
  gsub(/%/, "", $5)
```

```
  if ($5+0 >= t) printf "WARN %s at %s%%\n", $6, $5
```

```
}'
```

`df -P` forces POSIX output (one line per filesystem, stable columns). `awk` strips the % from the use-percent column with `gsub`, coerces it to a number (`$5+0`), and prints the mount point (`$6`) if over threshold. Passing the threshold via `-v` keeps it parameterized.

Inode exhaustion is a sneaky cousin: a disk can be 30% full but out of inodes (many tiny files). Check `df -iP` too — a frequent cause of "no space left" with free space showing.

 **Start here (official docs)**

- [Python documentation](#)

Q15 How do you run commands in parallel in Bash and wait for all to finish?

Bash / Shell

 **Spoken answer** · ~90 s

The simplest approach is to start each job in the background with an ampersand, collect their process IDs, and then call `wait` on them. Starting jobs in a loop, appending dollar bang, the PID of the last background process, to an array, then looping over the array calling `wait` on each PID, lets you capture each exit status individually, which a bare `wait` with no arguments does not do. Check each status and track failures so the script exits non-zero if any job failed. Uncontrolled parallelism can overwhelm a machine or a remote service, so limit concurrency: use `xargs` with `-P` to run `N` processes at once, such as `xargs -P 8 -n 1`, or GNU `parallel`, which also handles output grouping and retries. In newer Bash, `wait -n` returns when any one job finishes, which lets you implement a simple worker pool by launching a new job whenever one completes. Other considerations: output from parallel jobs interleaves, so write each job's output to its own file or prefix lines; use `set -e` carefully, because failures in background jobs don't trigger it; trap signals to kill children on interrupt; and make sure the jobs don't share mutable state or temp files without locking. For heavy orchestration, move to a job runner or Python's concurrent futures.

Likely follow-ups

- How do you capture each background job's exit status?
- How do you limit parallelism to `N` jobs?

```
# background jobs + wait
```

```
for host in web1 web2 web3; do
    ssh "$host" 'systemctl restart app' &
done
wait # block until all background jobs complete

# bounded parallelism with xargs

printf '%s\n' web{1..9} | xargs -P4 -I{} ssh {} uptime
```

Appending & backgrounds each job; wait with no args blocks until they all finish. For controlled concurrency, xargs -P4 runs at most 4 at once — important so you don't fork 500 SSH sessions and exhaust resources. GNU parallel is the richer tool if available.

Capture failures: Plain wait doesn't tell you which job failed. Collect PIDs and wait \$pid individually, or write per-host exit codes to files, if you need to know what broke.



Official docs

- [GNU Bash reference manual](#)

Q16 What does 'local' do in a Bash function and why does it matter?

Bash / Shell

Spoken answer · ~84 s

By default, variables in a Bash function are global: assigning to a variable inside a function changes it for the whole script. The local keyword declares a variable with scope limited to the function and any functions it calls, using dynamic scoping, so it shadows a global of the same name for the duration of the call and then restores it. It matters for correctness and maintainability. Without it, a function using a loop counter called i can silently overwrite a caller's i, producing baffling bugs, especially in recursive functions or in sourced libraries where names collide. Using local keeps functions self-contained, so they can be reused and tested. Practical guidance: declare local at the top of each function for every variable it uses, including loop variables; use local with assignment on a separate line from command substitution, because combining them, as in local x=\$(cmd), masks the exit status of cmd, which hides failures under set -e. Functions return an exit status via return, and output via standard output or by setting a named global or using a nameref; local doesn't help return values. Remember local only works inside functions, and is a Bash feature rather than strict POSIX, though widely supported.

Likely follow-ups

- Why does local x=\$(cmd) hide failures?
- How is Bash scoping different from lexical scoping?

```
counter=0
```

```
bump() { local counter=99; echo "inside: $counter"; }
```

```
bump          # inside: 99
```

```
echo "$counter" # 0 (global untouched)
```

By default all Bash variables are global — assigning inside a function mutates the caller's scope, a notorious source of action-at-a-distance bugs. `local` scopes a variable to the function and its callees. Always declare function-internal variables `local` to avoid clobbering globals or other functions' state.

Subtle trap: `local var=$(cmd)` masks `cmd`'s exit code (local always returns 0). Split into `local var;`
`var=$(cmd) || return 1` if you need to check it.

Official docs

- [GNU Bash reference manual](#)

Q17 How do you safely use variables in a command that might be empty or undefined?

Bash / Shell

Spoken answer · ~85 s

Unset or empty variables in commands are dangerous: `rm -rf` of a variable followed by a slash becomes `rm -rf slash` if the variable is empty. Defences. Turn on `set -u` so referencing an unset variable aborts. Use parameter expansion operators for explicit handling: the colon-question form, dollar brace `VAR` colon question message, fails with a message if the variable is unset or empty, which makes it a required-variable check; the colon-dash form supplies a default value; the colon-plus form expands only if set. Always double-quote expansions, so an empty value remains an empty argument instead of disappearing, and so spaces don't split it. Validate before use: test for non-empty with the `-n` test, check that a path exists and is under an expected prefix, and refuse dangerous values like slash or the home directory. For destructive commands, add guards such as the double dash to end option parsing, and use `rm` without `-r` where possible, or move to a trash location first. Use arrays for commands built from parts instead of string concatenation and `eval`. Add a dry-run mode, and use `shellcheck` in CI, which flags unquoted expansions and many of these hazards. Treat anything from the environment or user input as untrusted.

Likely follow-ups

- What does `${VAR:?message}` do?
- Why is `eval` dangerous when building commands?

```
${VAR:-default} # use default if VAR unset OR empty
```

```
{ ${VAR:=default}    # assign default to VAR if unset/empty
```

```
{ ${VAR:?must be set} # exit with error message if unset/empty
```

```
{ ${VAR:+value}      # use 'value' only if VAR IS set
```

These parameter-expansion forms handle missing config cleanly. `:` supplies a fallback, `:=` sets it, `:?` fails fast with a message (great for required env vars), and `:+` conditionally adds something only when set (e.g. appending a flag). Combined with `set -u`, `:?` gives you self-documenting required-variable checks.

Pattern: `:"${DATABASE_URL:?DATABASE_URL is required}"` at the top of a script makes missing config fail immediately with a clear message instead of mysteriously later.

 **Start here (official docs)**

- [Python documentation](#)

Q18 Write a healthcheck script that returns proper exit codes for a load balancer.

Bash / Shell

 **Spoken answer** · ~81 s

A load balancer health check calls an endpoint or script and treats the result as pass or fail, so the script needs clear, fast, accurate exit codes. For an HTTP check, the script or endpoint returns 200 when healthy and 503 when not; for command-based checks, exit zero means healthy and non-zero unhealthy. Design: keep it fast, with a strict timeout, because slow checks cause flapping; verify what actually matters for serving traffic, such as the process is running, the port is listening, a local request to the application returns a successful response, and that critical local dependencies like disk space and a required socket are available. Be careful about deep checks of remote dependencies: if the database is slow and every instance fails, the load balancer removes everything, turning a partial failure into total outage; prefer degraded reporting, with separate liveness and readiness semantics. Return distinct codes where the platform allows, such as 1 for unhealthy and 2 for warning, and include a short reason to stderr or the response body, for debugging. Support a drain mode: a maintenance flag file that makes the check fail so the instance is removed gracefully before deployment.

Likely follow-ups

- Why avoid checking remote dependencies in a health check?
- How does a drain flag help deployments?

```
#!/usr/bin/env bash

set -euo pipefail

if curl -fsS --max-time 2 http://localhost:8080/healthz >/dev/null; then

    exit 0    # healthy

else

    exit 1    # unhealthy → LB removes from rotation

fi
```

Health checks communicate via exit codes: 0 = healthy, non-zero = unhealthy. `curl -f` makes HTTP ≥ 400 return non-zero, `-sS` stays quiet but still shows real errors, and `--max-time` bounds the check so a hung backend doesn't hang the probe. The LB or orchestrator interprets the exit code to route traffic.

Check the right thing: A good health endpoint verifies the app can do real work (DB reachable, deps OK) — but keep liveness cheap and local so a downstream blip doesn't cause mass restarts.



Official docs

- [Kubernetes: Determine the reason for Pod failure](#)
- [Cloudflare Learning: Load balancing](#)

 Spoken answer · ~88 s

I debug Bash scripts by making execution visible and catching mistakes early. Run with `bash -x`, or `set -x` inside the script, to print each command after expansion, with a prefix customised via `PS4` to include the script name, line number, and function, such as setting `PS4` to include `BASH_SOURCE` and `LINENO`. Use `set -x` only around the suspicious section, with `set +x` afterwards, so output isn't overwhelming. Combine with `set -euo pipefail` to expose silent failures, and add a trap on `ERR` that prints the failing line and command. Run `bash -n` to check syntax without executing, and run `shellcheck`, which catches quoting mistakes, unused variables, and portability problems, ideally in an editor and in CI. Add logging functions with levels and timestamps, and print variable values with `declare -p`, which shows quoting, arrays, and hidden characters. Reproduce in a minimal case, isolating the failing command, and check the environment: differences in `PATH`, locale, current directory, or the shell used when run from cron, which has a minimal environment. Check line endings, since carriage returns from Windows files cause strange errors. Verify the shebang and whether the script runs under `sh` instead of `bash`. For interactive stepping, there's the `DEBUG` trap or the `bashdb` debugger, though trace output is usually enough.

Likely follow-ups

- How do you make `set -x` output show line numbers?
- Why do scripts behave differently under cron?

```
bash -x script.sh          # trace every expanded command
```

```
set -x ... set +x          # trace just a section
```

```
PS4='+ ${BASH_SOURCE}:${LINENO}:${FUNCNAME[0]:-}: ' # richer trace
```

```
bash -n script.sh          # syntax check without running
```

```
shellcheck script.sh       # static analysis – catches most bugs
```

`set -x` prints each command after expansion so you see exactly what ran with what values — the single most useful Bash debugging tool. Customizing `PS4` adds file/line/function context. `bash -n` checks syntax without executing. And `shellcheck` is essential: it statically flags quoting bugs, unset vars, and footguns before they bite.

Run shellcheck in CI: Treat its warnings as errors. It catches the vast majority of the quoting and subtlety bugs that make Bash scripts fragile.

 Official docs

- [GNU Bash reference manual](#)

 Spoken answer · ~84 s

Bash is excellent glue for short sequences of commands, but I switch when the script outgrows its strengths. Signals: more than roughly a hundred lines, or complex logic with nested conditionals; the need for real data structures beyond flat arrays, such as nested maps or JSON manipulation; heavy string processing or parsing of structured formats like JSON, YAML, or CSV; error handling needs beyond exit codes, such as retries, exceptions, and cleanup across steps; making HTTP calls with proper authentication, pagination, and timeouts; concurrency beyond simple background jobs; unit testing requirements; cross-platform portability; and when the script will be maintained by a team who need readable, testable code. Python is the usual next step for scripting and automation, with a rich standard library, libraries for cloud APIs and Kubernetes, and good testing tools. Go is better when you need a single static binary with no runtime dependencies, strong concurrency, performance, and distribution to many machines, which is common for CLIs and operators. A pragmatic path is to keep Bash for thin wrappers, glue, and entrypoints, and move the logic into Python or Go behind a clear interface. Also consider whether a configuration management or infrastructure tool should own the task instead.

Likely follow-ups

- What signals tell you a script has outgrown Bash?
- When would you pick Go over Python?

Stay in Bash: glue, pipelines, <100 lines, calling other CLIs

Switch when: real data structures, JSON/HTTP parsing, error handling,

arithmetic/floats, testing, or it exceeds ~100–200 lines

Bash excels at orchestrating other programs and text pipelines. But once you need structured data (parsing JSON, nested maps), robust error handling, floating-point math, unit tests, or the script grows past a couple hundred lines, Bash becomes a liability — fragile quoting, no real types, painful debugging. That's the signal to move to Python (or Go for a distributed binary).

Rule of thumb: "If I'm reaching for `jq` three times and an array of structs, this should be Python." Knowing when to switch is itself a senior signal.

 Official docs

- [GNU Bash reference manual](#)
- [Python Documentation](#)

Q21 **What's the difference between a list, tuple, set, and dict — and when do you use each?**

Python

 Spoken answer · ~87 s

These are Python's core built-in collections, each with a different trade-off. A list is an ordered, mutable sequence allowing duplicates, with fast append and index access; use it for ordered collections that change, like a queue of tasks to process. A tuple is an ordered, immutable sequence; use it for fixed records, such as a coordinate pair or a function returning multiple values, and because it's hashable when its elements are, it can serve as a dictionary key; immutability also signals intent and is slightly lighter. A set is an unordered collection of unique, hashable items with average constant-time membership tests and set algebra, union, intersection, and difference; use it for deduplication and fast lookups, like checking whether a host is in an allowlist or finding which servers are in one inventory but not another. A dict is a mapping from keys to values, insertion-ordered since Python 3.7, with average constant-time lookup; use it for structured records, indexes, counters, and caching. Performance guidance: membership checks on a list are linear, so convert to a set for repeated lookups. Related tools: `collections.defaultdict`, `Counter`, `deque` for fast pops from both ends, and dataclasses or named tuples for readable records. Choose by asking whether order, uniqueness, mutability, or keyed lookup matters.

Likely follow-ups

- Why is membership testing faster in a set?
- When would you use a tuple instead of a list?

`list` `[..]` ordered, mutable, dupes ok → sequences

`tuple` `(..)` ordered, immutable, hashable → fixed records, dict keys

`set` `{..}` unordered, unique, $O(1)$ membership → dedup, fast "in"

`dict` `{k:v}` key→value, $O(1)$ lookup, ordered(3.7+) → mappings

Choose by access pattern. Need ordered mutable data → list. Need an immutable, hashable group (or a dict key) → tuple. Need uniqueness or fast membership tests → set (checking `x in big_set` is $O(1)$ vs $O(n)$ for a list). Need key-based lookup → dict. Picking the wrong one turns $O(1)$ operations into $O(n)$ and is a top cause of "why is my script slow on big input."

SRE example: Diffing two large lists of hostnames? Convert to sets — `set_a - set_b` is near-instant; nested list loops are $O(n^2)$.

 **Start here (official docs)**

- [Python documentation](#)

 Spoken answer · ~86 s

A list comprehension builds a list in a single expression: an output expression, a loop, and an optional condition, such as squares of numbers that are even. It's concise, readable for simple transformations and filters, and usually faster than an equivalent loop with `append`, because the iteration is optimised in the interpreter and avoids repeated method lookups. Dict and set comprehensions follow the same pattern, and generator expressions, with parentheses, produce items lazily. They're better when the logic is a straightforward map or filter that fits comfortably on one or two lines. They're worse, or at least a poor choice, when: the logic is complex, with nested loops and multiple conditions, which become hard to read, where a regular loop with clear names is better; you need side effects like logging or writing, since a comprehension should build a value, not act; you need to handle exceptions per item, which a comprehension can't do; or the result is huge and you only need to iterate once, where a generator expression saves memory by not materialising the entire list. Another gotcha: in Python 3, the comprehension variable doesn't leak into the enclosing scope. Rule of thumb: if you have to think hard to read it, write a loop.

Likely follow-ups

- When is a generator expression preferable?
- Why shouldn't a comprehension be used for side effects?

```
squares = [x*x for x in nums if x > 0]
```

```
by_host = {h: ping(h) for h in hosts}      # dict comp
```

```
uniq = {line.strip() for line in f}        # set comp
```

```
# generator (lazy, memory-friendly):
```

```
total = sum(x*x for x in nums)
```

Comprehensions are concise and fast for building a collection by transforming/filtering an iterable. Prefer them for simple map/filter logic. Use a generator expression (parentheses) when you only iterate once or the data is large — it's lazy and avoids building the whole list in memory. Drop back to a plain loop when there's side effects, multiple statements, or the logic hurts readability.

Don't over-nest: A triple-nested comprehension with two conditions is write-only code. If you can't read it aloud, use a loop.

 Start here (official docs)

- [Python documentation](#)

 Spoken answer · ~88 s

The memory-efficient way is to iterate over the file object directly: opening the file with a context manager and writing a for loop over the file handle. Python reads the file in buffered chunks and yields one line at a time, so memory use stays small regardless of file size. By contrast, `read()` loads everything into memory, and `readlines()` builds a list of all lines; both can exhaust memory on large logs. Good practice: open with a specified encoding and an errors policy such as `replace` for dirty logs; strip the newline; handle each line streaming, perhaps by passing through a generator pipeline that filters, parses, and aggregates, so the stages compose without intermediate lists. For binary or fixed-size processing, read in chunks with a loop reading `N` bytes at a time. For compressed files, use the `gzip` module's text mode, which also iterates line by line. For following a growing file, tail-like behaviour requires reading until the end, sleeping, and handling rotation, or using a library. For truly large data and parallelism, split the file into byte ranges and process in multiple processes, or use tools like `pandas` with `chunksize`. Keep state small by aggregating counts into dictionaries, and be aware of extremely long lines, which can still use memory.

Likely follow-ups

- Why is `readlines()` risky on large files?
- How do generator pipelines keep memory flat?

```
# GOOD: iterates lazily, constant memory
```

```
with open("huge.log") as f:
```

```
    for line in f:
```

```
        process(line)
```

```
# BAD: loads entire file into RAM
```

```
lines = open("huge.log").read().splitlines()
```

A file object is itself a lazy iterator — `for line in f` reads one line at a time with buffering, so memory stays flat regardless of file size. Using `.read()` or `.readlines()` pulls the whole file into RAM and will OOM on multi-GB logs. The `with` block guarantees the file handle closes even on exception.

For huge files, combine with a generator pipeline: a function that yields parsed records keeps the whole flow streaming and composable.

 Official docs

- [Git LFS](#)

Q24 **Write a function that parses a log line into a structured dict. Handle malformed lines.**

Python

🔊 Spoken answer · ~90 s

I write a function that takes a line string and returns a dictionary, or None, or raises a specific error when the line doesn't match. For a common log format, I compile a regular expression once at module level, with named groups for fields like the client address, timestamp, method, path, status, and size. In the function, I call match on the line; if there's no match, I treat it as malformed. Then I convert types deliberately: status and size to integers, handling a dash for missing size, and the timestamp parsed with datetime using the expected format and time zone. Handling malformed lines: don't let one bad line crash the pipeline. Either return None and let the caller skip and count them, or raise a custom ParseError that the caller catches, logging a truncated sample of the line and incrementing a counter metric, so you notice if the malformed rate spikes. Also handle encoding problems, truncated lines, and unexpected extra fields. Keep the function pure and easy to test: unit tests with valid lines, missing fields, garbage, empty strings, and extremely long lines. For performance on large logs, precompiled regex is fast enough, and for structured JSON logs, json.loads in a try block is simpler. Consider returning a dataclass or typed dict for clarity.

Likely follow-ups

- Why compile the regex once at module level?
- Should a parser skip bad lines or fail the whole run?

```
import re

PATTERN = re.compile(

    r'(?P<ip>\S+) \S+ \S+ \[(?P<ts>[^\]]+)\] '

    r'""(?P<method>\S+) (?P<path>\S+)[^"]*" (?P<status>\d+)'

def parse(line):

    m = PATTERN.match(line)

    if not m:

        return None          # caller skips/counts bad lines

    d = m.groupdict()

    d["status"] = int(d["status"])

    return d
```

Use a compiled regex with named groups for readable, fast parsing (compile once at module level, not per call). Return None (or raise a custom exception) on non-matching lines so the caller decides how to handle bad input — real logs always contain garbage lines, partial writes, and format drift. Convert types (status → int) at the boundary.

Robustness: Count and log the malformed-line rate. A sudden spike in unparseable lines is itself a signal (deploy changed the log format, or something is corrupting output).

Official docs

- [Google SRE Book: Distributed tracing & logs \(Monitoring chapter\)](#)

Q25 Explain *args and **kwargs with a practical example.

Python

Spoken answer · ~85 s

The star-args and double-star-kwargs let a function accept a variable number of arguments. Within the definition, star args collects extra positional arguments into a tuple, and double-star kwargs collects extra keyword arguments into a dictionary. For calling, the same operators unpack: a star before a list expands it into positional arguments, and double-star before a dict expands into keyword arguments. A practical example is a wrapper or decorator that forwards whatever arguments it receives to the wrapped function without needing to know its signature: the inner function takes star args and double-star kwargs, does something like timing or logging, and calls the original function with the same arguments. Another is a thin client function that builds a request: it accepts a URL and then passes remaining keyword options like timeout, headers, or params through to requests.get. They are also useful for configuration merging, passing options down through layers, and subclass constructors that call the parent with super, forwarding unknown arguments. Cautions: they hide the real signature, hurting readability and tool support, so prefer explicit parameters when the set of arguments is known; document accepted keys; and validate unexpected ones, because typos silently pass through. Keyword-only parameters after a bare star improve clarity and safety.

Likely follow-ups

- How do wrappers use *args and **kwargs?
- What is the downside of overusing them?

```
def run(cmd, *args, timeout=30, **opts):  
  
    full = [cmd, *args]  
  
    print(full, timeout, opts)  
  
run("kubectl", "get", "pods", timeout=10, namespace="prod")  
  
# ['kubectl', 'get', 'pods'] 10 {'namespace': 'prod'}
```

`*args` collects extra positional arguments into a tuple; `**kwargs` collects extra keyword arguments into a dict. They let you write flexible wrappers (decorators, CLI runners, retry helpers) that forward arbitrary arguments to an inner function via `inner(*args, **kwargs)`. The same `*/**` also *unpack* a list/dict back into a call.

Forwarding pattern: A retry/timing decorator typically signs as `def wrapper(*args, **kwargs): return fn(*args, **kwargs)` so it works for any wrapped function.

Official docs

- [AWS Builders' Library: Timeouts, retries and backoff with jitter](#)

Q26 What is a context manager and how do you write one with contextlib?

Python

Spoken answer · ~89 s

A context manager defines setup and teardown around a block of code, used with the `with` statement, and guarantees cleanup even if an exception occurs. Classic examples are opening files, acquiring locks, database transactions, temporary directories, and timing blocks. The protocol is two methods on a class: `enter`, which runs on entry and returns the resource, and `exit`, which runs on exit and receives the exception details, and can suppress the exception by returning `true`. The `contextlib` module makes it easier with the `contextmanager` decorator: you write a generator function that does setup, yields the resource, and places teardown in a `finally` block after the `yield`; any exception in the `with` body is raised at the `yield` point, so a `try-finally` ensures cleanup runs. For example, a context manager that changes the working directory temporarily, then restores it, or one that creates a temp directory and removes it. `contextlib` also provides `closing`, `suppress`, `redirect_stdout`, `ExitStack` for managing a dynamic number of contexts, and in newer versions asynchronous equivalents for `async with`. Benefits: resource safety, less boilerplate than `try-finally`, and clear scoping. Pitfalls: forgetting `finally` in the generator version so cleanup is skipped on error, and swallowing exceptions accidentally in `exit`. They compose well, and you can place multiple contexts in one `with` statement.

Likely follow-ups

- Why use `try/finally` around the `yield`?
- What is `ExitStack` for?

```
from contextlib import contextmanager
```

```
import time
```

```
@contextmanager
```

```
def timed(label):
```

```
    start = time.monotonic()
```

```
try:
```

```
    yield
```

```
finally:
```

```
    print(f"{label}: {time.monotonic()-start:.3f}s")
```

```
with timed("query"):
```

```
    run_query()
```

A context manager guarantees setup/teardown around a block via `with` — the canonical use is releasing resources (files, locks, connections) even on exception. The `@contextmanager` decorator turns a generator into one: code before `yield` is setup, code in `finally` after `yield` always runs as teardown. Use `time.monotonic()` for durations (it never goes backward, unlike `wall-clock`).

Why finally matters: Putting teardown in `finally` ensures cleanup runs even if the wrapped block raises — the whole point over manual setup/teardown.

 **Start here (official docs)**

- [Python documentation](#)

Spoken answer · ~89 s

Proper exception handling means catching only what you can meaningfully handle, being specific, preserving information, and cleaning up reliably. Good practice: catch specific exceptions, such as `KeyError` or `requests.Timeout`, not a bare `except` or a broad `Exception`, which swallows programmer errors and even keyboard interrupts in the bare form. Keep the `try` block small, wrapping only the statement that can fail. Use `else` for code that runs only when no exception occurred, and `finally`, or a context manager, for cleanup. Re-raise when you can't handle something, using a bare `raise` to preserve the traceback, or `raise from` to chain a more meaningful exception while keeping the cause. Define custom exceptions for domain errors. Log the exception with `traceback`, using `logger.exception`, at the right level. For retries, catch transient errors only. Anti-patterns: bare `except`; `except Exception` followed by `pass`, which silently hides failures; using exceptions for normal control flow in hot paths; returning error codes or `None` when an exception is clearer; catching and re-raising without adding value; losing the original traceback by raising a new exception without chaining; printing instead of logging; and overly large `try` blocks that mask which line failed. At top level, a single handler should log and exit non-zero. Also follow the principle that errors should never pass silently.

Likely follow-ups

- What does `raise from` do?
- Why is a bare `except` dangerous?

```
# GOOD: specific, logged, re-raised or handled
```

```
try:
```

```
    resp = client.get(url, timeout=5)
```

```
except requests.Timeout:
```

```
    log.warning("timeout on %s", url); raise
```

```
except requests.HTTPError as e:
```

```
    log.error("http %s", e.response.status_code); raise
```

```
# BAD:
```

```
try: ...
```

```
except: pass          # swallows everything, incl KeyboardInterrupt
```

Catch the most specific exceptions you can handle, do something meaningful (log with context, retry, clean up), and re-raise if you can't actually recover. The cardinal sins: bare except: (catches SystemExit/KeyboardInterrupt too), except Exception: pass (silently hides bugs), and catching broadly just to log-and-continue. Silent failures are how data corruption and "why didn't the alert fire" incidents happen.

Never swallow silently: except: pass is the most dangerous two lines in production Python — it converts crashes into invisible wrong behavior.

Official docs

- [Microsoft: Anti-patterns \(performance antipatterns\)](#).

Q28 What's the difference between is and == ? Show a gotcha.

Python

Spoken answer · ~88 s

The equality operator, double equals, compares values by calling the object's equality method; the is operator compares identity, whether two names refer to the exact same object in memory. Use is for singletons, such as None, True, and False; use equals for value comparison. The gotcha: small integers and some strings are cached or interned by the interpreter, so is may appear to work with them and then fail unpredictably. For example, two variables each assigned 256 may satisfy is, while two assigned 1000 computed at runtime may not, and the result can differ between the REPL and a script, or between Python versions. Likewise, two equal lists are equal but not identical: comparing a list to a copy with equals gives true, with is gives false. Using is to compare strings or numbers is a bug that sometimes passes tests. Another subtlety: a class can define its own equality, so equal objects can be distinct, and a float NaN is not equal to itself, though it is identical to itself. Also write x is None, not x equals None, because a class could override equality in surprising ways. Newer Python versions emit a syntax warning for is with a literal. Rule: is for None and sentinels, equals for everything else.

Likely follow-ups

- Why does is sometimes work for small integers?
- Why write x is None instead of x == None?

```
a == b    # value equality (calls __eq__)
```

```
a is b    # identity: same object in memory
```

```
x = [1,2]; y = [1,2]
```

```
x == y    # True   (same contents)
```

```
x is y    # False  (different objects)
```

```
if val is None: ...    # correct None check
```

`==` compares values; `is` compares object identity. Use `is` only for singletons — `None`, `True`, `False` (if `x is None`). Using `is` for value comparison appears to work for small ints and short strings due to interning, then breaks unpredictably for larger values. That intermittent breakage is the classic gotcha.

Always `if x is None`, never `if x == None` — and never `is` for numbers/strings you actually want to compare by value.

 **Start here (official docs)**

- [Python documentation](#)

Q29 Explain mutable default arguments — the classic Python footgun.

Python

 Spoken answer · ~83 s

Default argument values are evaluated once, when the function is defined, not each time it's called. So if the default is a mutable object like a list or dict, all calls that omit the argument share the same object, and modifications persist across calls. For example, a function with a parameter `items` defaulting to an empty list that appends to it will accumulate values from previous calls, producing baffling behaviour, and in a long-running service it becomes a memory leak and a data leak between requests. The fix is to use `None` as the default and create a new object inside the function: if the argument is `None`, set it to a fresh list. Using a sentinel object is useful when `None` is itself a valid value. Immutable defaults such as numbers, strings, tuples, and `None` are safe. In dataclasses, mutable defaults are disallowed directly and require field with `default_factory`, which creates a new instance per object. Occasionally the behaviour is exploited deliberately, as a cache or memo dictionary, but that should be explicit and commented, or better replaced with `functools.lru_cache`. Linters such as `pylint` and `flake8` bugbear flag this pattern, so enabling them in CI catches it early.

Likely follow-ups

- What is the safe pattern for a default list?
- How do dataclasses handle mutable defaults?

```
# BUG: default list is created ONCE, shared across calls
```

```
def add(item, bucket=[]):
```

```
    bucket.append(item); return bucket
```

```
add(1) # [1]
```

```
add(2) # [1, 2] ← surprise! same list
```

```
# FIX:
```

```
def add(item, bucket=None):
```

```
    bucket = [] if bucket is None else bucket
```

```
    bucket.append(item); return bucket
```

Default argument values are evaluated *once* at function-definition time, not per call. A mutable default (list, dict, set) is therefore shared across every call that doesn't override it, accumulating state — a subtle, data-dependent bug. The fix is the `None` sentinel pattern: default to `None` and create a fresh object inside the function.

Interview staple: This is asked constantly because it trips up even experienced devs and silently corrupts state in long-running services.

Official docs

- [Python Documentation](#)

Spoken answer · ~84 s

For reliable HTTP calls, set timeouts and retries explicitly, because the requests library has no default timeout, so a call can hang forever. The recommended approach is a Session, which reuses connections, with an HTTPAdapter mounted for the https and http prefixes, configured with a urllib3 Retry object: a total number of retries, a backoff factor for exponential delays, a status_forcelist of transient codes like 429, 500, 502, 503, and 504, and allowed methods limited to idempotent ones such as GET, since retrying POST can duplicate side effects unless it has an idempotency key. Honour the Retry-After header. Pass a timeout on each request, as a tuple of connect and read seconds, such as three and ten. After the response, call raise_for_status so HTTP errors become exceptions, and catch requests exceptions: Timeout, ConnectionError, and HTTPError, handling each sensibly. Add jitter to avoid synchronised retries, an overall deadline for the operation, and a User-Agent header. Reuse the session across calls, and close it with a context manager. Log failures with the URL and status without leaking secrets. Verify TLS, never disabling verification. For more advanced needs, libraries such as tenacity add flexible retry policies, and httpx offers async support with similar controls.

Likely follow-ups

- Why is there no default timeout a concern?
- Why restrict retries to idempotent methods?

```
import requests

from requests.adapters import HTTPAdapter, Retry

s = requests.Session()

retries = Retry(total=4, backoff_factor=0.5,
                status_forcelist=[429, 500, 502, 503, 504],
                allowed_methods=["GET", "PUT", "DELETE"])

s.mount("https://", HTTPAdapter(max_retries=retries))

r = s.get(url, timeout=(3, 10)) # (connect, read)

r.raise_for_status()
```

Always set an explicit timeout — without it, requests can hang forever and freeze your service. The Retry adapter gives automatic exponential backoff (backoff_factor) only on retryable statuses, and only on idempotent methods by default (don't blindly retry POSTs). raise_for_status() turns 4xx/5xx into exceptions so failures aren't silently ignored.

The #1 production bug: Forgetting timeout=. A single hung upstream call with no timeout cascades into your service's thread pool and takes you down.

Official docs

- [AWS Builders' Library: Timeouts, retries and backoff with jitter](#)
- [Microsoft: Cloud design pattern – Retry](#)

Q31 What's the difference between deepcopy and a shallow copy?

Python

Spoken answer · ~88 s

Both create copies of objects, but they differ for nested structures. A shallow copy, made with `copy.copy`, the `copy` method of lists and dicts, slicing, or the constructor, creates a new outer container but fills it with references to the same inner objects. So if you shallow copy a list of lists and then modify an inner list through the copy, the original sees the change, because both outer lists point to the same inner list. A deep copy, `copy.deepcopy`, recursively copies all nested objects, so the new structure is fully independent. It also handles cyclic references via a memo dictionary. Use shallow copies when the elements are immutable, like numbers and strings, or when sharing inner objects is intended, as it's cheaper. Use deep copies when you need an independent snapshot of nested mutable data, such as a configuration you will modify per environment without altering the template. Costs and caveats: `deepcopy` is slower and uses more memory, can fail on objects that can't be copied, such as open files, locks, or sockets, and may copy more than you intended. You can customise behaviour with `copy` and `deepcopy` methods on classes. A common bug is assigning with equals, which copies nothing and just creates a second name for the same object.

Likely follow-ups

- Why doesn't assignment create a copy?
- What kinds of objects can't be deep-copied?

```
import copy
```

```
a = {"tags": ["x"]}
```

```
b = a.copy()           # shallow: new dict, SAME inner list
```

```
b["tags"].append("y")
```

```
a["tags"]              # ['x', 'y'] ← mutated!
```

```
c = copy.deepcopy(a)   # fully independent nested copy
```

A shallow copy duplicates the top-level container but the nested objects are shared references — mutating a nested list/dict shows up in both copies. `copy.deepcopy` recursively copies everything, giving full independence (at higher cost). This bites when you copy config dicts or default templates and then mutate them per-iteration.

Cheaper alternative: If your data is simple/serializable, prefer immutable structures or rebuild rather than `deepcopy` in hot paths — `deepcopy` is slow on large nested objects.

 Start here (official docs)

- [Python documentation](#)

Q32 How do you parse and emit JSON safely, including from a subprocess?

Python

Spoken answer · ~85 s

For JSON, use the `json` module: `json.loads` parses a string, `json.load` reads from a file object, `json.dumps` serialises to a string, and `json.dump` writes to a file. Safe handling means expecting failure and validating. Wrap parsing in a try block catching `json.JSONDecodeError`, a subclass of `ValueError`, and report where the error occurred. Never trust the structure: use get with defaults, or validate against a schema using a library like `jsonschema` or `pydantic`, checking types and required fields before use. Never use `eval` to parse JSON, and avoid `pickle` for untrusted input, since it can execute code. For output, use `dumps` with `sort_keys` and `indent` for readable, deterministic output, `ensure_ascii` as appropriate, and a default function for non-serialisable types such as `datetime` or `Decimal`. For floats and precision, consider `parse_float` with `Decimal` when handling money. From a subprocess: run with `subprocess.run`, passing a list of arguments, `capture_output` and `text` set, a timeout, and `check` set to `true` so a failing command raises; then check that `stdout` isn't empty and call `json.loads` on it. Tools often print warnings to `stderr`, so keep `stdout` and `stderr` separate, so warnings don't corrupt the JSON. For large or streaming data, use newline-delimited JSON and parse one line at a time.

Likely follow-ups

- Why keep `stderr` separate from `stdout` when parsing JSON?
- Why avoid `pickle` for untrusted data?

```
import json, subprocess

out = subprocess.run(["kubectl", "get", "pods", "-o", "json"],
                     capture_output=True, text=True, check=True).stdout

data = json.loads(out)

names = [p["metadata"]["name"] for p in data["items"]]
```

```
print(json.dumps({"pods": names}, indent=2))
```

Get machine-readable output (`-o json`) and parse with `json.loads` rather than scraping text. `subprocess.run(..., check=True)` raises if the command fails (don't ignore exit codes), `text=True` decodes bytes to str, and `capture_output` grabs stdout. Emit with `json.dumps` for structured output other tools can consume.

Never `subprocess.run(cmd, shell=True)` with interpolated user input — that's shell injection. Pass a list of args (no shell) so arguments can't be reinterpreted.

Official docs

- [Python: subprocess](#)

Q33 What are generators and why are they useful for SRE data processing?

Python

Spoken answer · ~87 s

A generator is a function that uses `yield` to produce values one at a time, pausing and resuming its state between calls, instead of building and returning a full collection. Calling it returns an iterator; each next call runs the function until the next `yield`. Generator expressions offer the same laziness in a compact form. The key benefit is memory efficiency: you process items one by one, so you can handle datasets far larger than memory, such as multi-gigabyte logs, huge API result sets, or streams of metrics, with constant memory. They also enable composable pipelines: one generator reads lines, another filters errors, another parses fields, another aggregates, and data flows through lazily with no intermediate lists, much like Unix pipes. Laziness means work is done only when needed, so you can stop early after finding the first match. They model infinite or unbounded streams, like following a log file or polling a queue. For SRE tasks: paginating through a cloud API with a generator that fetches the next page on demand, reading rotated logs in sequence, and batching items with a chunking generator. Caveats: a generator can be consumed only once; you can't index or take its length; exceptions occur at consumption time; and debugging is slightly harder.

Likely follow-ups

- Why can a generator only be consumed once?
- How would you paginate an API with a generator?

```
def read_errors(path):  
  
    with open(path) as f:  
  
        for line in f:  
  
            if "ERROR" in line:
```

```
yield line.rstrip()
```

```
# pipeline stays lazy end-to-end:
```

```
errs = read_errors("app.log")
```

```
first_ten = list(itertools.islice(errs, 10))
```

A generator produces values lazily with `yield`, computing each only when requested. This keeps memory flat over huge or infinite streams and lets you build composable pipelines (`filter` → `map` → `take`) that never materialize the full dataset. Perfect for log processing, paginated API consumption, and streaming transforms where the data won't fit in RAM.

Composability: Chain generators — each stage pulls one item at a time — so a multi-GB log flows through filtering and parsing using constant memory.

 **Start here (official docs)**

- [Python documentation](#)

Q34 How do you use `argparse` to build a proper CLI tool?

Python

 **Spoken answer** · ~86 s

The `argparse` module from the standard library builds a command-line interface with help text, validation, and error messages. I create an `ArgumentParser` with a description and examples in an epilog, then add arguments: positional ones, like a required target; optional flags with short and long names; types for automatic conversion, such as `int` or a custom function; choices to restrict values, for example environment names; defaults; required flags; action `store_true` for boolean switches; and `nargs` for lists. Mutually exclusive groups prevent conflicting options. For tools with multiple operations, subparsers create subcommands, like `git`-style `deploy` and `rollback`, each with its own arguments. Good practice: put the logic in a main function that takes the parsed arguments or a list, so it's testable; call `parse_args` and return meaningful exit codes, zero on success and non-zero on failure, via `sys.exit`; print errors to `stderr` and normal output to `stdout`, so the tool works in pipelines; support a verbose flag that configures logging levels; read secrets from environment variables, not command-line arguments visible in process lists; support a dry-run flag for risky operations; and use the version action. `Argparse` automatically generates the usage message and help. For richer interfaces with less boilerplate, libraries such as `click` or `typer` are popular.

Likely follow-ups

- Why keep logic in a main function rather than at module level?
- Why avoid passing secrets as command-line arguments?

```
import argparse
```

```
p = argparse.ArgumentParser(description="deploy tool")

p.add_argument("service")

p.add_argument("-e", "--env", required=True, choices=["dev", "prod"])

p.add_argument("--dry-run", action="store_true")

p.add_argument("-n", "--replicas", type=int, default=3)

args = p.parse_args()

if args.dry_run: print("would deploy", args.service)
```

argparse gives you typed args, required flags, choices validation, defaults, and auto-generated `--help` and error messages for free. Use `type=int` for conversion, `action="store_true"` for booleans, and choices to constrain values (catching typos like `--env prdo` at parse time). A `--dry-run` flag is essential for any tool that mutates infrastructure.

For larger CLIs with subcommands, argparse subparsers or libraries like `click`/`typer` scale better and reduce boilerplate.



Official docs

- [Python: argparse](#)

Spoken answer · ~88 s

A decorator is a function that takes a function and returns a new function that wraps it, adding behaviour before or after without changing the original code. The at-sign syntax above a function definition is shorthand for reassigning the name to the decorated version. A retry decorator is a good example, and since it needs parameters like the number of attempts and the delay, it's a decorator factory with three layers: the outer function takes the parameters and returns the decorator; the decorator takes the function and returns the wrapper; the wrapper takes star args and double-star kwargs, loops for the number of attempts, tries to call the function and return its result, catches the specified exception types, sleeps with exponential backoff and jitter, and re-raises the last exception after the final attempt. Use `functools.wraps` on the wrapper to preserve the original function's name, docstring, and signature metadata, which matters for debugging, logging, and introspection. Cautions: only retry idempotent operations and transient exceptions; add logging of each retry; don't stack behaviours so deeply that flow becomes opaque; and mind that decorators run at import time. Other uses include timing, caching with `functools.lru_cache`, access control, and registering handlers. For async functions, the wrapper must itself be async and use `await` with `asyncio.sleep`.

Likely follow-ups

- Why use `functools.wraps`?
- How do you write a decorator that takes arguments?

```
import functools, time

def retry(times=3, delay=1):

    def deco(fn):

        @functools.wraps(fn)

        def wrap(*a, **kw):

            for i in range(times):

                try: return fn(*a, **kw)

                except Exception:

                    if i == times-1: raise

                    time.sleep(delay * 2**i)

            return wrap
```

```
return deco
```

```
@retry(times=5)
```

```
def fetch(): ...
```

A decorator wraps a function to add behavior (retries, timing, auth, caching) without touching its body. This three-layer form takes arguments: outer `retry` captures config, `deco` receives the function, `wrap` runs the logic and forwards `*a`, `**kw`. `@functools.wraps` preserves the original name/docstring — omit it and `tracebacks/introspection` break.

Always use `@functools.wraps` on the inner wrapper, or your decorated functions all report as `wrap` in logs and stack traces.

Official docs

- [AWS Builders' Library: Timeouts, retries and backoff with jitter](#)

Q36 What is the difference between threading, multiprocessing, and asyncio? When use each?

Python

Spoken answer · ~88 s

The right choice depends on whether the work is waiting on I/O or burning CPU, and on Python's global interpreter lock, which lets only one thread execute Python bytecode at a time in standard CPython. Threading provides concurrency for I/O-bound work: while one thread waits for a network or disk response, the lock is released, so others run. It's simple, fits blocking libraries, such as checking many hosts or calling APIs, but it won't speed up CPU-bound code, and shared state needs locks. Multiprocessing runs separate processes, each with its own interpreter and memory, so it uses multiple cores for CPU-bound tasks like parsing, compression, or number crunching. The costs are process startup, memory, and the need to serialise data between processes. Asyncio uses a single thread with an event loop and cooperative multitasking: tasks await I/O and yield control, scaling to thousands of concurrent connections with low overhead, ideal for network-heavy services, but it requires async-aware libraries, and one blocking call stalls everything. Rules of thumb: many concurrent I/O operations with async libraries, use asyncio; I/O with blocking libraries and modest concurrency, use threads, via `concurrent.futures.ThreadPoolExecutor`; CPU-bound, use multiprocessing or `ProcessPoolExecutor`. Note that newer free-threaded Python builds are changing this picture. Always bound concurrency to protect remote systems.

Likely follow-ups

- Why doesn't threading speed up CPU-bound code?
- What happens if you call a blocking function inside asyncio?

