ESTUDAR E REALMENTE APRENDER DEVOPS & CLOUD

Caderno de Respostas

Gabarito comentado: 66 aulas de prática, do #01 ao #66

Linux · Git · Docker · Terraform · Kubernetes · 12 módulos · 73 questões · 2 desafios integradores

Autor Desconhecido · theknowledgeisle.com

Consulte este gabarito só depois de tentar.

As respostas são comentadas para ensinar, não só para conferir. Nos exercícios de código existem outras soluções igualmente válidas: o que importa é entender por que a sua funciona e o que ela troca em relação à mostrada aqui.

Os programas com um quadro verde de **Saída** foram executados com o Bash 5.2, o Git 2.54 e o Python 3.13 no Linux (Debian), e a saída mostrada é a real.

Sumário

Módulo	Tema	Aulas	Questões
MÓDULO 1	Linux e o terminal	#01–#06	Q1–Q10
MÓDULO 2	Git e colaboração	#07–#11	Q11–Q17
MÓDULO 3	Containers	#12–#18	Q18–Q23
MÓDULO 4	CI/CD	#19–#24	Q24–Q28
MÓDULO 5	Infraestrutura como código	#25–#30	Q29–Q35
MÓDULO 6	Observabilidade	#31–#34	Q36–Q42
MÓDULO 7	AWS na prática	#35–#38	Q43–Q48
MÓDULO 8	Kubernetes	#39–#41	Q49–Q54
MÓDULO 9	Segurança, custos, resiliência e cultura	#42–#47	Q55–Q61
MÓDULO 10	Capstone: o sistema completo	#48–#52	Q62–Q64
MÓDULO 11	Outras nuvens e plataformas	#53–#60	Q65–Q68
MÓDULO 12	IA em operações e extras	#61–#66	Q69–Q73
DESAFIOS	Desafios integradores	todas	D1–D2

MÓDULO 1

Linux e o terminal

O terminal, o sistema de arquivos, permissões, processos e serviços, shell script e SSH.

Artigos 01 a 06

#01 O Terminal Não é o Inimigo
#02 Navegando pelo Sistema de Arquivos com Confiança
#03 Permissões, Usuários e Grupos no Linux
#04 Processos, Serviços e o Comando `systemctl`
#05 Shell Script do Zero
#06 SSH: Conectando e Gerenciando Servidores Remotos

Perguntas conceituais — respostas

Q1 Caminho absoluto e relativo

> *Estando em `/home/ana/projetos`, para onde levam `cd ..`, `cd ../..`, `cd ~/docs`, `cd -` e `cd /var/log`? Por que um script agendado no cron deve usar caminhos absolutos?*

- `cd ..`: `/home/ana` (o diretório pai).
- `cd ../..`: `/home`.
- `cd ~/docs`: `/home/ana/docs` (o `~` é o diretório pessoal do usuário).
- `cd -`: volta ao diretório **anterior**, onde se estava antes do último `cd`.
- `cd /var/log`: `/var/log`, porque um caminho que começa com `/` é absoluto e não depende de onde se está.

O cron executa os *scripts* com um diretório de trabalho e um `PATH` mínimos, diferentes do terminal interativo: um caminho relativo como `backups/` aponta para outro lugar (normalmente o diretório pessoal do usuário), e comandos fora de `/usr/bin` e `/bin` podem nem ser encontrados. Um *script* agendado usa caminhos absolutos ou começa com `cd "$(dirname "$0")"`.

Q2 Lendo permissões

> *O que significa `-rwxr-x---` num arquivo, e qual o equivalente numérico? Por que `chmod 777` é quase sempre um erro, e o que o bit de execução significa num diretório?*

O primeiro caractere é o tipo (`-` arquivo, `d` diretório, `l` *link*). Depois vêm três trios: **dono** `rwx` (ler, escrever, executar), **grupo** `r-x` (ler e executar) e **outros** `---` (nada). Em octal, r=4, w=2 e x=1: `750`. O `chmod 777` dá escrita a **qualquer usuário** do sistema: um processo comprometido de outro serviço pode alterar o arquivo, e, se for um *script* executado por *root* ou um arquivo servido pelo site, isso vira escalada de privilégio ou *defacement*. Quase sempre o problema real é de **dono** ou de **grupo**, e se resolve com `chown` e permissões como `755` e `644`. Num diretório, o `x` é a permissão de **atravessar**: entrar nele e acessar o que há dentro pelo nome; o `r` só permite listar os nomes.