```
threading      → I/O-bound, shared memory, limited by the GIL for CPU
```

multiprocessing→ CPU-bound, true parallelism, separate memory

asyncio → high-concurrency I/O (1000s of connections), single thread

Python's GIL means threads don't run CPU-bound Python code in parallel — so use **threads** for I/O-bound work (waiting on network/disk) where the GIL is released during the wait. Use **multiprocessing** for CPU-bound work to get real parallelism across cores. Use **asyncio** when you have huge numbers of concurrent I/O operations and want efficient single-threaded cooperative concurrency without thread overhead.

SRE lens: A script that fans out to 500 hosts is I/O-bound → threads or asyncio. A script crunching numbers over a big dataset is CPU-bound → multiprocessing.

Official docs

- [Python: asyncio](#)

Q37 How do you run shell commands from Python the right way?

Python

Spoken answer · ~86 s

The right way is the subprocess module, specifically subprocess.run, passing the command as a list of arguments rather than a single string, which avoids invoking a shell and therefore avoids shell injection and quoting problems. Avoid shell equals True with any untrusted input, and avoid os.system. Useful parameters: capture_output equals True to collect stdout and stderr; text equals True to decode to strings; check equals True to raise CalledProcessError on a non-zero exit; timeout to prevent hanging, which raises TimeoutExpired and kills the child; cwd and env for the working directory and environment; and input to feed stdin. Examine returncode, stdout, and stderr afterwards. For streaming large output, use Popen and read incrementally, to avoid memory blowups or deadlocks when buffers fill; use communicate to avoid deadlocks when reading both streams. For pipelines, either connect Popen objects explicitly or do the filtering in Python, which is often clearer. When you truly need shell features like pipes and globs with a trusted fixed string, use shell equals True deliberately and quote any variable parts with shlex.quote. Log the command and exit status, never secrets, and decide how to handle failures. If a pure-Python library does the job, such as an API client, prefer it over shelling out.

Likely follow-ups

- Why pass a list instead of a string?
- How can reading output cause a deadlock?

```
import subprocess
```

```
# preferred: list args, no shell, check errors
```

```
r = subprocess.run(["df", "-h", "/"], capture_output=True,
```

```
text=True, check=True, timeout=10)
```

```
print(r.stdout)
```

```
# streaming for long-running:
```

```
with subprocess.Popen(["tail", "-f", "log"], stdout=subprocess.PIPE,
```

```
text=True) as p:
```

```
    for line in p.stdout: handle(line)
```

Use `subprocess.run` with a list of arguments and `shell=False` (the default) to avoid shell-injection.

`check=True` raises on non-zero exit, `timeout=` prevents hangs, and `text=True` gives str instead of bytes. For long-running or streaming commands use `Popen` and iterate stdout line by line.

Avoid `os.system` and `shell=True` with any interpolated input — both are injection-prone and hard to error-check. The list form is safer and clearer.

Official docs

- [Python: subprocess](#)
- [Python Documentation](#)

 Spoken answer · ~84 s

Use the standard logging module instead of print, so you get levels, timestamps, destinations, and structure that can be configured without changing code. Structure: in each module, create a logger with `logging.getLogger` of the module name; libraries only create loggers and never configure handlers, while the application's entry point configures logging once, with `basicConfig` or `dictConfig`, setting the level, handlers, and formatter. Use levels meaningfully: debug for diagnostic detail, info for normal milestones, warning for unexpected but handled conditions, error for failures, and critical for severe problems. Use lazy formatting, passing arguments to the logger instead of pre-formatting strings, and `logger.exception` inside except blocks to include the traceback. For services, emit structured JSON logs to stdout, which log collectors pick up, containing a timestamp, level, logger name, message, and context fields like request ID, user, and trace ID; context can be attached via `LoggerAdapter`, filters, or `contextvars`. Avoid logging secrets and personal data, and be careful with log volume, using sampling or rate limits on noisy messages. Make the level configurable by environment variable so you can turn on debug temporarily. Don't catch an exception just to log it and re-raise at every layer, producing duplicate entries; log once at the boundary.

Likely follow-ups

- Why shouldn't libraries configure logging handlers?
- What is the value of structured JSON logs?

```
import logging, sys

logging.basicConfig(

    level=logging.INFO, stream=sys.stdout,

    format='%(asctime)s %(levelname)s %(name)s %(message)s')

log = logging.getLogger(__name__)

log.info("starting", extra={"svc": "api"})

log.error("failed: %s", err)    # lazy %-format, not f-string
```

Use the logging module, not print — it gives levels, timestamps, module names, and configurable handlers/formatters. Log to **stdout** in containers (let the platform collect it; don't manage files). Get a logger per module with `getLogger(__name__)`. Pass args lazily (`log.info("x %s", val)`) so formatting is skipped when the level is disabled.

Structured logging: In production, emit JSON logs (via `python-json-logger` or `structlog`) so fields are queryable in Loki/Elasticsearch — far more useful than free-text lines.

Official docs

- [Python: logging](#)
- [Python Documentation](#)

Q39 What are type hints and how do they help in platform code?

Python

Spoken answer · ~86 s

Type hints are optional annotations that declare the expected types of function parameters, return values, and variables, such as a parameter annotated as a string and a return annotated as a list of dictionaries. Python doesn't enforce them at runtime; static checkers such as mypy, pyright, and pyre analyse the code before it runs. In platform and infrastructure code, which is long-lived, shared across teams, and often manipulates complex configuration, the benefits are substantial. They catch bugs early, such as passing None where a value is required, mixing up argument order, or misusing an API, before they hit production. They serve as checked documentation, making function contracts explicit and improving editor autocomplete, navigation, and safe refactoring. They make large-scale changes safer because the checker finds every broken call site. Useful features: Optional or union with None to force null handling; TypedDict for dictionary-shaped data such as parsed JSON; dataclasses and Protocol for structural typing; Literal for allowed values; generics for containers; and Callable for callbacks. Libraries like pydantic use hints for runtime validation of configuration and API payloads. Adopt gradually, starting with public interfaces, run the type checker in CI, and avoid the Any escape hatch where possible. Remember hints add some verbosity and don't replace tests.

Likely follow-ups

- What does the type checker do that Python itself doesn't?
- How does pydantic use type hints at runtime?

```
from typing import Optional

def scale(svc: str, replicas: int,
         ns: str = "default") -> Optional[dict]:
    ...

# check statically, no runtime cost:

$ mypy deploy.py
```

Type hints annotate expected types; they're not enforced at runtime but enable static checkers (mypy, pyright) to catch type bugs before deploy, and they dramatically improve editor autocomplete and readability. In platform/tooling code consumed by other engineers, hints act as machine-checked documentation of your function contracts.

Adopt gradually: Add hints to public function signatures first and run mypy in CI. Even partial typing catches a surprising number of "passed the wrong thing" bugs.

 **Start here (official docs)**

- [Python documentation](#)

Q40 How do you write a unit test with pytest, including a fixture and a mock?

Python

 Spoken answer · ~87 s

Pytest tests are plain functions whose names start with test, using bare assert statements, with pytest giving detailed failure introspection. A fixture is a function decorated with `pytest.fixture` that provides setup, and optionally teardown via `yield`, injected by naming it as a test parameter. For example, a fixture that creates a temporary configuration file using the built-in `tmp_path` fixture, or a sample client object, can be reused across tests, with scope controlling whether it runs per test, module, or session. For mocking, I replace external dependencies so tests are fast and deterministic: use `unittest.mock`, or the pytest `monkeypatch` fixture, or the `mock` fixture from `pytest-mock`. For instance, to test a function that calls an HTTP API, patch `requests.get` where it's used, not where it's defined, make it return a mock response with a status code and JSON body, call my function, then assert on the result and on how the mock was called, with `assert_called_once_with`. Test error paths too, using `side_effect` to raise a timeout and `pytest.raises` to assert the exception. Parametrisation with `pytest.mark.parametrize` runs one test over many inputs. Keep tests independent and focused, avoid over-mocking so the test still checks real behaviour, and complement unit tests with a few integration tests. Run them in CI with coverage.

Likely follow-ups

- Why patch where a name is used rather than defined?
- What does `parametrize` give you?

```
import pytest

from unittest.mock import patch

@pytest.fixture
def sample(): return {"cpu": 95, "host": "web1"}

def test_alerts_on_high_cpu(sample):

    assert should_alert(sample) is True

@patch("mymod.requests.get")
```

```
def test_fetch(mock_get):
```

```
    mock_get.return_value.json.return_value = {"ok": True}
```

```
    assert fetch_status() == {"ok": True}
```

pytest uses plain assert and gives rich failure output. A `@pytest.fixture` provides reusable test data/setup injected by parameter name. `@patch` replaces external dependencies (HTTP, subprocess, cloud SDKs) with mocks so tests are fast, deterministic, and don't hit real systems. Test behavior at the boundaries: high-CPU → alert, API failure → handled.

Mock the edges, not your logic: Patch network/cloud calls so tests are hermetic, but exercise your real decision logic — that's where the bugs are.

Official docs

- [Python: unittest.mock / testing](#)
- [pytest documentation](#)

Q41 Explain Big-O. Why does an SRE/platform engineer care about it?

Algorithms & DS

Spoken answer · ~83 s

Big-O notation describes how an algorithm's running time or memory grows as the input size grows, ignoring constants and lower-order terms. $O(1)$ is constant, like a dictionary lookup; $O(\log n)$ grows slowly, like binary search; $O(n)$ is linear, like scanning a log; $O(n \log n)$ is typical of efficient sorting; $O(n^2)$ comes from nested loops over the same data and is where scripts start to fall over. An SRE or platform engineer cares because automation runs against growing data: a script that compares every host with every other host works for 50 servers and takes hours for 50,000. Complexity predicts that cliff before production hits it. It guides choices, such as using a set or dictionary for lookups instead of searching a list inside a loop, which turns a quadratic job into a linear one. It also matters for memory: loading a whole log file is $O(n)$ memory, while streaming is $O(1)$. It applies beyond code: a query without an index scans the whole table, an alert rule that joins large series is expensive, and a retry loop can multiply load. Big-O isn't everything, since constants and cache behaviour matter for small inputs, so measure with profiling too.

Likely follow-ups

- Why does a nested loop over hosts become a problem at scale?
- When do constants matter more than Big-O?

$O(1)$ constant – dict/set lookup

$O(\log n)$ logarithmic – binary search, balanced tree

$O(n)$ linear – single loop / scan

$O(n \log n)$ – good sorts

$O(n^2)$ quadratic – nested loop over same data ← danger at scale

Big-O describes how runtime/memory grows with input size, ignoring constants. It matters because tooling that's fine on 100 items can melt on 1,000,000 — the classic SRE incident is a script that worked in staging and timed out on prod-scale data. Recognizing an accidental $O(n^2)$ (a list in check inside a loop) and fixing it to $O(n)$ with a set is everyday platform work.

Real example: "Find duplicates" with nested loops is $O(n^2)$; with a set it's $O(n)$. On a million-row log that's the difference between seconds and hours.

Official docs

- [Big-O / algorithms: Python TimeComplexity wiki](#)

Q42 Find the duplicate hostnames across two large lists efficiently.

Algorithms & DS

Spoken answer · ~90 s

The naive approach checks each hostname in the first list against the whole second list, which is $O(n \text{ times } m)$ and collapses on large inputs. The efficient way is to put one list into a set, which has average constant-time membership, then scan the other list once and keep hostnames present in the set. Better still, use set intersection directly: building two sets and intersecting them runs in roughly $O(n \text{ plus } m)$. To save memory, build the set from the smaller list and stream the larger one from the file line by line, instead of loading it. Normalise before comparing: strip whitespace, lowercase, and remove trailing dots from fully qualified names, otherwise equal hosts look different. If you also want duplicates inside one list, use a Counter and filter counts above one. For files too big for memory, the Unix way works: sort both files and use `comm` or `uniq -d`, which is $O(n \log n)$ with external sorting; or partition by a hash of the hostname into buckets that each fit in memory and compare bucket by bucket. For approximate answers on enormous data, a Bloom filter gives a compact membership test with a small false positive rate. State the trade-offs: exact versus approximate, memory versus disk, and what you do with the result.

Likely follow-ups

- Why normalise hostnames before comparing?
- How do you handle lists too large for memory?

$O(n)$ with sets

```
a = set(list_a)
```

```
b = set(list_b)
```

```
common = a & b      # in both
```

```
only_a = a - b      # in a, not b
```

```
missing = b - a     # in b, not a
```

Convert both to sets and use set algebra: intersection (&), difference (-), symmetric difference (^). Each set build is $O(n)$ and lookups are $O(1)$, so the whole diff is $O(n)$ — versus the naive nested-loop approach at $O(n^2)$. This is the bread-and-butter pattern for reconciling inventories (DNS vs cloud, desired vs actual state).

Drift detection: "What's in Terraform state but not in the cloud?" is exactly `state - actual` — a one-line set difference.

Official docs

- [Big-O / algorithms: Python TimeComplexity wiki](#)

Q43 Count the frequency of each status code in a log — in one pass.

Algorithms & DS

Spoken answer · ~87 s

I do it in a single pass with a dictionary that maps status code to count, streaming the file so memory stays small. In Python, `collections.Counter` is ideal: open the file, loop over lines, parse out the status field, and increment the counter for it. The parse step must be robust: for a combined access log, use a precompiled regex or split carefully, since the request line is quoted and may contain spaces; skip and count malformed lines rather than crashing. The work is $O(n)$ time and $O(k)$ memory, where k is the number of distinct status codes, which is tiny. In shell, `awk` does the same with an associative array: increment an entry keyed by the status field, then print all entries in an `END` block, piped to `sort`. Variations I'd mention: group by status class like 2xx, 4xx, and 5xx using integer division; add a time dimension by keying on minute and status to see error spikes; compute error rate as 5xx divided by total; and process several files with the same code, merging counters, since counters support addition, which also allows map-reduce style parallelism across files. For rotated logs, handle gzip transparently. If this is ongoing, export the counts as metrics instead of re-parsing logs.

Likely follow-ups

- Why is memory usage small here?
- How would you merge counts from many files?

Log frequency count



```
from collections import Counter
```

```
counts = Counter(parse(l)["status"] for l in open("a.log"))
```

```
print(counts.most_common(5))
```

```
# manual equivalent with defaultdict:
```

```
from collections import defaultdict
```

```
c = defaultdict(int)
```

```
for l in open("a.log"): c[parse(l)["status"]] += 1
```

`collections.Counter` is the idiomatic $O(n)$ frequency counter — feed it any iterable and it tallies, with `.most_common(k)` giving the top- k for free. `defaultdict(int)` is the manual version (auto-initializes missing keys to 0, avoiding `KeyError`). Both do it in a single streaming pass with constant extra logic.

Counter does math too: You can add/subtract Counters to combine tallies from multiple files — handy for aggregating per-shard stats.



Official docs

- [RFC 9110: HTTP Semantics \(status codes, methods\)](#)
- [Python: collections](#)

Spoken answer · ~87 s

Binary search finds a target in a sorted sequence by repeatedly halving the search range: compare the target with the middle element, discard the half that can't contain it, and repeat. It runs in $O(\log n)$, so searching a million sorted items takes about twenty comparisons.

Requirements: the data must be sorted and support random access. Pitfalls: off-by-one errors in the boundaries, infinite loops when the midpoint isn't updated correctly, and integer overflow in languages with fixed-size integers when computing the midpoint. In Python, the `bisect` module provides tested implementations, and also finds insertion points, which handle duplicates and range queries. Platform use cases: looking up a timestamp in a huge sorted log file to jump to a time window without scanning everything, which is how some log tools seek; finding the first failing commit in `git bisect`, which is binary search over history, an excellent real example; locating the threshold at which latency degrades by bisecting load levels; finding which of many configuration changes caused a regression by halving the set; and searching sorted lists of IP ranges or version numbers. The general idea, searching on a monotonic condition, extends beyond arrays, for example finding the smallest instance count that meets an SLO if more capacity never hurts.

Likely follow-ups

- How does `git bisect` use binary search?
- What precondition must the data satisfy?

```
import bisect

# find first log index at/after a target timestamp

i = bisect.bisect_left(timestamps, target_ts)

# classic implementation:

lo, hi = 0, len(arr)

while lo < hi:

    mid = (lo+hi)//2

    if arr[mid] < x: lo = mid+1

    else: hi = mid
```

Binary search finds an element (or insertion point) in a *sorted* array in $O(\log n)$ by halving the search range each step. Python's `bisect` module gives this for free. SRE use: jumping to a timestamp in a sorted log/time-series, or "git bisect" style — finding the first bad commit/deploy among many in $\log(n)$ steps instead of checking each.

git bisect is literally binary search over your commit history to find which change introduced a regression — the same $O(\log n)$ idea applied to debugging.

 Official docs

- [Big-O / algorithms: Python TimeComplexity wiki](#)

Q45 How would you implement a simple rate limiter (token bucket)?

Algorithms & DS

 Spoken answer · ~85 s

A token bucket allows bursts while enforcing an average rate. The bucket holds up to a capacity of tokens and refills at a fixed rate, say 10 tokens per second. Each request consumes a token; if one is available, the request proceeds, otherwise it is rejected or delayed. Implementation without a background thread: store the current token count and the timestamp of the last update. On each request, compute the elapsed time since the last check, add elapsed times the refill rate to the tokens, cap at capacity, update the timestamp, and then try to subtract one. Use a monotonic clock, `time.monotonic`, so system clock changes don't break it, and use a lock to make it thread-safe. For a distributed limiter, keep the state in Redis and perform the refill-and-consume atomically in a Lua script, keyed per client. Return how long the caller should wait, to support a `Retry-After` header. Design choices: bucket capacity controls burst size, and the refill rate sets the sustained throughput; per-client buckets need cleanup of idle entries to prevent memory growth. Alternatives are a leaky bucket for smoothing and sliding windows for stricter accuracy. Test with simulated time, injecting a fake clock, so tests are fast and deterministic.

Likely follow-ups

- Why compute refill lazily instead of with a timer thread?
- Why use a monotonic clock?

```
import time

class TokenBucket:

    def __init__(self, rate, cap):

        self.rate=rate; self.cap=cap

        self.tokens=cap; self.ts=time.monotonic()

    def allow(self, n=1):

        now=time.monotonic()

        self.tokens=min(self.cap, self.tokens+(now-self.ts)*self.rate)
```

```
self.ts=now
```

```
if self.tokens>=n: self.tokens-=n; return True
```

```
return False
```

A token bucket refills tokens at a steady rate up to a cap; each request consumes tokens, and is allowed only if enough are available. It permits short bursts (up to capacity) while bounding the long-run average rate — the standard algorithm behind API rate limiting and client-side throttling. Using `monotonic()` avoids clock-adjustment bugs.

Why token bucket: Unlike a fixed window, it smooths bursts and has no boundary spikes; it's what most API gateways and SDKs use under the hood.

Official docs

- [RFC 6585: HTTP 429 Too Many Requests](#)
- [Microsoft: Cloud design pattern – Throttling](#)
- [Stripe Engineering: Scaling your API with rate limiters](#)

Q46 What's a hash map collision and why does dict performance matter?

Algorithms & DS

Spoken answer · ~88 s

A hash map stores key-value pairs by computing a hash of the key and using it to pick a bucket. A collision happens when two different keys map to the same bucket. Collisions are unavoidable, since there are more possible keys than buckets, so implementations handle them. Separate chaining keeps a list of entries per bucket; open addressing, which CPython's dict uses, probes other slots in a defined sequence until an empty one is found. With a good hash function and a bounded load factor, the table resizes as it fills, keeping average lookup, insert, and delete at $O(1)$. Performance matters because dictionaries underpin so much: caches, counters, indexes, and sets. When collisions become frequent, due to a poor hash function, a nearly full table, or deliberately crafted keys, operations degrade toward $O(n)$. That has security implications: hash-flooding denial of service, where an attacker sends many keys that collide, which is why Python randomises string hashing per process. Practical points: keys must be hashable and immutable, with equality consistent with hashing; custom classes defining equality must define a matching hash; avoid mutable keys; and resizing causes occasional latency spikes, which matter in tight latency-sensitive code. Preallocating or choosing structures purposely can help. Understanding this explains why dict lookups beat list scans.

Likely follow-ups

- What is hash-flooding and how does Python mitigate it?
- Why must equal keys have equal hashes?

avg case: $O(1)$ insert / lookup / delete

collisions resolved by probing/chaining

worst case: $O(n)$ if many keys collide (rare, adversarial)

A dict/hash map stores keys by hashing them to buckets; a collision is two keys hashing to the same bucket, resolved by open addressing or chaining. Average operations are $O(1)$, which is why dicts/sets are the go-to for fast lookup. Worst case degrades to $O(n)$ under pathological/adversarial keys, but for normal SRE data this effectively never happens — the practical takeaway is "lookups are cheap, use them."

Keys must be hashable: Only immutable types (str, int, tuple) can be dict keys — that's why you can't key by a list, and why tuples exist as frozen records.

Official docs

- [Big-O / algorithms: Python TimeComplexity wiki](#)

Q47 How do you find the top-K items from a massive stream without sorting everything?

Algorithms & DS

Spoken answer · ~85 s

To find the top K items without sorting everything, use a min-heap of size K. Stream the items through: push each one onto the heap; when the heap grows beyond K, pop the smallest. At the end, the heap contains the K largest. Time is $O(n \log K)$ and memory is $O(K)$, versus $O(n \log n)$ time and $O(n)$ memory for a full sort. In Python, `heapq` provides this, and `heapq.nlargest` does it directly. Top-K by frequency over a stream, such as the most common client IPs, needs counts: when the number of distinct items fits in memory, use a `Counter` and its `most_common` method, which uses a heap internally. When there are too many distinct items, use approximate streaming algorithms: Count-Min Sketch estimates frequencies in fixed memory, combined with a small heap of candidates; the Space-Saving or Misra-Gries algorithms track heavy hitters with a bounded set of counters. For data spread across machines, compute local top-K on each, then merge them, being careful that local top lists can miss globally heavy items that are moderately frequent everywhere, so send more than K from each. Use cases: heaviest endpoints, noisiest log sources, biggest consumers of bandwidth. Mention the accuracy and memory trade-offs explicitly.

Likely follow-ups

- Why a min-heap rather than a max-heap?
- When would you use a Count-Min Sketch?

```
import heapq
```

```
# top 10 slowest requests by latency, streaming
```

```
top = heapq.nlargest(10, stream, key=lambda r: r["ms"])
```

```
# or maintain a fixed min-heap of size K:

h=[]

for r in stream:

    heapq.heappush(h, (r["ms"], r))

    if len(h)>K: heapq.heappop(h) # drop smallest
```

Sorting the whole stream is $O(n \log n)$ and needs all data in memory. A min-heap of size K keeps only the current top- K : push each item, and when the heap exceeds K , pop the smallest. That's $O(n \log K)$ time and $O(K)$ memory — ideal for "top 10 slowest endpoints" over a huge log you can't fully load.

heapq.nlargest/nsmallest does exactly this internally — reach for it before writing a manual heap.



Official docs

- [Python: collections](#)
- [Python: heapq](#)

Q48 Explain the sliding window technique with an SRE example.

Algorithms & DS

Spoken answer · ~86 s

The sliding window technique processes a contiguous range of a sequence that moves forward, updating a running result incrementally instead of recomputing from scratch. For a fixed-size window, add the new element, subtract the element leaving, and update the aggregate in constant time, so a pass over n items is $O(n)$, not $O(n \text{ times window size})$. For variable windows, two pointers expand and shrink the window to satisfy a condition. SRE examples: computing a moving average of latency or request rate over the last five minutes to smooth noise; detecting whether the error rate exceeded a threshold in any ten-minute window; implementing a sliding window rate limiter that counts requests in the last 60 seconds, using a deque of timestamps and dropping old ones from the left; finding the longest period with no errors; and burn-rate alerts, which compare error budget consumption across a short and a long window. In Python, `collections.deque` is the natural structure, with `append` on the right and `popleft` on the left, both $O(1)$; for sums, maintain a running total. Considerations: define the window by time or by count; handle gaps and out-of-order data; mind the memory used for the window contents; and be careful with floating point drift in running sums.

Likely follow-ups

- How does a deque make window updates $O(1)$?
- What is a burn-rate alert's use of two windows?

```
# max requests in any 60s window from sorted timestamps
```

```

from collections import deque

win=deque(); best=0

for t in timestamps:      # sorted

    win.append(t)

    while win[0] <= t-60: win.popleft()

    best=max(best, len(win))

```

A sliding window maintains a moving range over a sequence, adding new elements and evicting ones that fall out of range — turning many $O(n^2)$ "check every subarray" problems into $O(n)$. A deque gives $O(1)$ appends and pops from both ends. SRE use: computing peak request rate over a rolling time window, or a moving average of latency.

Rolling metrics: Burn-rate alerting and "requests in the last N seconds" are sliding-window computations — the same pattern you'd code by hand.



Official docs

- [Big-O / algorithms: Python TimeComplexity wiki](#)

Q49 How do you safely chunk a huge list to process in batches (e.g. API bulk calls)?

Algorithms & DS

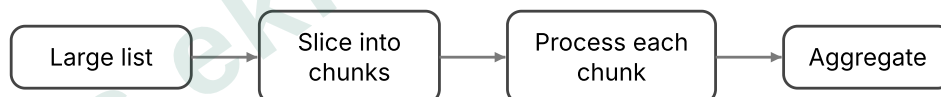
🗣 Spoken answer · ~83 s

Chunking splits a large sequence into fixed-size batches so you can respect API limits, control memory, and make progress with partial failure handling. In Python 3.12, `itertools.batched` does it; otherwise write a generator that takes an iterable and a size, uses `itertools.islice` to take `n` items at a time, and yields each non-empty chunk until the iterator is exhausted. It works with any iterable, including generators, so you never need the entire list in memory. Using a generator-based chunker with streaming input keeps memory at $O(\text{batch size})$. Safe usage for bulk API calls: respect the API's maximum batch size, for example 25 for some DynamoDB batch writes or 1,000 for some delete calls; handle partial failures by inspecting the response for unprocessed items and retrying only those, with backoff; add rate limiting or sleeps between batches; make each batch idempotent so retries are safe; log progress and checkpoint so a crash can resume rather than start over; and consider concurrency with a bounded pool, balancing speed against throttling. Be careful that the last chunk may be smaller, and that slicing a list with a step is the simple option for in-memory lists. For payload-size limits, chunk by bytes, not count.

Likely follow-ups

- Why handle partial failures within a batch?
- How does a chunking generator keep memory low?

🚩 Batching



```
def chunks(seq, size):  
    for i in range(0, len(seq), size):  
        yield seq[i:i+size]  
  
for batch in chunks(host_ids, 100):  
    api.bulk_update(batch)    # 100 at a time  
  
# lazy version for iterators (3.12+): itertools.batched  
  
from itertools import batched
```

Batching avoids hitting API payload limits, request-size caps, or memory blowups when operating on huge collections. The generator yields successive slices of size; for arbitrary iterators (not just lists), `itertools.batched` (3.12+) or an `islice` loop works without slicing. Combine with rate limiting and retries for

robust bulk operations.

Tune the batch size: Too small = many round-trips (slow); too large = timeouts/memory. Pick based on the API's documented limits, not a guess.

 **Start here (official docs)**

- [Python documentation](#)

Q50 What's the difference between $O(n)$ memory and $O(1)$ memory, and why prefer streaming?

Algorithms & DS

 Spoken answer · ~85 s

$O(n)$ memory means the memory needed grows linearly with input size, for example reading an entire file or building a list of every record. $O(1)$ memory, constant space, means usage stays fixed regardless of input size, such as processing a file line by line while keeping only a running counter. Streaming is preferred because data volumes grow faster than machine memory, and production incidents tend to involve the biggest inputs. A script that loads a one-gigabyte log works on a laptop test and then gets killed by the out-of-memory killer in a container with a 512 megabyte limit, often at the worst time. Streaming also gives lower latency to first result, since you start producing output immediately, and enables pipelines where stages overlap. It makes resource usage predictable, which matters for capacity planning, Kubernetes memory limits, and avoiding noisy neighbours. Techniques: iterate over file objects, use generators, read in chunks, process pages from APIs one at a time, aggregate incrementally with counters and sketches, and write results as you go. Caveats: some tasks inherently need more state, like sorting or joining, which require external sort, partitioning, or a database; streaming code is harder to restart or re-iterate, since a generator can be consumed once.

Likely follow-ups

- Which tasks inherently need more than $O(1)$ memory?
- How do container memory limits make streaming important?

```
#  $O(n)$  memory – whole file in RAM
```

```
data = open("huge.log").readlines()
```

```
errors = [l for l in data if "ERR" in l]
```

```
#  $O(1)$  memory – stream, count only
```

```
n = sum(1 for l in open("huge.log") if "ERR" in l)
```

$O(n)$ memory grows with input size; $O(1)$ (constant) memory does not. Streaming processes one record at a time and keeps only a small running state (a count, a sum, a fixed-size heap), so it handles arbitrarily large inputs that would never fit in RAM. For SRE log/metric processing, always ask "do I need all the data at once,

or can I stream?" — usually streaming wins.

OOM-killer territory: Loading a 50GB log with `.readlines()` on a 4GB box gets your process killed. Stream it.

Official docs

- [Big-O / algorithms: Python TimeComplexity wiki](#)

Q51 How would you deduplicate log lines while preserving order?

Algorithms & DS

Spoken answer · ~88 s

The goal is to drop repeated lines but keep the first occurrence of each in its original order. In Python, keep a set of lines already seen and a loop that yields a line only if it hasn't been seen, adding it to the set as you go. Membership checks are $O(1)$ on average, so the whole pass is $O(n)$ time, with memory proportional to the number of unique lines. As a generator, it streams, writing output as it reads. The classic shell equivalent is `awk` with a not-seen test on an associative array: print the line when its counter for that line was zero, then increment, which preserves order, unlike `sort -u`, which reorders. `Dict.fromkeys` on a list is another order-preserving idiom, since dictionaries keep insertion order, but it needs the whole list in memory. If the number of unique lines is huge, memory becomes the limit: store hashes of lines instead of the lines themselves, which cuts memory at the cost of a tiny collision risk, or use a Bloom filter for approximate deduplication, accepting false positives that drop a few legitimate lines. Sometimes the right notion is near-duplicate collapse: normalise lines first by removing timestamps, IDs, and numbers, then count by pattern, which is more useful for log analysis.

Likely follow-ups

- Why does `sort -u` not preserve order?
- How can hashing reduce memory in deduplication?

```
seen = set(); out = []

for line in stream:

    if line not in seen:

        seen.add(line); out.append(line)

# 3.7+ shortcut (insertion-ordered dict):

deduped = list(dict.fromkeys(stream))
```

A plain `set(stream)` dedups but loses order. To keep first-seen order, track a seen set for $O(1)$ membership and append only new items — $O(n)$ time. The slick one-liner `dict.fromkeys` exploits dict's guaranteed insertion order (3.7+) to dedup-and-preserve in one expression.

Memory note: Dedup needs $O(\text{unique})$ memory to remember what was seen. For truly massive cardinality, a probabilistic structure (Bloom filter) trades a tiny false-positive rate for huge memory savings.

 **Start here (official docs)**

- [Python documentation](#)

Q52 Explain recursion vs iteration — and the risk of recursion in Python.

Algorithms & DS

 **Spoken answer · ~77 s**

Iteration repeats using loops and explicit state; recursion has a function call itself on a smaller subproblem until reaching a base case. Recursion is natural for self-similar structures: walking a directory tree, traversing nested JSON or Kubernetes manifests, parsing expressions, and divide-and-conquer algorithms. The code is often shorter and closer to the problem definition. Iteration is usually faster and uses constant stack space. Python's risks with recursion: it doesn't optimise tail calls, so each call adds a stack frame; the default recursion limit is about 1,000, and exceeding it raises `RecursionError`; deep recursion also costs memory and time from call overhead. Deeply nested data, or a linked list of ten thousand nodes, can crash a recursive solution. Also, a missing or wrong base case causes infinite recursion, and naive recursion can repeat work exponentially, like a naive Fibonacci, which memoisation with `functools.lru_cache` fixes. Mitigations: convert to iteration with an explicit stack or queue, which also gives control over depth-first versus breadth-first order; use generators with `yield from` for tree walks; raise the recursion limit only with care, since the real C stack can overflow.

Likely follow-ups

- Why does Python hit a recursion limit?
- How do you convert a recursive tree walk to iteration?

```
# recursive dir walk – risks RecursionError on deep trees
```

```
def walk(path):
```

```
    for e in os.scandir(path):
```

```
        if e.is_dir(): yield from walk(e.path)
```

```
        else: yield e.path
```

```
# Python default recursion limit ≈ 1000
```

Recursion expresses tree/nested problems naturally; iteration uses an explicit loop/stack. Python has no tail-call optimization and a default recursion limit (~1000 frames), so deep recursion raises `RecursionError` and each frame costs memory. For deep or unbounded structures (huge directory trees, large graphs), convert to an iterative approach with an explicit stack/queue.

Practical rule: Recursion is fine for shallow, bounded depth (parsing, small trees). For anything that could be thousands deep, iterate with a stack.

 Official docs

- [Python Documentation](#)

Q53 How do you merge K sorted streams (e.g. logs from many servers) by timestamp?

Algorithms & DS

 Spoken answer · ~82 s

Merging K sorted streams, such as per-server logs each ordered by time, is solved with a min-heap holding the next item from each stream. Initialise by pushing the first item from every stream, as tuples of timestamp, stream index, and the item. Repeatedly pop the smallest, output it, and push the next item from the same stream, if any. Each output costs $O(\log K)$, so merging N total items takes $O(N \log K)$ time, with only $O(K)$ memory, since you never load whole streams. In Python, `heapq.merge` does exactly this lazily, taking any number of sorted iterables and a key function, so you can merge open file iterators directly: parse the timestamp in the key function. Details to handle: all inputs must already be sorted; if a stream is slightly out of order, buffer a small window or sort within that window; ties need a stable tie-breaker so comparison never falls back to comparing non-comparable items, which is why the stream index is included; timestamps must be parsed into comparable values with consistent time zones, not compared as strings, unless in a sortable ISO format; and clocks across servers may skew, so merged order is approximate.

Likely follow-ups

- Why include the stream index in the heap tuple?
- How does clock skew affect merged ordering?

```
import heapq
```

```
# each src is an iterator of (timestamp, line), already sorted
```

```
merged = heapq.merge(*sources, key=lambda x: x[0])
```

```
for ts, line in merged:
```

```
    print(ts, line)
```

`heapq.merge` lazily merges multiple already-sorted iterables into one sorted stream using a heap, in $O(\text{total} \log K)$ time and $O(K)$ memory — it never loads all logs at once. This is exactly how you'd interleave per-server log files into a single chronological view for incident analysis without sorting the combined multi-GB dataset.

Precondition: Each input must already be sorted by the merge key (per-server logs usually are by time).
heapq.merge only does the K-way interleave, not a full sort.

 Official docs

- [Python: heapq](#)

Q54 **What is an LRU cache and how do you add one in Python?**

Algorithms & DS

 Spoken answer · ~88 s

An LRU cache, least recently used, keeps a bounded number of items and evicts the one that was used longest ago when full, based on the assumption that recently used items are likely to be used again. A correct implementation needs $O(1)$ get and put, which combines a hash map for lookup with a doubly linked list to track recency order: on access, move the node to the front; on insert when full, remove the node at the tail. In Python, `collections.OrderedDict` gives this directly: `move_to_end` on access, and `popitem` with `last` equal to `False` to evict the oldest. For caching function results, the standard library offers `functools.lru_cache`, a decorator with a `maxsize` parameter, which memoises on hashable arguments, exposes `cache_info` for hits and misses, and `cache_clear`. Caveats: arguments must be hashable; cached results are returned by reference, so mutable results can be modified by callers; there's no time-based expiry, so stale data persists, and for TTLs, use a library like `cachetools`; in methods, `lru_cache` keeps references to self and can cause memory leaks; it is thread-safe for the cache structure, but the function may still run twice on concurrent misses. Use cases: caching DNS lookups, metadata API results, compiled regexes, and parsed configuration. Measure hit ratio to size it properly.

Likely follow-ups

- Why is a hash map plus a linked list used?
- What problems does `lru_cache` have on instance methods?

```
from functools import lru_cache
```

```
@lru_cache(maxsize=1024)
```

```
def resolve(hostname):    # expensive DNS / API lookup
```

```
    return slow_lookup(hostname)
```

```
# inspect / clear:
```

```
resolve.cache_info()
```

```
resolve.cache_clear()
```

An LRU (Least Recently Used) cache stores recent results and evicts the least-recently-used entry when full, giving $O(1)$ get/put. `@functools.lru_cache` memoizes a function's return value keyed by its arguments — perfect for expensive, repeatable lookups (DNS, config fetches, API calls with stable answers). Bound `maxsize` so the cache can't grow unbounded.

Cache invalidation: `lru_cache` never expires entries on its own — only on eviction. For data that changes, you need TTL logic (use `cachetools.TTLCache`) or you'll serve stale results.

Official docs

- [Redis: Key eviction](#)
- [Python Documentation](#)

Q55 How do you detect anomalies in a metric series with simple statistics?

Algorithms & DS

 Spoken answer · ~86 s

Simple statistical methods detect unusual values by comparing each point to expected behaviour derived from recent history. The z-score approach computes the mean and standard deviation over a window and flags points more than, say, three standard deviations away. It assumes roughly normal data and is sensitive to outliers distorting the mean and deviation. A more robust variant uses the median and the median absolute deviation, which isn't thrown off by the very anomalies you're trying to find. The interquartile range method flags values beyond 1.5 times the IQR outside the quartiles. For time series, compare with a rolling baseline: a moving average or an exponentially weighted moving average, and flag large residuals. Account for seasonality, since traffic differs by hour and weekday: compare with the same hour last week, or subtract a seasonal baseline, which avoids false alarms each morning. Practical points: use rates or percentiles instead of raw counters; require several consecutive breaches or a minimum duration to reduce noise; set a minimum absolute threshold so tiny changes in near-zero metrics don't alert; handle missing data; and tune using historical incidents to balance false positives against missed detections. Static thresholds are simpler and often adequate; statistical detection helps with metrics whose normal level shifts.

Likely follow-ups

- Why is median absolute deviation more robust than standard deviation?
- How do you handle daily seasonality?

```
import statistics as st
```

```
mean = st.mean(window); sd = st.pstdev(window)
```

```
# flag points more than 3 standard deviations out
```

```
for x in series:
```

```
if sd and abs(x-mean) > 3*sd:
```

```
    flag(x)
```

A simple, explainable anomaly check is the z-score: flag points more than ~3 standard deviations from the rolling mean. It's cheap and good for roughly-normal metrics (latency, request rate). For real systems prefer a rolling/EWMA baseline so it adapts, and be aware it assumes normality — spiky or seasonal metrics need percentile-based or seasonal methods instead.

Know the limits: z-score breaks on skewed or seasonal data (daily traffic cycles). Mention MAD (median absolute deviation) or percentile thresholds as more robust alternatives.

 **Start here (official docs)**

- [Python documentation](#)

Q56 Reverse-engineer: a script is slow. How do you profile and find the bottleneck?

Algorithms & DS

 **Spoken answer · ~88 s**

I profile before optimising, since intuition about bottlenecks is often wrong. First, reproduce the slowness with a realistic input and get a baseline timing, using the time command for wall, user, and system time: high system time suggests I/O or syscalls, high user time suggests computation, and wall time much higher than both suggests waiting on network or locks. For Python, run cProfile, such as `python -m cProfile -s cumulative script.py`, to see which functions consume the most time, and visualise with snakeviz; use line_profiler for hot lines, and py-spy to sample a running process without modifying it, producing flame graphs, useful in production. Use tracemalloc for memory. For shell scripts, use `set -x` with timestamps in PS4 to see slow commands, or time sections, and look for loops launching many subprocesses, which is a common culprit. At the system level, `strace -c` summarises syscalls, and checks for slow DNS or repeated network calls. Typical findings: an accidentally quadratic loop, repeated API calls that could be batched or cached, reading files repeatedly, unbuffered output, or sequential network calls that could be concurrent. Fix the biggest hotspot first, re-measure to confirm improvement, and stop when it's fast enough. Keep a benchmark to prevent regressions. Avoid premature micro-optimisation, and prefer algorithmic fixes over tweaks.

Likely follow-ups

- What does it tell you if wall time far exceeds CPU time?
- Why use py-spy in production?

```
python -m cProfile -s cumtime script.py    # function-level timing
```

```
# line-level: pip install line_profiler; @profile + kernprof
```

```
import timeit; timeit.timeit(stmt, number=1000) # micro-bench
```

```
# memory: pip install memory_profiler
```

Measure before optimizing — intuition about hot spots is usually wrong. cProfile shows where cumulative time goes by function; line_profiler drills to specific lines; timeit micro-benchmarks small snippets; memory_profiler finds memory hogs. Find the actual bottleneck, fix that one thing, then re-measure.

Amdahl's law: Optimizing code that's 2% of runtime is wasted effort. Profile, attack the biggest slice (often an accidental $O(n^2)$ or repeated I/O), re-profile.

 **Start here (official docs)**

- [Python documentation](#)

Q57 How do you implement exponential backoff with jitter correctly?

Algorithms & DS

 **Spoken answer · ~88 s**

Exponential backoff increases the wait between retries multiplicatively, so a struggling service gets progressively more breathing room. The basic delay is a base value times two to the power of the attempt number, capped at a maximum. Jitter adds randomness so many clients don't retry in sync, which would create waves of load. The recommended approach is full jitter: pick a random delay uniformly between zero and the capped exponential value. Another is decorrelated jitter, where each delay is random between the base and three times the previous delay. Correct implementation details: put a maximum number of attempts and an overall deadline, so retries can't continue forever; cap the delay, for instance at 30 or 60 seconds; retry only on transient errors, such as timeouts, connection resets, 429, and 5xx, never on client errors like 400 or 401; honour the Retry-After header when present; make the operation idempotent or use idempotency keys, since a timed-out request may have succeeded; use a monotonic clock and an interruptible sleep so signals can stop the loop; log each attempt with the reason and delay; and consider a retry budget limiting retries to a fraction of total traffic, to prevent retry storms across the system. In Python, libraries like tenacity and urllib3's Retry implement these correctly.

Likely follow-ups

- Why is full jitter better than a fixed delay?
- What is a retry budget?

```
import random
```

```
def backoff(attempt, base=0.5, cap=30):
```

```
    # full jitter: random between 0 and the capped exp delay
```

```
    exp = min(cap, base * (2 ** attempt))
```

```
return random.uniform(0, exp)
```

Exponential backoff doubles the wait per attempt ($\text{base} * 2^{*n}$) up to a cap, so retries don't hammer a struggling service. Jitter randomizes the delay so many clients don't retry in lockstep — without it, every client that failed at the same instant retries simultaneously, creating a synchronized thundering herd. "Full jitter" (random between 0 and the delay) is the AWS-recommended variant.

Jitter is not optional: Backoff without jitter still produces correlated retry spikes after an outage. The randomization is what actually spreads the load.

Official docs

- [AWS Builders' Library: Timeouts, retries and backoff with jitter](#)

Q58 What data structure powers a Bloom filter and when would an SRE use one?

Algorithms & DS

Spoken answer · ~63 s

A Bloom filter is a probabilistic set membership structure built on a bit array of m bits and k independent hash functions. To add an item, hash it with each function and set the corresponding k bits. To query, hash and check those bits: if any bit is zero, the item is definitely not in the set; if all are one, it is probably in the set. So it has no false negatives but can have false positives, with a rate controlled by the bit array size, number of hashes, and number of items; roughly ten bits per item with seven hashes gives about a one percent false positive rate. It uses far less memory than storing the items and gives constant-time operations. Limitations: you can't remove items in the basic version, though counting Bloom filters allow it; you can't list the items; and it fills up and degrades if overloaded.

Likely follow-ups

- Why can a Bloom filter never give a false negative?
- Why can't you delete from a basic Bloom filter?

```
bit array + k hash functions
```

```
add(x):    set bits hash1(x)..hashk(x)
```

```
maybe(x): all those bits set? → "possibly present"
```

```
guarantee: no false negatives; small false-positive rate
```

A Bloom filter is a compact probabilistic set: it tells you an element is *definitely not* present or *possibly* present, using a bit array and several hash functions — a fraction of the memory of a real set. SRE uses: "have I already alerted on this fingerprint?", dedup at massive scale, or a cheap pre-check before an expensive lookup (skip the DB hit if the Bloom filter says "definitely not there").

Trade-off: You accept a tunable false-positive rate in exchange for huge memory savings. Never use it where a false "yes" is unacceptable without a confirming check.

 **Start here (official docs)**

- [Python documentation](#)

Q59 Write a Python script to page through a paginated REST API and collect all results.

Automation / API

 **Spoken answer · ~86 s**

Paginated APIs return results in pages with a way to fetch the next, typically a cursor or next token, a next URL, or page numbers with offsets. The script loops: request a page, append its items, find the continuation indicator, and repeat until none remains. In Python I'd write a generator that yields items page by page, so callers can stream results instead of holding them all in memory. Use a requests Session with a timeout and retries; pass the cursor from the previous response; stop when the next token is absent or the page is empty. Details that matter: prefer cursor-based pagination to offsets, because offsets can skip or duplicate items when data changes during iteration; respect rate limits, with sleeps or by handling 429 and Retry-After; set a maximum page count or total as a safety against infinite loops caused by a bug, such as a server that returns the same cursor; follow Link headers when the API uses them; handle partial failure by checkpointing the last cursor so a rerun can resume; and request the largest allowed page size to reduce calls. Log progress. Cloud SDKs such as boto3 provide paginators that handle this, so use them where available, instead of re-implementing the loop.

Likely follow-ups

- Why is cursor-based pagination safer than offsets?
- How do you guard against an infinite pagination loop?

```
def fetch_all(url, session):  
  
    items=[]  
  
    while url:  
  
        r=session.get(url, timeout=10); r.raise_for_status()  
  
        body=r.json()  
  
        items.extend(body["items"])  
  
        url=body.get("next")      # None when no more pages  
  
    return items
```

Most APIs cap results per page and return a cursor/next-link. Loop until there's no next page, accumulating items. Reuse a Session (connection pooling + shared retry/auth), always set a timeout, and check status. For very large result sets, yield each page instead of building one giant list, to stay memory-friendly.

Respect rate limits: Honor Retry-After / X-RateLimit-Remaining headers and back off — hammering a paginated API across thousands of pages will get you throttled or banned.

Official docs

- [REST API pagination \(Google AIP-158\)](#)
- [Google API Design Guide](#)
- [Python Documentation](#)

Q60 How do you authenticate to a cloud API and avoid hardcoding credentials?

Automation / API

Spoken answer · ~89 s

The principle is never to put long-lived credentials in code, repositories, images, or command lines. Prefer ambient, short-lived identities provided by the platform. On cloud VMs and containers, use the attached role: AWS instance profiles or IAM Roles for Service Accounts, Azure managed identities, and GCP service accounts via workload identity. SDKs such as boto3 or Google's client libraries automatically discover these through a default credential chain, so the script contains no secrets at all. For CI systems, use OIDC federation, where the pipeline presents a signed token and exchanges it for temporary cloud credentials, with no stored keys. For humans running scripts locally, use SSO login that produces temporary credentials cached by the CLI, rather than long-term access keys. When a secret is unavoidable, such as a third-party API key, fetch it at runtime from a secrets manager like Vault, AWS Secrets Manager, or Azure Key Vault, or inject it as an environment variable or mounted file from the orchestrator, and keep it in memory only. Scope permissions with least privilege, give each automation its own identity, rotate and set short TTLs, and audit usage. Avoid logging credentials, and scan repositories with secret scanners to catch accidents. If a key leaks, revoke and rotate immediately. Document how scripts obtain identity so it's reproducible.

Likely follow-ups

- What is OIDC federation for CI?
- Why prefer a default credential chain in SDKs?

NEVER hardcode keys. Prefer, in order:

1. Workload identity / IRSA / instance metadata (no static keys)

2. Secrets manager (Vault, AWS SM, GCP Secret Manager)

3. Env vars injected at runtime

```
import boto3          # auto-uses role/metadata creds
```

```
s3 = boto3.client("s3") # no keys in code
```

The credential hierarchy: best is keyless — workload identity (GKE), IRSA (EKS), or instance metadata, where the cloud injects short-lived rotated credentials your SDK picks up automatically. Next best is a secrets manager fetched at runtime. Env vars are acceptable when injected by the platform. Never commit keys to code or Git — it's the most common cause of cloud breaches.

Static long-lived keys are the enemy: They leak via Git history, logs, and screenshots. Use short-lived, automatically-rotated identity wherever possible.

 **Start here (official docs)**

- [Python documentation](#)

Q61 **Write an idempotent script to ensure a resource exists (create-or-update pattern).** Automation / API

 **Spoken answer · ~82 s**

The create-or-update pattern makes a script converge a resource to a desired state, safe to rerun. Steps: define the desired state in a declarative form; look up the current resource by a stable identifier, such as name or ID; if it doesn't exist, create it; if it exists, compare the relevant fields with the desired ones, and update only if they differ; then verify and report what changed. Make the comparison precise: compare only the fields you manage, ignoring server-generated ones like timestamps and IDs, and normalise representations, such as ordering of lists, so you don't produce spurious diffs. Handle races: two runs may both see 'not found' and both create, so rely on the API's uniqueness constraint and treat an 'already exists' error as a signal to re-read and update, or use native upsert or idempotency tokens, and apply optimistic concurrency with ETags or resource versions to avoid overwriting concurrent changes. Support a dry-run mode that prints the planned changes without applying them, and exit codes or output that indicate changed or unchanged. Log decisions. Be careful with deletion; if you manage a set, only delete things you can prove you own, using tags or labels.

Likely follow-ups

- How do you avoid a race between two concurrent runs?
- Why compare only the fields you manage?

Idempotent script



```
def ensure_bucket(client, name):

    existing = {b["Name"] for b in client.list_buckets()["Buckets"]}

    if name in existing:

        log.info("bucket %s exists", name); return False

    client.create_bucket(Bucket=name)

    log.info("created %s", name); return True
```

Idempotent automation checks current state and only acts if needed, so re-running causes no errors or duplicates — essential because automation reruns on retries, cron, and CI. The "ensure" pattern: read desired vs actual, converge the difference. Return whether a change was made so callers can report drift. This is the philosophy behind Terraform/Ansible — declare the end state, not the steps.

Reconcile, don't imperatively create: "Make it so X exists" is safe to run repeatedly; "create X" fails the second time. Always write the former.



Official docs

- [RFC 9110: HTTP Semantics \(status codes, methods\)](#)
- [Idempotence in scripts \(Ansible concepts\)](#)

Spoken answer · ~87 s

The key point is to use `yaml.safe_load`, never `yaml.load` with the default or full loader on untrusted input, because the unsafe loaders can construct arbitrary Python objects, enabling code execution. `safe_load` restricts parsing to basic types: dicts, lists, strings, numbers, booleans, and `None`. Kubernetes manifests often contain multiple documents separated by triple dashes, so use `yaml.safe_load_all`, which yields each document; skip empty ones. Handle errors: catch `yaml.YAMLError` and report the line and column using the problem mark, so users can fix the file. Validate the structure after parsing, because valid YAML isn't a valid manifest: check required keys such as `apiVersion`, `kind`, and `metadata.name`, types, and allowed values, using a schema such as `jsonschema`, `pydantic` models, or `kubeconform` and the official OpenAPI schemas; use `kubectl apply` with `dry-run` server for the strongest check. Beware YAML gotchas: the Norway problem, where the unquoted value `no` becomes boolean `false` in YAML 1.1 parsers; version numbers like `1.10` parsed as floats and turned into `1.1`; octal-looking numbers; and tabs not allowed. When you must modify and write back while preserving comments and formatting, use `ruamel.yaml`, as `PyYAML` discards comments. When writing, use `safe_dump` with `default_flow_style` set to `False` and `sort_keys` `False` to preserve key order. Never include secrets in committed YAML.

Likely follow-ups

- Why is `yaml.load` dangerous?
- What is the Norway problem?

```
import yaml

with open("deploy.yaml") as f:

    doc = yaml.safe_load(f)          # NOT yaml.load

    replicas = doc["spec"]["replicas"]

# multi-document files (--- separated):

for d in yaml.safe_load_all(open("all.yaml")):

    print(d["kind"])
```

Always use `yaml.safe_load`, never `yaml.load` without a safe loader — the unsafe loader can instantiate arbitrary Python objects and execute code from a malicious YAML file (a real RCE vector). For K8s manifests with multiple documents separated by `---`, use `safe_load_all` which yields each document.

Security: `yaml.load(untrusted)` is a known remote-code-execution risk. `safe_load` only builds plain dicts/lists — use it for anything you didn't write yourself.

Q63 Write a script that sends a Slack/webhook alert with structured payload.

Automation / API

🗣 Spoken answer · ~89 s

Sending a webhook alert means building a JSON payload and POSTing it, with care for reliability and secrets. Take the webhook URL from an environment variable or secrets manager, because it acts as a credential. Build a structured payload: for Slack, a text fallback for notifications plus blocks or attachments with fields such as service, environment, severity, summary, a link to the dashboard or runbook, and the timestamp; use consistent field names so messages are scannable. Colour or an emoji indicates severity. Send it with `requests.post`, passing `json` equals the payload rather than a hand-built string, with a timeout, and call `raise_for_status`. Handle failures: retry with backoff on 5xx and 429, honouring Retry-After for Slack's rate limits, but never let an alerting failure crash the main job; catch exceptions, log them, and fall back to another channel such as email or stderr. Avoid alert spam by deduplicating, using a fingerprint and a cooldown, grouping related events, and rate limiting; include a resolved notification when the issue clears. Do not include secrets or sensitive data in the message. For paging, integrate with Alertmanager, PagerDuty, or Opsgenie using their APIs instead of ad hoc webhooks, since they handle routing, escalation, and acknowledgement. Test with a dedicated test channel, and truncate long text to Slack's size limits.

Likely follow-ups

- Why should alert failures not crash the main job?
- How do you prevent alert spam?

```
import requests, os

def notify(text, level="warning"):
    hook = os.environ["SLACK_WEBHOOK"] # from secret, not code

    payload = {"attachments":[{"
        "color": {"warning":"#daa","error":"#f00"}.get(level,"#ccc"),
        "text": text}]}

    r = requests.post(hook, json=payload, timeout=5)

    r.raise_for_status()
```

POST a JSON payload to the incoming-webhook URL (kept in an env var/secret, never in code). Set a short timeout so a slow Slack doesn't block your script, and check the response. Structure the message (color by severity, fields) so on-call can triage at a glance. This is the building block for custom alerting hooks from scripts and cron jobs.

Don't spam: Add dedup/throttling so a flapping condition doesn't fire 500 Slack messages — that's how alert fatigue starts.

 **Start here (official docs)**

- [Python documentation](#)

Q64 How do you write a script that's safe to run from cron (locking, logging, exit codes)?

Automation / API

 Spoken answer · ~75 s

Cron runs scripts in a minimal environment with no terminal, so scripts must be defensive. Locking: prevent overlapping runs if a job takes longer than its interval, using flock on a lock file, such as flock -n with a lock path wrapping the command, or a lock inside the script, so the second instance exits quickly. Use absolute paths for commands and files, and set PATH and other environment variables explicitly, since cron's environment differs from your shell; set the working directory deliberately. Use set -euo pipefail in Bash, plus a trap for cleanup. Logging: cron emails stdout and stderr, which is noisy or lost, so redirect output to a log file or send it to syslog via logger, with timestamps and a run ID; rotate logs with logrotate. Don't produce output on success unless needed. Exit codes: return zero on success and non-zero on failure, and make failures visible by alerting, with a dead-man's-switch heartbeat, using a service like healthchecks.io, so you're notified if the job doesn't run at all, which is the failure cron itself can't tell you about.

Likely follow-ups

- How does flock prevent overlapping runs?
- What does a dead-man's-switch detect?

```
#!/usr/bin/env bash
```

```
set -Eeuo pipefail
```

```
exec 9>/var/lock/myjob.lock
```

```
flock -n 9 || { echo "already running"; exit 0; }
```

```
exec >>/var/log/myjob.log 2>&1 # all output to log
```

```
echo "$(date -Is) start"
```

```
# ... work ...
```

Cron jobs need three things: a lock so overlapping runs don't collide (`flock -n` exits if another instance holds it), explicit logging (cron's environment is minimal and output otherwise vanishes or emails root), and correct exit codes. Also set full paths and required env vars — cron runs with a bare PATH and no profile, a classic "works in my shell, fails in cron" trap.

The cron environment is not your shell: No aliases, minimal PATH, different locale. Always use absolute paths and source needed env explicitly.

Official docs

- [Kubernetes: Determine the reason for Pod failure](#)

Q65 How do you template config files in Python (e.g. generate per-environment configs)?

Automation / API

Spoken answer · ~83 s

The standard approach is a template engine, usually Jinja2, with a single template per config file and a small set of per-environment values supplied from data files. Structure: keep a base values file for shared settings and an overlay file per environment, such as dev, staging, and prod, deep-merged so each environment only declares differences. The script loads the values using `safe_load`, builds a Jinja2 Environment with the template directory, and importantly sets `StrictUndefined`, so a missing variable raises an error instead of rendering blank. Render the template with the merged values and write the output file. Use `autoescape` appropriately for HTML, but not for configs. Good practices: validate the values against a schema before rendering, and validate the output, for instance by parsing the rendered YAML or JSON or running the target tool's config test, such as `nginx -t`; write files atomically via a temp file and rename; make rendering deterministic, so reruns produce identical output, and diffs are meaningful in code review; keep secrets out of values files, injecting them at deploy time from a secret store; and commit templates and values to version control. Avoid putting heavy logic in templates; compute complex values in Python.

Likely follow-ups

- Why use `StrictUndefined`?
- Why validate the rendered output?

```
from jinja2 import Environment, FileSystemLoader
```

```
env = Environment(loader=FileSystemLoader("templates"),
```

```
                undefined=__import__("jinja2").StrictUndefined)
```

```
tpl = env.get_template("app.conf.j2")
```

```
out = templ.render(env="prod", replicas=5, region="us-east1")
```

```
open("app.conf", "w").write(out)
```

Jinja2 templating generates config from a template plus variables — the engine behind Ansible and Helm-style rendering. Use `StrictUndefined` so a missing variable raises an error instead of silently rendering blank (which causes broken configs that fail mysteriously at runtime). Keep templates in version control and render per-environment.

Fail loud on missing vars: `StrictUndefined` turns "forgot to set region" into an immediate template error, not a config with an empty region that breaks in prod.



Official docs

- [Python Documentation](#)

Q66 Write a script to back up a directory and rotate old backups.

Automation / API

Spoken answer · ~86 s

A backup script creates a dated archive of the directory, verifies it, then deletes old ones according to a retention policy. Steps. Create the archive with a timestamped name in a dedicated backup directory, using tar with gzip or zstd compression; write to a temporary name and rename on success, so partial archives are never mistaken for good ones. For consistency, snapshot first if the data is changing, using filesystem or LVM snapshots, or quiesce the application. Verify: check the tar's exit code, test the archive can be listed, and record a checksum. Copy offsite, since a backup on the same disk isn't protection: sync to object storage with encryption, and consider immutability. Rotation: keep, for example, the last seven daily, four weekly, and twelve monthly backups; implement by sorting archives by name or modification time, and deleting those not matching the keep rules. Be extremely careful with deletion: match only files with your expected naming pattern, never delete the newest backup even if the policy says otherwise, and refuse to delete if the new backup failed, so a failing job doesn't erode the good copies. Use locking to prevent overlap, log actions, alert on failure, and monitor backup age and size, with sudden size changes flagged.

Likely follow-ups

- Why write to a temporary name and then rename?
- Why is a restore test essential?

```
ts=$(date +%Y%m%d-%H%M%S)
```

```
tar czf "/backups/data-$ts.tgz" -C /var/lib/app data
```

```
# keep only the 7 most recent
```

```
ls -lt /backups/data-*.tgz | tail -n +8 | xargs -r rm -f
```

Timestamp each backup for ordering and traceability. The rotation pipeline lists backups newest-first (`ls -lt`), skips the 7 most recent (`tail -n +8`), and removes the rest — `xargs -r` avoids running `rm` with no args. This bounds disk usage so backups don't silently fill the volume (a frequent self-inflicted outage).

Untested backups aren't backups: Add a periodic restore test. Plenty of teams discover their backups were corrupt or empty only during a real incident.

 **Start here (official docs)**

- [Python documentation](#)

Q67 How do you handle secrets in a script without leaking them in logs or process list?

Automation / API

 Spoken answer · ~85 s

Secrets leak through several channels, so close each. Command-line arguments are visible to every user in the process list via `ps` and in `/proc`, and often end up in shell history, so never pass passwords as arguments; use environment variables, `stdin`, or files with strict permissions. Environment variables are safer than arguments but still readable in `/proc` by the same user and `root`, inherited by child processes, and often dumped in crash reports and debug output, so scope them narrowly. Better: read secrets from a mounted file with mode 600 or from a secrets manager at runtime, and keep them in memory only for as long as needed. Logging: never print secrets; disable shell tracing, set `-x`, around sensitive lines, since it echoes expanded values; redact known patterns and tokens in log output with filters; avoid logging full request headers or URLs containing tokens; and be careful with exception messages and debug dumps. In CI, use masked variables, though masking is best effort and shouldn't be relied on, and avoid echoing them. Pass secrets to subprocesses through `stdin` or a specific environment, not `argv`. Prefer short-lived credentials and workload identity so there's little to leak. Prevent commits with secret scanners and pre-commit hooks.

Likely follow-ups

- Why are command-line arguments visible to other users?
- Why doesn't masking in CI fully protect secrets?

```
# BAD: visible in ps/history
```

```
curl -H "Authorization: Bearer $TOKEN" ... # token in ps args? no-env is ok
```

```
mysql -u root -p"$PASS" # ← password visible in ps!
```

```
# GOOD: read from file / stdin / env
```

```
mysql --defaults-extra-file=/run/secrets/my.cnf
```

```
read -rs TOKEN < /run/secrets/token
```

Secrets passed as command-line arguments appear in `ps`, shell history, and process listings — any user on the box can read them. Pass secrets via environment variables, files with restricted permissions (0600), or stdin instead. Also avoid `set -x` while handling secrets (it echoes them to logs), and scrub secrets from error output.

Two leak vectors people forget: command-line args (visible in `ps`) and `set -x` trace output. Both expose secrets to anyone with shell/log access.

 **Start here (official docs)**

- [Python documentation](#)

Q68 Write a Python health-aggregator that checks N endpoints concurrently. Automation / API

 **Spoken answer · ~89 s**

A health aggregator checks many endpoints in parallel and summarises the results. Because the work is I/O-bound, use concurrency: either a `ThreadPoolExecutor` with a bounded number of workers, or `asyncio` with an HTTP client like `aiohttp` or `httpx`. With a thread pool, submit a check function per endpoint, then collect results with `as_completed`. Each check does an HTTP GET with a strict timeout, measures latency with a monotonic clock, treats non-2xx status codes, timeouts, and connection errors as failures, and returns a small result object with name, status, latency, and error message. One slow or hung endpoint must not block others, so apply per-check timeouts and an overall deadline. Bound the concurrency to avoid exhausting file descriptors or hammering the targets. After collection, aggregate: count healthy and unhealthy, list failures, compute latency percentiles, and decide an overall status. Exit non-zero if any critical endpoint failed, with a distinction between critical and optional ones, and optionally output JSON for machine consumption and a readable table for humans. Add retries for transient failure to cut false alarms, or require consecutive failures. Design points: reuse connections through a session, verify TLS certificates and optionally check expiry, keep the endpoint list in configuration, and export results as metrics. Make it cron or monitoring friendly, with logging and alerting.

Likely follow-ups

- Why is concurrency appropriate for this task?
- Why bound the number of workers?

```
from concurrent.futures import ThreadPoolExecutor, as_completed
```

```
def check(url):
```

```
try: return url, requests.get(url, timeout=3).status_code
```

```
except Exception as e: return url, f"ERR {e}"
```

```
with ThreadPoolExecutor(max_workers=20) as ex:
```

```
futs=[ex.submit(check,u) for u in urls]
```

```
for f in as_completed(futs): print(*f.result())
```

Health checks are I/O-bound (waiting on the network), so threads give real speedup despite the GIL. `ThreadPoolExecutor` bounds concurrency (don't open 5000 sockets at once), `submit` schedules work, and `as_completed` yields results as they finish. Each check catches its own exceptions so one bad endpoint doesn't crash the whole sweep.

Bound the pool: `max_workers` caps concurrent connections so you don't exhaust file descriptors or overwhelm the targets you're checking.

Official docs

- [Python Documentation](#)

Q69 How do you make a long-running script gracefully handle SIGTERM (e.g. in a container)?

Automation / API

Spoken answer · ~82 s

Containers stop by receiving SIGTERM from the orchestrator, followed by SIGKILL after a grace period, 30 seconds by default in Kubernetes. A script that ignores SIGTERM gets killed mid-work, possibly leaving partial writes or locks. So register a signal handler that sets a shutdown flag, and have the main loop check the flag between units of work, finishing the current item, flushing buffers, releasing locks, deregistering, committing offsets, and then exiting with a clean status. In Python, use `signal.signal` for SIGTERM and SIGINT, with the handler setting a `threading.Event`; keep handlers minimal, since they interrupt arbitrary code. Make blocking waits interruptible: use `event.wait` with a timeout instead of `time.sleep`, and set timeouts on network calls. In Bash, use `trap` on TERM and INT; note that a Bash script waiting on a foreground child won't run its trap until the child returns, so run the child in the background and use `wait`, which is interruptible. A key container gotcha: signals go to PID 1. If the entrypoint is a shell script that launches your program as a child, SIGTERM may not be forwarded; use `exec` to replace the shell with your process, or run an init like `tini`.

Likely follow-ups

- Why does PID 1 matter for signals in containers?
- How do you make a sleeping loop respond to SIGTERM?

```

import signal, sys

stop=False

def handle(sig, frame):

    global stop; stop=True

signal.signal(signal.SIGTERM, handle)

signal.signal(signal.SIGINT, handle)

while not stop:

    do_work_chunk()

cleanup(); sys.exit(0)

```

Kubernetes sends SIGTERM before SIGKILL during pod shutdown. A graceful handler sets a flag, finishes the current unit of work, flushes/commits, and exits cleanly within the termination grace period — avoiding corrupted state, lost messages, or half-done writes. Don't do heavy work *inside* the handler; just signal the main loop to wind down.

Respect the grace period: If cleanup takes longer than `terminationGracePeriodSeconds`, K8s SIGKILLS you. Keep shutdown fast or checkpoint progress.



Official docs

- [Kubernetes: Pod lifecycle & termination](#)
- [Linux man-pages: signal\(7\)](#)

Spoken answer · ~83 s

Diffing desired against actual state is the heart of declarative automation. First, obtain both in a comparable form: the desired state from a source of truth such as files in Git, and the actual state from the system's API, such as listing current resources. Normalise both: drop read-only and server-generated fields like timestamps, UIDs, and status; sort lists whose order doesn't matter; convert to canonical types; and index each by a stable key, such as a name, so comparison is by key. Then compute the plan with set operations on keys: keys only in desired are creates, keys only in actual are deletions or ignores, and keys in both are compared field by field for updates. Present the plan as a readable diff, and support a dry-run mode that stops there; require approval for destructive actions, and only delete resources you own, identified by tags or labels, to avoid removing manually created ones. Apply in a safe order respecting dependencies, make each action idempotent, handle partial failure by being able to rerun, and verify afterwards by re-reading the state and confirming zero diff. Add drift detection by running the diff on a schedule and alerting when actual differs.

Likely follow-ups

- Why only delete resources you own?
- How do you detect configuration drift?

```
desired = load_desired()          # {name: spec}

actual  = fetch_actual()

to_create = desired.keys() - actual.keys()

to_delete = actual.keys() - desired.keys()

to_update = {k for k in desired&actual.keys()

              if desired[k] != actual[k]}

for k in to_create: create(k, desired[k]) # etc.
```

This reconciliation loop is the core of declarative automation (Terraform, K8s controllers): compute the set difference between desired and actual state, then create/update/delete only what differs. Using sets makes the diff $O(n)$ and the intent explicit. Always support a `--dry-run` that prints the plan without applying — so humans can review before infrastructure changes.

Plan then apply: Show the diff first (like `terraform plan`). Surprising deletes are how reconciliation scripts cause outages.

 **Start here (official docs)**

- [Python documentation](#)

Q71 **Write a script to roll-restart pods/services one at a time, verifying health between each.**

Automation / API

 **Spoken answer** · ~88 s

A rolling restart restarts instances one at a time and verifies each is healthy before moving on, so capacity never drops too far and a bad change halts early. For Kubernetes, the best answer is often not a script at all: `kubectl rollout restart` on a Deployment performs a rolling update governed by `maxUnavailable` and `maxSurge`, readiness probes, and pod disruption budgets, and `kubectl rollout status` waits for completion and returns non-zero on failure. For a custom script across VMs or services: get the ordered list of targets; for each one, optionally drain it from the load balancer and wait for connections to finish; restart; then poll a health check with a timeout until it reports ready, requiring several consecutive successes; re-enable it in the load balancer; wait a short settle time; and only then move on. If any step fails or times out, stop immediately, leave the remaining nodes untouched, alert, and optionally roll back the one that failed. Safety features: a limit on how many can be unhealthy at once, a canary first, awareness of quorum for clustered systems like databases, so you restart followers before the leader and never break the majority, a dry-run mode, logging of each step with timestamps, and idempotence so reruns skip already restarted targets.

Likely follow-ups

- Why stop the whole run on the first failure?
- How does quorum affect the restart order?

```
for pod in $(kubectl get pods -l app=api -o name); do  
  
    kubectl delete "$pod"  
  
    kubectl rollout status deploy/api --timeout=120s || exit 1  
  
    sleep 5    # let it settle before the next  
  
done
```

A safe rolling restart takes down one instance at a time and waits for the replacement to become healthy before proceeding — preserving capacity and catching a bad rollout early (the `|| exit 1` stops the loop if health never recovers). This "verify between steps" pattern prevents the disaster of restarting everything at once and discovering too late that the new version is broken.

One at a time + health gate: Restarting all replicas simultaneously turns a bad deploy into a full outage. The gate between steps is what makes it safe.

 **Start here (official docs)**

- [Python documentation](#)

Q72 How do you safely do find-and-replace across many config files?

Automation / API

 **Spoken answer** · ~84 s

Mass edits are risky, so I make them reviewable and reversible. First, find the targets precisely: use `grep` with the exact pattern, or find with name filters, and list the matching files and lines before changing anything; restrict to the right directories and file types, and exclude generated files, binaries, and vendor directories. Do the work inside version control: run from a clean working tree on a branch so the result is a reviewable diff and can be reverted. Use a proper tool: `sed` with the in-place option, with a backup suffix when not under version control; `perl` for multi-line replacements; or for structured files, parse and edit with a YAML or JSON library, since regex on structured data breaks formatting or matches the wrong key. Use word boundaries and anchors so replacing `foo` doesn't alter `foobar`, and escape special characters, being careful with delimiters in paths. Handle filenames safely with `find -print0` and `xargs -0`. Test first with a dry run printing the substitutions or a diff, or try the command on a copy. Apply it, then inspect the git diff, run linters, validators, and tests such as `kubectl apply with dry run` or `terraform validate`, and look for unintended matches.

Likely follow-ups

- Why is regex a poor tool for editing structured YAML?
- Why work on a clean git tree?

```
# preview first – NO in-place edit yet
```

```
grep -rl 'old.host.com' /etc/app/
```

```
# then edit, with a backup suffix
```

```
grep -rl 'old.host.com' /etc/app/ \
```

```
| xargs sed -i.bak 's/old\.host\.com/new.host.com/g'
```

Always preview the affected files (`grep -rl`) before mutating anything. Then `sed -i.bak` edits in place but keeps a `.bak` backup so you can revert. Escape regex metacharacters (the dots in a hostname) to avoid matching unintended text. For anything important, do it in version control so the diff is reviewable and reversible.

Unescaped dots bite: `s/old.host.com/.../` matches `oldXhostYcom` too. Escape literal dots (`\.`) or you'll corrupt unexpected lines.

 **Start here (official docs)**

- [Python documentation](#)

Q73 How do you build a small CLI that wraps kubectl/cloud commands for your team?

Automation / API

 **Spoken answer** · ~88 s

A team CLI wrapper encodes the right way to do common operations, reducing mistakes and the need to remember long commands. Start by identifying the frequent, error-prone workflows, such as switching contexts, viewing logs for a service across pods, restarting a deployment, or opening a shell in a pod. Choose the language: Python with argparse, click, or typer for quick iteration, or Go with cobra for a single static binary that's easy to distribute. Design: subcommands with consistent naming, sensible defaults, and help text with examples; shell completion; clear errors that explain how to fix the issue; both human-friendly and machine-readable output, such as a JSON flag. Implementation: for cloud and Kubernetes, prefer the official client libraries and APIs over parsing the text output of shell commands; if you must call kubectl, run it through subprocess with argument lists, never shell strings, and parse JSON output. Safety: confirm destructive actions, especially in production, requiring the environment name be typed; support dry-run; show the context and cluster about to be affected; and respect RBAC by using the user's own credentials, never embedding shared ones. Provide a version command and a self-update or package distribution via internal package registry or Homebrew tap; add tests, and telemetry or audit logs for sensitive actions.

Likely follow-ups

- Why prefer official client libraries over parsing CLI text?
- How do you make destructive commands safer?

```
import subprocess, click
```

```
@click.group()
```

```
def cli(): pass
```

```
@cli.command()
```

```
@click.argument("svc")
```

```
@click.option("--ns", default="default")
```

```
def logs(svc, ns):
```

```
    subprocess.run(["kubectl", "logs", "-l", f"app={svc}", "-n", ns])
```

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

An internal CLI encodes team conventions and guardrails over raw commands — standard namespaces, label selectors, safety checks — so engineers don't memorize long invocations or make dangerous mistakes. `click` (or `typer`) gives subcommands, options, and help with minimal boilerplate. This is platform engineering in miniature: paving a golden path as a tool.

Guardrails in the wrapper: e.g. refuse to run destructive commands against prod without a confirmation flag — the CLI becomes where you enforce safety.

 **Start here (official docs)**

- [Python documentation](#)

Q74 How do you read configuration from multiple sources with precedence (flags > env > file > default)?

Automation / API

 Spoken answer · ~71 s

The conventional precedence, from highest to lowest, is command-line flags, then environment variables, then configuration files, then built-in defaults. The reasoning is that the more specific and immediate a source, the higher its priority: a flag expresses explicit intent for this run, an environment variable is per environment or container, a file is per installation, and defaults are the fallback. Implementation: define defaults in code; load the config file, if present, and overlay it; overlay environment variables, using a consistent prefix like `MYTOOL_` and a mapping from names to keys, with type conversion; and finally overlay explicitly supplied flags. With `argparse`, a subtlety is distinguishing flags the user passed from defaults, so set flag defaults to `None`, or use a sentinel, and only override when the value is not `None`. Merge nested configuration deeply instead of replacing whole sections, if desired. Validate the final merged configuration with a schema, such as `pydantic settings`, which supports environment variables and files out of the box, giving clear errors for missing or invalid values.

Likely follow-ups

- How do you tell whether a flag was actually passed?
- Why show where each effective value came from?

```
def get(key, cli_val=None):
```

```
    if cli_val is not None: return cli_val          # 1 flags
```

```
    if key.upper() in os.environ:                  # 2 env
```

```
        return os.environ[key.upper()]
```

```
    if key in file_cfg: return file_cfg[key]       # 3 file
```

```
return DEFAULTS[key]
```

```
# 4 default
```

The 12-factor precedence chain: explicit CLI flags override environment variables, which override a config file, which overrides built-in defaults. This gives operators flexibility (override one value at runtime without editing files) while keeping sane defaults. Resolve each setting through the same ordered lookup so behavior is predictable and documented.

Make precedence visible: A `--show-config` that prints the effective values and where each came from saves hours of "why is it using that value?" debugging.

 **Start here (official docs)**

- [Python documentation](#)

Q75 Write a script to wait for a condition (e.g. service ready) with timeout.

Automation / API

 **Spoken answer** · ~80 s

A wait-for-condition helper polls a check at intervals until it succeeds or a timeout expires. The structure: record a deadline using a monotonic clock; loop, calling the check function; if it returns true, return success; otherwise, if the deadline has passed, fail with a clear timeout error that includes the last observed state or error; else sleep before the next try. Choices that matter: the polling interval, short enough for responsiveness and long enough not to hammer the service, often with a mild backoff and jitter; a per-attempt timeout on the check itself, so a hung request can't blow past the overall deadline; treating expected transient errors, like connection refused while the service starts, as not-ready rather than fatal, while surfacing unexpected errors; and requiring several consecutive successes to avoid declaring ready on a flapping service. Return a meaningful exit code in a script, zero on success and non-zero on timeout, and log progress periodically. In Bash, a loop using a SECONDS-based deadline, curl with a fail flag and max time, and sleep works. In Python, a generic wait_until function accepting a callable, timeout, and interval is reusable, or use the tenacity library.

Likely follow-ups

- Why avoid a fixed sleep?
- Why require consecutive successes?

```
deadline=$(( $(date +%s) + 120 )) # 2-min timeout
```

```
until curl -fsS localhost:8080/ready >/dev/null 2>&1; do
```

```
(( $(date +%s) > deadline )) && { echo "timed out"; exit 1; }
```

```
echo "waiting..."; sleep 3
```

```
done
```

```
echo "ready"
```

Polling-with-timeout is everywhere in deploy automation: wait for a service to pass readiness, a DB migration to finish, a resource to provision. Always bound the wait with a deadline so a stuck dependency fails the pipeline instead of hanging forever. Use a fixed poll interval (or backoff) and emit progress so logs show it's alive, not frozen.

Bounded waits only: An unbounded "wait until ready" in CI will hang a build agent indefinitely when something's actually broken. The timeout converts a hang into a clear failure.

Official docs

- [AWS Builders' Library: Timeouts, retries and backoff with jitter](#)

Q76 How do you make automation observable — metrics/logs from your scripts?

Automation / API

Spoken answer · ~87 s

Automation that nobody can see is automation nobody trusts, so scripts should emit the same signals as services. Logs: use structured logging with a run ID, timestamps, levels, and key fields such as target, action, and result, written to stdout so the platform collects them; log what was decided as well as what was done, including skips and dry-run plans. Metrics: for batch jobs that exit quickly, a scrape-based model doesn't fit, so push metrics to a Prometheus Pushgateway, or write them to the node exporter's textfile collector, or send to a StatsD or OpenTelemetry endpoint. Useful metrics: last success timestamp, last run duration, items processed, items failed, and exit status. Alert on staleness of the last success timestamp, which catches jobs that stopped running entirely, and on failure counts. Use heartbeats or dead-man's-switch pings to an external service. Traces: for multi-step workflows, create spans per step, to see where time goes. Events: record changes made, who or what triggered them, and what changed, into an audit log, and annotate dashboards with deployment or maintenance markers. Exit codes should be meaningful so schedulers record failure. Include cardinality discipline in labels. Provide a clear summary at the end of each run, such as 120 updated, 3 skipped, 1 failed.

Likely follow-ups

- How do short-lived jobs expose metrics to Prometheus?
- Why alert on the last success timestamp?

```
# emit a Prometheus-style metric to the pushgateway
```

```
cat <<EOF | curl --data-binary @- \
```

```
http://pushgw:9091/metrics/job/backup
```

```
backup_success{host="$(hostname)"} 1
```

```
backup_duration_seconds $SECONDS
```

```
E0F
```

Cron jobs and automation fail silently unless you instrument them. Emit a success/failure metric and duration to a Pushgateway (for short-lived jobs) or write structured logs your shipper collects — then alert on "backup_success == 0" or "job hasn't reported in 25h". This turns invisible batch failures into actionable alerts instead of discovering a week-old broken backup during an incident.

Alert on absence: The most valuable batch-job alert is "this job didn't run/report when expected" — a job that silently stops running emits no failure at all.



Start here (official docs)

- [Python documentation](#)

Q77 How do you extract specific fields from kubectl output for scripting?

K8s / Cloud

Spoken answer · ~88 s

The reliable way is to request structured output and parse it, instead of scraping the human-readable table. Use the output flag with JSON, such as `kubectl get pods -o json`, and then process it in code or with `jq`. For simple field extraction, `kubectl` supports `JSONPath`: `-o jsonpath` with an expression, for example listing pod names, or getting one field like a deployment's image; ranges iterate over items. The custom-columns output creates tidy tables for humans or scripts, with chosen columns and `JSONPath` expressions for each, and the `no-headers` option helps parsing. For names only, `-o name` gives resource slash name strings, useful in loops. For anything beyond trivial extraction, pipe JSON into `jq`, which supports filtering, mapping, grouping, and conditionals. Inside Python, parse JSON from subprocess, or better use the Kubernetes client library directly. Good habits: don't parse the default columns with `awk`, since their format can change between versions and truncates or wraps values; use selectors, label and field selectors, to narrow results server-side, which reduces data; handle empty results and missing fields gracefully, with the `jq` alternative operator or defaults; quote `JSONPath` expressions properly to avoid shell interpretation; and specify the namespace and context explicitly in scripts, so they never act on whatever your current context happens to be.

Likely follow-ups

- Why avoid `awk` on default `kubectl` output?
- What is the advantage of server-side selectors?

```
# jsonpath – built into kubectl
```

```
kubectl get pods -o jsonpath='{.items[*].metadata.name}'
```

```
# custom columns
```

```
kubectl get pods -o custom-columns=NAME:.metadata.name,NODE:.spec.nodeName
```

```
# or pipe JSON to jq
```

```
kubectl get pods -o json | jq -r '.items[].metadata.name'
```

For reliable scripting, get structured output (`-o json/-o jsonpath`) and parse it — never `grep/awk` the human-readable table, whose columns and spacing change between versions. `jsonpath` and `custom-columns` are built in; `jq` is more powerful for filtering and transforming. This is the difference between automation that survives a `kubectl` upgrade and automation that silently breaks.

Never parse the default table: Column order, widths, and added columns vary by version and flags. Structured output is the contract; the table is not.

 **Start here (official docs)**

- [Python documentation](#)

Q78 Write a jq filter to list pods not in Running state.

K8s / Cloud

 **Spoken answer** · ~82 s

With `kubectl get pods -o json` piped to `jq`, I filter items whose status phase isn't Running. The filter selects from the items array those where the phase is not equal to Running and prints the namespace and name. For example: `jq -r` with a pipeline that iterates the items, applies `select` on `status.phase` not equal to Running, and then formats an output string joining `metadata.namespace` and `metadata.name`, and optionally the phase. Add `-A`, or all namespaces, to `kubectl` to cover the whole cluster. Alternatively, `kubectl` can do this server-side with a field selector: `--field-selector status.phase!=Running`, which is more efficient. Caveats worth mentioning: a pod can be in the Running phase while containers are failing, such as `CrashLoopBackOff` with the pod still reported as Running at the phase level, or not ready, so for a real health view also check container statuses and the Ready condition; completed Jobs show as Succeeded, which you probably want to exclude, so filter for phases other than Running and Succeeded; and Pending pods deserve their own look at events to see why they're unschedulable. In `jq`, use the alternative operator to cope with missing fields and `map` or `select` combinators.

Likely follow-ups

- Why isn't the Running phase proof the pod is healthy?
- How does a field selector differ from `jq` filtering?

```
kubectl get pods -A -o json | jq -r '
```

```
.items[]
```

```
| select(.status.phase != "Running")
```

```
| "\(.metadata.namespace)/\(.metadata.name) \(.status.phase)"'
```

jq is the standard tool for slicing JSON on the command line. `.items[]` iterates the list, `select(...)` filters, and `"\(.field)"` string-interpolates for clean output. `-r` emits raw strings (no quotes). This one-liner finds problem pods across all namespaces — a common triage starting point during an incident.

Phase isn't the whole story: A pod can be Running but not Ready (failing readiness). For real health, also check `.status.conditions[]` where `type=="Ready"`.

Official docs

- [jq manual](#)

Q79 How do you script against the Kubernetes API directly in Python?

K8s / Cloud

Spoken answer · ~86 s

The official Kubernetes Python client lets you call the API directly, avoiding shelling out to `kubectl`. Load configuration first: `config.load_incluster_config` when running inside a pod, using the service account token, or `config.load_kube_config` for local development, falling back between the two. Create API objects, such as `CoreV1Api` for pods, services, and nodes, and `AppsV1Api` for deployments. Then call methods like `list_namespaced_pod`, `read_namespaced_deployment`, or `patch_namespaced_deployment`. Results are typed model objects, not text, so you access fields as attributes, avoiding parsing fragility. For efficiency, use label and field selectors, pagination with limit and continue tokens for large lists, and the watch API via the watch module to stream events, with handling for expired resource versions and reconnects, rather than polling. For changes, prefer patch operations, including strategic merge or server-side apply, over replace, so you don't clobber concurrent changes, and handle conflicts using resource versions. Handle errors via `ApiException`, checking the status for 404, 409 conflict, and 429 throttling, with retries. Set request timeouts. Security: run with a dedicated service account and minimal RBAC, granting only the verbs and resources the script needs. For building controllers or operators, frameworks like `kopf` or `controller-runtime` provide informers and work queues. For simple needs, `kubectl` with JSON output remains a pragmatic choice.

Likely follow-ups

- Why prefer patch over replace?
- Why use the watch API instead of polling?

```
from kubernetes import client, config
```

```
config.load_incluster_config() # or load_kube_config() locally
```

```
v1 = client.CoreV1Api()
```

```
for p in v1.list_pod_for_all_namespaces().items:

    if p.status.phase != "Running":

        print(p.metadata.namespace, p.metadata.name, p.status.phase)
```

The official Python client talks to the API server directly — better than shelling out to `kubectl` for anything non-trivial (typed objects, watch streams, error handling). `load_incluster_config()` uses the pod's service-account token when running inside the cluster; `load_kube_config()` uses your local kubeconfig. This is how you build controllers, operators, and custom automation.

RBAC scope it: Give the service account only the verbs/resources it needs (least privilege). A script that only reads pods shouldn't have delete on secrets.

Official docs

- [Kubernetes Python client](#)
- [Python Documentation](#)

Q80 How do you safely run a command across many EC2 instances / VMs?

K8s / Cloud

Spoken answer · ~86 s

Running commands across many instances demands controlled scope, safety, and visibility. Prefer managed mechanisms over SSH loops: AWS Systems Manager Run Command or Session Manager, Azure Run Command, or GCP OS Config, which use IAM instead of distributed SSH keys, need no open inbound ports, log every execution, and report per-instance results; or configuration management tools such as Ansible. Targeting: select instances by tags or inventory groups, never by a guessed IP list; show the resolved target list and require confirmation, with a dry run. Rollout: do not hit everything at once; start with a canary instance, then proceed in batches, controlling concurrency and maximum error percentage, so a bad command stops after limited damage. Commands should be idempotent and non-interactive, with timeouts per instance. Capture exit status, stdout, and stderr per host, and present a summary of succeeded, failed, and timed out, with nonzero overall status if any failed. Security: least privilege on who can run commands and which commands, audit logs, avoiding secrets in command lines, and using `sudo` narrowly. If using SSH with `pssh` or Ansible, use short-lived certificates or bastion access, host key checking, and parallelism limits. Treat destructive actions with extra care: confirm the environment, require a ticket or approval in production.

Likely follow-ups

- Why is Systems Manager safer than SSH loops?
- Why start with a canary instance?

```
# Prefer agent-based (SSM) over raw SSH at scale
```

```
aws ssm send-command --document-name AWS-RunShellScript \
```

```
--targets Key=tag:env,Values=prod \
```

```
--parameters 'commands=["systemctl restart app"]'
```

```
# raw SSH fan-out (small scale), bounded parallelism:
```

```
printf '%s\n' "${hosts[@]}" | xargs -P5 -I{} ssh {} 'uptime'
```

At scale, prefer a managed mechanism (AWS SSM, Ansible, Salt) over hand-rolled SSH loops — they handle auth, auditing, bounded concurrency, and partial failures. If you do fan out over SSH, bound parallelism (`xargs -P`), set `ConnectTimeout`, and capture per-host results so you know exactly which hosts failed rather than a vague aggregate error.

Blast radius: A "restart everything" fan-out with no canary or batching can take down a whole fleet at once. Batch it, verify health between batches.

 **Start here (official docs)**

- [Python documentation](#)

Q81 Write a script to clean up untagged/old cloud resources (cost hygiene).

K8s / Cloud

 **Spoken answer · ~84 s**

A cleanup script for cost hygiene finds resources that are unused or abandoned, but because deletion is destructive it must be cautious. Discovery: list candidates through the cloud API with pagination, such as unattached volumes, old snapshots and images, unassociated elastic IPs, idle load balancers with no targets, stopped instances with attached disks, and untagged resources. Define staleness explicitly: older than N days by creation time, no attachments, and for volumes, no recent I/O in monitoring metrics. Protect what matters: honour a keep or do-not-delete tag, exclude production and shared accounts unless explicitly included, and require resources to carry an owner tag before auto-deleting. Safety: default to dry-run, printing a report of what would be deleted with name, age, size, and estimated monthly cost; require an explicit flag to execute; take snapshots or move to a quarantine stage first; and use a two-step lifecycle, where a first run tags resources with a pending-deletion date and notifies the owner, and a later run deletes those whose grace period has expired without objection. Log every action for audit, handle rate limits and partial failures, and keep the script idempotent. Run it on a schedule with least-privilege credentials limited to the delete calls it needs.

Likely follow-ups

- Why use a tag-then-delete grace period?
- Which resources are the most common cost leaks?

```

import boto3, datetime as dt

ec2 = boto3.client("ec2")

cutoff = dt.datetime.now(dt.timezone.utc) - dt.timedelta(days=30)

for v in ec2.describe_volumes(

    Filters=[{"Name": "status", "Values": ["available"]}])["Volumes"]:

    if v["CreateTime"] < cutoff:

        print("would delete", v["VolumeId"]) # dry-run first!

```

Orphaned resources (unattached EBS volumes, old snapshots, idle load balancers) quietly burn money. A cleanup script finds resources matching "wasteful" criteria (unattached + older than N days) and removes them. **Always dry-run first** — print what would be deleted, have a human review, and ideally require an explicit tag or grace period before any destructive action.

Destructive automation needs guardrails: dry-run by default, an allow/deny tag (e.g. `keep=true`), and a confirmation flag. Auto-deleting cloud resources has ended careers.

Start here (official docs)

- [Python documentation](#)

 Spoken answer · ~89 s

The simplest tool is `kubectl logs` with a label selector and the `follow` flag, plus the `prefix` option and a limit on concurrent requests, so lines from all matching pods stream together with each line labelled by pod name. By default `kubectl` limits parallel log streams to five, adjustable with a `max-log-requests` flag. Add the `since` flag to bound history, the `timestamps` flag for ordering, and the `all-containers` flag for sidecars. Pods are ephemeral, so a plain follow stops when a pod restarts or gets replaced during a rollout; scripts must handle reconnects, either by looping or by using a tool built for it such as `stern`, which tails multiple pods and containers by regex, adds colours, and picks up new pods automatically. For a custom script, the Kubernetes client lets you watch pods, and for each running pod start a thread or task that streams `read_namespaced_pod_log` with `follow` set, merging lines into a queue with the pod name attached. Handle the stream ending, backoff and resume with `since-time` to avoid duplicates, and clean up threads on exit. Remember logs on the node are rotated and lost with the pod, so for anything beyond live debugging, query the central logging system instead. Filter noise with `grep` using line buffering, and avoid printing secrets.

Likely follow-ups

- Why does `kubectl logs -f` stop during a rollout?
- What does `stern` add?

```
# all pods of a deployment, prefixed, following
```

```
kubectl logs -f -l app=api --all-containers --prefix --tail=50
```

```
# or use stern for multi-pod tailing
```

```
stern api --since 10m
```

Modern `kubectl logs` can follow (`-f`) a label selector across pods with `--prefix` to show which pod each line came from. For richer multi-pod tailing across deployments, `stern` is the de-facto tool (color-coded by pod, regex filtering). During an incident, aggregated live logs across replicas are essential to spot which instance is misbehaving.

Bound it: `--tail` and `--since` prevent pulling gigabytes of history. For real analysis, query your log backend (Loki) rather than streaming from the cluster.

 Start here (official docs)

- [Python documentation](#)

Q83 How do you compute resource usage / find the noisy-neighbor pod by script?

K8s / Cloud

 Spoken answer · ~85 s

To find a noisy-neighbour pod, look at resource usage per pod relative to requests and limits, and per node. Quick view: `kubectl top pods` with all namespaces and sorting by CPU or memory, and `kubectl top nodes` to see which node is hot; this uses metrics-server, giving current usage only. For analysis over time, query Prometheus, using container CPU usage rate and memory working set metrics from cAdvisor, grouped by pod and node, with `topk` to rank the heaviest. Compare usage to requests: pods using far more than they requested are the problem, since the scheduler packed the node assuming the request. CPU contention shows up as throttling, so check the container CPU throttled periods ratio for the victims; a high throttle rate means limits are too low, while node-level CPU pressure suggests neighbours are consuming. For memory, look for OOM kills in events and the last state of containers, and node memory pressure. Other shared resources matter: disk I/O, network bandwidth, and ephemeral storage, which need node exporter and pressure stall information metrics. A script can combine `kubectl get pods` with JSON output for requests, join with `top` output, compute the ratio of usage to request, and print the worst offenders per node.

Likely follow-ups

- Why does exceeding requests create noisy neighbours?
- How do you detect CPU throttling?

```
# top pods by CPU across the cluster (needs metrics-server)
```

```
kubectl top pods -A --sort-by=cpu | head
```

```
# memory:
```

```
kubectl top pods -A --sort-by=memory | head
```

```
# script: flag pods near their limits via /metrics or PromQL
```

`kubectl top` (backed by metrics-server) gives live CPU/memory per pod; sorting surfaces the heaviest consumers — the first step in finding a noisy neighbor or a leaking pod. For real analysis, query Prometheus: compare usage against requests/limits to find pods being throttled (CPU) or at OOM risk (memory near limit). Scripted, this feeds rightsizing and capacity work.

Usage vs limits: High CPU isn't bad if under limit; the signal is sustained throttling or memory creeping toward the limit (OOMKill incoming).

 **Start here (official docs)**

- [Python documentation](#)

 Spoken answer · ~86 s

A deploy script should block until the rollout is verified, and fail loudly. The core is kubectl rollout status for the deployment with a timeout flag, which watches until all replicas are updated and available, exiting non-zero if the timeout passes or the deployment's progress deadline is exceeded. In the script, apply the manifest or set the image, run rollout status, and check the exit code; on failure, run kubectl rollout undo to roll back, then fail the pipeline with diagnostics. Complement it with kubectl wait for specific conditions, such as wait for the condition available with a timeout, or for pods with a ready condition by label selector. Verify more than Kubernetes' own view: run a smoke test against the service endpoint, check the new version is actually serving by querying a version endpoint, and look at error rate or health metrics for a few minutes before declaring success. Set the right foundations: readiness probes that reflect real readiness, a progressDeadlineSeconds on the deployment, and maxUnavailable and maxSurge settings. Print useful diagnostics on failure: describe the deployment, recent events, and logs from failing pods. Specify the namespace and context explicitly, use set -e or explicit checks, and make the script idempotent, so rerunning is safe.

Likely follow-ups

- What does kubectl rollout status return on failure?
- Why verify beyond Kubernetes readiness?

```
kubectl apply -f deploy.yaml
```

```
if ! kubectl rollout status deploy/api --timeout=180s; then
```

```
  echo "rollout failed – rolling back" >&2
```

```
  kubectl rollout undo deploy/api
```

```
  exit 1
```

```
fi
```

kubectl rollout status blocks until the new ReplicaSet is fully available or the timeout hits, returning non-zero on failure — so your deploy script can detect a bad rollout and automatically rollout undo to the previous revision. This automated rollback-on-failure is what makes scripted deploys safe: a broken image self-heals instead of sitting half-deployed.

Mitigate before diagnose: Auto-rollback restores service first; you investigate the bad build afterward. That's the SRE deploy reflex encoded in a script.

 Official docs

- [Kubernetes: Deployments](#)

Spoken answer · ~88 s

Generating manifests programmatically avoids copy-paste drift across environments. Options by complexity. Templating with Jinja2 or Helm: a template with values per environment, rendered in a script. Kustomize: a base plus overlays that patch differences, without templating, built in to kubectl. Programmatic generation in code: build the objects as Python dictionaries or typed objects from the Kubernetes client models, or use CDK8s, Pulumi, or Jsonnet and CUE to generate manifests with loops, functions, and type checking, which helps for many similar services. Whichever method, treat generated YAML as build output. Validation in layers. Syntax and schema: kubeconform or kubeval validate against the Kubernetes OpenAPI schemas for the target version, including custom resource schemas. Policy: conftest with OPA, or Kyverno CLI, check rules such as resource limits, no privileged containers, required labels, and approved registries. Server-side: kubectl apply with --dry-run=server runs admission and validation against the real cluster without persisting changes, and kubectl diff shows the actual change. Lint with tools like kube-linter or kube-score for best-practice issues. Put these in CI, so invalid or non-compliant manifests fail before merge. Keep generation deterministic so diffs are reviewable, pin versions of generators and API versions, avoid embedding secrets, and commit the rendered output or use GitOps tools that render at deploy time.

Likely follow-ups

- What does kubectl apply --dry-run=server check that kubeconform doesn't?
- Why keep generation deterministic?

```
# render then validate before applying
```

```
python gen_manifests.py > out.yaml
```

```
kubectl apply --dry-run=server -f out.yaml # API-side validation
```

```
kubeconform -strict out.yaml # schema validation
```

```
# in Python, build dicts and yaml.safe_dump them
```

Generate manifests as Python dicts and `yaml.safe_dump` them, or template with Jinja/Helm. Critically, validate before applying: `--dry-run=server` runs real API admission (catches policy/webhook rejections), and `kubeconform/kubeval` checks schema offline in CI. Catching a malformed manifest in CI beats discovering it when the apply half-succeeds in prod.

Two validation layers: offline schema check (fast, in CI) plus server dry-run (catches admission webhooks/policies the schema can't). Use both.

Q86 How do you handle pagination and throttling when listing many cloud resources?

K8s / Cloud

 Spoken answer · ~86 s

Cloud APIs return results in pages and throttle callers, so listing many resources needs both mechanisms handled. Pagination: use the SDK's built-in paginators, such as boto3 paginators, which follow continuation tokens automatically and can be used as generators, so you process page by page without loading everything. Request the maximum page size, and use server-side filters to reduce data. If writing manual loops, pass the next token until none remains and guard against loops. Throttling: APIs return errors like 429 or service-specific ones such as `RequestLimitExceeded` or `ThrottlingException`. Handle with retries using exponential backoff with jitter, honour `Retry-After` headers, and use the SDK's retry configuration, such as boto3's adaptive or standard retry mode, which includes client-side rate limiting. Reduce calls: cache results, avoid per-resource follow-up calls (the N+1 pattern) by using bulk describe operations or batch APIs, and prefer event-driven or inventory services such as AWS Config, Azure Resource Graph, or Cloud Asset Inventory for large-scale queries, which are far more efficient than crawling APIs. Scale out carefully: parallelise across regions and accounts, but bound concurrency per API since limits are per account and per region. Be robust to eventual consistency, where newly created resources may not appear immediately, and to resources changing during the listing.

Likely follow-ups

- What is the N+1 problem with cloud APIs?
- Why use a resource graph service instead of crawling APIs?

```
import boto3
```

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

```
# paginators handle NextToken + retries automatically
```

```
for page in ec2.get_paginator("describe_instances").paginate():
```

```
    for r in page["Reservations"]:
```

```
        for i in r["Instances"]: handle(i)
```

Cloud list APIs paginate and throttle. SDK paginators handle the `NextToken` loop for you and integrate with the SDK's built-in exponential-backoff retry on throttling errors — far more robust than a hand-rolled loop. For very large accounts, process each page as it arrives (don't accumulate everything) and respect rate limits to

avoid RequestLimitExceeded.

Don't reinvent retries: The SDK already does exponential backoff on throttling. Hand-rolled list loops usually forget this and fail on large accounts.

Official docs

- [Microsoft: Cloud design pattern – Throttling](#)
- [REST API pagination \(Google AIP-158\)](#)

Q87 Write a script that snapshots then validates a database backup.

K8s / Cloud

Spoken answer · ~85 s

The script snapshots the database, then proves the snapshot is usable, because an untested backup isn't a backup. Snapshot step: use the database-native consistent method, such as `pg_dump` or `pg_basebackup` for PostgreSQL, `mysqldump` with single-transaction or a physical tool for MySQL, or cloud snapshot APIs for managed databases and volumes, and for volume snapshots, make sure the data is consistent by flushing or using a snapshot-capable approach. Name the artefact with a timestamp, record its size, compute a checksum, encrypt it, and upload to storage in another region or account, ideally with immutability. Validation, in increasing strength. Basic checks: the command exit status, non-zero size, within an expected range compared with previous backups, a readable archive. Integrity: verify the checksum after upload. Structural: for dumps, run a listing of contents, such as `pg_restore list`. Best: an actual restore test, spinning up a temporary database from the backup, running sanity queries like row counts for key tables and the latest record timestamp, perhaps application-level smoke checks, then tearing it down. Measure the restore duration, which informs your RTO. Alert on any failure and on missing backups, using a dead-man's-switch. Automate on a schedule, keep logs and a history, rotate according to retention policy, and protect credentials.

Likely follow-ups

- Why is a restore test the strongest validation?
- What does restore duration tell you?

```
set -Eeuo pipefail
```

```
snap="bk-$(date +%F-%H%M)"
```

```
pg_dump -Fc "$DATABASE_URL" -f "/backups/$snap.dump"
```

```
# validate: restore to a throwaway DB and run a sanity query
```

```
createdb verify_$snap
```

```
pg_restore -d verify_$snap "/backups/$snap.dump"
```

```
psql -d verify_$snap -c 'SELECT count(*) FROM users' | grep -q '[1-9]'
```

```
dropdb verify_$snap
```

A backup you haven't restored is a hope, not a backup. This script dumps, then actually restores into a throwaway database and runs a sanity query to confirm the data is present and the dump isn't corrupt — then cleans up. Automating restore-verification is what separates teams that recover in an incident from teams that discover their backups were empty.

Test restores on a schedule: Corruption, partial dumps, and permission changes silently break backups. Only a real restore proves they work.

 **Start here (official docs)**

- [Python documentation](#)

Q88 How do you write a script that's portable across Linux distros / shells?

K8s / Cloud

 **Spoken answer · ~85 s**

Portability means the script works across distributions, and sometimes shells, without surprises. Shell choice: if you target only Linux with Bash available, declare the shebang as `env bash` and use Bash features freely; if you need to run on minimal systems like Alpine, which uses BusyBox `ash`, or Debian's `dash` as `sh`, restrict yourself to POSIX `sh`: no arrays, no double brackets, no process substitution, and no local arrays; test with `dash` and `shellcheck` set to the target shell. Tools: GNU and BSD or BusyBox versions of `sed`, `grep`, `date`, `readlink`, and `stat` differ in options; avoid GNU-only flags, or detect and adapt; use `printf` instead of `echo -e`; test with the real target images. Package managers differ, so abstract installation behind a function that detects `apt`, `dnf`, `yum`, or `apk` from the OS release file, and avoid parsing the output of tools meant for humans. Paths: don't assume binary locations; use `command -v` to check for dependencies at the start and fail with a clear message. Use `/etc/os-release` to detect distribution and version. Systemd versus other init systems, and differences in default firewall and logging tools, matter for service management. Handle line endings and locale by setting `LC_ALL` equal to `C` for predictable parsing.

Likely follow-ups

- Why use `command -v` to check dependencies?
- What breaks when moving from Bash to `dash`?

```
#!/bin/sh          # POSIX sh, not bash, for max portability
```

```
# avoid bashisms: [[ ]], arrays, ${x^^}, <<<
```

```
command -v jq >/dev/null 2>&1 || { echo "need jq" >&2; exit 1; }
```

```
# detect package manager:
```

```
if command -v apt-get; then PM="apt-get"; elif command -v dnf; then PM="dnf"; fi
```

For scripts that must run anywhere (containers, Alpine, BSD, minimal images), target POSIX sh and avoid bash-only features. Check for required tools with `command -v` and fail clearly if missing. Detect distro differences (package manager, paths) rather than assuming. Alpine uses busybox with leaner utilities, so flags you rely on in GNU coreutils may not exist.

Test on the real target: A script that works on your Ubuntu laptop may break in an Alpine container (busybox sed/grep differ). Run it where it'll actually run.

 **Start here (official docs)**

- [Python documentation](#)

Q89 Why is Go popular for platform/SRE tooling over Python?

Go

 **Spoken answer · ~89 s**

Go has become the language of cloud infrastructure, with Docker, Kubernetes, Terraform, and Prometheus all written in it, for practical reasons. It compiles to a single static binary with no runtime or dependencies, so distributing a tool means copying one file, which is far easier than managing Python versions and virtual environments across machines and containers. It cross-compile trivially for different operating systems and architectures. Startup is fast, important for CLIs and short-lived containers, and the memory footprint is small, useful for sidecars and agents. Concurrency is built in, with goroutines and channels making it simple to write programs that handle thousands of concurrent connections or checks. It is statically typed, catching a whole class of errors at compile time, with a fast compiler and a simple language that is easy to read and maintain across a team. The standard library is strong for networking, HTTP, JSON, and testing. The ecosystem includes official clients and libraries for Kubernetes and cloud APIs, so building operators and controllers is natural. Python still wins for quick scripts, data wrangling, and glue where its libraries and rapid iteration matter, and for one-off automation. A good answer is choose Go for long-lived services, agents, operators, and distributed CLIs, where reliability and deployment simplicity matter, and Python for quick automation.

Likely follow-ups

- Why does a static binary simplify distribution?
- When would you still choose Python?

+ single static binary – no runtime/deps to ship

+ fast, compiled, low memory

+ first-class concurrency (goroutines, channels)

+ the ecosystem language: K8s, Terraform, Prometheus, Docker

– more verbose, slower to prototype than Python

Go compiles to a single static binary you can drop on any host or into a scratch container — no interpreter, no dependency hell. It's fast and memory-light, has built-in concurrency, and is the language the entire cloud-native stack is written in (Kubernetes, Terraform, Prometheus, etcd, Docker). For distributed tools and anything you ship as a binary, Go wins; for quick scripts and glue, Python is faster to write.

Rule of thumb: Quick automation/glue → Python/Bash. A daemon, controller, or CLI you distribute as a binary → Go.

Official docs

- [Python Documentation](#)
- [The Go Programming Language documentation](#)

Q90 Explain goroutines and channels with a worker-pool example.

Go

Spoken answer · ~87 s

A goroutine is a lightweight function execution managed by the Go runtime, started with the `go` keyword. They start with a few kilobytes of stack and are multiplexed onto operating system threads, so you can run hundreds of thousands. A channel is a typed conduit for communicating between goroutines; sends and receives block until the other side is ready for unbuffered channels, giving synchronisation, and buffered channels hold a limited number of values. The Go proverb is to share memory by communicating. A worker pool: create a jobs channel and a results channel; start `N` worker goroutines, each running a loop that receives jobs from the jobs channel, processes them, and sends results to the results channel; a `sync.WaitGroup` tracks the workers, and when the producer finishes sending, it closes the jobs channel, which ends the workers' range loops; a separate goroutine waits for the group and then closes the results channel, so the consumer's range over results terminates. The pool bounds concurrency, protecting downstream systems. Practical additions: pass a context for cancellation and timeouts, handle errors by sending them with results, avoid goroutine leaks by ensuring every goroutine can exit, such as by selecting on the context's Done channel, and only close channels from the sender side.

Likely follow-ups

- Who should close a channel?
- How does closing the jobs channel stop the workers?

```
jobs := make(chan string, 100)
```

```
results := make(chan int, 100)
```

```

for w := 0; w < 5; w++ {           // 5 workers

    go func() {

        for j := range jobs { results <- process(j) }

    }()

}

for _, j := range tasks { jobs <- j }

close(jobs)

```

A goroutine is a lightweight thread (thousands are cheap); `go func(){}` launches one. Channels pass data between them safely without locks. The worker-pool pattern bounds concurrency: a fixed set of goroutines pulls jobs off a channel, so you process work in parallel without spawning unbounded goroutines. This is the idiomatic way to fan out network/IO work in Go tooling.

"Share memory by communicating": Channels let goroutines coordinate without manual mutexes — fewer race conditions than thread+lock models.

Official docs

- [Effective Go](#)

 Spoken answer · ~88 s

Go treats errors as ordinary values: functions that can fail return an error as the last return value, and callers check it explicitly, typically with an `if err != nil` block. There's no try-catch, and panics exist but are reserved for truly unrecoverable programmer errors. The reasoning behind the design: explicit control flow, where you can see every place a function can fail by reading the code, with no hidden jumps from exceptions propagating invisibly up the stack; forcing developers to consider failure at each call site; simplicity and predictability; and low overhead. The cost is verbosity, with repeated error checks. Good practice: add context as the error propagates by wrapping with `fmt.Errorf` using the `percent-w` verb, so the chain of what failed is preserved; inspect wrapped errors with `errors.Is` for comparing against sentinel errors, and `errors.As` to extract a typed error; define custom error types or sentinel values where callers need to branch; don't ignore errors by assigning to the blank identifier; handle errors once, either log or return, not both; and use `defer` for cleanup so it runs on all return paths. `Panic` and `recover` are used sparingly, for example to protect a server from crashing on a single request, but not for normal control flow.

Likely follow-ups

- What does wrapping with `%w` enable?
- When is panic appropriate?

```
f, err := os.Open(path)

if err != nil {

    return fmt.Errorf("opening %s: %w", path, err) // wrap

}

defer f.Close()
```

Go returns errors as explicit values instead of throwing exceptions, forcing you to handle (or deliberately ignore) each one at the call site — verbose, but it makes failure paths visible and hard to silently skip. Wrapping with `%w` preserves the error chain for `errors.Is`/`errors.As` inspection. `panic` exists but is reserved for truly unrecoverable programmer errors, not normal failures.

Add context when wrapping: `fmt.Errorf("doing X: %w", err)` builds a breadcrumb trail so the final error tells you the whole failure path, not just the leaf cause.

 Official docs

- [Effective Go](#)

 Spoken answer · ~86 s

The defer statement schedules a function call to run when the surrounding function returns, whether normally, through an early return, or because of a panic. Deferred calls run in last-in, first-out order, and their arguments are evaluated at the time of the defer statement, not when they execute. It keeps cleanup next to the acquisition, which makes code harder to get wrong: open a file and immediately defer its close; acquire a mutex and defer its unlock; start a transaction and defer a rollback that is a no-op after commit; create a temp directory and defer its removal; start a timer or trace span and defer ending it; and cancel a context with defer cancel. For SREs, it's the Go equivalent of trap and finally, ensuring locks are released, connections closed, and temporary state cleaned up on every path, including errors. Gotchas: defer in a loop accumulates until the function returns, so loops opening many files should wrap the body in a function; deferred closes can return errors that get ignored, so for writes check the close error, using named return values to propagate it; deferred function arguments are evaluated immediately, which surprises people logging values that change later; and defer has small overhead, which rarely matters.

Likely follow-ups

- Why is defer inside a loop a problem?
- In what order do deferred calls run?

```
resp, err := http.Get(url)
```

```
if err != nil { return err }
```

```
defer resp.Body.Close() // runs when function returns
```

```
mu.Lock(); defer mu.Unlock()
```

defer schedules a call to run when the surrounding function returns, regardless of how it returns (normal or error). It keeps cleanup right next to acquisition — open then defer Close(), lock then defer Unlock() — so you never forget to release resources. Deferred calls run in LIFO order. It's Go's equivalent of a context manager / finally.

Defer in loops: defer fires at function return, not loop-iteration end — deferring inside a long loop leaks resources until the function exits. Close explicitly per-iteration or wrap the body in a function.

 Official docs

- [Effective Go](#)

Spoken answer · ~86 s

For a minimal image, use a multi-stage Docker build. The first stage uses the full Go image to compile: copy the module files, download dependencies first so Docker caches that layer, then copy the source and build with CGO disabled so the binary is statically linked, and with flags to strip debug info, such as `ldflags minus s minus w`, and optionally `trimpath` for reproducible builds. Set `GOOS` and `GOARCH` for the target platform. The second stage starts from a tiny base: `scratch`, which is empty, works for fully static binaries if you also copy CA certificates and time zone data when needed; `distroless static` images include certificates and a non-root user and are a good default; `Alpine` is small but adds a shell and `musl libc`. Copy the compiled binary from the build stage, run as a non-root user, set the entrypoint to the binary using `exec` form, and expose only what's needed. Result: an image of roughly 5 to 20 megabytes, with a minimal attack surface because there's no shell or package manager, and faster pulls and starts. Extras: scan images for vulnerabilities, sign them, generate an SBOM, pin base image digests, use a read-only root filesystem at runtime, and build multi-architecture images with `buildx`.

Likely follow-ups

- Why copy `go.mod` and `go.sum` before the source?
- What do you lose by using `scratch`?

```
# build a static binary
```

```
CGO_ENABLED=0 GOOS=linux go build -o app ./cmd/app
```

```
# multi-stage Dockerfile → tiny image
```

```
FROM golang:1.22 AS build
```

```
... go build ...
```

```
FROM gcr.io/distroless/static # or scratch
```

```
COPY --from=build /app /app
```

```
ENTRYPOINT ["/app"]
```

Go's static binaries enable tiny, secure images. Build with `CGO_ENABLED=0` for a fully static binary, then use a multi-stage Dockerfile: a fat builder stage compiles, and the final stage is `distroless/static` or `scratch` with just the binary — a few MB, minimal attack surface, no shell or package manager for attackers to abuse.

Distroless > Alpine for Go: Since the binary is static, you don't need a `libc` or shell at all. Smaller image, far fewer CVEs to patch.

Official docs

- [The Go Programming Language documentation](https://golang.org/doc/)

Q94 What's the Go concurrency gotcha with loop variables and goroutines?

Go

Spoken answer · ~90 s

The gotcha is that in a for loop, the loop variable is a single variable reused across iterations, in Go versions before 1.22. If you start goroutines inside the loop that reference the variable in a closure, they capture the variable itself, not its value at that moment. By the time the goroutines run, the loop has usually advanced, so many goroutines see the last value, or a mix, producing duplicate or missing work and data races. Example: looping over a slice of hosts and launching a goroutine that checks the host often ends up checking the final host repeatedly. Fixes: pass the variable as an argument to the goroutine's function, so each gets its own copy; or create a new variable inside the loop body shadowing the original, with `host := host`, so each iteration has its own. Go 1.22 changed the semantics, so each iteration gets its own copy of the loop variable when the module declares that language version, which removes the bug for modern code, but older modules and code reviewers still need to understand it. Detection: the race detector, go vet's loopclosure check, and linters. Related gotchas: forgetting to wait for goroutines with a `WaitGroup`, leaking goroutines, and unbounded goroutine creation. Always bound concurrency with a worker pool or semaphore.

Likely follow-ups

- What changed in Go 1.22?
- How do you detect this bug?

```
// pre-1.22 BUG: all goroutines saw the final i
for i := 0; i < 3; i++ {
    go func() { fmt.Println(i) }() // often prints 3,3,3
}

// fix (pre-1.22): capture per iteration
for i := 0; i < 3; i++ { i := i; go func(){ fmt.Println(i) }() }
```

Before Go 1.22, loop variables were shared across iterations, so goroutines closing over `i` all read its final value — a notorious concurrency bug. The fix was shadowing (`i := i`) or passing as an argument. Go 1.22 changed loop semantics so each iteration gets its own variable, fixing it by default — but you'll still see the old pattern in existing code and older interview questions.

Know your version: On Go <1.22 this bites; on ≥1.22 it's fixed. Mention both — it shows you track language

evolution.

Official docs

- [Effective Go](#)
- [Go concurrency patterns: Pipelines and cancellation](#)

Q95 What makes a script production-grade vs a throwaway one-liner?

Best Practices

Spoken answer · ~84 s

A throwaway one-liner works once for its author; a production-grade script must work repeatedly, for other people, in unattended conditions. Characteristics. Safety: strict error handling, input validation, quoting, no destructive action without guards, a dry-run mode, confirmation for risky operations, and least-privilege credentials. Idempotency: safe to run twice, converging on the desired state. Robustness: timeouts on every external call, retries with backoff for transient failures, locking against concurrent runs, cleanup of temporary files on exit, and handling signals. Observability: structured logging with levels, meaningful exit codes, metrics or heartbeats, and clear error messages that say what failed and what to do. Usability: argument parsing with help text, sensible defaults, documented configuration, and a README with examples and required permissions. Maintainability: readable code, small functions, no hardcoded environment details or secrets, version control, code review, and linting with tools such as shellcheck or ruff. Testing: unit tests for logic, integration tests against a safe environment, and tested failure paths. Operations: packaged and versioned, run by a scheduler or pipeline with alerting on failure, owned by a team, with a runbook. Also scalability considerations like streaming large inputs. A quick test: could someone else run it at 3 a.m. safely without asking me?

Likely follow-ups

- What makes a script idempotent?
- Why does a dry-run mode matter?

✓ strict mode / error handling, non-zero exit on failure

✓ idempotent + dry-run for anything destructive

✓ logging, observability, clear messages

✓ input validation + required-config checks

✓ timeouts, retries, graceful shutdown

✓ tests + shellcheck/lint in CI; documented usage

A throwaway script just needs to work once on known input. Production-grade means it survives bad input, partial failures, reruns, and interruption — with error handling, idempotency, dry-run, logging, timeouts, and tests. The mindset shift: assume everything can fail and the script will run unattended at 3am. That's what separates tooling other people can trust from a personal hack.

The 3am test: "Would I trust this to run unattended in prod and fail safely?" If not, it needs error handling, idempotency, and observability first.

 Start here (official docs)

- [Python documentation](#)

Q96 How do you make any mutating script safe with a dry-run mode?

Best Practices

Spoken answer · ~85 s

A dry-run mode shows what a script would change without changing anything, letting people review before trusting it. Design it from the start, not as an afterthought. Structure the script as two phases: first compute a plan, a list of intended actions based on desired versus current state, with no side effects; then execute the plan. Dry-run prints the plan and stops. The simplest implementation is a single wrapper function through which every mutating operation passes: in dry-run mode it logs 'would run: ...' and returns success, otherwise it runs the command. All reads still happen for real, so the plan reflects reality. Be disciplined: if any mutating call bypasses the wrapper, the dry run is a lie. Make it the default for destructive scripts, requiring an explicit apply or execute flag, as Terraform plan and apply do. Output should be clear: the action, target, and reason, plus a summary count, and optionally a machine-readable plan that can be saved, reviewed, and then applied exactly, so what is approved is what runs. Mention limitations: dry-run can't always predict dependent effects, such as values only known after creation, and the state may change between the plan and the apply, so re-validate at apply time.

Likely follow-ups

- Why compute a plan before executing?
- What can go wrong between plan and apply?

```
DRY_RUN=${DRY_RUN:-1}    # default to SAFE
```

```
run() {
```

```
    if [[ "$DRY_RUN" == 1 ]]; then echo "DRY: $*"; else "$@"; fi
```

```
}
```

```
run rm -rf "$old_dir"
```

```
run kubectl delete pod "$p"
```

Route every destructive action through a run wrapper that either prints (dry-run) or executes. Default to dry-run so the safe behavior is the accidental one — a human must consciously pass `DRY_RUN=0` to actually mutate. This lets operators preview exactly what a script will do before committing, turning "oops it deleted prod" into "the plan looked wrong, I stopped."

Default to safe: If dry-run is opt-in, someone will run the destructive path by accident. Make execution the explicit, deliberate choice.

 Start here (official docs)

- [Python documentation](#)

Q97 A script worked in staging but fails in prod. How do you debug systematically?

Best Practices

Spoken answer · ~87 s

Systematically, I treat it as a difference-finding exercise between environments. First, get the exact error: full logs, exit code, and stack trace, with timestamps, and reproduce it if I can. Then enumerate what differs. Inputs: data volume and shape, edge cases that only exist in production, character encodings, and null or empty values. Configuration: environment variables, config files, feature flags, and secrets. Identity and permissions: IAM roles, RBAC, service accounts, and security group or network policies that are stricter in production; permission errors and timeouts are the most common culprits. Network: DNS, proxies, egress rules, TLS certificates, and connectivity to dependencies. Runtime: operating system and distribution, tool and library versions, shell, locale, time zone, resource limits, and the user the script runs as. Scale: timeouts and rate limits that only trigger at production size, memory limits, and ordering or concurrency issues. State: leftover or unexpected existing resources making a create step fail. Use a bisection approach: run the failing command manually in the production context, using the same identity, from the same host or pod, with verbose or trace flags like `set -x`, then diff the environments, such as env output and tool versions. Check recent changes. Don't change production blindly; reproduce in a safe copy when possible.

Likely follow-ups

- Why are permission differences so common in these failures?
- How do you reduce staging and production drift?

1. Read the actual error + exit code (don't guess)

2. Diff the environments: versions, env vars, perms, data scale, network

3. Reproduce minimally; add `set -x` / verbose logging

4. Check scale: $O(n^2)$ or memory fine on 100 rows, dies on 10M

5. Check perms/secrets/network policy differences

"Works here, fails there" almost always comes down to an environment difference: tool versions, missing env vars, tighter permissions/RBAC, network policy, locale, or — very commonly — data scale (prod has 1000x the rows and exposes an $O(n^2)$ or memory bug). Debug methodically: capture the real error, diff the two environments systematically, reproduce small, then turn on tracing. Resist guessing.

Scale is the usual culprit: Code that's instant on staging's tiny dataset can time out or OOM on prod volume. Test with prod-like data sizes.

 **Start here (official docs)**

- [Python documentation](#)

Q98 How do you prevent and detect common shell-injection / command-injection bugs?

Best Practices

 Spoken answer · ~75 s

Injection happens when untrusted data is interpreted as code, such as a filename or parameter becoming extra commands. Prevention. Avoid the shell when possible: in Python, call `subprocess.run` with a list of arguments and without `shell` equals `True`, so the data is passed as arguments, never parsed by a shell; use library functions or APIs instead of building command lines. When a shell is unavoidable, quote correctly: in Bash, double-quote every expansion, and in Python use `shlex.quote` for pieces embedded in a shell string. Never use `eval` or `source` on untrusted data, avoid backticks and string concatenation, and be careful with xargs without null delimiters. Guard against argument injection, where a value starting with a dash is interpreted as an option: place a double dash before user-supplied arguments, or prefix paths with `./`. Validate input with allowlists: expected formats, such as a hostname pattern or numeric range, rather than trying to blacklist bad characters. Use least privilege so a compromise has limited impact, and avoid running as root. Also beware environment-based injection through `PATH` or `LD_PRELOAD`, and use absolute paths.

Likely follow-ups

- What is argument injection?
- Why are allowlists better than blocklists?

```
# VULNERABLE: user input in a shell string
```

```
os.system(f"ping {user_host}") # user_host="x; rm -rf /"
```

```
# SAFE: arg list, no shell
```

```
subprocess.run(["ping", "-c", "1", user_host])
```

```
# Bash: quote everything, validate input
```

```
[[ "$host" =~ ^[a-zA-Z0-9.-]+$ ]] || exit 1
```

Command injection happens when untrusted input is interpolated into a shell string — the attacker smuggles in `;`, `&&`, or backticks to run their own commands. The fix is to never build shell strings from input: pass arguments as a list with `shell=False`, so the OS treats them as literal arguments. In Bash, quote all expansions and validate input against a strict allowlist pattern.

The root cause is string-building: Any time user/external data becomes part of a command *string*, you have a potential injection. Pass args as a list, not a string.

 **Official docs**

- [OWASP Top Ten](#)

Q99 How do you structure a larger automation codebase so a team can maintain it?

Best Practices

 **Spoken answer** · ~87 s

A larger automation codebase needs to be treated like software, not a pile of scripts. Structure: a version-controlled repository with a clear layout, such as a package with modules separated by concern: clients for external systems, domain logic, CLI entry points, and configuration; thin entry points that parse arguments and call tested library code. Keep logic separate from side effects, so the logic can be unit tested with fakes, and wrap external systems behind small interfaces. Standards: consistent naming, type hints, formatting and linting enforced by pre-commit hooks and CI, such as `ruff` and `black` for Python or `shellcheck` for shell, and an agreed style guide. Dependency management: pinned dependencies with a lock file and a reproducible environment, using `pyproject` and a tool like `uv` or `poetry`, or Go modules. Testing: unit tests, plus integration tests against sandboxes, run in CI on every pull request. Configuration and secrets handled consistently, with environment-specific values separated from code. Shared libraries for common needs like logging, retries, and authentication, to avoid duplication. Documentation: README with setup and examples, docstrings, and runbooks for operational scripts. Delivery: versioned releases, packaged as a container or wheel, deployed through a pipeline, with changelogs. Ownership: code owners, review requirements, and a deprecation process for old scripts.

Likely follow-ups

- Why separate logic from side effects?
- What belongs in a shared library?

```
cli/          entrypoints (argparse/click)
```

```
lib/          reusable modules (clients, parsing, config)
```

```
tests/      pytest unit + integration
```

```
pyproject.toml  pinned deps, lint/format config
```

```
README + --help + type hints + CI (lint+test)
```

Past a few scripts, treat automation like real software: separate thin CLI entrypoints from reusable library logic, write tests, pin dependencies, and enforce lint/format/type-check in CI. Add type hints and a README so the next engineer (or you in six months) can understand and safely change it. The goal is that anyone on the team can fix a bug without fear — not a pile of one-off scripts only the author understands.

Bus-factor: Tested, documented, linted automation survives the author leaving. A folder of clever undocumented scripts is technical debt waiting to break.



Start here (official docs)

- [Python documentation](#)

Q100 Live coding: parse a log, compute error rate per endpoint, alert if any exceeds 5%. Walk through it.

Best Practices

🗣 Spoken answer · ~75 s

I'd walk through it in stages. Clarify first: the log format, whether it's a file or a stream, the time window, and what counts as an error, typically 5xx statuses, and whether the 5 percent threshold has a minimum request count. Then the approach: stream the file once, parsing each line with a precompiled regex or split into fields, extracting endpoint and status. Normalise the endpoint by stripping query strings and replacing IDs with placeholders, otherwise every URL is unique and cardinality explodes. Maintain two dictionaries, or a defaultdict of counters: total requests and error requests per endpoint. After the pass, compute error rate as errors divided by total for each endpoint, filter those above 0.05, and ignore endpoints with fewer than, say, 20 requests, to avoid noise from one failure out of three requests. Sort by rate and print a table. For alerting, exit with a non-zero code or post to a webhook when any exceed the threshold, including the counts and examples. Handle malformed lines by counting and skipping. Complexity is $O(n)$ time and $O(e)$ memory for e endpoints.

Likely follow-ups

- Why require a minimum request count?
- How do you avoid exploding endpoint cardinality?

```
from collections import defaultdict
```

```
total=defaultdict(int); errors=defaultdict(int)
```

```
for line in open("access.log"):
```

```

rec=parse(line)          # {path,status}; skip None

if not rec: continue

total[rec["path"]]+=1

if rec["status"]>=500: errors[rec["path"]]+=1

for path,n in total.items():

    rate=errors[path]/n

    if rate>0.05 and n>=100:    # ignore low-traffic noise

        alert(f"{path}: {rate:.1%} errors ({errors[path]}/{n})")

```

The structure interviewers want: stream the file (constant memory), parse defensively (skip malformed lines), tally totals and errors per endpoint with defaultdict in one pass ($O(n)$), then compute the rate and alert. The crucial nuance is the **minimum-volume guard** ($n \geq 100$): a path with 2 requests and 1 error is 50% but statistically meaningless — alerting on it creates noise. Mention edge cases out loud: empty file, all-bad lines, division by zero.

What earns the offer: Streaming not loading, defensive parsing, one-pass $O(n)$ counting, and especially the low-traffic guard — it shows you think about alert quality, not just code that "works."

Official docs

- [Python: re \(regular expressions\)](#)

Q101 How do you use dataclasses, and when are they better than a plain dict or NamedTuple?

Python

🔊 Spoken answer · ~88 s

A dataclass is a class decorator that generates boilerplate for classes mostly holding data: the constructor, repr, and equality, from annotated fields. Fields can have defaults and use field with a default_factory for mutable defaults. Options include frozen equals True for immutable, hashable instances, order equals True for comparison methods, and slots equals True to save memory. A post-init hook allows validation or derived fields. Compared with a plain dictionary: a dataclass has a defined set of fields, attribute access with autocomplete, type hints that static checkers can verify, and catches typos, because a misspelled attribute raises an error, whereas a dict silently returns a missing key. It can have methods and defaults. Compared with a named tuple: dataclasses are mutable by default, support inheritance, defaults, and methods more naturally, and aren't iterable or indexable like tuples; named tuples are immutable, lightweight, and unpackable, good for simple fixed records and as return values. Use a dict for truly dynamic or unknown keys, such as raw parsed JSON before validation. Use a dataclass for internal structured records, configuration objects, and results with a known shape. For parsing and validating external input, pydantic models add runtime validation and coercion, where plain dataclasses don't check types at runtime. Prefer frozen dataclasses for value objects.

Likely follow-ups

- Why use default_factory for list fields?
- When is a plain dict the better choice?

```
from dataclasses import dataclass, field
```

```
@dataclass(frozen=True, slots=True)
```

```
class Host:
```

```
    name: str
```

```
    cpu: int = 0
```

```
    tags: list = field(default_factory=list) # never a mutable default!
```

```
h = Host('web1', cpu=4)
```

A @dataclass auto-generates `__init__`, `__repr__`, and `__eq__` from typed fields — far more readable and self-documenting than a bare dict, with attribute access and type hints. `frozen=True` makes instances immutable and hashable (usable as dict keys/set members); `slots=True` cuts memory and speeds attribute access. Use `field(default_factory=list)` for mutable defaults to avoid the shared-default footgun.

Over dict: typed fields catch typos (`h.cpuu` is an `AttributeError`, `d['cpuu']` a silent `KeyError` later), and the class documents the shape. Over `NamedTuple`: dataclasses support defaults, mutation, and inheritance more cleanly.

 **Start here (official docs)**

- [Python documentation](#)

Q102 Explain the GIL precisely — what it does and does not prevent.

Python

 Spoken answer · ~90 s

The global interpreter lock, or GIL, is a mutex in CPython that ensures only one thread executes Python bytecode at a time. It exists to protect the interpreter's internal state, particularly reference counting for memory management, making the C implementation simpler and single-threaded code fast. What it prevents: true parallel execution of Python code across multiple threads, so CPU-bound multi-threaded Python programs do not scale across cores and can even run slower due to contention. What it does not prevent: concurrency for I/O-bound work, since the lock is released during blocking operations like network calls, file I/O, and sleep, so threads overlap waiting; also C extensions and libraries such as NumPy can release the GIL during heavy computation, achieving parallelism. It does not make your code thread-safe: operations like incrementing a shared counter consist of several bytecodes, and a thread switch between them can lose updates, so you still need locks or queues for shared mutable state. It doesn't affect multiprocessing, because each process has its own interpreter and GIL. The practical consequences: use threads or asyncio for I/O-bound tasks, multiprocessing or native extensions for CPU-bound ones. Recent Python versions offer experimental free-threaded builds without the GIL, and other implementations like Jython lack one. So the answer is: it limits parallel bytecode execution, not concurrency.

Likely follow-ups

- Why isn't Python code automatically thread-safe despite the GIL?
- When is the GIL released?

```
# GIL = Global Interpreter Lock: one thread runs Python bytecode at a time
```

```
CPU-bound threads  -> NO speedup (serialized by GIL)
```

```
I/O-bound threads  -> YES speedup (GIL released during I/O waits)
```

```
multiprocessing    -> true parallelism (separate interpreters, no shared GIL)
```

```
# Python 3.13+: experimental free-threaded build (PEP 703) can disable it
```

The GIL is a mutex that lets only one thread execute Python bytecode at a time, so pure-Python CPU work does not parallelize across threads. It does *not* block I/O concurrency: the GIL is released during blocking syscalls (network, disk), so threads overlap I/O effectively. C extensions (numpy) can also release it during heavy compute. For CPU parallelism use multiprocessing (or 3.13's experimental free-threaded build).

Why it exists: it simplifies CPython's memory management (reference counting) and C-extension safety. The free-threaded work (PEP 703) is the long-term path to removing it without breaking the ecosystem.

 Official docs

- [Python: concurrent.futures](#)

Q103 How do you profile memory and find a leak in a long-running Python service?

Python

Spoken answer · ~85 s

A memory leak in a long-running Python service appears as steadily growing resident memory until it's killed by the OOM killer. First confirm it's real by graphing RSS over time, distinguishing leaks from legitimate caching and fragmentation. Then find what's growing. In-process tools: `tracemalloc`, in the standard library, takes snapshots and compares them, showing which lines allocated the most new memory; take one snapshot early, another after load, and compare. The `objgraph` library shows which object types are increasing and who references them; `gc.get_objects` can count types; `heapy` or `memray` provide detailed allocation profiling, with `memray` able to attach to a running process and show flame graphs. For production, `py-spy` and `memray` allow attaching without code changes, or expose a debug endpoint temporarily.

Common causes: unbounded caches or dictionaries keyed by ever-growing values, such as request IDs; lists or queues that accumulate; global registries and listeners never removed; reference cycles holding large objects, though the cycle collector usually handles those unless objects define problematic finalisers; `lru_cache` on methods retaining instances; unclosed connections, file handles, or threads; exception tracebacks retained in variables; and memory held by C extensions. Fixes: bound caches with size limits or TTLs, use weak references, and close resources with context managers.

Likely follow-ups

- How does `tracemalloc` help find the leak source?
- What's a common cause involving caches?

```
import tracemalloc

tracemalloc.start()

snap1 = tracemalloc.take_snapshot()

# ... run workload ...

snap2 = tracemalloc.take_snapshot()

for stat in snap2.compare_to(snap1, 'lineno')[:10]:

    print(stat)          # top growth by source line
```

also: tracemalloc, objgraph, or memory_profiler / py-spy for live procs

tracemalloc (stdlib) snapshots allocations by source line; diffing two snapshots taken over time pinpoints what keeps growing — the signature of a leak. In Python, 'leaks' are usually unbounded growth: an ever-growing cache/list, lingering references in a module-level dict, or a closure capturing large objects. py-spy can profile a running process without restarting it, which is invaluable in prod.

Common culprit: an unbounded @lru_cache (no maxsize), a module-level list that's appended forever, or registering callbacks/observers without ever deregistering them. Bound your caches and audit module-level mutable state.

Official docs

- [Python Documentation](#)

Q104 What is the difference between `__str__` and `__repr__`, and why do both matter?

Python

Spoken answer · ~77 s

Both convert objects to strings, but for different audiences. The repr method is meant to be unambiguous and aimed at developers: ideally it returns a string that looks like valid Python code to recreate the object, and at least includes the class name and key fields. It's what you see in the interactive shell, in debuggers, inside containers when printing lists, and in log messages using the percent-r format. The str method is meant to be readable and aimed at end users: friendly output for display, used by print and format. If only repr is defined, str falls back to it, but not vice versa, so a good default is to always implement repr. Why both matter in practice: logs and error messages with repr show exactly what a value is, including quotes around strings and escapes for hidden characters, which makes it much easier to debug than str; for example, a string with trailing whitespace or a newline is visible in repr but invisible in str. Printing a list shows each element's repr, so a poor repr makes collections unreadable. Dataclasses generate repr automatically.

Likely follow-ups

- Which one does print() use?
- Why should repr avoid including secrets?

```
class Job:

    def __init__(self, id): self.id = id

    def __repr__(self): return f'Job(id={self.id!r})' # unambiguous, dev-facing

    def __str__(self): return f'job {self.id}' # readable, user-facing

j = Job('x'); print(j) # uses __str__ -> 'job x'
```

```
j # uses __repr__ -> Job(id='x')
```

```
[j] # containers use __repr__
```

`__repr__` is the unambiguous developer representation (ideally something you could paste back to recreate the object); it's what you see in the REPL, in tracebacks, and inside containers. `__str__` is the human-readable form used by `print()` and `str()`. If you define only one, define `__repr__` — Python falls back to it for `str()`, and good reprs make debugging logs and stack traces vastly more useful.

SRE relevance: a meaningful `__repr__` turns a useless `<Job object at 0x7f...>` in your error logs into `Job(id='abc')` — the difference between a fast diagnosis and a guessing game.

 **Start here (official docs)**

- [Python documentation](#)

Q105 How do you write async code with asyncio to fan out 1000 HTTP requests efficiently?

Python

 **Spoken answer · ~90 s**

To fan out a thousand HTTP requests efficiently, use `asyncio` with an async HTTP client such as `aiohttp` or `httpx`, because the work is I/O-bound and a single thread can manage thousands of concurrent connections. Create one shared client session, reused for connection pooling, instead of a new session per request. Define an async function that performs one request with a timeout. Launch tasks and gather results with `asyncio.gather`, or better, `asyncio.TaskGroup` in modern Python, which cancels siblings on failure. The crucial part is bounding concurrency: firing a thousand requests at once can exhaust file descriptors, trigger rate limits, or overwhelm the target. Use an `asyncio.Semaphore`, with a limit like 50, acquired around each request, or a connector limit on the client; or a worker pool pulling from an `asyncio.Queue`. Handle errors per request: use `return_exceptions=True` in `gather`, or catch exceptions inside the task and return a result object, so one failure doesn't discard the others. Add retries with backoff and jitter for transient errors, and honour 429 responses. Set overall and per-request timeouts. Never call blocking functions, such as `time.sleep` or `requests`, inside async code, since they stall the event loop; if unavoidable, use `run_in_executor`. Process results as they complete with `as_completed` to stream. Start the program with `asyncio.run`, and measure throughput.

Likely follow-ups

- Why bound concurrency with a semaphore?
- What happens if you call a blocking function in a coroutine?

```
import asyncio, aiohttp
```

```
async def fetch(session, url):
```

```
async with session.get(url, timeout=aiohttp.ClientTimeout(total=5)) as r:
```

```
    return url, r.status
```

```
async def main(urls):
```

```
    sem = asyncio.Semaphore(50)          # bound concurrency
```

```
    async with aiohttp.ClientSession() as s:
```

```
        async def bound(u):
```

```
            async with sem: return await fetch(s, u)
```

```
    return await asyncio.gather(*(bound(u) for u in urls))
```

asyncio runs thousands of concurrent I/O operations on a single thread via an event loop — ideal for high-fan-out network work without the overhead of thousands of threads. `asyncio.gather` runs coroutines concurrently and collects results; a `Semaphore` bounds how many run at once so you don't open 1000 sockets and exhaust file descriptors or hammer the target. Always set per-request timeouts.

Don't block the loop: a single synchronous/CPU-bound call inside a coroutine freezes the entire event loop and stalls every other task. Keep async code non-blocking; offload CPU work to an executor.

Official docs

- [Python: asyncio](#)

 Spoken answer · ~88 s

Floating-point numbers are stored in binary with finite precision, so many decimal values cannot be represented exactly. The classic example is that 0.1 plus 0.2 does not equal 0.3 in floating point; it comes out as 0.30000000000000004. So comparing floats with the equality operator can fail unexpectedly, even though the math looks right. In production this bites in several ways: a threshold check or equality condition silently never triggers; accumulated rounding errors in running sums of metrics or money drift over many operations; sorting and deduplication by float keys behave oddly; and results can differ across platforms or operation order, since floating-point addition is not associative. Also, NaN is never equal to itself, which breaks equality-based logic, and comparisons with NaN are always false. Fixes: compare with a tolerance, using `math.isclose` with relative and absolute tolerances, or `numpy.isclose`; use the decimal module or integers in the smallest unit, like cents, for money and anything requiring exact decimal arithmetic; use fractions where exact rational arithmetic is needed; avoid using floats as dictionary keys or in equality-based deduplication; round for display, not for storage; and use summation functions like `math.fsum`, which reduce error accumulation. Be careful when parsing numbers from JSON or metrics, and design alert thresholds with greater-than-or-equal comparisons plus tolerance.

Likely follow-ups

- What does `math.isclose` do?
- Why use integers or Decimal for money?

```
0.1 + 0.2 == 0.3          # False!  -> 0.30000000000000004
```

```
import math
```

```
math.isclose(0.1+0.2, 0.3)          # True (relative tolerance)
```

```
math.isclose(a, b, abs_tol=1e-9)     # for values near zero
```

```
# money: use Decimal, never float
```

```
from decimal import Decimal; Decimal('0.1') + Decimal('0.2') # exact 0.3
```

Floats are binary IEEE-754 and cannot represent most decimal fractions exactly, so `0.1 + 0.2 != 0.3`. Never compare floats with `==`; use `math.isclose` with appropriate tolerances. For money or anything requiring exact decimal arithmetic, use `Decimal` — float rounding errors in billing or financial calculations are real incidents.

The classic bug: summing many floats accumulates error, and `==` threshold checks flake intermittently. For currency, `Decimal` (or integer cents) is mandatory, not optional.

Q107 How does Python's import system work, and what causes circular import errors?

Python

 Spoken answer · ~87 s

When Python executes an import statement, it first checks `sys.modules`, a cache of already loaded modules. If the module is there, it's returned immediately; otherwise Python searches for it along `sys.path` using finders, locates the source, compiles it to bytecode, creates a module object, registers it in `sys.modules`, and then executes the module's top-level code. Importantly, the module is placed in `sys.modules` before its code finishes running, which is what makes circular imports possible and also fragile. A circular import error arises when module A imports module B at the top, and B imports A. When A starts, it begins importing B; B then imports A, finding a partially initialised A in `sys.modules`, and tries to use a name from it that hasn't been defined yet, causing an `ImportError` or `AttributeError`. Fixes: restructure to remove the cycle, which is usually best: move shared code into a third module that both import, or invert the dependency; import the module rather than names from it, so attribute lookups happen later at call time; move the import inside the function that needs it, a local import, deferring it until after both modules are initialised; or use type-checking-only imports with the `typing.TYPE_CHECKING` guard and string annotations, for cycles caused by type hints.

Likely follow-ups

- Why does the module appear in `sys.modules` before it finishes loading?
- How does `TYPE_CHECKING` break type-hint cycles?

```
# circular: a.py imports b, b.py imports a -> ImportError / partial module
```

```
# fixes:
```

```
import b                # import the MODULE, use b.func() lazily
```

```
def handler():
```

```
    from b import thing    # deferred import inside the function
```

```
# or: extract the shared piece into a third module both import
```

Importing runs the module top-to-bottom once, caching it in `sys.modules`. A circular import fails when module A (mid-execution) imports B, which imports A again — but A isn't finished, so the name B wants isn't defined yet. Fixes: import the module rather than names from it (resolve attributes lazily at call time), defer the import inside a function, or refactor the shared dependency into a third module. Circular imports usually signal a layering problem.

Design signal: a circular import often means two modules are too entangled. The cleanest fix is extracting the shared abstraction, not just deferring the import.

 Official docs

- [Python Documentation](#)

Q108 How do you safely handle and clean up resources when multiple can fail (ExitStack)?

Python

Spoken answer · ~85 s

When a function acquires several resources, each acquisition can fail, and everything acquired so far must still be released. Nested with statements work for a fixed number of resources, and a single with statement can list several context managers. For a dynamic or conditional number of resources, use `contextlib.ExitStack`. It's a context manager that holds a stack of cleanup callbacks: you enter it with a with statement, then call `enter_context` on each resource as you acquire it, and register arbitrary cleanup functions with `callback`. When the block exits, normally or by exception, `ExitStack` unwinds in last-in, first-out order, running every cleanup even if earlier ones raise, and it chains exceptions so none are lost silently. This solves the problem of try-finally pyramids and the partial-failure case: if the third resource fails to open, the first two are still closed. Use cases: opening a list of files whose length is only known at runtime, starting several temporary services in a test, acquiring locks or connections in a loop, or conditionally entering a context. The `pop_all` method transfers ownership of the cleanups to the caller, which is useful for constructing objects whose constructor acquires multiple resources and should clean up only on failure. Async code has `AsyncExitStack`.

Likely follow-ups

- What happens if one cleanup callback raises?
- What does `pop_all` do?

```
from contextlib import ExitStack
```

```
with ExitStack() as stack:
```

```
    files = [stack.enter_context(open(p)) for p in paths]
```

```
    conn = stack.enter_context(db.connect())
```

```
    # all of files + conn are cleaned up in reverse order on exit,
```

```
    # even if one of them (or the body) raises
```

ExitStack manages a dynamic number of context managers — you don't know how many files/connections until runtime. Each `enter_context` registers cleanup; when the `with` block exits (normally or via exception), everything is closed in reverse (LIFO) order. This avoids deeply nested `with` statements and guarantees partial setup is fully unwound if a later resource fails to open.

Why it matters: opening N resources where resource K fails would leak the first $K-1$ without ExitStack. It makes 'open many, clean up all, even on partial failure' correct and readable.

 Start here (official docs)

- [Python documentation](#)

Q109 What are Python's f-string pitfalls and the right way to log structured data?

Python

Spoken answer · ~88 s

F-strings are convenient and fast, but they have several pitfalls, especially in logging. Eager evaluation: an f-string is built before the logging call runs, so even if the log level is disabled, you pay the formatting cost, including expensive `repr` calls. Logging's own lazy formatting, with percent placeholders and arguments passed separately, defers formatting until the record is actually emitted. Structured logging: embedding values into the message text makes logs hard to search and aggregate, because every message is unique; instead, keep the message constant and pass fields as structured data, using the `extra` parameter, a structured logging library like `structlog`, or a JSON formatter, so logs carry fields such as user, request ID, and duration that can be filtered. Security: f-strings can leak secrets by embedding them in messages, and interpolating untrusted input into logs enables log injection via newlines; interpolating into SQL or shell commands causes injection vulnerabilities, so use parameterised queries and argument lists. Other gotchas: quotes and backslash restrictions inside expressions in older Python versions; accidentally formatting `None` or objects whose `str` is unhelpful, where the exclamation `r` conversion shows `repr`; format specs for numbers and widths; and the debug form with an equals sign, which prints the expression and value, handy for debugging but not for production.

Likely follow-ups

- Why is lazy formatting preferred in logging calls?
- Why are constant log messages better for aggregation?

```
# BAD: f-string formats EVEN when the log level is disabled (wasted work)
```

```
log.debug(f'processing {expensive()}')
```

```
# GOOD: lazy %-args, formatted only if DEBUG is enabled
```

```
log.debug('processing %s', expensive_repr)
```

```
# structured:
```

```
log.info('request done', extra={'path': path, 'ms': dur, 'status': code})
```

f-strings are great for general formatting but wrong for logging: the string is built eagerly even when that log level is suppressed, wasting CPU on hot paths. Use the logger's lazy %s arguments so formatting only happens if the message is actually emitted. For machine-queryable logs, attach fields via `extra=` (or use `structlog`) so you can filter by path/status in Loki/Elasticsearch instead of regexing free text.

Subtle perf bug: `log.debug(f'...')` in a tight loop computes the string a million times even in production where `DEBUG` is off. The %s form skips it.

Official docs

- [Google SRE Book: Distributed tracing & logs \(Monitoring chapter\)](#)
- [Python Documentation](#)

Q110 How do you pin dependencies and build reproducible Python environments?

Python

Spoken answer · ~88 s

Reproducible Python environments need exact versions of both direct and transitive dependencies. Declare direct dependencies with sensible constraints in `pyproject.toml` or `requirements.in`, then resolve and lock the full dependency tree with exact versions, using a tool such as `pip-tools`, which compiles `requirements.in` into a pinned `requirements.txt`, or `uv`, `poetry`, or `pipenv`, which maintain lock files. Prefer including hashes, so `pip install` with `require-hashes` verifies the artefacts weren't tampered with, protecting against supply chain attacks. Commit the lock file, and install from it in CI and production, so every environment gets identical packages. Use virtual environments to isolate projects, and pin the Python version too, via a `.python-version` file, `pyenv`, or the base container image tag, ideally by digest. In containers, install dependencies in an early Docker layer from the lock file for caching, and use multi-stage builds. Separate runtime from development dependencies. Update deliberately: use automated tools such as `Dependabot` or `Renovate` to propose upgrades through pull requests that run tests, rather than floating versions in production. Watch for platform-specific wheels and build requirements, vulnerability scanning with `pip-audit`, and private index configuration. The trade-off: pinning gives reproducibility but means you must actively update to receive security fixes, so schedule regular dependency refreshes. Libraries should specify ranges, while applications should pin.

Likely follow-ups

- Why include hashes in a lock file?
- Why do libraries and applications pin differently?

```
# modern: pyproject.toml + a lockfile
```

```
uv pip compile pyproject.toml -o requirements.txt # fully pinned + hashes
```

```
uv pip sync requirements.txt # exact, reproducible install
```

```
# or pip-tools: pip-compile / pip-sync
```

```
pip install --require-hashes -r requirements.txt # verify integrity
```

Reproducibility means every install resolves to the identical versions. Declare loose constraints in `pyproject.toml`, then compile a fully-pinned lockfile (with hashes) committed to the repo, and install from that in CI/prod. Hashes (`--require-hashes`) verify supply-chain integrity. Tools like `uv` or `pip-tools` separate 'what I want' (`pyproject`) from 'exactly what's installed' (lockfile) — the same pattern as `package-lock.json` or `Cargo.lock`.

Unpinned = irreproducible: `pip install` requests with no lock gives different versions on different days, causing 'works on my machine' and surprise breakages when a transitive dep ships a bad release.

Official docs

- [Python Documentation](#)

Q111 Explain `__enter__`/`__exit__` — write a class-based context manager for a lock with timeout.

Python

Spoken answer · ~84 s

A class-based context manager implements `enter` and `exit`. For a lock with a timeout: in the constructor, store the underlying lock, such as a `threading.Lock`, and the timeout value. The `enter` method tries to acquire the lock with the timeout; if it succeeds, it returns the lock or `self`; if it fails, it raises a `TimeoutError`, so the `with` block is never entered. The `exit` method receives the exception type, value, and traceback, releases the lock, and returns `False` or `None` so any exception from the body propagates rather than being suppressed. Important subtlety: if `enter` raises, `exit` is not called, so only release in `exit` when the acquire succeeded, which here is guaranteed because the block runs only after a successful `enter`. Make the release safe against errors, and keep the state tracking explicit if the object can be reused. Usage: with `TimeoutLock(lock, 5)` as `l`, then the critical section runs, and the lock is always released, even on exceptions. For inter-process locks, such as cron jobs, the same pattern can wrap a file lock using `fcntl.flock` with non-blocking attempts in a retry loop until the timeout. Add a poll interval, logging when waiting, and an owner or PID for debugging.

Likely follow-ups

- Why isn't `__exit__` called if `__enter__` raises?
- How would you extend this to a cross-process file lock?

```
import time
```

```
class FileLock:
```

```

def __init__(self, path, timeout=10):

    self.path, self.timeout = path, timeout

def __enter__(self):

    deadline = time.monotonic() + self.timeout

    while not self._try_acquire():

        if time.monotonic() > deadline: raise TimeoutError(self.path)

        time.sleep(0.1)

    return self

def __exit__(self, exc_type, exc, tb):

    self._release()    # runs even if the with-body raised

    return False      # don't suppress exceptions

```

A class context manager implements `__enter__` (setup, returns the resource) and `__exit__` (teardown, always runs). `__exit__` receives any exception info; returning `False` lets the exception propagate (returning `True` would swallow it — rarely what you want). This pattern guarantees the lock is released even if the protected block raises, and the timeout prevents waiting forever on a stuck lock.

Always release in `__exit__`: putting `release` there (not after the `with`-body) is what makes it exception-safe. Return `False` so you don't accidentally hide errors that occurred inside the block.



Official docs

- [AWS Builders' Library: Timeouts, retries and backoff with jitter](#)
- [Go: context package](#)

Spoken answer · ~88 s

External input, such as API payloads, configuration files, and environment variables, is untrusted and must be validated at the boundary. Pydantic lets you declare a model with typed fields, and when you construct it from a dictionary or JSON, it parses, coerces types where sensible, such as the string 8080 into the integer 8080, validates constraints, and raises a `ValidationError` listing every problem with field paths, instead of failing on the first. Features: field constraints for ranges, lengths, and regex patterns; enums and Literal types for allowed values; nested models for structured data; optional fields with defaults; custom validators for cross-field rules; strict mode to disable coercion; and extra equals forbid to reject unknown keys, which catches typos in configuration. The settings extension, `pydantic-settings`, loads configuration from environment variables and files. Benefits over manual dictionary checks: far less code, consistent error messages, type safety for the rest of the program, because after validation you work with typed objects, and automatic JSON schema generation for documentation. Practice: validate once at the boundary and pass typed objects inward; return clear errors to users; never include secrets in error output, using `SecretStr` for sensitive fields; and keep validation separate from business logic. Also set limits on size and depth to defend against abuse.

Likely follow-ups

- Why forbid unknown keys in configuration models?
- What is the benefit of validating at the boundary?

```
from pydantic import BaseModel, field_validator

class Config(BaseModel):

    replicas: int

    env: str

    @field_validator('replicas')

    @classmethod

    def positive(cls, v):

        if v < 1: raise ValueError('replicas must be >= 1')

        return v

cfg = Config(**yaml.safe_load(open('c.yaml')))) # raises on bad input
```

Parsing untrusted/external data (config files, API payloads, env) is where bugs and security issues enter. pydantic validates types and constraints at the boundary and produces typed objects, so the rest of your code works with guaranteed-valid data. It fails fast with clear messages on bad input instead of letting a malformed value cause a confusing error deep in the call stack. This is the 'parse, don't validate' discipline.

Validate at the edge: convert raw dicts to validated models the moment data enters your program. Downstream code then trusts the types, and errors point at the actual source (the bad input), not a distant symptom.

 Start here (official docs)

- [Python documentation](#)

Q113 What is the walrus operator and a genuinely useful SRE use case?

Python

Spoken answer · ~87 s

The walrus operator, written as colon-equals, is an assignment expression: it assigns a value to a variable and returns that value, within a larger expression. It was added in Python 3.8. Its main value is avoiding repeated computation or separate statements when you need to both test and use a result. Classic uses: in a while loop that reads chunks, while a chunk is assigned from reading the file and is non-empty, process it, removing the duplicated read call before and inside the loop; in an if statement checking a regex match, if a match is assigned from the search and not None, use it directly, instead of a separate assignment line; and in comprehensions, filtering on an expensive computed value while also keeping it, avoiding calling the function twice. SRE examples: parsing log lines, where the match object is both tested and used for its groups; polling a status API until it returns a terminal state; reading a stream of messages until an empty one arrives; and checking a lookup in a dictionary and using the value in one expression. Use sparingly: it can hurt readability when nested or clever, and has scoping subtleties, especially in comprehensions, where the target variable is bound in the enclosing scope.

Likely follow-ups

- Why does it simplify a read-until-empty loop?
- What are its scoping subtleties in comprehensions?

```
# assign-and-test in one expression (3.8+)
```

```
while (chunk := f.read(8192)):
```

```
    process(chunk)
```

```
# avoid calling expensive thing twice:
```

```
if (m := PATTERN.search(line)) is not None:
```

```
    handle(m.group('ip'))
```

```
# filter + reuse in a comprehension:
```

```
results = [y for x in data if (y := transform(x)) is not None]
```

The walrus `:=` assigns within an expression. Its best uses: reading in chunks until empty (assign and test the read in the loop condition), capturing a regex match once and reusing it, and computing a value in a comprehension's filter to reuse in the output without calling the function twice. Used sparingly it removes duplicate calls and tightens loops; overused it hurts readability.

Use it for: 'compute once, test, then reuse' patterns — especially regex matches and chunked reads. Don't sprinkle it everywhere; clarity beats cleverness.

 **Start here (official docs)**

- [Python documentation](#)

Q114 How do you handle timezones correctly in Python (a frequent production bug)?

Python

 **Spoken answer** · ~88 s

Time zone bugs are common in production, and the rule is to work in UTC internally and convert only at the edges. Distinguish naive datetimes, which have no time zone information, from aware ones, which do. Mixing them raises errors, but worse, naive values are silently misinterpreted: `datetime.now` returns local time without zone info, and `utcnow` returns a naive UTC value, which is deprecated. Always create aware datetimes: `datetime.now` with `timezone.utc`, or `datetime.fromtimestamp` with a zone. Store and transmit timestamps in UTC, in ISO 8601 format with an explicit offset, or as epoch seconds; convert to the user's local zone only for display, using the standard library `zoneinfo` module with IANA names such as `America/New_York`, not fixed offsets, because daylight saving time rules change. Pitfalls: arithmetic across DST transitions, where adding a day to a local time may produce a non-existent or ambiguous time; use UTC for arithmetic, or normalise afterwards; scheduling jobs in local time causes skipped or repeated runs when clocks change, so schedule in UTC where possible; comparing naive with aware datetimes; parsing timestamps without offsets; and servers whose system time zone isn't UTC. In logs, always include the zone. Test with edge cases around DST changes, leap days, and year boundaries. Use `pytest` with frozen time libraries.

Likely follow-ups

- Why store timestamps in UTC?
- What goes wrong when scheduling in local time around DST?

```
from datetime import datetime, timezone
```

```
from zoneinfo import ZoneInfo          # stdlib 3.9+
```

```
now = datetime.now(timezone.utc)       # ALWAYS aware, in UTC
```

```
# convert for display only:
```

```
now.astimezone(ZoneInfo('Asia/Kolkata'))
```

```
# parse ISO with offset:
```

```
datetime.fromisoformat('2026-01-15T10:00:00+05:30')
```

```
# AVOID: datetime.now() -> naive, ambiguous, bug magnet
```

Store and compute in UTC with timezone-aware datetimes; convert to local time only for display. Naive datetimes (no tzinfo) are ambiguous and the source of countless bugs — comparing naive and aware raises, and `datetime.now()` silently uses local time. Use `datetime.now(timezone.utc)` and `zoneinfo` for conversions. For durations use `time.monotonic()`, which is immune to clock changes and DST.

The DST/log-correlation trap: mixing naive local timestamps across servers in different zones makes log correlation impossible and breaks scheduling around DST transitions. UTC everywhere internally is the only safe rule.



Official docs

- [Python Documentation](#)

Q115 What's the difference between shallow iteration and copying when mutating during a loop?

Python

🗣 Spoken answer · ~88 s

Modifying a list while iterating over it is a classic bug. The for loop uses an internal index; if you remove an element, the following elements shift left, and the iterator's index moves forward, so the element that slid into the current position is skipped. For example, removing items that match a condition while looping over the same list will leave some matching neighbours behind. Adding elements while iterating can cause infinite loops or unexpected extra iterations. For dictionaries and sets, changing size during iteration raises a `RuntimeError`, because their iteration order depends on internal structure. Safe approaches: iterate over a copy, such as a slice copy of the list, or a list of the dictionary's keys, while mutating the original; build a new collection with a list comprehension or filter, keeping only items you want, which is usually the clearest and avoids mutation; collect items to remove in a separate list and remove them after the loop; or iterate backwards by index when removing from a list, so the shifting doesn't affect unvisited items. For dictionaries, build a new dictionary with a comprehension or loop over a list of items. Note a shallow copy still shares the underlying element objects, so mutating the elements themselves is visible through both.

Likely follow-ups

- Why does removing during iteration skip elements?
- What is the cleanest way to filter a list?

```
# BUG: modifying a list while iterating it skips elements / errors
```

```
for x in items:
```

```
    if bad(x): items.remove(x)    # WRONG
```

```
# FIX: iterate a copy, or build a new list
```

```
for x in items[:]:                # iterate a shallow copy
```

```
    if bad(x): items.remove(x)
```

```
items = [x for x in items if not bad(x)]    # cleaner: rebuild
```

```
# dict: for k in list(d): if ...: del d[k]
```

Mutating a collection while iterating it corrupts the iteration — lists skip elements (indices shift), and dicts/sets raise `RuntimeError: dictionary changed size during iteration`. The fixes: iterate over a copy (`items[:]` or `list(d)`) so the original can be mutated safely, or — usually cleaner — build a new filtered collection with a comprehension instead of removing in place.

Prefer rebuilding: `items = [x for x in items if keep(x)]` is clearer and bug-free versus in-place removal. Reach for it before copy-and-mutate.

 **Start here (official docs)**

- [Python documentation](#)

Q116 How do you make a CPU-bound task use all cores with multiprocessing safely?

Python

 Spoken answer · ~86 s

For CPU-bound work in Python, threads don't help because of the GIL, so use multiple processes. The standard route is `concurrent.futures.ProcessPoolExecutor` or `multiprocessing.Pool`, creating a pool of workers, typically sized to the CPU count from `os.cpu_count`, and distributing chunks of work with `map` or `submit`. Safety points. Always guard the entry point with the `if name equals main` idiom, because on platforms using the `spawn` start method, such as Windows and macOS by default, child processes re-import the main module, and unguarded code would recursively spawn processes. Arguments and results are pickled and transferred between processes, so they must be picklable, and large data transfer is expensive; reduce overhead by passing chunks rather than individual tiny tasks, using the `chunksize` parameter, and by giving workers file paths or indexes instead of big objects; shared memory facilities exist for large arrays. Avoid shared mutable state; have workers return results and let the parent aggregate, or use queues. Handle exceptions, which are re-raised in the parent when you retrieve results, and make sure a crashing worker doesn't hang the pool. Set timeouts, handle signals for clean shutdown, and be careful with forking in programs that use threads or hold locks. Consider memory: each process has its own copy.

Likely follow-ups

- Why is the `__main__` guard essential?
- What limits speedup from multiprocessing?

```
from concurrent.futures import ProcessPoolExecutor
```

```
def crunch(chunk): return heavy_compute(chunk)
```

```
if __name__ == '__main__':          # REQUIRED on spawn platforms
```

```
    with ProcessPoolExecutor() as ex:
```

```
        results = list(ex.map(crunch, chunks))
```

For CPU-bound work, `ProcessPoolExecutor` runs tasks in separate processes (each with its own GIL), achieving true multicore parallelism. The `if __name__ == '__main__'` guard is mandatory on spawn-based platforms (Windows/macOS default) — without it, child processes re-import and re-run your module, causing

infinite process spawning. Arguments and return values must be picklable, since they're serialized between processes.

Two real gotchas: (1) the `__main__` guard, or you fork-bomb yourself; (2) IPC overhead — passing big objects between processes can cost more than the parallelism saves. Chunk work coarsely to amortize the serialization cost.

Official docs

- [Python: concurrent.futures](#)

Q117 How do you write a generator-based streaming pipeline for ETL on huge data?

Python

Spoken answer · ~66 s

A generator-based pipeline chains small generator functions, each consuming an iterator and yielding transformed items, so data flows through one item at a time with constant memory. For ETL on a huge file: an extract generator opens the source and yields raw records line by line, or in chunks; a parse generator takes those and yields structured records, skipping or routing malformed ones to a dead-letter output; a transform generator cleans, enriches, and filters; and a batch generator groups records into fixed-size chunks; the load stage writes each batch to the destination, such as a database bulk insert or a Parquet file part. Nothing materialises the full dataset, and processing begins immediately. Composition looks like a nested call or a series of assignments where each stage wraps the previous, much like a Unix pipeline. Advantages: low memory, easy testing of each stage independently with small lists, reusability, and natural backpressure, since a stage pulls items only when ready.

Likely follow-ups

- Why does a generator pipeline give natural backpressure?
- How do you make the load stage safely retryable?

```
def read(path):  
    with open(path) as f:  
        yield from f  
  
def parse(lines):  
    for line in lines:  
        if (r := try_parse(line)): yield r  
  
def enrich(records):  
    for r in records: yield {**r, 'host': lookup(r['ip'])}
```

```
# compose – stays lazy, constant memory end to end:
```

```
for out in enrich(parse(read('huge.log'))):
```

```
    sink(out)
```

Chaining generators builds a pull-based pipeline where each stage processes one item at a time and passes it on — no stage ever materializes the full dataset, so memory stays flat regardless of input size. Each function does one transform (read → parse → enrich), making the pipeline composable and testable in isolation. This is the idiomatic Python pattern for ETL over multi-GB logs or unbounded streams.

Why streaming wins: the whole flow uses constant memory and starts producing output immediately (no waiting to load everything). Each stage is independently unit-testable with a small fake iterable.

 **Start here (official docs)**

- [Python documentation](#)

Q118 How do you debug a hung Python process in production without restarting it?

Python

🗣 Spoken answer · ~81 s

To debug a hung Python process without restarting it, you need to inspect it from outside. The best tool is py-spy: run `py-spy dump` with the process ID, and it prints the current Python stack of every thread, showing where each is blocked, such as waiting on a lock, a socket read, or in an infinite loop; it works without modifying the program, although it needs permissions like `ptrace` access, and in containers may require added capabilities. `py-spy top` shows live hot functions, and `record` produces flame graphs, which separate a busy loop from a blocked wait. If you prepared in advance, the `faulthandler` module can dump tracebacks on a signal: register `faulthandler` with a signal like `SIGUSR1` at startup, then send the signal to print all thread stacks to `stderr`, or use `faulthandler.dump_traceback_later` as a watchdog. Python 3.14 also adds remote debugging attach facilities. Other tools: `gdb` with Python extensions to inspect stacks at the C level, `strace` to see which syscall it is stuck in, such as `futex` for lock waits or `read` on a socket, `lsof` to inspect open connections, and checking CPU usage to tell a spin from a block.

Likely follow-ups

- What does `py-spy dump` show?
- How do you tell a busy loop from a blocked wait?

```
py-spy dump --pid 12345          # instant stack trace of every thread, no restart
```

```
py-spy top --pid 12345          # live flame-graph-style view
```

```
# built-in: send SIGQUIT for a faulthandler traceback (if enabled)
```

```
python -X faulthandler app.py # then faulthandler dumps on fatal signals
```

```
# or attach: import pdb; or signal a thread to print stacks
```

py-spy is the SRE's best friend: it samples a running Python process from outside (no code changes, no restart, negligible overhead) and dumps every thread's stack — instantly showing where a hang is stuck (a blocking call with no timeout, a deadlock, an infinite loop). Restarting a hung prod process destroys the evidence; py-spy lets you diagnose the live state first. faulthandler is the stdlib fallback for crash tracebacks.

The #1 finding: a hung process is usually blocked on a network call with no timeout. py-spy dump points straight at the offending stack frame, turning a mystery hang into a one-line fix.

Official docs

- [Python Documentation](#)

Q119 How do you write an ERR trap that prints the failing command and line number?

Bash / Shell

Spoken answer · ~87 s

An ERR trap runs a handler whenever a command fails, and combined with Bash's built-in variables, it can show where. Define a function that captures the exit status immediately, since later commands overwrite it, and prints the failing command, line number, and source file, using BASH_COMMAND for the command text, LINENO for the line, and BASH_SOURCE for the file. Then register it with trap handler ERR. Also set errtrace with set -E, so the trap is inherited by functions, command substitutions, and subshells; without it, failures inside functions don't trigger the trap. Combine with set -e so the script exits after reporting, and set -u and pipefail for strictness. The handler can also print a call stack by looping over the FUNCNAME and BASH_LINENO arrays, which is useful in larger scripts. Send the message to stderr, and perhaps include a timestamp and the exit code. Caveats: the trap fires for failures that Bash considers errors, so commands in conditionals, such as the test of an if, or part of an and-or list, don't trigger it, matching the set -e rules; the line number reported may refer to the start of a multi-line command; and in pipelines, with pipefail, the failing stage is reported only by the overall status.

Likely follow-ups

- Why is set -E needed for the trap to work in functions?
- Which variables identify the failing command and line?

```
#!/usr/bin/env bash
```

```
set -Eeuo pipefail
```

```
trap 'echo "ERROR line $LINENO: $BASH_COMMAND (exit $?)" >&2' ERR
```

```
# now any failing command reports exactly where and what failed
```

```
some_command_that_might_fail
```

An ERR trap fires whenever a command exits non-zero (with `set -e`). Printing `$LINENO` and `$BASH_COMMAND` tells you exactly which line and command failed — turning a silent abort into an actionable error. The `-E` flag is what makes the ERR trap propagate into functions and subshells; without it, failures inside functions won't trigger the trap.

Production pattern: combine strict mode + an ERR trap that logs line/command, so a 200-line deploy script that dies tells you precisely where instead of just exiting with a bare code.

 **Start here (official docs)**

- [Python documentation](#)

Q120 What is the difference between a subshell and a command group, and when does it matter?

Bash / Shell

 Spoken answer · ~75 s

Both group commands, but they differ in the process model. A subshell, written with parentheses, runs the commands in a child shell process, a forked copy of the current shell. Changes made inside, such as setting variables, changing the working directory, defining functions, or altering shell options, don't affect the parent. A command group, written with curly braces, runs the commands in the current shell, so changes persist; the syntax is stricter, requiring spaces after the opening brace and a semicolon or newline before the closing one. When it matters: use a subshell when you want isolation, such as changing directory for a few commands without affecting the rest of the script, or temporarily modifying the environment or umask; the isolation means you can't set a variable inside and read it afterwards. Use a command group when you want to redirect or pipe the combined output of several commands as a unit, such as sending the output of three commands to one file, without the cost of a fork, and when the group must modify variables or exit the script.

Likely follow-ups

- Why do variables set in a piped while loop vanish?
- When would you use a command group for redirection?

```
( cd /tmp; do_stuff )      # SUBSHELL: cd is local, doesn't affect caller
```

```
{ cmd1; cmd2; }            # GROUP: same shell, cd/vars persist
```

```
x=1; ( x=2 ); echo $x      # 1 (subshell change is lost)
```

```
x=1; { x=2; }; echo $x     # 2 (group mutates current shell)
```

```
result=$(cmd)
```

```
# command substitution also runs in a subshell
```

() runs commands in a **subshell** — a child process with its own copy of variables and working directory, so changes (cd, variable assignments) don't leak back. { } is a **command group** running in the current shell, so its side effects persist. This matters constantly: a cd inside \$(...) or a pipe stage won't affect your script, and variables set inside a piped while loop vanish because the pipe spawns a subshell.

The classic gotcha: `cmd | while read x; do count=$((count+1)); done` — count is 0 afterward because the loop ran in a subshell. Use process substitution `while ...; done <<(cmd)` to keep the loop in the current shell.



Start here (official docs)

- [Python documentation](#)

Q121 How do you use process substitution and why is it better than temp files?

Bash / Shell

🗣 Spoken answer · ~75 s

Process substitution lets a command's output or input appear as a filename. The form with a less-than sign and parentheses runs the command and gives the outer command a path, like `/dev/fd/63`, that it can read as a file; the form with a greater-than sign gives a path that feeds the command's input. Bash creates it using a named pipe or a file descriptor, so no data is written to disk. Typical uses: comparing the output of two commands with `diff`, giving each command's output as a substituted file, with no need to save them first; feeding a `while read` loop from a command without a pipeline, so the loop runs in the current shell and variables set inside it survive, which a pipe would lose; and `tee` to send a stream to multiple consumers. Benefits over temp files: no disk I/O, no race conditions or predictable names, no cleanup needed, no leftover files if the script is interrupted, no exposure of sensitive data on disk, and data streams concurrently instead of waiting for the first command to finish.

Likely follow-ups

- Why does it preserve variables in a while-read loop?
- What can't you do with a substituted path?

```
# compare output of two commands without temp files
```

```
diff <(kubectl get pods -o name) <(cat expected.txt)
```

```
# feed a command's output where a FILE is expected
```

```
while read -r line; do echo "$line"; done < <(generate)
```

```
# tee to multiple processors
```

```
generate | tee >(gzip > out.gz) >(wc -l > count.txt) > /dev/null
```

Process substitution `<(cmd)` presents a command's output as a filename (a `/dev/fd` pipe), so tools that expect file arguments (`diff`, `comm`) can consume command output directly — no temp files to create, secure, and clean up. `>(cmd)` works the other direction for output. Crucially, `while read < <(cmd)` keeps the loop in the current shell (unlike `cmd | while`), so variables set inside persist.

Two wins: no temp-file race conditions or leaks, and it sidesteps the subshell-variable-loss problem of pipes. `diff <(a) <(b)` is the canonical example.

 **Start here (official docs)**

- [Python documentation](#)

🗣 Spoken answer · ~86 s

Bash's `getopts` handles only short options, so for long options like double-dash `env` equals `prod`, I write a manual loop over the arguments with a case statement. The loop runs while arguments remain; it matches patterns like the long option followed by an equals sign, extracting the value using parameter expansion that strips the prefix, and also supports the separated form, where the value is the next argument, handled by shifting. Short aliases are matched in the same case branch. A double dash terminates option parsing, so everything after is treated as positional, and an unknown option prints usage and exits non-zero. After the loop, validate required options and allowed values, such as restricting the environment to `dev`, `staging`, or `prod`. Gotchas: handle missing values, such as an option at the end with no argument, and check that the next argument isn't itself another option; support combining forms consistently; quote everything; and keep defaults before parsing. The external `getopt` command, GNU enhanced version, can parse long options and reorder arguments, but behaviour differs on BSD and macOS, hurting portability, so use it only when you control the platform. For anything with subcommands, validation, or generated help text, consider Python `argparse` or a Go CLI library instead.

Likely follow-ups

- Why can't `getopts` handle long options?
- How do you support both `--env=prod` and `--env prod`?

```
while [[ $# -gt 0 ]]; do
    case "$1" in
        --env=*)  env="${1#*=}"; shift ;;          # --env=prod
        --env)    env="$2"; shift 2 ;;             # --env prod
        --dry-run) dry=1; shift ;;
        --)       shift; break ;;                  # end of options
        -*)       echo "unknown: $1" >&2; exit 2 ;;
        *)       args+=("$1"); shift ;;             # positional
    esac
done
```

`getopts` only handles short flags, so for long options you write a manual `while/case` loop. Handle both `--env=prod` (split on `=` with `${1#*=}`) and `--env prod` (consume `$2`, `shift 2`) forms, collect positionals into an array, and reject unknown flags. The `--` sentinel signals end-of-options so remaining args starting with `-` are

treated as positional.

When the flag logic gets complex, that's the signal to switch to Python's argparse — robust long-option parsing in Bash is verbose and easy to get subtly wrong.

Official docs

- [GNU Bash reference manual](#)

Q123 Explain mapfile/readarray and why it's better than a for-loop over command output.

Bash / Shell

Spoken answer · ~78 s

The mapfile builtin, also named readarray, reads lines from standard input into an indexed array in one step. With the -t option, it strips trailing newlines, so each element is a clean line. The common usage is mapfile -t lines with input redirected from a process substitution of a command, for example the list of files or hosts. It is better than looping over command output for several reasons. Looping with for over an unquoted command substitution relies on word splitting, so lines containing spaces break into multiple items, and globbing characters get expanded, producing wrong results. A while-read loop piped from a command is correct but runs in a subshell, losing variable changes. Mapfile with process substitution runs in the current shell, preserves each line exactly, including spaces and glob characters, and gives you an array with a known length for indexing, slicing, and re-iterating, which a stream can't do. It can also read null-delimited input with the -d option set to an empty string, ideal for filenames from find -print0, and has options to skip lines, limit counts, and call a callback every N lines.

Likely follow-ups

- Why does a for loop over \$(command) break on spaces?
- How do you read filenames with newlines safely?

```
# read all lines into an array safely (handles spaces, no word-split)
```

```
mapfile -t pods < <(kubectl get pods -o name)
```

```
echo "got ${#pods[@]} pods"
```

```
for p in "${pods[@}"; do echo "$p"; done
```

```
# vs the fragile classic: for p in $(kubectl ...) -> splits on whitespace
```

mapfile -t (aka readarray) reads lines into an array, one element per line, with -t stripping the trailing newline. It correctly handles entries with spaces and avoids the word-splitting and glob-expansion bugs of for x in \$(cmd). You then iterate with "\${arr[@]}" (quoted) to preserve each element. It's the safe, modern way to capture command output line-by-line into a list.

Why it's safer: `for f in $(ls)` breaks on any filename with a space and expands globs; `mapfile -t arr < <(find ...)` with quoted expansion handles every case correctly.

 **Start here (official docs)**

- [Python documentation](#)

Q124 How do you safely create and use temporary files/dirs (avoiding /tmp races)?

Bash / Shell

 Spoken answer · ~77 s

Temporary file races happen when a script uses a predictable name in a shared directory like /tmp. Another user can pre-create a file or symlink with that name, so your script writes into or reads from something the attacker controls, a symlink attack, or two runs collide on the same name. Safe approach: use `mktemp`, which creates the file or directory atomically with a unique random name and restrictive permissions, owner-only, returning the path. Use `mktemp -d` for a private directory and create files inside it. Never build names from the process ID or date alone, and never test for existence and then create separately, which introduces a time-of-check to time-of-use window. Store the path in a variable and quote it, and register a trap on EXIT to remove it, so cleanup occurs on errors and interrupts. Prefer a private per-user or per-application directory, such as one under `XDGRUNTIME_DIR`, instead of the world-writable /tmp, and in containers, mount a tmpfs for scratch data. Set a restrictive umask before creating sensitive files. For data that doesn't need a file at all, use pipes or process substitution.

Likely follow-ups

- What is a time-of-check to time-of-use race?
- Why create temp files on the same filesystem as the destination?

```
tmp=$(mktemp)           # unique file, mode 600, in $TMPDIR
```

```
tmpd=$(mktemp -d)       # unique directory
```

```
trap 'rm -rf "$tmp" "$tmpd"' EXIT INT TERM
```

```
# NEVER: tmpfile=/tmp/myapp.$$ (predictable name = symlink-attack risk)
```

`mktemp` atomically creates a uniquely-named temp file/dir with safe permissions (0600/0700), defeating the predictable-name race where an attacker pre-creates or symlinks `/tmp/myapp.PID` to hijack or read your data. Pair it with a `trap ... EXIT` cleanup so temp files never leak on exit, error, or interrupt. Hard-coded /tmp paths are both a security hole and a source of collisions between concurrent runs.

Predictable temp names are a CVE class: `/tmp/foo.$$` is guessable; an attacker can symlink it to a sensitive file before you write. Always `mktemp`.

 [Start here \(official docs\)](#)

- [Python documentation](#)

Q125 What does 'set -o pipefail' fix, and what's the remaining gotcha with pipelines?

Bash / Shell

 Spoken answer · ~90 s

By default, a pipeline's exit status is that of the last command only, so a failure in an earlier stage is hidden. For example, a failing curl piped into grep that succeeds reports success, and with set -e the script carries on with bad or empty data. Setting pipefail makes the pipeline's status the rightmost non-zero status among its commands, or zero if all succeeded, so with set -e a failure anywhere stops the script. The array PIPESTATUS also records every stage's individual status, useful for inspecting which stage failed. Remaining gotchas. Some commands legitimately return non-zero in normal use: grep returns 1 when it finds no match, so a pipeline ending in grep fails under pipefail when there are simply no matches, requiring a guard such as appending an or-true clause, or checking explicitly. A producer in a pipeline can receive SIGPIPE when the consumer exits early, as with head: the writer is killed and returns 141, so pipefail may report failure for a perfectly normal pipeline like yes piped to head. Pipeline stages run in subshells, so variable assignments inside are lost. And errexit itself has exceptions inside conditions. So pipefail improves safety, but you still need deliberate handling for commands where a non-zero status is expected, and explicit checks of critical steps.

Likely follow-ups

- What is the PIPESTATUS array?
- Why can head cause a pipeline to appear to fail?

```
set -o pipefail
```

```
cmd_that_fails | grep x      # now returns failure, not grep's success
```

```
# inspect each stage's exit code:
```

```
false | true | false
```

```
echo "${PIPESTATUS[@]}"      # 1 0 1 -> per-stage exit codes
```

By default a pipeline's exit status is only the *last* command's, so `failing_cmd | grep x` reports success if `grep` succeeds — silently masking the real failure. `set -o pipefail` makes the pipeline fail if *any* stage fails. The remaining nuance: to know *which* stage failed, inspect the `PIPESTATUS` array, which holds each command's individual exit code.

Debugging tip: `${PIPESTATUS[@]}` right after a pipeline tells you exactly which stage broke — invaluable when a multi-stage pipe fails and pipefail only tells you 'something did.'

Official docs

- [GNU Bash reference manual](#)

Q126 How do you write a portable script that detects and adapts to GNU vs BSD tools?

Bash / Shell

 Spoken answer · ~85 s

GNU and BSD versions of common tools, such as those on Linux versus macOS, differ in options and behaviour. For example, `sed` in-place editing needs an empty-string argument on BSD, `date` has different flags for relative times and formatting, `readlink -f` is missing on older BSDs, `stat` has different format syntax, and `grep -P` exists only in GNU. Detection approach: test capability rather than guessing from the operating system name. For instance, run the tool with a GNU-only flag, such as `sed --version` or `date --version`, and check whether it succeeds, then set a variable or define wrapper functions that choose the right options. Alternatively, check `uname` to select a branch, but feature tests are more robust because users may install GNU tools on macOS with a `g` prefix, like `gsed` and `gdate`. Another approach is to define variables such as `SED` pointing to `gsed` when available, and use that consistently. Best practice: stick to the POSIX subset of options where possible, which both support; use `printf` instead of `echo` with escapes; avoid non-portable features; and check dependencies early with `command -v`, failing with a clear message. For complex logic like date arithmetic, Perl or Python are more consistent across platforms than shell tools.

Likely follow-ups

- Why test for a capability rather than check `uname`?
- What's the `sed` in-place difference on BSD?

```
# sed -i differs: GNU 'sed -i' vs BSD 'sed -i ""'
```

```
if sed --version >/dev/null 2>&1; then
```

```
    SED_INPLACE=(sed -i)          # GNU
```

```
else
```

```
    SED_INPLACE=(sed -i '')       # BSD/macOS
```

```
fi
```

```
"${SED_INPLACE[@]}" 's/a/b/' file
```

```
# also: date, stat, readlink, grep -P differ across GNU/BSD/busybox
```

GNU coreutils (Linux), BSD utils (macOS), and busybox (Alpine) have subtly different flags — `sed -i`, date formatting, `stat`, `readlink -f`, and `grep -P` are common divergences. Detect the variant (e.g. `sed --version` succeeds only on GNU) and branch, storing the command in an array. For maximum portability, stick to POSIX-specified flags, or require GNU coreutils explicitly and check for them up front.

The frequent offender: `sed -i` needs an empty backup arg on BSD/macOS but not GNU. A script that works on your Mac silently corrupts files in an Alpine container, or vice versa.

Official docs

- [POSIX Shell Command Language](#)

Q127 How do you implement a mutex/lock so only one instance of a script runs at a time?

Bash / Shell

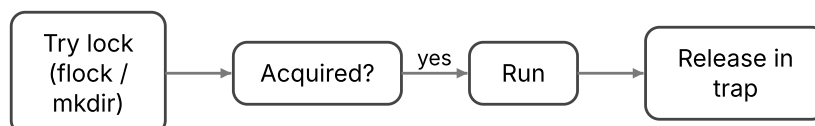
Spoken answer · ~85 s

To ensure only one instance of a script runs at a time, use an atomic lock rather than a check-then-create approach that races. The best option on Linux is `flock` from `util-linux`: open a lock file on a file descriptor, then call `flock` with the non-blocking option on that descriptor; if it fails, another instance holds the lock, so print a message and exit. The kernel releases the lock automatically when the process exits, even if it crashes or is killed, so there's no stale lock problem. Place the lock file in a stable directory such as `/var/lock` or `XDG_RUNTIME_DIR`. You can also wrap the whole command externally, running `flock -n` with the lock file and the script, which is handy in cron. Alternative: `mkdir`, which is atomic, as a lock: succeeds only if the directory doesn't exist; but it leaves a stale lock if the process dies unless cleaned by a trap, and traps can't catch `SIGKILL`. PID files add checking whether the recorded process is still alive, which is racy and subject to PID reuse. Design choices: decide whether to wait with a timeout or fail fast, log when skipping, and consider alerting if a run is blocked for too long.

Likely follow-ups

- Why is `flock` safer than a PID file?
- Why is `mkdir` used as an atomic lock?

Single-instance lock



```
exec 9>/var/lock/myjob.lock      # open fd 9 on a lock file
```

```
if ! flock -n 9; then           # non-blocking: fail if held
```

```
    echo 'another instance is running' >&2; exit 0
```

```
fi
```

```
# fd 9 (and thus the lock) is auto-released when the script exits
```

```
# blocking variant with timeout: flock -w 30 9 || exit 1
```

`flock` on a dedicated lock file gives a robust mutex. Opening the file on a file descriptor and locking it means the lock is automatically released when the process exits (even on crash) — no stale lock files to clean up, unlike a manual `[ -f lockfile ]` check which leaks on kill. `-n` fails immediately if another instance holds it; `-w N` waits up to `N` seconds. Essential for cron jobs that must not overlap.

Don't roll your own with a flag file: `if [ -f lock ]; then exit; fi; touch lock` has a race (two instances pass the check) and leaks the file on crash. `flock` is atomic and self-cleaning.

 **Start here (official docs)**

- [Python documentation](#)

Q128 How do you safely handle 'rm -rf' with variables to avoid catastrophic deletion?

Bash / Shell

 Spoken answer · ~85 s

The classic disaster is `rm -rf slash dollar variable slash star` where the variable is empty, which becomes `rm -rf slash star`, deleting everything. Layers of protection. Make unset or empty variables fatal: `set -u` catches unset, and the colon-question parameter expansion, dollar brace `VAR colon question message`, aborts if the variable is unset or empty. Validate the value before deleting: ensure it is non-empty, is an absolute path, resolves with `realpath` to a location under an allowed base directory, is not slash or the home directory, and exists and is the expected type of directory. Better yet, avoid constructing paths from variables for deletion: use `mktemp`-created directories tracked in a variable and removed via `trap`, with checks. Use the double dash to prevent names starting with dashes being read as options, and quote expansions. Reduce blast radius: delete specific files with `find` using `-maxdepth` and name patterns with `-delete` after a dry run listing the matches, prefer `rm` without `-r` when possible, and don't run as root. Provide a dry-run mode that prints what would be removed. Consider moving to a trash or quarantine directory first, deleting later. GNU `rm`'s `preserve-root` option, default on modern systems, blocks removing slash itself but not slash star.

Likely follow-ups

- What does `${VAR:?}` do?
- Why doesn't `--preserve-root` protect against `rm -rf /*`?

```
# DANGER: if $dir is empty/unset -> rm -rf / (with set -u it errors first)
```

```
rm -rf "$dir"/
```

```
# SAFE patterns:
```

```
: "${dir:?dir must be set}"          # fail loud if empty/unset
```

```
[[ -n "$dir" && "$dir" != "/" ]] || exit 1
```

```
rm -rf -- "${dir:?}"                # :? guards empty; -- ends options
```

The infamous `rm -rf "$dir"/` deletes everything if `$dir` is empty (becomes `rm -rf /`). Defenses: `set -u` to error on unset vars, the `${dir:?msg}` expansion to fail loudly when empty, an explicit non-empty/non-root check, and `--` to stop a path starting with `-` being read as a flag. For destructive operations, validate the target rigorously and prefer a dry-run that prints what would be deleted.

Real outages have happened from an unset variable in `rm -rf "$VAR/"`. Always guard with `${VAR:?}` and a non-root check before any recursive delete.

 **Start here (official docs)**

- [Python documentation](#)

Q129 How do you read a config/env file into a script without 'source'-ing untrusted code?

Bash / Shell

🗣 Spoken answer · ~84 s

Sourcing a file executes it as shell code, so a malicious or malformed config file can run arbitrary commands, including through command substitution or backticks in values. For untrusted or semi-trusted files, parse them as data. For a simple KEY=value format, read line by line with a while loop using IFS set to the equals sign and `read -r`, which splits key from value without evaluating anything. Skip blank lines and comments, validate keys against a pattern such as uppercase letters, digits, and underscores, and optionally an allowlist of expected keys, then assign using `declare` or `printf -v` with the validated name, never `eval`. Strip surrounding quotes explicitly if supported. Don't expand values: treat dollar signs and backticks as literal text. For environment files, tools like systemd's `EnvironmentFile`, Docker's `env-file`, or `direnv` parse them without executing. Alternatives: use a structured format with a real parser, such as JSON read through `jq`, or YAML or TOML read in Python, which don't execute code. If you must source, restrict risk by checking file ownership and permissions: it must be owned by root or the service user and not world- or group-writable, and live in a protected directory; and run the script with least privilege.

Likely follow-ups

- Why is sourcing a config file risky?
- How do you assign a variable with a validated name without `eval`?

```
# UNSAFE: source runs ARBITRARY code in the file
```

```
source config.env          # a line like  rm -rf /  would execute
```

```
# SAFER: parse KEY=VALUE lines explicitly
```

```
while IFS='=' read -r key val; do
```

```
[[ $key =~ ^[A-Z_][A-Z0-9_]*$ ]] || continue  # validate key
```

```
declare "$key=$val"
```

```
done < config.env
```

`source file` executes the file as shell code — if the file is attacker-controlled or has a typo with a command, it runs. For a simple `KEY=VALUE` config, parse it explicitly: split on `=`, validate the key matches an identifier pattern, and assign with `declare`. This treats the file as data, not code, eliminating the command-execution risk.

Sourcing config is a code-execution vector: any command in the file runs with your script's privileges. Only source files you fully control; parse anything external.

 **Start here (official docs)**

- [Python documentation](#)

Q130 What's the difference between 'exec', backgrounding, and 'nohup' for process control?

Bash / Shell

 Spoken answer · ~87 s

These control how a process relates to the shell and terminal. The `exec` builtin replaces the current shell process with the specified command, with no new process: the shell is gone, and the command takes over the same process ID. It's used at the end of wrapper scripts and container entrypoints, so the real program becomes PID 1 and receives signals such as `SIGTERM` directly, and no extra shell lingers. Used with only redirections, `exec` changes the shell's own file descriptors, such as redirecting all later output to a log. Backgrounding with an ampersand starts a command as a child in the background; the shell continues immediately, you can wait for it, and it is still a child of the shell, in the same session. However, when the terminal closes, the shell sends `SIGHUP` to its jobs, which typically kills it. The `nohup` command makes the command ignore `SIGHUP` and redirects output to `nohup.out` if attached to a terminal, so it survives logout, though it is not a supervisor. Alternatives for long-running work: `disown` to remove a job from the shell's job table, `setsid` to start a new session, `tmux` or `screen` for resumable sessions, and for real services, `systemd` units, which handle restarts, logging, resource limits, and dependencies.

Likely follow-ups

- Why use `exec` in container entrypoint scripts?
- What does `nohup` actually change?

```
exec myprog          # REPLACE the shell with myprog (no new process)
```

```
myprog &             # run in background, dies if shell/terminal closes
```

```
nohup myprog &       # background, survives terminal hangup (SIGHUP)
```

```
setsid myprog        # fully detach into a new session (daemon-like)
```

```
exec 2>/var/log/x     # exec with redirects: redirect script's own fds
```

`exec cmd` replaces the current shell process with `cmd` — useful as the last line of a wrapper (no extra shell lingering), and for redirecting the script's own file descriptors (`exec >log 2>&1`). `&` backgrounds a job but it receives `SIGHUP` when the terminal closes. `nohup` ignores `SIGHUP` so the job survives logout. `setsid` fully detaches into its own session. Knowing these is key for launching long-running processes correctly.

Container entrypoints use `exec`: `exec myapp` as the final line makes your app PID 1, so it receives signals (`SIGTERM`) directly for graceful shutdown — wrapping it in a shell that doesn't `exec` swallows those signals.

Official docs

- [Linux man-pages: `signal\(7\)`](#)

 Spoken answer · ~84 s

Measure first: wall time with the `time` command, and compare user, system, and real times. Shell scripts are usually slow because of process creation: every external command in a loop forks and execs a new process, costing milliseconds, so running a loop of a hundred thousand iterations calling `sed` or `awk` or `grep` each time is dramatically slower than a single invocation. Improvements: replace external commands with Bash builtins and parameter expansion for string operations, such as substring removal, replacement, and case conversion, and arithmetic with double parentheses instead of `expr`; avoid piping into `cat`; process whole files in one `awk`, `sed`, or `grep` pass instead of per-line calls; use `read` with here-strings; avoid command substitutions in tight loops; and avoid reading a file repeatedly. Use `find` with plus-terminated `exec` or `xargs` to batch arguments rather than one process per file, and use `xargs -P` or GNU `parallel` for parallelism when work is independent. For network or I/O-bound steps, batch requests and reuse connections, such as one `curl` invocation handling many URLs. To locate the hotspot, enable tracing with `set -x` and a PS4 containing timestamps, using the `EPOCHREALTIME` variable, or time sections with `SECONDS`; `strace -c -f` counts syscalls and forks.

Likely follow-ups

- Why is a loop calling external commands so slow?
- How do you find the slow section of a script?

```
time ./script.sh                # wall/user/sys time overview
```

```
PS4='+ $(date +%s.%N) ${LINENO}: ' ; set -x    # timestamped trace
```

```
# common fix: avoid spawning a process per line
```

```
# SLOW: while read l; do echo "$l" | grep x; done    (fork per line)
```

```
# FAST: do it in ONE awk/grep pass over the whole file
```

```
awk '/x/{...}' file
```

Shell scripts are usually slow because they fork a new process per iteration (calling `grep/sed/cut` inside a loop over thousands of lines). `time` gives the overview; a timestamped PS4 trace shows where time goes. The big win is almost always replacing per-line external commands with a single pass through `awk` (or moving to Python) — turning thousands of `fork+exec` calls into one.

The rule: if you're calling an external command inside a loop body, you can usually replace the whole loop with one `awk` invocation that's 100x faster. Process spawning is the hidden cost.

 Official docs

- [GNU Bash reference manual](#)

Q132 How do you handle signals and ensure children are cleaned up (no orphans)?

Bash / Shell

 Spoken answer · ~86 s

Orphans arise when a script exits or is killed while its background children keep running. To avoid them, track children and clean up on exit. Record child PIDs when launching background jobs, using dollar bang. Register a trap on EXIT, plus INT and TERM, that kills the recorded children and waits for them. A common idiom: in the trap handler, call kill on the job PIDs from the jobs builtin with the -p option, or kill the whole process group. Using process groups: start the script as a group leader, with setsid or set -m job control, then kill the whole group with a negative PID, which also reaches grandchildren. Because signals sent to the script do not automatically propagate to children, forwarding is required. When the script waits on a foreground child, Bash defers trap execution until that child exits, so run the child in the background and use wait, which is interruptible by signals; then the trap runs promptly. In containers, the PID 1 problem matters: PID 1 doesn't get default signal handling and must reap zombies, so use an init process such as tini or Docker's init flag, or exec the main program. Use exec for the last command, so no wrapper remains.

Likely follow-ups

- Why do traps wait for a foreground child to finish?
- How does a negative PID in kill reach grandchildren?

```
child_pids=()

cleanup() { kill "${child_pids[@]}" 2>/dev/null; wait; }

trap cleanup EXIT INT TERM

long_task & child_pids+=($!)      # record each child PID

another_task & child_pids+=($!)

wait
```

When a script spawns background children and is then killed, the children can be orphaned and keep running. Track each child's PID (\$! after &), and in a trap on EXIT/INT/TERM, kill them and wait for cleanup. This guarantees that interrupting the parent tears down the whole process tree — important for scripts that launch workers, port-forwards, or tunnels you don't want leaking.

Orphaned port-forwards/tunnels are a classic leak: a script that backgrounds `kubectl port -forward` and exits without killing it leaves a dangling process holding the port. Trap + tracked PIDs fix it.

 **Start here (official docs)**

- [Python documentation](#)

Q133 How do you correctly quote a variable used inside a command run over SSH?

Bash / Shell

Spoken answer · ~82 s

Commands run over SSH are parsed twice: once by your local shell, and once by the remote shell. So quoting that is right locally may break remotely, and variables may expand on the wrong side. Key question: where should each variable be expanded? If a variable should expand locally, so the remote receives its value, but the value may contain spaces or special characters, you must quote it for the remote shell. A robust approach is to use `printf` with the percent-q format, or Bash's at-Q expansion, to produce a shell-quoted version of the value that is safe to embed in the remote command string. If the remote command should use remote variables, wrap the command in single quotes locally so the local shell doesn't expand them. Another safer pattern is passing the script on standard input: `ssh host bash -s` with arguments, and a here-document with a quoted delimiter, so nothing is expanded locally; positional parameters pass values explicitly. Also consider sending data through environment variables with `SendEnv` and `AcceptEnv`, though usually restricted by server configuration. Never build the remote command by naive concatenation of untrusted input, because that creates command injection on the remote host.

Likely follow-ups

- Why is a command parsed twice with `ssh`?
- What does `printf %q` do?

```
# The remote shell re-parses the command -> double quoting needed
```

```
host=server1; path='/var/log/my app'
```

```
# BAD: remote splits on the space in the path
```

```
ssh "$host" "ls $path"
```

```
# GOOD: let printf %q produce a remote-safe quoted string
```

```
ssh "$host" "ls $(printf '%q' "$path")"
```

SSH concatenates its arguments and runs them through the *remote* shell, which re-parses and re-splits the string — so local quoting alone isn't enough; the value must survive a second round of shell parsing. `printf '%q'` produces a string safely re-quoted for shell input, so a path with spaces or special characters is interpreted literally on the remote side. This double-quoting subtlety is a frequent source of remote-command bugs.

Remember the two shells: your local shell parses once, the remote shell parses again. `printf %q` (or here-doc with quoted delimiter) is how you make values survive the second parse intact.

 **Start here (official docs)**

- [Python documentation](#)

Q134 How do you build a robust deploy/rollback shell function with state verification?

Bash / Shell

 Spoken answer · ~84 s

A robust deploy function is built around explicit state, verification, and a defined way back. Before deploying, record the current version, from the live system, not a file that may be stale, so rollback knows its target. Pre-flight checks: validate inputs, confirm the artefact exists and passes checks, ensure the target environment is healthy and nobody else is deploying, using a lock, and check capacity. Deploy step: apply the change in a controlled manner, using rolling batches or switching a symlink or traffic pointer atomically, rather than editing in place. Verify after: poll health endpoints with a timeout, require several consecutive passes, confirm the running version matches the one intended, and check key metrics, such as error rate and latency, against a baseline for a settle period. If verification fails or times out, roll back automatically: redeploy the recorded previous version, verify again, and exit non-zero with diagnostics, so the pipeline fails visibly. Implement with functions for deploy, verify, and rollback, a trap so interruption mid-deploy triggers safe handling, set -euo pipefail, and careful handling of the case where rollback itself fails, which should alert loudly. Log each step with timestamps, support a dry-run, and make operations idempotent so a rerun works.

Likely follow-ups

- Why record the previous version before deploying?
- What if the rollback itself fails?

```
deploy() {  
    local prev; prev=$(current_version) || return 1  
    if ! apply_new_version; then  
        echo 'apply failed' >&2; return 1  
    fi  
    if ! verify_healthy; then  
        echo 'health check failed, rolling back' >&2  
        rollback_to "$prev"; return 1  
    fi
```

```
echo "deployed, previous was $prev"
```

```
}
```

A safe deploy function captures the current state *before* changing anything (so rollback has a target), applies the change, then verifies health — and automatically rolls back to the captured previous version if verification fails. This 'capture → apply → verify → auto-rollback' structure encodes the SRE deploy reflex: never leave the system in a broken half-deployed state, and restore service first, investigate later.

Capture-before-change is the key: recording prev up front is what makes rollback possible. A deploy that mutates first and figures out rollback later has no safe state to return to.

Official docs

- [Kubernetes: Deployments](#)

Q135 When and how should you use 'trap DEBUG' or 'set -T' for advanced debugging?

Bash / Shell

Spoken answer · ~87 s

The DEBUG trap runs a specified command before every simple command executes, which allows stepping, tracing, and inspecting state. Used with set -T, or functrace, the trap is inherited by functions and command substitutions. Uses: print each command and line number as it runs, via BASH_COMMAND and LINENO; build a poor man's step debugger that pauses before each command with a read prompt, letting you inspect variables with declare -p and continue or abort; log variable changes by comparing values before each command; or measure the time spent per line by recording timestamps in the trap. With shopt -s extdebug, a non-zero return from the DEBUG trap can skip the command, enabling real debugger behaviour, which is what bashdb builds on. How to use it in practice: add trap with a handler on DEBUG near the section being investigated, and remove it with trap minus DEBUG afterwards. Costs and cautions: it fires for every command, slowing the script dramatically, so use it only temporarily and narrowly; it makes output noisy; it interacts with other traps and set -e in subtle ways; and it is Bash-specific. In most cases, simpler tools are enough: set -x with a customised PS4, ERR trap for the failure location, and shellcheck for static mistakes.

Likely follow-ups

- What does extdebug add?
- Why is DEBUG trap expensive?

```
# print every command with its line number before it runs
```

```
trap 'echo "[L$LINENO] $BASH_COMMAND" >&2' DEBUG
```

```
# time each command (rough):
```

```
trap 'echo "$(date +%s.%N) $BASH_COMMAND"' DEBUG
```

```
set -T      # make DEBUG/RETURN traps inherit into functions
```

A DEBUG trap runs before *every* command, giving finer-grained tracing than `set -x` — you can log line numbers, time individual commands, or inspect state at each step. `set -T` makes the DEBUG and RETURN traps propagate into functions (analogous to how `-E` propagates ERR). This is a heavier debugging tool for when `set -x` and `shellcheck` haven't found a subtle ordering or state bug.

Use sparingly: DEBUG traps fire constantly and slow execution, so reserve them for isolating a tricky bug — then remove. For everyday debugging, `set -x + shellcheck` cover most cases.

 **Start here (official docs)**

- [Python documentation](#)

Q136 How does context.Context work, and why is it essential in Go services?

Go

 Spoken answer · ~85 s

The context package carries cancellation signals, deadlines, and request-scoped values across API boundaries and between goroutines. A Context is an immutable interface with a Done channel that closes when the context is cancelled or times out, an Err method explaining why, a Deadline, and Value for scoped data. You derive contexts: WithCancel returns a child and a cancel function; WithTimeout and WithDeadline cancel automatically at a time; cancelling a parent cancels all descendants. It is essential in services because it solves resource leaks and wasted work: when a client disconnects or a request times out, the handler's context is cancelled, and downstream calls, database queries, and goroutines can stop instead of continuing for a response nobody will read. That prevents goroutine leaks and cascading overload, and lets you enforce end-to-end deadlines across microservices. Conventions: pass ctx as the first parameter of functions that do I/O or may block; never store a context in a struct; always call the cancel function, typically with defer, to release resources; check the Done channel in a select within long loops; use the context-aware variants of standard library calls, like `http.NewRequestWithContext` and `db.QueryContext`; and use context values sparingly for request-scoped data such as trace IDs, never for optional parameters.

Likely follow-ups

- Why must you call the cancel function?
- What should and shouldn't go in context values?

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

```
defer cancel()                                // always cancel to free resources
```

```
req, _ := http.NewRequestWithContext(ctx, 'GET', url, nil)
```

```
resp, err := http.DefaultClient.Do(req) // aborts if ctx times out
```

```
// propagate ctx down the call chain so cancellation flows everywhere
```

```
func work(ctx context.Context) error { select { case <-ctx.Done(): return ctx.Err(); ... } }
```

`context.Context` carries deadlines, cancellation signals, and request-scoped values across API boundaries and goroutines. It's how you implement timeouts and propagate cancellation: when a request is cancelled or times out, every downstream call watching the context (`ctx.Done()`) stops, freeing goroutines and connections. Passing `ctx` as the first parameter through your call chain is idiomatic Go and the foundation of well-behaved services.

Why SREs care: without context propagation, a cancelled client request leaves backend goroutines and DB queries running — leaking resources under load. Context is how cancellation flows end-to-end so work stops when the caller gives up.

Official docs

- [Go: context package](#)

Q137 Explain Go's race detector and how you find concurrency bugs.

Go

Spoken answer · ~90 s

A data race occurs when two goroutines access the same variable concurrently, at least one is writing, and there is no synchronisation. The result is undefined behaviour: corrupted data, crashes, or rare, hard-to-reproduce bugs. Go's race detector is built on ThreadSanitizer. You enable it by adding the race flag to `go test`, `go run`, or `go build`. The instrumented program tracks memory accesses and synchronisation events, and when it observes unsynchronised conflicting accesses, it prints a report showing both goroutines' stack traces, the accessed location, and where each goroutine was created. Important properties: it detects races that actually happen during that execution, not all possible ones, so coverage depends on tests exercising concurrent paths; it has no false positives, so every report is real; and it adds roughly 5 to 10 times CPU and memory overhead, so use it in tests and CI, and occasionally in staging or canary builds, not normally in production. Practice: run all tests with race in CI, write tests that exercise concurrent code in parallel with many iterations, and fix by choosing the appropriate synchronisation: mutexes, channels, atomic operations, or confining data to one goroutine. Other tools: `go vet`'s copylocks check, which finds copied mutexes; deadlock detection through goroutine dumps; and `pprof` goroutine profiles. Design to minimise shared mutable state.

Likely follow-ups

- Does the race detector have false positives?
- Why not run it in production?

```
go test -race ./... // run tests with the race detector
```

```
go run -race main.go // or run the program with it
```

```
// it instruments memory access and reports data races at runtime:
```

```
// WARNING: DATA RACE
```

```
// Read at 0x... by goroutine 7 / Write at 0x... by goroutine 8
```

A data race is two goroutines accessing the same memory concurrently with at least one write and no synchronization — undefined behavior that manifests as intermittent corruption. Go's built-in race detector (-race) instruments memory accesses and reports the exact conflicting reads/writes with stack traces when they occur. Run your tests and integration suites with -race in CI; it catches bugs that are nearly impossible to find by inspection.

It only finds races that actually execute: the detector is dynamic, so you need test coverage that exercises the concurrent paths. A race in an untested code path won't be caught — run race-enabled tests with realistic concurrency.

 **Start here (official docs)**

- [Python documentation](#)

Q138 How do you synchronize goroutines correctly — channels vs sync.Mutex vs sync.WaitGroup?

Go

 Spoken answer · ~84 s

Choose by what you're coordinating. A WaitGroup waits for a group of goroutines to finish: add before launching, done when each completes, wait to block until all are done. It coordinates completion; it doesn't pass data or protect shared state. A Mutex protects shared memory: lock around the critical section, unlock afterwards, usually with defer; use RWMutex when reads vastly outnumber writes. It suits protecting a cache, a counter map, or other state accessed by many goroutines, and is simple and fast for that. Channels transfer ownership of data and signal events between goroutines: pipelines, worker pools, fan-in and fan-out, cancellation signals, and producer-consumer queues. The Go proverb: do not communicate by sharing memory; share memory by communicating. Practical guidance: if goroutines are passing work or results, use channels; if multiple goroutines read and write a shared structure, use a mutex; if you need to wait for completion, use a WaitGroup, or errgroup when errors and cancellation matter; for simple counters, use atomic operations. Avoid common mistakes: copying a mutex, holding locks across blocking calls, forgetting to unlock on early return, sending on a closed channel, closing a channel from the receiver, and deadlocks from inconsistent lock ordering. Keep critical sections small.

Likely follow-ups

- When is a mutex clearer than a channel?
- What does errgroup add to WaitGroup?

```
var mu sync.Mutex; mu.Lock(); shared++; mu.Unlock() // protect shared state

var wg sync.WaitGroup

for _, j := range jobs { wg.Add(1); go func(j Job){ defer wg.Done(); do(j) }(j) }

wg.Wait() // wait for all to finish

ch := make(chan int); go func(){ ch <- compute() }(); v := <-ch // pass data
```

Pick the tool by intent. **Channels** pass data/ownership between goroutines ('share by communicating') — best for pipelines and handing off results. **sync.Mutex** protects shared mutable state accessed by multiple goroutines (a counter, a map). **sync.WaitGroup** waits for a set of goroutines to complete. Use a mutex when goroutines genuinely share memory; use channels when you can pass ownership instead — fewer locks means fewer deadlocks.

Idiom: 'Don't communicate by sharing memory; share memory by communicating.' Prefer channels for coordination, reserve mutexes for genuinely shared state like a concurrent counter or cache.

Official docs

- [Effective Go](#)
- [Go concurrency patterns: Pipelines and cancellation](#)

Q139 What are Go interfaces, and how do they enable testable, decoupled code?

Go

Spoken answer · ~73 s

A Go interface is a set of method signatures, and a type satisfies it implicitly by implementing those methods, with no explicit declaration of intent. That structural typing is central: the consumer defines the interface it needs, small and focused, and any type that has the methods works, even types from other packages that never knew about the interface. Go idiom: accept interfaces, return concrete types, and keep interfaces small, like `io.Reader` with a single method. This enables testable, decoupled code. A function that needs to fetch data takes an interface with just the needed method, instead of a concrete database client. In production, pass the real client; in tests, pass a small fake or stub implementing the interface, returning canned data or errors, so tests run fast without network or database and can simulate failures that are hard to trigger for real. It decouples high-level logic from low-level details, such as swapping a cloud provider, storage backend, or clock. Examples: injecting a `Clock` interface to test time-dependent logic, or an interface around the Kubernetes client.

Likely follow-ups

- Why define interfaces on the consumer side?
- What is the nil interface gotcha?

```
type Store interface {           // small interface = the dependency

    Get(ctx context.Context, k string) ([]byte, error)

}

func Handler(s Store) { ... }    // depends on the interface, not a concrete type

// real: type RedisStore struct{}; func (r RedisStore) Get(...){...}

// test: type fakeStore struct{}; func (f fakeStore) Get(...){ return ..., nil }
```

Go interfaces are satisfied implicitly — any type with the right methods implements them, no `implements` keyword. Depending on a small interface (rather than a concrete type) decouples your code from implementations: production code injects the real one, tests inject a fake/mock. Idiomatic Go favors small interfaces (often one method) defined where they're *used*, not where the implementation lives — 'accept interfaces, return structs.'

Accept interfaces, return structs: functions take narrow interfaces (so callers can substitute fakes in tests), but

return concrete types (so callers get full functionality). This is the core of testable Go design.

Official docs

- [Effective Go](#)
- [The Go Programming Language documentation](#)

Q140 How do you avoid goroutine leaks, and how do you detect them?

Go

Spoken answer · ~85 s

A goroutine leak is a goroutine that never exits, usually blocked forever, holding memory and resources; over time they accumulate until the process runs out of memory or slows down. Common causes: sending or receiving on a channel that no one will ever service, such as a worker blocked sending a result after the consumer returned early; a missing channel close, so a range loop never ends; waiting on a WaitGroup that never reaches zero; a goroutine waiting on a network call without a timeout; lack of cancellation handling in loops; and tickers never stopped. Prevention. Give every goroutine a clear exit path and an owner who knows how it ends. Pass a context and select on its Done channel in any blocking operation. Use buffered channels sized appropriately for early-return cases, or select with a Done case when sending. Close channels from the sender when done. Always set timeouts on I/O. Stop timers and tickers with defer. Use errgroup, which manages cancellation and waiting. Detection. Monitor the goroutine count as a metric, using runtime.NumGoroutine or the process collector, and alert on steady growth. Capture a goroutine profile through pprof's debug endpoint, which lists goroutine stacks grouped by location, making the blocked ones obvious.

Likely follow-ups

- How does pprof reveal leaked goroutines?
- What does goleak do in tests?

```
// LEAK: goroutine blocks forever on a channel no one reads/closes
```

```
go func(){ ch <- result }()    // if nothing ever receives, it leaks
```

```
// FIX: use context cancellation or a buffered/closed channel
```

```
go func(){ select { case ch <- result: case <-ctx.Done(): } }()
```

```
// detect: runtime.NumGoroutine(), or goleak in tests
```

```
defer goleak.VerifyNone(t)    // uber-go/goleak fails test on leaks
```

A goroutine leaks when it blocks forever — typically waiting to send/receive on a channel that's never read/closed, or looping without a cancellation path. Leaked goroutines accumulate, consuming memory and eventually degrading the service. Prevent it by giving every goroutine a way out: a select with ctx.Done(), a

closed channel, or a buffered channel. Detect leaks with `runtime.NumGoroutine()` monitoring or `uber-go/goleak` in tests.

The silent killer under load: a handler that spawns a goroutine per request which can block forever will slowly exhaust memory. Every goroutine needs a guaranteed termination path — usually via context.

Official docs

- [Effective Go](#)
- [Go concurrency patterns: Pipelines and cancellation](#)

Q141 What is the difference between value and pointer receivers, and when does it matter?

Go

Spoken answer · ~83 s

A method receiver can be a value or a pointer. With a value receiver, the method gets a copy of the value, so modifications inside don't affect the caller's variable. With a pointer receiver, the method gets the address, so it can modify the original and avoids copying large structs. When it matters. Mutation: if the method needs to change the receiver's state, it must use a pointer receiver. Copy cost: large structs are expensive to copy on every call, favouring pointers. Consistency: if any method needs a pointer receiver, make all of the type's methods pointer receivers, as mixing is confusing, and the Go style guidance recommends it. Types containing a mutex or other no-copy fields must use pointer receivers, since copying breaks them. Interface satisfaction: the method set of a value type includes only value-receiver methods, while the method set of the pointer type includes both. So if the interface methods are defined on pointer receivers, only a pointer to the type satisfies the interface, and assigning a plain value gives a compile error. Maps, slices, and channels are reference-like but their headers are copied, so appending inside a value-receiver method may not be visible to the caller.

Likely follow-ups

- Why does the method set differ between `T` and `*T`?
- When are value receivers appropriate?

```
type Counter struct{ n int }
```

```
func (c Counter) Get() int { return c.n } // value: gets a COPY
```

```
func (c *Counter) Inc() { c.n++ } // pointer: mutates original
```

```
// value receiver on a big struct copies it every call (cost)
```

```
// pointer receiver needed to MUTATE, and for sync.Mutex fields (don't copy a lock!)
```

A value receiver operates on a copy, so it can't mutate the original and copies the whole struct each call. A pointer receiver shares the original, enabling mutation and avoiding copies of large structs. Rules of thumb: use pointer receivers if the method mutates, if the struct is large, or if it contains a `sync.Mutex` (copying a locked mutex is a bug). Keep receiver types consistent across a type's methods.

Copying a mutex is a real bug: a value receiver on a struct containing a `sync.Mutex` copies the lock, so the copy guards nothing. `go vet` flags this. Use pointer receivers for any struct holding a lock.

 **Start here (official docs)**

- [Python documentation](#)

Q142 How do you write table-driven tests in Go (the idiomatic pattern)?

Go

 Spoken answer · ~88 s

Table-driven tests are the idiomatic Go pattern for testing many cases with one test function. You define a slice of anonymous structs, each holding a descriptive name, the inputs, and the expected output or error. Then loop over the cases, and for each, call `t.Run` with the case name, which creates a subtest. Inside, call the function under test and compare the result with the expected value, reporting differences with `t.Errorf` that include the input, the actual, and the expected values, or use a comparison helper such as `cmp.Diff` from `go-cmp`, which produces readable diffs. Benefits: adding a case is a single line, so it encourages thorough coverage of edge cases; the cases are visible in one table; subtests give individual names in output, can be run individually with the `run` flag, and can be parallelised with `t.Parallel`. For parallel subtests in older Go versions, capture the loop variable correctly. Include edge cases: empty input, nil, boundaries, invalid input, and error paths, with a field for whether an error is expected, or for the expected error. Keep each test case independent, avoid shared mutable state, and use helper functions marked with `t.Helper` for common setup. Combine with fakes for dependencies, golden files for large outputs, and fuzz tests for unexpected inputs.

Likely follow-ups

- What do subtests give you?
- How do you run a single case?

```
func TestParse(t *testing.T) {  
  
    tests := []struct{ name, in string; want int; wantErr bool }{  
  
        {'valid', '200', 200, false},  
  
        {'bad', 'xx', 0, true},  
  
    }
```

```

for _, tt := range tests {

    t.Run(tt.name, func(t *testing.T) {

        got, err := Parse(tt.in)

        if (err != nil) != tt.wantErr { t.Fatalf('err = %v', err) }

        if got != tt.want { t.Errorf('got %d want %d', got, tt.want) }

    })

}

}

```

Table-driven tests define a slice of test cases (input + expected output) and loop over them, running each as a subtest with `t.Run`. This is the idiomatic Go testing style: adding a case is one line, each subtest reports independently (and can run in parallel), and the structure forces you to enumerate edge cases explicitly. It scales far better than copy-pasted test functions.

Why it's standard: subtests give per-case names in output, failures pinpoint the exact case, and the table makes the set of behaviors you're verifying visible at a glance. It's how the Go standard library tests itself.

Start here (official docs)

- [Python documentation](#)

🗣 Spoken answer · ~89 s

Go's `encoding/json` maps structs to JSON using field tags. Marshalling: export fields with capital letters, and use tags to set JSON names, omit empty values with `omitempty`, or skip fields with a dash. Unmarshalling is where robustness matters. Always check the returned error. Be strict where needed: use a `json.Decoder`, and call `DisallowUnknownFields` to reject unexpected keys, catching typos in configuration; use `Decoder.More` or check for trailing data after the first value; and limit input size with `http.MaxBytesReader` to prevent memory abuse. Understand type behaviours: numbers decode into `float64` when the target is `interface{}`, losing precision for large integers, so use `UseNumber` or typed structs; missing fields leave zero values, so you cannot distinguish absent from zero unless you use pointer fields or custom types; null handling similarly needs pointers or `sql.Null`-style types. Use `time.Time` with RFC 3339 formatting, and custom `UnmarshalJSON` or `MarshalJSON` methods for special formats, like durations given as strings. Use `json.RawMessage` to delay decoding polymorphic parts, based on a type field. Validate after decoding: required fields, ranges, and allowed values, since JSON decoding doesn't enforce them; libraries like `go-playground validator` help. For large streams, use `Decoder` to read values one at a time instead of loading everything. Handle HTML escaping if relevant. Write tests with malformed and edge-case payloads.

Likely follow-ups

- How do you distinguish a missing field from a zero value?
- Why use `DisallowUnknownFields`?

```
type Pod struct {  
    Name    string `json:'name'`  
    Status string `json:'status,omitempty'` // omit if empty  
    Labels map[string]string `json:'labels,omitempty'`  
}  
  
var p Pod  
  
if err := json.Unmarshal(data, &p); err != nil { return err } // always check!  
  
out, _ := json.Marshal(p)  
  
// unknown/dynamic shape: json.RawMessage or map[string]any
```

Struct tags (`json:'name'`) map Go fields to JSON keys; `omitempty` drops zero-valued fields from output. Always check the `Unmarshal` error — malformed JSON is common with external data. For unknown or dynamic structures, unmarshal into `map[string]any` or defer parsing with `json.RawMessage`. Be aware that

absent fields become Go zero values, so you sometimes need a pointer (*int) to distinguish 'absent' from 'zero'.

Zero vs absent: a missing JSON number unmarshals to 0, indistinguishable from an explicit 0. Use a pointer field (*int) when you must tell 'not provided' from 'provided as zero' — a frequent config-parsing subtlety.

 **Start here (official docs)**

- [Python documentation](#)

Q144 What does 'go vet' and static analysis catch, and how do you wire up linting?

Go

 **Spoken answer** · ~85 s

The go vet tool performs static analysis for suspicious constructs that compile fine but are likely bugs. It catches printf-style format string mismatches, copying of locks by value, unreachable code, misuse of struct tags, incorrect use of sync primitives, loop variable capture in closures, unkeyed struct literal issues, impossible comparisons, and wrong use of defer in loops for some patterns. It ships with Go, and go test runs a subset automatically. Beyond vet, linters find more: staticcheck detects deprecated API use, unused code, inefficient constructs, and many bug patterns; errcheck flags ignored error returns, a major source of production bugs; gosec looks for security issues like weak crypto or injectable commands; ineffassign and unused find dead assignments and code; and gofmt or goimports enforce formatting. The golangci-lint tool runs many linters together quickly with caching and a shared configuration file, where you pick which linters to enable, set exclusions, and configure thresholds. Wiring: add a configuration file to the repository; run the linter in CI on every pull request, failing the build on findings, ideally only reporting issues in new code to ease adoption on legacy projects; run it locally through pre-commit hooks or editor integration for fast feedback; pin the tool version for reproducibility.

Likely follow-ups

- Why is errcheck valuable?
- How does golangci-lint fit into CI?

```
go vet ./...           // built-in: suspicious constructs
```

```
golangci-lint run      // aggregates many linters (staticcheck, etc.)
```

```
// catches: printf format mismatches, copied mutexes, unreachable code,
```

```
//          shadowed err, unchecked errors, nil-map writes, ...
```

```
gofmt -l . / goimports // formatting + import hygiene
```

go vet catches correctness issues the compiler allows but are almost always bugs: Printf format/argument mismatches, copied locks, unreachable code, struct-tag typos. golangci-lint bundles many linters (staticcheck, errcheck, ineffassign) into one fast pass. Run both in CI alongside gofmt/goimports for

consistent formatting. This catches a class of bugs before code review and keeps a codebase uniform across a team.

CI gate: `go vet + golangci-lint + gofmt -l` (fail if anything's unformatted) as required checks keeps Go code correct and consistent — the equivalent of shellcheck for Bash or mypy for Python.



Start here (official docs)

- [Python documentation](#)

Q145 How do you implement graceful shutdown of a Go HTTP server?

Go

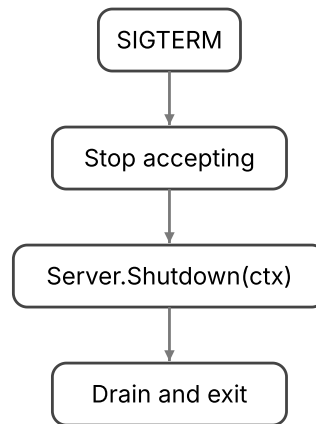
Spoken answer · ~88 s

Graceful shutdown lets in-flight requests finish before the process exits, which matters during deployments and scaling, so users don't see errors. In Go, create the `http.Server` explicitly, start it in a goroutine calling `ListenAndServe`, and treat `http.ErrServerClosed` as the normal result, not a failure. In the main goroutine, set up signal handling with `signal.NotifyContext` for `SIGINT` and `SIGTERM`, which Kubernetes sends on pod termination, and wait for it. When the signal arrives, create a context with a timeout, say 25 seconds, shorter than the platform's termination grace period, and call `server.Shutdown` with it. Shutdown stops accepting new connections, closes idle ones, and waits for active requests to complete; if the timeout expires, it returns an error, and you can call `Close` to force termination. Then clean up other resources in order: stop background workers, flush logs and metrics, close database connections, and finish. Additional practices: make the readiness probe start failing as soon as shutdown begins, and wait a few seconds so the load balancer and service endpoints stop sending new traffic before closing listeners, avoiding a race that causes connection errors; set server read, write, and idle timeouts; use the request context in handlers, so long operations can be cancelled; and handle long-lived connections like WebSockets explicitly, with `RegisterOnShutdown`.

Likely follow-ups

- Why should the shutdown timeout be shorter than the grace period?
- Why fail readiness before closing the listener?

🚩 Go graceful shutdown



```
srv := &http.Server{Addr: ':8080', Handler: mux}
```

```
go func(){ srv.ListenAndServe() }()
```

```
stop := make(chan os.Signal, 1)
```

```
signal.Notify(stop, syscall.SIGINT, syscall.SIGTERM)
```

```
<-stop // wait for signal
```

```
ctx, cancel := context.WithTimeout(context.Background(), 30*time.Second)
```

```
defer cancel()
```

```
srv.Shutdown(ctx) // stop accepting, drain in-flight
```

Graceful shutdown stops accepting new connections while letting in-flight requests finish within a deadline — critical in Kubernetes, which sends SIGTERM before SIGKILL. You catch the signal, then call `srv.Shutdown(ctx)`, which closes listeners and waits for active handlers to complete (up to the context timeout) before returning. Without this, a rolling deploy drops in-flight requests, causing user-visible errors during every deploy.

Match the grace period: the shutdown timeout should fit within Kubernetes' `terminationGracePeriodSeconds`, or the pod gets SIGKILLED mid-drain. Combine with a readiness probe flipping to not-ready so traffic stops arriving first.

📖 Official docs

- [Kubernetes: Pod lifecycle & termination](#)
- [Go: net/http](#)

Spoken answer · ~80 s

Go is efficient, but several pitfalls cause memory and performance problems in production. Memory: goroutine leaks, covered earlier, are the top cause of steady growth; slices holding references to a large backing array when you keep only a small sub-slice, preventing the garbage collector from freeing the rest, fixed by copying; maps never shrink after deletions, and unbounded maps used as caches grow forever; string and byte slice conversions allocate; and large numbers of small allocations pressure the garbage collector. The GC: high allocation rates increase GC frequency and latency; mitigate by reducing allocations, reusing buffers with `sync.Pool`, preallocating slices and maps with known capacity, and avoiding unnecessary pointers, which GC must scan. Tune `GOGC` and, in containers, set `GOMEMLIMIT` so the runtime respects the container's memory limit; and set `GOMAXPROCS` appropriately, since older Go versions saw the host's CPU count rather than the container's CPU limit, causing throttling, fixed by `automaxprocs` or by newer runtime versions. Other pitfalls: `defer` in hot loops, excessive use of `fmt` and reflection, lock contention on shared mutexes, unbuffered channels causing goroutine pile-ups, and HTTP clients without timeouts or response bodies not being closed, which leaks connections.

Likely follow-ups

- Why can a small sub-slice keep a huge array alive?
- Why does `GOMEMLIMIT` matter in containers?

```
// 1. slice holding a huge backing array alive:
```

```
small := big[:10]           // still references big's full array -> copy if needed
```

```
// 2. appending in a loop without pre-allocating:
```

```
s := make([]T, 0, n)        // pre-size to avoid repeated reallocations
```

```
// 3. string concatenation in a loop -> use strings.Builder
```

```
// 4. defer in a hot loop -> accumulates until function returns
```

Key Go performance traps: a small slice of a large array keeps the *whole* backing array in memory (copy the sub-slice to release it); growing a slice without pre-allocating capacity causes repeated reallocations and copies; concatenating strings with `+` in a loop is $O(n^2)$ (use `strings.Builder`); and `defer` inside a tight loop accumulates deferred calls until the function returns (leaking resources). Profile with `pprof` to find the real hot spots.

pprof is built in: `import _ 'net/http/pprof'` exposes CPU/heap/goroutine profiles over HTTP — the standard way to find allocation hot spots and goroutine leaks in a running Go service.

Q147 How do you structure a Go project and manage dependencies with modules?

Go

🗣 Spoken answer · ~85 s

A Go project is organised as a module, defined by a `go.mod` file at the root, declaring the module path and Go version, and listing dependencies with required versions. `go.sum` records cryptographic checksums, so downloads are verified; commit both. Create it with `go mod init`, add dependencies simply by importing them and running `go mod tidy`, which adds missing and removes unused requirements. Go uses semantic versioning with minimal version selection, which picks the minimum version satisfying all requirements, making builds deterministic and predictable. Major versions 2 and above include the version in the import path. Update with `go get`, inspect with `go list` and `go mod graph`, and use `go mod vendor` if you need a vendored copy for offline builds. Private modules need `GOPRIVATE` and credentials. Common layout conventions: a `cmd` directory with one subdirectory per executable containing a small main package; an `internal` directory for code that must not be imported by other modules, enforced by the compiler; packages named by purpose, small and focused, avoiding utility dumping grounds; `pkg` for importable libraries if you publish; tests alongside code in `_test.go` files; and a `Makefile` or `Taskfile` for build, test, and lint commands. Keep main thin, wiring dependencies and calling testable code.

Likely follow-ups

- What does `go mod tidy` do?
- What does the `internal` directory enforce?

```
go mod init github.com/org/tool // creates go.mod
```

```
go get github.com/x/y@v1.2.3 // add a pinned dependency
```

```
go mod tidy // prune unused, add missing
```

```
// layout:
```

```
// cmd/tool/main.go    entrypoint(s)
```

```
// internal/...        private packages (un-importable externally)
```

```
// pkg/...             public reusable packages
```

Go modules (`go.mod/go.sum`) pin dependencies with versions and checksums for reproducible, verifiable builds — commit both files. `go mod tidy` keeps them accurate. The conventional layout puts entrypoints under `cmd/`, private packages under `internal/` (the compiler forbids external imports of `internal`), and

shared libraries under `pkg/.go.sum` hashes give supply-chain integrity — a tampered dependency fails verification.

internal/ is enforced: packages under `internal/` can only be imported by code in the same module subtree — a compiler-enforced boundary for keeping implementation details private, unlike Python's convention-only underscore prefix.

 **Start here (official docs)**

- [Python documentation](#)

Q148 How do you wrap and inspect errors with `errors.Is` / `errors.As` / `%w`?

Go

 Spoken answer · ~74 s

Go errors can be wrapped to add context while preserving the original for inspection. To wrap, use `fmt.Errorf` with the percent-`w` verb: for example, wrapping an error from opening a config file with a message about loading the configuration. The resulting error contains both the message and the underlying error, which can be retrieved by unwrapping. Each layer adds context, producing a readable chain like `loading config: opening file: no such file or directory`. To inspect, never compare error strings. Use `errors.Is` to check whether any error in the chain matches a particular target value, such as `os.ErrNotExist` or a package-level sentinel error like `ErrNotFound`; it walks the chain, comparing with equality or a custom `Is` method. Use `errors.As` to check whether any error in the chain has a particular type and extract it, passing a pointer to a variable of the target type, which then gives access to its fields, for example to read the status code from a custom HTTP error type. Types can implement an `Unwrap` method to participate, and `errors.Join` combines multiple errors.

Likely follow-ups

- What's the difference between `errors.Is` and `errors.As`?
- When would you use `%v` instead of `%w`?

```
var ErrNotFound = errors.New('not found')
```

```
func get(k string) error { return fmt.Errorf('get %s: %w', k, ErrNotFound) }
```

```
// inspect the chain:
```

```
if errors.Is(err, ErrNotFound) { /* handle 404 */ }
```

```
var perr *os.PathError
```

```
if errors.As(err, &perr) { log.Print(perr.Path) } // extract a typed error
```

Wrapping with `%w` in `fmt.Errorf` preserves the underlying error in a chain while adding context. `errors.Is` checks whether any error in the chain matches a sentinel (like `ErrNotFound`) — robust to wrapping, unlike `==`. `errors.As` extracts a specific error type from the chain so you can read its fields. This lets you add context at each layer yet still make decisions on the root cause at the top.

Wrap with context, match on cause: each layer adds `fmt.Errorf('doing X: %w', err)` so the final message is a breadcrumb trail, while `errors.Is/As` at the top still classifies the original failure correctly.

 **Start here (official docs)**

- [Python documentation](#)

Q149 How do you build a small concurrent CLI that respects timeouts and cancellation?

Go

 **Spoken answer** · ~81 s

A concurrent CLI that respects timeouts and cancellation is built on context. In main, create a root context with `signal.NotifyContext` for interrupt and termination signals, so Ctrl-C cancels everything, and derive a timeout context from the user's timeout flag with `context.WithTimeout`, deferring cancel. Parse flags with the `flag` package or `cobra`. Fan out the work with `errgroup.WithContext`, which gives a group and a context that is cancelled when any task returns an error or the parent is cancelled. Bound concurrency with `group.SetLimit`, or a semaphore channel, to avoid overwhelming targets. For each target, such as a list of hosts or endpoints, launch a task that uses the context in its operations: `http.NewRequestWithContext` for HTTP calls, a `net.Dialer`'s `DialContext`, or database calls with the context, plus a per-task timeout derived from the parent. Collect results safely: send them over a channel to a single collector goroutine, or write into a pre-sized slice by index, avoiding shared map writes. After waiting, print a summary in a stable order, human-readable by default with a JSON flag, send errors to `stderr`, and set the exit code: zero when everything succeeded, non-zero on failures, with a distinct code for timeouts.

Likely follow-ups

- What does `errgroup.WithContext` provide?
- How do you bound concurrency in the group?

```
ctx, cancel := signal.NotifyContext(context.Background(), os.Interrupt)
```

```
defer cancel() // Ctrl-C cancels the context
```

```
g, ctx := errgroup.WithContext(ctx) // golang.org/x/sync/errgroup
```

```
for _, u := range urls {
```

```
    u := u
```

```
g.Go(func() error { return check(ctx, u) }) // bounded fan-out
```

```
}
```

```
if err := g.Wait(); err != nil { log.Fatal(err) } // first error cancels rest
```

`signal.NotifyContext` ties Ctrl-C/SIGTERM to context cancellation, so an interrupt cleanly stops all work. `errgroup` runs a group of goroutines, propagates the first error, and cancels the shared context so siblings stop early — exactly what you want for fan-out tasks (health checks, parallel fetches). Each goroutine watches `ctx` for cancellation. This is the idiomatic pattern for a robust concurrent CLI tool.

errgroup is the go-to: it combines `WaitGroup` semantics with error propagation and context cancellation — the clean way to fan out work where any failure should abort the rest, common in SRE tooling.

Official docs

- [AWS Builders' Library: Timeouts, retries and backoff with jitter](#)

Q150 Live coding: write a Go function to find the top-K slowest requests from a stream. Go
Walk through it.

Spoken answer · ~83 s

The problem: from a stream of requests, each with an ID and a duration, report the K slowest without storing everything. The right structure is a min-heap of size K, ordered by duration, so the root is the smallest of the current top K. For each incoming request: if the heap has fewer than K items, push it; otherwise compare its duration with the root; if it's larger, replace the root and restore heap order; if not, ignore it. Each item costs $O(\log K)$, total $O(n \log K)$ time, and $O(K)$ memory, regardless of stream length. In Go, use the `container/heap` package: define a type implementing the heap interface, `Len`, `Less`, `Swap`, `Push`, and `Pop`, over a slice of requests, with `Less` comparing durations to make it a min-heap. After the stream ends, pop all items, which come out smallest first, and reverse them for descending order, or sort the slice. Details to discuss: ties and a stable tie-breaker; K larger than the number of items; concurrent producers, which need a mutex around the heap or a single consumer goroutine reading from a channel; malformed values; and parsing the stream with a `bufio.Scanner` with an enlarged buffer for long lines.

Likely follow-ups

- Why a min-heap rather than a max-heap?
- How would you handle multiple producers?

```
import 'container/heap'
```

```
// maintain a MIN-heap of size K; the root is the smallest of the top-K
```

```

type Req struct{ Path string; MS int }

// heap holds the K largest seen so far

func topK(reqs []Req, k int) []Req {

    h := &minHeap{}; heap.Init(h)

    for _, r := range reqs {

        if h.Len() < k { heap.Push(h, r) } else if r.MS > (*h)[0].MS {

            heap.Pop(h); heap.Push(h, r)    // evict smallest, add bigger

        }

    }

    return *h

}

```

The interview-grade structure: maintain a min-heap of size K . Push until it holds K items; thereafter, if a new request is slower than the heap's smallest (the root), evict the root and push the new one. This keeps only the top- K in $O(n \log K)$ time and $O(K)$ memory — never sorting or holding the whole stream, so it scales to an unbounded log. Walk through edge cases aloud: k larger than the input, ties, empty stream.

What earns the offer: choosing a size- K min-heap over sorting everything ($O(n \log K)$ vs $O(n \log n)$, $O(K)$ vs $O(n)$ memory), explaining *why* the min-heap root is the eviction candidate, and naming the streaming/constant-memory property — that's the SRE-grade answer.

Official docs

- [Python: collections](#)
- [Python: heapq](#)

Q151 **How do you use here-documents and here-strings, and when are they better than echo?**

Bash / Shell

 Spoken answer · ~65 s

A here-document passes a block of text to a command's standard input, and a here-string passes a single string. They are cleaner than chains of echo for writing config files, SQL scripts or multi-line messages, because quotes and newlines do not need escaping. The important detail is quoting the delimiter. With a bare delimiter, the shell expands variables and command substitutions inside the block, which is useful for templating. With a quoted delimiter such as 'EOF', the text is literal, which is what I want when generating scripts or YAML containing dollar signs. A dash before the delimiter strips leading tabs, so documents can be indented inside functions. Here-strings are handy for feeding a variable to grep or read without a pipe, which also avoids running the command in a subshell and losing variable assignments. I use here-documents with caution for secrets, since a command line is not involved but the file permissions still matter.

Likely follow-ups

- What does quoting the delimiter change?
- Why is a here-string better than echo piped to read?

A here-doc feeds multi-line text to a command's stdin; a here-string feeds one string. They avoid quoting and escaping problems with echo.

```
cat <<'EOF' > app.conf
port=8080
EOF
```

```
grep -c error <<< "$LOG"
```

Interview tip: Quote the delimiter to stop expansion.

 Start here (official docs)

- [Python documentation](#)

 Spoken answer · ~61 s

Bash has indexed arrays and, from version four, associative arrays declared with `declare -A`. Arrays are the correct way to hold a list of items, especially filenames or command arguments, because a plain string would be split on whitespace and break on spaces. The critical habit is quoting the expansion as `"${arr[@]}"`, which keeps each element as a separate word even when it contains spaces. Using `"${arr[*]}"` joins them into one string. I build command lines in arrays, for example `cmd=(kubectl get pods -n "$ns")` and run `"${cmd[@]}"`, which avoids eval and injection risks. Associative arrays are useful for lookups such as mapping environment names to endpoints. Remember that they are not portable to POSIX sh or the old Bash on some systems, that iteration order is not guaranteed, and that arrays cannot be exported to child processes. When data gets complicated, I switch to Python.

Likely follow-ups

- Why avoid eval for building commands?
- How do you get the length of an array?

Indexed arrays hold lists; associative arrays (`declare -A`) map keys to values. Always expand with quotes: `"${arr[@]}"`.

```
files=(a.txt 'b c.txt')
for f in "${files[@]"; do echo "$f"; done
```

```
declare -A port=([web]=80 [db]=5432)
echo "${port[db]}"
```

Interview tip: Quote `"${arr[@]}"` to preserve elements with spaces.

Official docs

- [GNU Bash reference manual](#)

Q153 **How does parameter expansion help with defaults, trimming and substitutions?**

Bash / Shell

 Spoken answer · ~58 s

Parameter expansion lets Bash manipulate variables without spawning sed, awk or cut, which is faster and avoids quoting problems. The default operators handle missing input: dash gives a fallback value, colon-equals assigns it, and colon-question aborts with an error message, which is a neat way to require a variable. Hash and percent remove the shortest matching prefix or suffix, and doubled versions remove the longest, so I can strip extensions or directory parts, mirroring basename and dirname. Slash substitutes patterns, with a double slash replacing all occurrences. There is also substring extraction, length with a hash prefix, and case conversion in Bash four and later. These are handy in loops over thousands of files, where forking an external command for each one would be slow. The limitation is readability, so I keep expansions simple and comment non-obvious ones.

Likely follow-ups

- How do you require a variable to be set?
- What is the difference between # and ##?

`${var:-default}`, `${var#prefix}`, `${var%suffix}` and `${var//a/b}` do common string work without external tools.

```
echo "${ENV:-dev}"
```

```
f=report.tar.gz; echo "${f%.tar.gz}"
```

```
echo "${path//\\/_}"
```

Interview tip: Prefer built-in expansion over spawning sed.

 **Start here (official docs)**

- [Python documentation](#)

Q154 **How do you read a file line by line in Bash without breaking on spaces or backslashes?**

Bash / Shell

 Spoken answer · ~61 s

The reliable idiom is `while IFS= read -r line`, with the file redirected into the loop. Setting IFS to empty prevents read from trimming leading and trailing whitespace, and the minus r option stops it from treating backslashes as escape characters. Without these, lines with indentation or backslashes get silently altered. Redirecting with a less-than sign at the end, instead of piping from cat, keeps the loop in the current shell, so variables set inside the loop are still available afterwards. With a pipe, the loop runs in a subshell and changes are lost. If the last line has no trailing newline, read returns failure but still fills the variable, so I add an or-test on the variable to handle it. For delimited data I set IFS to the delimiter and read into several variables. For big files, awk is much faster than a Bash loop.

Likely follow-ups

- Why does a piped loop lose variables?
- How do you handle a missing final newline?

Use `while IFS= read -r line; do ...; done < file`. IFS= keeps whitespace, -r keeps backslashes.

```
while IFS= read -r line; do printf '%s\n' "$line"; done < hosts.txt
```

```
while IFS=, read -r name ip; do echo "$name $ip"; done < list.csv
```

Interview tip: Redirect the file in, do not pipe, to keep variables.

 Official docs

- [GNU Bash reference manual](#)

 Spoken answer · ~61 s

I write a small log function that prints a UTC timestamp, a level and the message. Logs go to standard error and only the script's real output goes to standard out, so the output can be piped into other commands without being polluted. I define a verbosity variable so debug messages show only when requested, and include the script name or a run identifier to correlate lines. For machine-readable output I emit JSON lines with fields like level and message, which a log shipper can parse. If the script runs under cron or systemd, I make sure the logs reach the journal or a file with rotation. I also log the exit status in the cleanup trap and set error handling so failures produce a clear final message. For anything complex I would use Python's logging module instead, but for small tools this pattern is enough.

Likely follow-ups

- Why send logs to stderr?
- How would you make the logs machine-readable?

Write a small log function with levels and timestamps, send logs to stderr, and keep stdout for data.

```
log(){ printf '%s %s %s\n' "$(date -u +%FT%TZ)" "$1" "${*:2}" >&2; }
```

```
log INFO 'starting backup'
```

Interview tip: stderr for logs, stdout for results.

 Official docs

- [Python: logging](#)
- [OpenTelemetry: Logs](#)
- [GNU Bash reference manual](#)

Q156 **What are the common Bash pitfalls with set -e, and when does it not stop the script?**

Bash / Shell

 Spoken answer · ~60 s

The errexit option stops the script when a command fails, but it has surprising exceptions. It is ignored for commands that are part of an if or while condition, for commands followed by and- and or or-or, and for a failure on the left of a pipe unless pipefail is set. Combining local and an assignment on one line, such as `local x=$(cmd)`, hides the command's exit status because local itself succeeds. Functions called in a condition also run with errexit disabled, so failures inside them are ignored. Because of these rules, I do not rely on it for correctness. I use it as a safety net together with nounset and pipefail, check critical commands explicitly with an or-branch that logs and exits, and use an ERR trap to report the failing line. For complex logic, explicit error handling or a different language is more predictable.

Likely follow-ups

- Why does `local x=$(cmd)` hide failures?
- What does an ERR trap give you?

`set -e` is ignored in conditions, after `&&` or `||`, in command substitutions inside assignments with `local`, and in pipelines without `pipefail`.

```
set -Eeuo pipefail
```

```
local out; out=$(cmd) # separate declaration from assignment
```

Interview tip: Treat `set -e` as a safety net, not a design.

Official docs

- [ShellCheck wiki](#)
- [Google Style Guide: Shell](#)
- [GNU Bash reference manual](#)

🗣 Spoken answer · ~56 s

Bash functions can only return an integer status between zero and 255, which is meant for success or failure. To return data, a function prints it to standard output and the caller captures it with command substitution. Anything else the function prints, such as debug logging, must go to stderr, otherwise it contaminates the result. Command substitution runs in a subshell, so the function cannot modify the caller's variables that way. When I need to return several values or modify caller variables without a subshell, I use a nameref with local -n, or set a global result variable by convention. I declare function variables local to avoid leaking names. I also check the status of the captured call, since the assignment's status is that of the substitution only when not combined with local or export.

Likely follow-ups

- Why can't a function return a string?
- What is a nameref?

Return status with return, data by printing to stdout and capturing with \$(...), or via a nameref for multiple values.

```
get_ip(){ hostname -I | awk '{print $1}'; }  
ip=$(get_ip)
```

```
fill(){ local -n out=$1; out=(a b c); }
```

Interview tip: Return codes are 0-255 only.

📖 Official docs

- [GNU Bash reference manual](#)

 Spoken answer · ~59 s

Filenames can contain spaces, quotes and even newlines, so a plain pipe from find to xargs breaks. The safe pattern is find with print0, which separates names with a null byte, and xargs with the zero option, which reads them that way. I also add the no-run-if-empty option so xargs does not run the command with no arguments. Xargs batches many arguments into each invocation, which is far faster than running a command per file, and the dash P option runs several batches in parallel, with dash n controlling batch size. An alternative is find with exec and a plus sign, which also batches without needing a pipe. Before a destructive command like rm, I first run the pipeline with echo or ls to preview the matches. For complex per-file logic, I use a while read loop with null delimiters instead.

Likely follow-ups

- Why use -print0?
- What does find -exec {} + do?

Use find -print0 with xargs -0 for safe filenames, and -P for parallelism, or use find -exec ... +.

```
find . -name '*.log' -mtime +7 -print0 | xargs -0 -r rm -f
```

```
find . -type f -print0 | xargs -0 -P4 -n50 gzip
```

Interview tip: -print0 and -0 always go together.

 Official docs

- [GNU coreutils manual](#)

Q159 **How do you write a Bash script that reads from stdin or a file argument like standard Unix tools?**

Bash / Shell

 Spoken answer · ~55 s

Unix filters usually accept a file name and fall back to standard input, with a single dash meaning stdin. I set input to the first argument with a default of a dash, then use cat, which understands that convention, or I handle it explicitly by redirecting. The double dash after options protects against file names starting with a hyphen. If I need to know whether input is piped or interactive, I use the test for a terminal on file descriptor zero. For multiple files, I loop over the arguments and treat none as stdin. I keep results on stdout and messages on stderr so the script composes in pipelines, and I return meaningful exit codes. Following these conventions makes small scripts reusable in pipelines, which is much of the value of shell tooling.

Likely follow-ups

- What does the dash argument mean?
- How do you detect piped input?

Default to stdin when no file is given, using "\${1:--}" and cat or redirect, and detect a terminal with test -t.

```
input="${1:--}"  
cat -- "$input" | wc -l
```

```
[ -t 0 ] && echo 'reading from terminal'
```

Interview tip: A dash means stdin by convention.

 Official docs

- [GNU Bash reference manual](#)

 Spoken answer · ~56 s

For simple timing, Bash has the SECONDS variable that counts seconds since the shell started, so I record it before and after and subtract. For finer resolution I use date with nanoseconds or the time keyword. For limiting runtime, the coreutils timeout command runs a command and sends a signal after a duration. By default it sends TERM, which a process can handle gracefully; adding the kill-after option sends KILL if it is still running after a grace period. Timeout returns exit status 124 when the limit hit, so the script can tell the difference between a failure and a timeout. This is useful for health checks, flaky network calls and CI steps that might hang. I also set timeouts on the commands themselves, such as curl's max-time, because layered timeouts avoid orphaned work.

Likely follow-ups

- What exit code does timeout return?
- Why add kill-after?

Use SECONDS or date +%s for timing, and the timeout command to kill long-running processes.

```
start=$SECONDS; run_job; echo "took $((SECONDS-start))s"
```

```
timeout --signal=TERM --kill-after=10 60 ./long.sh
```

Interview tip: Use kill-after to escalate.

 Official docs

- [AWS Builders' Library: Timeouts, retries and backoff with jitter](#)
- [GNU Bash reference manual](#)

 Spoken answer · ~65 s

Pathlib provides an object-oriented way to work with paths. Instead of calling `os.path.join`, `basename` and `exists` on strings, I build a `Path` with the slash operator and use attributes like `name`, `suffix`, `stem` and `parent`, and methods like `exists`, `is_dir`, `read_text`, `write_text`, `mkdir` with `parents` and `exist_ok`, and `rglob` for recursive searches. The code reads more naturally, handles the platform's separator, and avoids bugs from manual string concatenation. Paths are immutable and comparable, and many standard library functions accept them. I use `resolve` to get the absolute canonical path, which also helps prevent path traversal bugs when combined with a check that the result stays within an allowed directory. For very large directory scans, `os.scandir` may be faster, but for most automation scripts `pathlib` is clearer and less error-prone. I keep the code small and tested so the next engineer can trust and change it. I verify the behaviour with a quick test before relying on it.

Likely follow-ups

- How do you prevent path traversal?
- When is `os.scandir` better?

Path objects model filesystem paths with operators and methods, making code more readable and portable.

```
from pathlib import Path
p = Path('/var/log') / 'app' / 'x.log'
print(p.suffix, p.exists())
```

```
[f for f in Path('.').rglob('*.yaml')]
```

Interview tip: Use `/` to join paths.

 Official docs

- [Python: pathlib](#)

Q162 **How do you use the subprocess module to capture output, enforce timeouts and avoid shell injection?**

Python

 Spoken answer · ~60 s

I call `subprocess.run` with a list of arguments rather than a single string, which executes the program directly without a shell, so user-provided values cannot be interpreted as extra commands. `Shell equals true` is reserved for cases where I truly need shell features, and then I quote with `shlex.quote`. I set `capture_output` and `text` to get strings back, a timeout so a hung process cannot block the script forever, and `check equals true` so a non-zero exit raises `CalledProcessError`, which I catch to log `stderr` and return code. For long-running or streaming output I use `Popen` and read lines as they arrive, being careful about pipe buffers filling up and deadlocking. I pass an explicit environment when needed and avoid leaking secrets through arguments, since they are visible in the process list. Wrapping this in a helper keeps behaviour consistent across the codebase.

Likely follow-ups

- Why is `shell=True` risky?
- How do you stream output from a process?

Pass a list of arguments, avoid `shell=True`, set timeout and `check=True`, and capture text output.

```
r = subprocess.run(['kubectl', 'get', 'pods', '-o', 'json'], capture_output=True, text=True,
timeout=30, check=True)
```

```
json.loads(r.stdout)
```

Interview tip: A list of args avoids the shell.

 Official docs

- [AWS Builders' Library: Timeouts, retries and backoff with jitter](#)
- [OWASP Top Ten](#)
- [Python: subprocess](#)

Q163 How do you handle configuration with environment variables and validate it at startup?

Python

 Spoken answer · ~57 s

I centralise configuration in one module that reads environment variables once at startup and builds a typed settings object. Required values are accessed with a key lookup so a missing variable raises immediately with a clear message, and optional ones get safe defaults. I convert types explicitly, for example integers, booleans and durations, and validate ranges. Libraries like pydantic settings make this declarative and give good error messages, and they can also read a dotenv file for local development. Secrets come from the environment or a secret manager and are never printed; I mask them in any configuration dump. Failing fast at startup means a bad deployment is rejected by health checks instead of failing later under load. Keeping configuration separate from code follows the twelve-factor approach and makes the same image runnable in every environment.

Likely follow-ups

- Why fail fast at startup?
- How do you avoid logging secrets?

Read env vars into a typed settings object, apply defaults, and fail fast with clear errors when required values are missing.

```
import os
PORT = int(os.environ.get('PORT', '8080'))
DB = os.environ['DB_URL']
```

```
from pydantic_settings import BaseSettings
```

Interview tip: Fail at startup, not at 3 a.m.

Start here (official docs)

- [Python documentation](#)

Q164 **How do you write robust retry logic with the tenacity library or a custom decorator?**

Python

 Spoken answer · ~64 s

A good retry policy has four parts. First, selectivity: retry only transient failures like timeouts, connection resets and HTTP 429 or 503, never client errors like 400 or authentication failures. Second, spacing: exponential backoff with jitter so many clients do not retry in lockstep and hammer a recovering service. Third, limits: a maximum number of attempts or total time, so failures surface instead of looping forever. Fourth, visibility: log each retry with the reason and delay, and raise the final exception. Tenacity expresses all of this declaratively with decorators; a custom decorator works for simple cases. I also respect Retry-After headers. A crucial caveat is idempotency: retrying a non-idempotent operation like creating a payment can cause duplicates, so I use idempotency keys or check state before retrying. I also document the edge cases, because that is where production surprises usually appear. I verify the behaviour with a quick test before relying on it.

Likely follow-ups

- Why add jitter?
- How do idempotency keys help?

Retry only on transient errors, with exponential backoff and jitter, a stop condition, and logging of attempts.

```
from tenacity import retry, stop_after_attempt, wait_random_exponential,
retry_if_exception_type
@retry(stop=stop_after_attempt(5), wait=wait_random_exponential(max=30),
retry=retry_if_exception_type(ConnectionError))
```

Interview tip: Never retry non-idempotent calls blindly.

 Official docs

- [AWS Builders' Library: Timeouts, retries and backoff with jitter](#)

Q165 **What is the difference between a list comprehension, a generator expression and map/filter?**

Python

 Spoken answer · ~58 s

A list comprehension evaluates immediately and stores all results in memory, which is fine for small data and when I need indexing or multiple passes. A generator expression uses parentheses and produces items lazily, one at a time, so memory stays constant even for huge sequences, and it can be consumed only once. Functions like `sum`, `any`, `all` and `max` accept generators directly, so I avoid creating throwaway lists. `map` and `filter` also return lazy iterators in Python three, but comprehension syntax is usually more readable, especially with conditions. For processing large log files, I chain generators: `read lines`, `filter`, `parse` and `aggregate`, which streams the data without loading the whole file. If the logic needs a loop with several statements or error handling, a regular `for` loop or a generator function is clearer than a dense comprehension.

Likely follow-ups

- When do you need a list instead of a generator?
- Why can a generator be consumed only once?

A list comprehension builds a list now; a generator expression yields lazily; `map` and `filter` are lazy iterators but less readable.

```
squares = [x*x for x in range(10)]
```

```
total = sum(x*x for x in range(10**7))
```

Interview tip: Use generators for large pipelines.

 **Start here (official docs)**

- [Python documentation](#)

Q166 How do you use collections like Counter, defaultdict and deque for operational scripts?

Python

 Spoken answer · ~66 s

The collections module has structures that make common script tasks short and efficient. Counter counts hashable items and offers most_common, which is perfect for top talkers or status code frequencies. defaultdict creates missing values automatically from a factory, so grouping items by key needs no existence checks, for example defaultdict of list. Deque is a double-ended queue with constant time appends and pops at both ends, unlike lists where removing from the front is linear. With a maximum length it automatically discards old items, which gives a rolling window for tasks like the last hundred latencies or the last few log lines before an error. Others include namedtuple and OrderedDict. Choosing the right structure often reduces both code and complexity, and it signals to reviewers that the author understands performance characteristics. I keep the code small and tested so the next engineer can trust and change it. I verify the behaviour with a quick test before relying on it.

Likely follow-ups

- Why is a deque better than a list for queues?
- How do you group items with defaultdict?

Counter tallies, defaultdict removes key checks, deque gives fast appends and pops at both ends and bounded windows.

```
from collections import Counter, defaultdict, deque
Counter(codes).most_common(5)
```

```
last = deque(maxlen=100)
```

Interview tip: deque(maxlen) is a free sliding window.

Official docs

- [Python: collections](#)

Q167 How do you work with dates and durations, such as computing time since last deploy?

Python

 Spoken answer · ~65 s

I work in timezone-aware datetimes in UTC internally and convert to local time only for display. Naive datetimes without a timezone are a frequent source of bugs, especially around daylight saving changes. To compute elapsed time, I subtract two aware datetimes and get a `timedelta`, from which I can read `total_seconds` or compare against thresholds. For parsing and formatting I use ISO 8601 with `fromisoformat` and `isoformat`, or a library like `dateutil` for messy formats. For measuring durations within a process I use `time.monotonic` or `perf_counter`, because the wall clock can jump due to NTP corrections. In APIs and logs I standardise on UTC timestamps with an explicit offset. For recurring schedules and local rules, I use the `zoneinfo` module, not fixed offsets, since offsets change over the year. I also document the edge cases, because that is where production surprises usually appear. I verify the behaviour with a quick test before relying on it.

Likely follow-ups

- Why use monotonic time for timing?
- Why avoid naive datetimes?

Use timezone-aware datetime in UTC, `timedelta` for durations, and ISO 8601 for parsing and output.

```
from datetime import datetime, timezone, timedelta
now = datetime.now(timezone.utc)
age = now - datetime.fromisoformat('2026-10-01T10:00:00+00:00')
```

```
age > timedelta(hours=6)
```

Interview tip: Store UTC, convert for display.

Start here (official docs)

- [Python documentation](#)

Q168 How do you write and organise Python modules and packages for a team's internal tooling?

Python

 Spoken answer · ~66 s

Scripts copied around quickly become unmaintainable, so I structure internal tools as installable packages. The layout has a source directory with the package, a tests directory, and a pyproject file describing metadata, dependencies and console entry points, so the tool is invoked by name after installation. Modules have clear responsibilities: a thin CLI layer that parses arguments, a core library with logic that is easy to test, and separate clients for external systems. I pin or constrain dependencies, publish to an internal package index, and version with semantic versioning. Quality tooling includes a formatter, linter, type checker and pre-commit hooks, all run in CI. Documentation includes a readme with examples and a changelog. This approach lets teammates install, upgrade and trust the tool, instead of running whichever copy they found. I keep the code small and tested so the next engineer can trust and change it. I verify the behaviour with a quick test before relying on it.

Likely follow-ups

- What is an entry point?
- Why separate CLI from core logic?

Use a src layout, pyproject.toml, clear module boundaries, entry points, tests and a linter and formatter.

```
pyproject.toml
src/mytool/__init__.py
src/mytool/cli.py
tests/test_cli.py
```

```
pip install -e .
```

Interview tip: Make it installable, not a pile of scripts.

Official docs

- [Python Documentation](#)

 Spoken answer · ~57 s

The key principle is to stream and avoid loading everything into memory. For CSV, `csv.DictReader` iterates row by row with constant memory. For JSON, a single large document needs a streaming parser like `ijson`, but the better format for big data is JSON Lines, where each line is a separate record and can be read in a loop. With pandas I use `chunksize` to process pieces and aggregate partial results, and I specify column types and read only needed columns to cut memory. For heavy analytic queries, DuckDB can query CSV or Parquet files directly with SQL, very quickly and out of core. I also consider compressed formats and Parquet for repeated analysis. Before optimising, I check whether a simple `awk` or `grep` pre-filter reduces the data enough, and I measure memory and time with a sample.

Likely follow-ups

- What is JSON Lines?
- When would you use DuckDB?

Stream with `csv.reader` or `json lines`, process in chunks, and use pandas chunking or DuckDB for heavy analytics.

```
import csv
with open('big.csv', newline='') as f:
    for row in csv.DictReader(f): ...
```

```
pd.read_csv('big.csv', chunksize=100000)
```

Interview tip: Avoid `json.load` on multi-GB files.

 Official docs

- [Python: json](#)
- [Python Documentation](#)

Q170 **How do you use virtual environments and tools like pip-tools, Poetry or uv for reproducible installs?**

Python

 Spoken answer · ~65 s

Each project gets its own virtual environment so dependencies do not conflict with the system or other projects. For reproducibility I separate what I want from what I got: a short file of direct dependencies and a lock file with exact versions of everything, including transitive packages, ideally with hashes. Tools like pip-tools, Poetry and uv generate and update the lock file. CI and production install strictly from the lock, so builds are repeatable and a surprise upstream release cannot break them. Updates happen deliberately through reviewed pull requests, often automated by an update bot, with tests and vulnerability scans. I pin the Python version too, and in containers I install into the image at build time using a cache-friendly layer ordering. This prevents it-works-on-my-machine problems and makes security patching traceable. I also document the edge cases, because that is where production surprises usually appear. I verify the behaviour with a quick test before relying on it.

Likely follow-ups

- Why include hashes?
- How do you update locked dependencies?

Isolate per project with venv, lock exact versions with hashes, and install from the lock file in CI and production.

```
python -m venv .venv && . .venv/bin/activate
```

```
pip-compile --generate-hashes requirements.in
```

```
pip install --require-hashes -r requirements.txt
```

Interview tip: Separate abstract and locked requirements.

 **Start here (official docs)**

- [Python documentation](#)

Q171 **What are Python's common performance pitfalls in scripts, and how do you fix them?**

Python

 Spoken answer · ~60 s

Most slow scripts suffer from algorithmic or I/O problems rather than the interpreter. Testing membership in a list is linear, so for repeated checks I convert to a set or dict. Building strings with plus in a loop creates many copies; collecting parts and using join is better. Opening files or making network calls inside tight loops multiplies latency, so I batch requests, reuse sessions and connections, and use concurrency for I/O. Recomputing the same values calls for caching with `functools.lru_cache`. Reading entire files when streaming would do wastes memory. For numeric work, vectorised libraries beat Python loops. But I always profile first with `cProfile` or a sampling profiler to find the real hotspot, because guesses are often wrong, and I re-measure after each change. If it is still too slow, moving the hot path to a compiled extension or another language is an option.

Likely follow-ups

- How do you find the real bottleneck?
- When does caching help?

String concatenation in loops, repeated list membership tests, unnecessary I/O, and global lookups; use join, sets, batching and profiling.

```
items = set(big_list)
if x in items: ...
```

```
''.join(parts)
```

Interview tip: Profile first.

 Official docs

- [Python Documentation](#)

Q172 **How do you test code that calls external APIs or the cloud SDK without real calls?**

Python

 Spoken answer · ~58 s

I isolate external calls behind a small client class or function that I own, so tests can replace just that boundary. With `unittest.mock` or `pytest monkeypatch` I substitute fake responses, including error cases like timeouts, throttling and malformed data, which are hard to trigger for real. For cloud SDKs there are purpose-built tools, such as `moto` for AWS or the SDK's stubber, which simulate services more faithfully. Recorded responses with a library like `vcrpy` are another option, though they can go stale. I also test that my code handles pagination and retries. Mocks only prove my code works against my assumptions, so I keep a small set of integration tests that run against a sandbox account on a schedule or before release. Combining fast mocked unit tests with a few real tests gives good confidence without cost or flakiness.

Likely follow-ups

- Where should you place the mock boundary?
- Why keep some real integration tests?

Mock the client boundary with `unittest.mock` or a stubber, use recorded responses, and keep a few real integration tests separate.

```
from unittest.mock import patch
@patch('mytool.client.get_instances', return_value=[])
```

```
moto # mock AWS services
```

Interview tip: Mock at your own boundary, not deep internals.

 Official docs

- [Python: `unittest.mock` / testing](#)

Q173 **How do you build a simple REST API client class with sessions, auth, timeouts and pagination?**

Python

 Spoken answer · ~57 s

I wrap the HTTP library in a small client class. It owns a session, which reuses connections and keeps default headers like authorization, and it configures a retry adapter with backoff for transient status codes. Importantly, the requests library has no default timeout, so I set one on every call, separate connect and read values, or the program may hang forever. Credentials are injected from configuration rather than hard-coded. The client raises meaningful exceptions on non-success responses after logging the status and request identifier. For list endpoints it exposes a generator that follows next links or cursors and yields items, so callers do not care about pagination. I add a user agent string, handle rate limit headers by sleeping, and keep methods thin so tests can mock the transport. This pattern makes every integration consistent and testable.

Likely follow-ups

- Why is the default timeout dangerous?
- How would a pagination generator work?

Wrap `requests.Session`, set default headers and timeouts, retry adapters, and expose an iterator for pages.

```
s = requests.Session()
s.headers['Authorization'] = f'Bearer {token}'
s.mount('https://', HTTPAdapter(max_retries=Retry(total=3, backoff_factor=0.5,
status_forcelist=[429,500,502,503])))
```

Interview tip: requests has no default timeout.

 Official docs

- [AWS Builders' Library: Timeouts, retries and backoff with jitter](#)
- [REST API pagination \(Google AIP-158\)](#)
- [Google API Design Guide](#)

 Spoken answer · ~61 s

The brute force approach compares every pair, which is quadratic. A hash map reduces it to one pass: for each number, I check whether target minus the number is already in the map, and if so I return the stored index; otherwise I store the current number and index. Lookups and inserts are average constant time, so the whole algorithm is linear in time and uses linear extra memory. The same idea detects duplicates: add each item to a set and report when it is already present. In operations work this appears when finding duplicate hostnames, matching IDs between two inventories, or checking which resources appear in one list but not another, using set difference. I mention edge cases: duplicates in the input, using the same element twice, and unsorted input. If memory is constrained and the data is sorted, the two-pointer technique uses constant memory.

Likely follow-ups

- What is the complexity?
- When would two pointers be better?

Store seen values in a hash set or map and check complements in one pass for $O(n)$ time.

```
seen = {}
for i, x in enumerate(nums):
    if target - x in seen: return seen[target - x], i
    seen[x] = i
```

Interview tip: Trade memory for time.

 **Start here (official docs)**

- [Python documentation](#)

 Spoken answer · ~64 s

The two-pointer technique uses two indices that move toward each other or in the same direction, exploiting some order in the data so that each element is visited only once. In a sorted array, to find a pair with a given sum, I place one pointer at each end; if the sum is too small I advance the left pointer, if too large I move the right one, and I find the answer in linear time with constant memory. A same-direction variant handles removing duplicates in place or merging two sorted lists, such as combining sorted log streams. The sliding window is a close relative. The key condition is a monotonic property that tells me which pointer to move. If the data is unsorted, I either sort first, paying $n \log n$, or use a hash map. In interviews, I explain the invariant and test edge cases like empty input and a single element.

Likely follow-ups

- What property must the data have?
- How is it related to a sliding window?

 Two pointers

Left pointer	Right pointer
Starts at index 0	Starts at end
Moves right	Moves left

Two indexes move through a sorted or linear structure to solve in $O(n)$ what would take $O(n^2)$, such as pair sums or merging.

```
l, r = 0, len(a)-1
while l < r:
    s = a[l]+a[r]
    if s == t: break
    l, r = (l+1, r) if s < t else (l, r-1)
```

Interview tip: Needs sorted data or a monotonic property.

 Start here (official docs)

- [Python documentation](#)

Q176 **How do you detect a cycle or compute dependency order, such as for a deploy plan?**

Algorithms & DS

 Spoken answer · ~57 s

Dependencies between services, Terraform resources or build tasks form a directed graph. A valid execution order is a topological order, where each node appears after everything it depends on. Kahn's algorithm computes it by repeatedly taking nodes with zero incoming edges, removing them and updating counts; if some nodes remain at the end, there is a cycle and no valid order exists. Depth-first search with a visiting state also detects cycles. Python's graphlib module provides TopologicalSorter, which does this and raises CycleError with the offending nodes. Complexity is linear in nodes plus edges. This is useful for ordering a rollout, determining which Terraform modules to apply first, or deciding what can be built in parallel, since nodes at the same level are independent. In practice I also report the cycle path to help someone fix the configuration.

Likely follow-ups

- How do you detect a cycle?
- How does it help with parallel execution?

Model dependencies as a directed graph, run a topological sort, and a failure to order all nodes means a cycle.

```
from graphlib import TopologicalSorter
list(TopologicalSorter({'app':{'db','cache'},'db':set()}).static_order())
```

Interview tip: graphlib is in the standard library.

 Start here (official docs)

- [Python documentation](#)

Q177 How do you implement a simple circuit breaker or a sliding-window rate counter?

Algorithms & DS

 Spoken answer · ~60 s

A sliding-window counter keeps the timestamps of recent events in a deque. On each event I discard timestamps older than the window, append the new one, and the length of the deque is the count in the window. This supports rate limiting or error-rate tracking. A circuit breaker builds on it with three states. Closed means calls pass and failures are counted; when the failure count or ratio in the window passes a threshold, the breaker opens and calls fail immediately without hitting the struggling dependency. After a cooldown, it goes half-open and lets a few probe calls through; if they succeed it closes again, otherwise it re-opens. This protects the dependency from overload and my service from slow failures piling up. In production I use a tested library, add metrics for state changes, and combine it with timeouts and retries with backoff.

Likely follow-ups

- What are the three breaker states?
- Why fail fast when open?

Track recent failures or timestamps in a bounded structure and open the circuit when a threshold is exceeded, then probe after a cooldown.

```
from collections import deque
window = deque()
while window and window[0] < now-60: window.popleft()
window.append(now)
```

Interview tip: States: closed, open, half-open.

Official docs

- [Google SRE Book: Addressing Cascading Failures](#)
- [Microsoft: Cloud design pattern – Circuit Breaker](#)
- [Python: collections](#)

Q178 How do you find the most recent N events or compute a moving average efficiently?

Algorithms & DS

 Spoken answer · ~57 s

Recomputing the average over the last N values on every new data point costs order N each time. Instead I keep a running sum. A deque with a maximum length holds the window; when it is full and a new value arrives, I subtract the value about to be evicted from the sum, append the new value, and add it to the sum. Each update is constant time and memory is bounded by N. For an exponentially weighted moving average, I need only a single previous value and a smoothing factor, which reacts faster to recent changes and uses constant memory. Floating point drift can accumulate in a long-running sum, so for very long streams I occasionally recompute or use a more stable algorithm. These techniques are used in alerting, dashboards and anomaly detection on metric streams.

Likely follow-ups

- What is an exponential moving average?
- How can floating point drift affect it?

Use a deque with maxlen, maintaining a running sum so each update is $O(1)$.

```
from collections import deque
w = deque(maxlen=5); total = 0
for x in stream:
    if len(w)==w.maxlen: total -= w[0]
    w.append(x); total += x; avg = total/len(w)
```

Interview tip: Update incrementally, do not recompute.

 **Start here (official docs)**

- [Python documentation](#)

 Spoken answer · ~58 s

A heap is a tree-based structure that always gives fast access to the smallest or largest element. Insertion and removal of the top take logarithmic time, and peeking is constant. A priority queue is the abstract concept, usually implemented with a heap. Python's `heapq` module provides a min-heap on a list; for a max-heap I push negated values. Practical uses include a job scheduler that always runs the highest-priority task next, retry queues ordered by next attempt time, merging many sorted log streams by timestamp, and finding the top K items with a bounded heap of size K, which uses $O(n \log K)$ time and little memory. For ties, I include a counter in the tuple to keep comparisons stable and avoid comparing non-comparable payloads. Heaps are not good for arbitrary lookups or removals, where a different structure is better.

Likely follow-ups

- How do you build a max-heap in Python?
- How does a bounded heap give top-K?

A heap gives $O(\log n)$ insert and extract-min; use it for top-K, scheduling and merging sorted streams.

```
import heapq
heapq.heappush(h, (priority, item))
heapq.heappop(h)
heapq.nlargest(5, data, key=...)
```

Interview tip: Python `heapq` is a min-heap.

 Official docs

- [Python: heapq](#)

Q180 **How do you compute percentiles like p95 and p99 from large latency data, and what are the pitfalls?**

Algorithms & DS

 Spoken answer · ~65 s

For a modest dataset I can sort and index, or use numpy percentile. For huge or streaming data, exact computation requires too much memory, so I use approximate structures such as histograms with fixed buckets, HDR histograms or t-digest, which give accurate tail estimates in bounded memory and can be merged. The most important pitfall is aggregation: percentiles of subsets cannot be averaged to get the overall percentile. To combine data from many hosts, I merge the histograms and compute the percentile from the merged distribution. Another pitfall is choosing bucket boundaries that are too coarse around the SLO threshold. I also watch for coordinated omission in load tests, where a stalled system hides slow requests. Tail percentiles matter because users experience the slow requests, and averages hide them. I keep the code small and tested so the next engineer can trust and change it. I verify the behaviour with a quick test before relying on it.

Likely follow-ups

- Why can't you average percentiles?
- What is a t-digest?

Sort or use selection for exact values; use histograms or t-digest for streaming; never average percentiles.

```
import statistics
statistics.quantiles(lat, n=100)[94]
```

```
import numpy as np
np.percentile(lat, [95,99])
```

Interview tip: You cannot average p99s across hosts.

 Official docs

- [Prometheus: Histograms and summaries](#)

Q181 What is the difference between BFS and DFS, and where do they appear in infrastructure scripts?

Algorithms & DS

 Spoken answer · ~55 s

Breadth-first search uses a queue and visits nodes in order of distance from the start, so it finds the shortest path in an unweighted graph. Depth-first search uses a stack or recursion and follows one branch to the end before backtracking, which is natural for walking directory trees, detecting cycles and topological ordering. Both run in time proportional to nodes plus edges and need a visited set to avoid infinite loops in graphs with cycles. In infrastructure scripts, BFS might compute the blast radius of a failing service by finding all transitive dependents level by level, or the fewest hops between two network nodes. DFS fits recursively walking a resource hierarchy, a file tree or nested configuration. Recursion depth is limited in Python, so for deep graphs I use an explicit stack.

Likely follow-ups

- Which finds shortest paths?
- Why do you need a visited set?

BFS vs DFS

BFS	DFS
Queue, level by level	Stack / recursion
Shortest path unweighted	Cycle detection, topo sort

BFS explores level by level and finds shortest paths in unweighted graphs; DFS goes deep first and suits traversal, cycle detection and tree walks.

```
from collections import deque
q = deque([start]); seen = {start}
while q:
    n = q.popleft()
    for m in graph[n]:
        if m not in seen: seen.add(m); q.append(m)
```

Interview tip: Always track visited nodes.

Start here (official docs)

- [Python documentation](#)

Q182 **How do you build a script that safely rotates credentials or API keys with zero downtime?**

Automation / API

 Spoken answer · ~65 s

Safe rotation follows create, distribute, verify, retire. I generate the new credential while the old one stays valid, store it in the secret manager, and update consumers either by letting them re-read the secret or by triggering a rolling restart. Then I verify that the new credential works by checking authentication success and that no traffic still uses the old one, using access logs or last-used timestamps. Only then do I deactivate the old credential first, which is reversible, wait a period for any missed consumers to fail loudly, and finally delete it. The script is idempotent, can resume after failure, logs each step without printing secrets, and supports dry-run. Where possible, I avoid long-lived keys entirely by using short-lived tokens and workload identity, which turns rotation into an automatic process. I also document the edge cases, because that is where production surprises usually appear. I verify the behaviour with a quick test before relying on it.

Likely follow-ups

- Why deactivate before deleting?
- How do you verify no one uses the old key?

Create the new credential, update consumers, verify, then revoke the old one; support overlap and have a rollback.

```
aws iam create-access-key --user-name ci
```

```
aws iam update-access-key --access-key-id OLD --status Inactive
```

```
aws iam delete-access-key --access-key-id OLD
```

Interview tip: Deactivate before delete.

 **Start here (official docs)**

- [Python documentation](#)

Q183 **How do you write a script that audits cloud resources and produces a report?**

Automation / API

 Spoken answer · ~59 s

I start with a read-only role in each account, since an audit should never be able to change anything. The script iterates over accounts and regions, uses SDK paginators so no pages are missed, and normalises results into a common structure with fields like resource ID, type, owner tag, creation date and cost centre. Rules are expressed as small functions, for example unattached volumes, untagged instances, public buckets or old snapshots, each returning findings with severity. The output goes to JSON for machines and CSV or a dashboard for people, with a summary of counts. I handle throttling with retries, run accounts concurrently within limits, and record the run time and scope so reports are comparable. The report is scheduled, stored historically so trends are visible, and ideally linked to tickets for remediation, because an audit nobody acts on has little value.

Likely follow-ups

- Why use a read-only role?
- How do you handle API throttling?

List resources across accounts with pagination, normalise to a common structure, apply rules, and output CSV or JSON.

```
aws ec2 describe-instances --query 'Reservations[].Instances[].  
[InstanceId,State.Name,Tags]' --output json
```

```
paginator = client.get_paginator('describe_volumes')
```

Interview tip: Use paginators and read-only roles.

 Official docs

- [REST API pagination \(Google AIP-158\)](#)

 Spoken answer · ~58 s

A chat or webhook trigger is an entry point into production, so security comes first. I verify each request's signature with the shared secret using a constant-time comparison, and check timestamps to prevent replay. Then I authorise: the user must belong to an allowed group for the specific command, with stricter rules for production. Actions are limited to a defined set of safe operations, parameters are validated against allowlists, and nothing is built with string concatenation into shell commands. Mutating operations are idempotent, support dry-run, require confirmation for risky ones, and can ask for a second approver. Every action is logged with who, what, when and the result, and posted back to the channel for transparency. Rate limits prevent runaway loops, and the service runs with least privilege credentials. This gives teams speed while keeping control and an audit trail.

Likely follow-ups

- How do you prevent replay attacks?
- Why use an allowlist of actions?

Authenticate and verify requests, authorise by user and action, make actions idempotent and audited, and require confirmation for risky ones.

```
hmac.compare_digest(sig, expected)
```

```
# verify signing secret and timestamp
```

Interview tip: Verify signatures and limit blast radius.

Start here (official docs)

- [Python documentation](#)

 Spoken answer · ~56 s

Expired certificates cause avoidable outages, so I automate detection. A script or exporter connects to each endpoint, reads the certificate's not-after date and exposes days remaining as a metric; it also scans certificate stores, secrets and ingress objects for certificates that are not on public endpoints. Alerts fire at several thresholds, such as thirty, fourteen and seven days, with increasing severity and a clear owner. The best fix is removing the manual process: use automatic issuance and renewal with ACME or a cloud certificate manager, and cert-manager in Kubernetes, then monitor that renewal actually succeeds. I also check the full chain and the hostname, not just the expiry. An inventory of certificates and owners helps, and a regular test of the renewal path ensures it still works when the CA or DNS setup changes.

Likely follow-ups

- Why alert at several thresholds?
- What does cert-manager do?

Scan endpoints and stores for not-after dates, alert at multiple thresholds, and automate renewal where possible.

```
echo | openssl s_client -connect host:443 -servername host 2>/dev/null | openssl x509 -noout -enddate
```

```
openssl x509 -checkend 1209600 -noout -in cert.pem
```

Interview tip: Alert at 30, 14 and 7 days.

 Start here (official docs)

- [Python documentation](#)

Q186 How do you automate environment cleanup, like removing old preview environments?

Automation / API

 Spoken answer · ~59 s

Preview environments are easy to create and easy to forget, so cleanup must be automatic. When creating one, I label it with an owner, creation time and time to live. A scheduled job lists resources with the preview label, selects those past their expiry or whose pull request is closed, and deletes them in dependency order: workloads, then load balancers and DNS records, then volumes and namespaces, so nothing is orphaned. The job has a dry-run mode that prints what it would delete, an allowlist label to protect something that should stay, and a notification to owners before deletion. It only acts on resources carrying the preview marker, never relying on naming alone. Metrics on environments created, deleted and cost saved show its value. Deleting based on a pull request close event gives fast cleanup, with the scheduled sweep as backup.

Likely follow-ups

- Why delete in dependency order?
- How do you protect resources from cleanup?

Tag with owner and expiry, list by tag, delete in dependency order, and run on a schedule with dry-run and notifications.

```
kubectl get ns -l preview=true -o json | jq -r '.items[]|select(.metadata.creationTimestamp < "2026-10-01")|.metadata.name'
```

Interview tip: Label everything with ttl and owner.

Start here (official docs)

- [Python documentation](#)

Q187 How do you write a drift-detection script for configuration or infrastructure?

Automation / API

🗣 Spoken answer · ~57 s

Drift detection compares what the code says should exist with what actually exists. For Terraform, a scheduled plan with the detailed exit code returns zero for no changes and two when there are differences, which a pipeline can turn into an alert or ticket. For Kubernetes, `kubectl diff` or a GitOps controller shows live-versus-Git differences. For custom systems, I write a script that loads desired state from files, queries the real state through APIs, normalises both into comparable structures, and prints additions, removals and changes. I ignore fields controlled by other systems to reduce noise. Remediation can be automatic for low-risk resources, but for sensitive ones I report and require approval, because the drift might be an emergency fix that should be codified. I track drift counts over time to find processes that bypass the pipeline.

Likely follow-ups

- What does exit code 2 mean?
- Should drift always be auto-corrected?

🔗 Drift detection script



Compare desired state from code with actual state from APIs, report differences, and optionally remediate with approval.

```
terraform plan -detailed-exitcode
```

```
kubectl diff -f manifests/
```

Interview tip: Exit code 2 means changes.

📖 Start here (official docs)

- [Python documentation](#)

Q188 **How do you implement a safe bulk operation, such as updating tags on thousands of resources?**

Automation / API

 Spoken answer · ~56 s

Bulk changes can cause large damage quickly, so I make the process staged. First, a dry-run mode produces a list of intended changes for review. Then I apply to a small canary set, verify the effect, and proceed in batches with a pause between them to respect API rate limits and allow monitoring to catch problems. The operation is idempotent: it checks the current state and only changes what differs, so rerunning is safe, and it records progress so it can resume after interruption. I log every change with before and after values to a file, which also supports rollback, and I stop automatically if the error rate passes a threshold. Concurrency is limited. Finally, I verify the end state with a separate audit query. These controls turn a risky task into a repeatable procedure.

Likely follow-ups

- Why start with a canary subset?
- How does idempotency help resumption?

Plan first, batch, throttle, make it idempotent and resumable, log each change, and start with a small canary subset.

```
for chunk in chunks(resources, 20): apply(chunk); time.sleep(1)
```

Interview tip: Dry-run, canary, then scale.

 Start here (official docs)

- [Python documentation](#)

Q189 How do you use the official Kubernetes Python client to list and patch resources safely?

K8s / Cloud

 Spoken answer · ~58 s

I load configuration with `load_kube_config` on a laptop or `load_incluster_config` inside a pod, then create typed API objects like `CoreV1Api` and `AppsV1Api`. For lists, I use limit and continue tokens for pagination on large clusters, and label or field selectors to reduce data. Writes use patch with a small body rather than replacing whole objects, which avoids overwriting concurrent changes; a strategic merge or JSON patch states exactly what changes. I wrap calls in try-except for `ApiException` and check status codes: 404 for missing, 409 for conflicts, 429 for throttling with a retry. For watching changes, the watch API streams events, but production controllers should use informers or a framework, since raw watches need reconnect logic and resource version handling. The service account used must have least privilege RBAC, and mutating scripts should support dry-run through the server-side dry-run option.

Likely follow-ups

- Why patch instead of replace?
- How do you authenticate in a pod?

Load kubeconfig or in-cluster config, use typed API clients, handle pagination and `ApiException`, and patch with strategic merge.

```
from kubernetes import client, config
config.load_kube_config()
v1 = client.CoreV1Api()
v1.list_namespaced_pod('default', limit=100)
```

```
apps.patch_namespaced_deployment(name, ns, {'spec':{'replicas':3}})
```

Interview tip: Use in-cluster config inside pods.

Official docs

- [Kubernetes Python client](#)
- [Python Documentation](#)

Q190 **How do you write a script that drains and upgrades nodes one at a time safely?**

K8s / Cloud

 Spoken answer · ~57 s

The loop for each node is cordon to stop new pods landing, drain to evict existing pods gracefully, perform the upgrade or replacement, uncordon, and then wait until the node is ready and workloads are healthy before moving on. Drain respects PodDisruptionBudgets, so if a budget blocks eviction the script should wait and retry with a timeout, then alert a human, instead of forcing deletion. I ignore daemonset pods and deal with local storage consciously. Between nodes, I verify cluster health: node ready, pending pods not increasing, key deployments at desired replicas, and error rate stable. I process in small batches only if capacity allows, and keep a state file so the script can resume. In managed services, surge upgrades or node pool rotation automate this, but understanding the sequence is key for troubleshooting stuck drains.

Likely follow-ups

- What blocks a drain?
- Why not force delete pods?

Cordon, drain respecting PodDisruptionBudgets, upgrade, uncordon, and verify health before the next node.

```
kubectl cordon node1
kubectl drain node1 --ignore-daemonsets --delete-emptydir-data --timeout=300s
```

```
kubectl uncordon node1
```

Interview tip: PDBs may block drains, which is intentional.

 Official docs

- [Kubernetes: Safely drain a node](#)

 Spoken answer · ~55 s

Cloud CLIs return large JSON documents, so I reduce them early. AWS and Azure CLIs support a query option using JMESPath, and gcloud has format and filter flags. Server-side filters, such as filters on tags or states, are best because they limit data fetched. Then I project only the fields I need, using text or value output for simple shell consumption or JSON with jq for structured processing. In scripts I use machine-readable output and avoid parsing human-formatted tables, whose layout can change. I handle pagination, since CLIs may return only a page unless told otherwise, and I check for empty results explicitly. For anything beyond a few lines I move to the SDK in Python, which gives proper error handling, retries and testability. Always quote queries carefully to avoid shell interpretation.

Likely follow-ups

- Why prefer server-side filters?
- When should you switch to an SDK?

Use `--query` with JMESPath for server-side filtering and `--output json` piped into `jq` for further shaping.

```
aws ec2 describe-instances --filters Name=tag:env,Values=prod --query  
'Reservations[].Instances[].InstanceId' --output text
```

```
gcloud compute instances list --format='value(name,zone)'
```

Interview tip: Filter at the API where possible.

 Official docs

- [jq manual](#)

 Spoken answer · ~62 s

A controller implements a control loop. It watches resources, and on every change or periodically calls a reconcile function that reads the desired spec, observes the actual world, and performs the minimal actions to converge them, then updates the status. Reconcile must be idempotent and level-triggered: it should work from the current state, not from the specific event that triggered it, because events can be lost or duplicated. Frameworks like Kubebuilder or Operator SDK in Go, or Kopf in Python, handle caching, queues, retries with backoff and leader election. Finalizers ensure cleanup of external resources on deletion. An operator is worth building when you manage complex stateful applications or external systems declaratively and need continuous healing; for a one-off task a simple job or script is better. Building one adds a long-term maintenance cost, so I check whether an existing operator or Helm chart solves the need first.

Likely follow-ups

- What does level-triggered mean?
- What is a finalizer for?

A controller watches objects, compares desired and actual state, and reconciles idempotently; use one for domain logic that needs continuous convergence.

```
kubebuilder init --domain example.com
```

```
# reconcile(): get object, diff state, create/update, update status
```

Interview tip: Reconcile must be idempotent and level-triggered.

 Official docs

- [Kubernetes: Operator pattern](#)

 Spoken answer · ~60 s

In Go, functions return errors as values, so after every call that can fail I check err immediately and either handle it or return it with added context using `fmt.Errorf` and the percent w verb, so callers can still inspect the underlying cause. After successfully opening a file, I defer its `Close`, which guarantees cleanup on every return path. For reading line by line I use `bufio.Scanner`, remembering it has a default token size limit, so for very long lines I raise the buffer, and I check `scanner.Err` after the loop. For writes, I use a buffered writer, flush it, and check the error from `Close`, since write errors can surface there. For small files `os.ReadFile` and `os.WriteFile` are simpler. For atomic updates I write to a temporary file in the same directory and rename it. This explicit style makes failure paths visible.

Likely follow-ups

- Why check the error from `Close` on writes?
- How do you do atomic file updates?

Use `os` and `bufio`, check every error, defer `Close`, and wrap errors with context.

```
f, err := os.Open(path)
if err != nil { return fmt.Errorf("open %s: %w", path, err) }
defer f.Close()
sc := bufio.NewScanner(f)
```

Interview tip: Check the error from `Close` on writes.

 Start here (official docs)

- [Python documentation](#)

Q194 How do you make HTTP requests in Go with timeouts, and why is the default client risky?

Go

 Spoken answer · ~55 s

The package-level helper functions use a default client with no timeout, so a stalled server can hang a goroutine forever. I create my own `http.Client` with a overall `Timeout`, and for finer control I configure the transport with dial, TLS handshake and response header timeouts and sensible connection pool limits. I build requests with a context so I can cancel them when the caller gives up, which also propagates deadlines. After a successful `Do` call I must close the response body, usually with `defer`, and for connection reuse I read it to completion or drain it; otherwise connections leak. I check the status code explicitly because non-2xx responses are not errors. For retries I use backoff on idempotent requests only. Reusing one client across the program lets the transport pool connections efficiently.

Likely follow-ups

- Why must the body be closed?
- How do context and Timeout differ?

The default client has no timeout. Create an `http.Client` with `Timeout`, use context per request, and close the response body.

```
c := &http.Client{Timeout: 10 * time.Second}
req, _ := http.NewRequestWithContext(ctx, "GET", url, nil)
resp, err := c.Do(req)
defer resp.Body.Close()
```

Interview tip: Always close and drain the body.

Official docs

- [AWS Builders' Library: Timeouts, retries and backoff with jitter](#)
- [Requests: Advanced usage \(sessions, retries\)](#)
- [curl documentation](#)

 Spoken answer · ~56 s

The select statement waits on multiple channel operations and proceeds with whichever is ready first, choosing randomly if several are ready. It is the main tool for timeouts and cancellation: one case receives the result, another receives from the context's Done channel so the work stops when the caller cancels, and a default case makes the operation non-blocking. Using time.After inside a hot loop creates a new timer on every iteration, which can pile up until it fires, so for loops I create one timer and reset it, or rely on the context deadline. Select is also used to multiplex producers, to send or receive without blocking, and to implement graceful shutdown. A nil channel blocks forever, which can be used to disable a case dynamically. Always make sure goroutines can exit, otherwise they leak.

Likely follow-ups

- Why is time.After in a loop wasteful?
- What does a nil channel do in select?

select waits on several channel operations; combine with time.After or ctx.Done for timeouts and cancellation.

```
select {
case r := <-results: use(r)
case <-ctx.Done(): return ctx.Err()
case <-time.After(5*time.Second): return errTimeout
}
```

Interview tip: Prefer ctx.Done over time.After in loops.

 Official docs

- [AWS Builders' Library: Timeouts, retries and backoff with jitter](#)
- [Effective Go](#)

Q196 **How do you use `sync.WaitGroup` and `errgroup` to run concurrent tasks and collect errors?**

Go

 Spoken answer · ~65 s

A `WaitGroup` counts running goroutines: I add before starting each, call `Done` when it finishes, and `Wait` blocks until all complete. It does not carry errors, so I would use a channel or mutex-protected slice to gather them. The `errgroup` package builds on this: it runs functions in goroutines, `Wait` returns the first non-nil error, and the group created with a context cancels that context when any function fails, so the others can stop early. `SetLimit` bounds concurrency, which prevents launching thousands of goroutines against an API and exceeding rate limits. I make sure each worker respects the context. For results, I write into per-index slots of a pre-sized slice, which avoids locking, or use a channel. These patterns replace manual coordination and make concurrent fan-out safer and shorter. I keep the code small and tested so the next engineer can trust and change it. I verify the behaviour with a quick test before relying on it.

Likely follow-ups

- What does `errgroup` add over `WaitGroup`?
- How do you bound concurrency?

`WaitGroup` waits for goroutines; `errgroup` also propagates the first error and cancels siblings with a context.

```
g, ctx := errgroup.WithContext(ctx)
g.SetLimit(8)
for _, u := range urls { u := u; g.Go(func() error { return fetch(ctx, u) }) }
err := g.Wait()
```

Interview tip: Limit concurrency with `SetLimit`.

 Official docs

- [Go concurrency patterns: Pipelines and cancellation](#)

 Spoken answer · ~64 s

For small tools the standard flag package is enough; for tools with subcommands, persistent flags and help generation, Cobra is the common choice. I structure the program so main only calls a run function that returns an error, prints it to stderr, and calls os.Exit with a non-zero code. Keeping the exit in main matters because os.Exit skips deferred functions, so cleanup in run still executes. Logic lives in testable functions that take their dependencies and writers as parameters, rather than reading globals or printing directly. I support machine-readable output with a json flag, and send diagnostics to stderr so stdout stays clean for pipelines. Version information is injected at build time with linker flags. Cross-compiling produces single static binaries for each platform, which makes distribution easy. I also document the edge cases, because that is where production surprises usually appear. I verify the behaviour with a quick test before relying on it.

Likely follow-ups

- Why keep os.Exit only in main?
- How is the version embedded?

Use the flag package for simple tools or Cobra for subcommands, return errors from run functions, and exit non-zero in main only.

```
func main(){ if err := run(os.Args[1:]); err != nil { fmt.Fprintln(os.Stderr, err);  
os.Exit(1) } }
```

Interview tip: Keep os.Exit in main so defers run.

 Official docs

- [Kubernetes: Determine the reason for Pod failure](#)
- [Cobra CLI library](#)
- [The Go Programming Language documentation](#)

 Spoken answer · ~65 s

Go has benchmarking built in: a function named `BenchmarkSomething` loops `b.N` times, and running `go test` with the `bench` flag reports nanoseconds per operation, and with `benchmem` it also shows allocations per operation, which often matter more than raw speed. For deeper analysis I collect profiles. A CPU profile shows where time is spent, a heap profile shows memory allocation, and goroutine and block profiles help with concurrency issues. Running services can expose the `net/http/pprof` endpoints, restricted to internal access, so I can capture profiles from live systems. The `pprof` tool offers flame graphs and top views. I compare before and after with `benchstat` to avoid being fooled by noise. Typical wins come from reducing allocations, reusing buffers with `sync.Pool`, avoiding unnecessary conversions, and fixing algorithmic issues. I keep the code small and tested so the next engineer can trust and change it. I verify the behaviour with a quick test before relying on it.

Likely follow-ups

- Why check allocations?
- How do you profile a live service?

Write `BenchmarkXxx` functions with `b.N`, run `go test -bench`, and use CPU and heap profiles via `pprof` to find hotspots.

```
go test -bench=. -benchmem
```

```
go test -cpuprofile cpu.out -bench .  
go tool pprof -http=:8080 cpu.out
```

Interview tip: Measure allocations too.

 Start here (official docs)

- [Python documentation](#)

Q199 How do you review someone else's automation script before it runs in production?

Best Practices

 Spoken answer · ~58 s

I review for safety before style. First scope and blast radius: what can this touch, is the target explicit, and is there a way to preview with dry-run? Then correctness under failure: what happens if it fails halfway or runs twice, is it idempotent, are errors checked, and are timeouts and retries sensible? Security: how are credentials obtained, could input be injected into commands or queries, and do logs leak secrets? Operations: is there logging, a clear exit status, locking against concurrent runs and a rollback plan? Then quality: static analysis such as shellcheck or linters, tests including edge cases, readable names and comments explaining why. Finally I ask for a staged rollout, running first in a non-production environment and then on a small subset. Questions in review are more effective than assumptions, and I record the decisions.

Likely follow-ups

- What do you check first?
- What does idempotent mean for a script?

Check safety first: scope, idempotency, dry-run, error handling, secrets, blast radius, logging and rollback; then readability and tests.

```
shellcheck script.sh
```

```
git diff main -- scripts/
```

Interview tip: Ask what happens if it runs twice, or half-fails.

 **Start here (official docs)**

- [Python documentation](#)

Q200 When do you choose a script versus a proper service or tool, and how do you decide?

Best Practices

 Spoken answer · ~58 s

A script is perfect for a quick one-off or a small automation owned by one person, because it is fast to write and easy to change. The signals to promote it into a real tool or service are growth beyond a couple of hundred lines, use by multiple people, running unattended or on a schedule, handling money or production changes, needing state, concurrency, retries or authentication, or requiring tests and an audit trail. At that point I choose a more maintainable language, add structure, tests, packaging, logging and metrics, and consider whether an existing product already solves the problem. Promotion should be deliberate, before an unreviewed script becomes critical infrastructure that nobody understands. I also set an owner and documentation. The cost of a rewrite is often smaller than the cost of operating a fragile script that everyone depends on.

Likely follow-ups

- What signals tell you to promote a script?
- Why assign an owner?

Scripts suit short, low-risk, single-owner tasks; move to a tool or service when it needs reliability, shared use, state, concurrency or auditing.

signals: >200 lines, multiple users, runs unattended, needs retries/state

Interview tip: Promote code deliberately before it becomes critical.

Start here (official docs)

- [Python documentation](#)