Q3 Processos e sinais

Qual a diferença entre `kill PID` e `kill -9 PID`? Por que o primeiro deve ser sempre tentado antes, e o que é um processo zumbi?

O `kill PID` envia o **SIGTERM** (15): um pedido educado para terminar. O processo pode tratá-lo, fechando arquivos, terminando a transação em andamento, apagando arquivos temporários e avisando os clientes. O `kill -9` envia o **SIGKILL**, que não pode ser tratado nem ignorado: o kernel encerra o processo na hora, sem nenhuma limpeza, o que pode deixar arquivos corrompidos, *locks* órfãos e transações pela metade. Por isso o SIGKILL é o último recurso, depois de alguns segundos de espera pelo SIGTERM, exatamente o que o `systemctl stop` e o `docker stop` fazem. Um **zumbi** é um processo que já terminou, mas cujo pai ainda não leu o código de saída (com `wait`): não consome CPU nem memória, só uma entrada na tabela de processos. Muitos zumbis indicam um pai com *bug*.

Q4 systemd

O que é uma unit de serviço do systemd? O que `systemctl enable --now nginx` faz, e onde ver o log de um serviço que não sobe?

Uma *unit* é um arquivo de configuração (em `/etc/systemd/system/` ou `/lib/systemd/system/`) que descreve um serviço: o comando que o inicia (`ExecStart`), o usuário, as dependências (`After=network.target`), a política de reinício (`Restart=on-failure`) e o ambiente. O `enable` cria os *links* para o serviço subir **no boot**; o `--now` também o inicia agora (sem ele, só no próximo boot). Os *logs* ficam no journal: `journalctl -u nginx -n 50` mostra as últimas 50 linhas, `-f` acompanha em tempo real, e `systemctl status nginx` mostra o estado e as últimas linhas. Depois de editar uma *unit*, é preciso `systemctl daemon-reload`.

Q5 set -euo pipefail

O que cada opção de `set -euo pipefail` faz no início de um script Bash, e que problema cada uma evita?

- ** `-e` *: *encerra o *script* no primeiro comando que falhar (saída diferente de zero). Sem ele, um `cd /pasta-errada` falha e o `rm -rf *` seguinte apaga o diretório atual.
- ** `-u` **: trata variável não definida como erro. Sem ele, um nome digitado errado vira texto vazio: `rm -rf "$DIRETORO/"` vira `rm -rf "/"`.
- ** `-o pipefail` *: *o código de saída de um *pipe* passa a ser o do primeiro comando que falhou, e não só o do último. Sem ele, `comando_que_falha | grep x` "dá certo" se o `grep` der certo.

O `-e` tem exceções (comandos dentro de `if`, à esquerda de `&&` e `||`), por isso não substitui a verificação explícita onde importa.

Q6 SSH com chaves

Por que autenticar no servidor com chave SSH é mais seguro que com senha? Que duas linhas do `sshd_config` todo servidor exposto à internet deveria ter?

Com senha, qualquer um na internet pode tentar adivinhá-la, e os servidores recebem milhares de tentativas automáticas por dia; uma senha fraca ou reutilizada cai. Com chave, o servidor guarda só a chave **pública** (em `~/.ssh/authorized_keys`), e o cliente prova que tem a **privada** sem enviá-la: não há nada a adivinhar. A chave privada fica no seu computador, protegida por uma *passphrase*. No `sshd_config`: `PasswordAuthentication no` (só chaves) e `PermitRootLogin no` (entra-se como usuário comum e usa-se `sudo`, o que deixa rastro de quem fez o quê). Antes de aplicar, teste o acesso por chave numa **segunda** sessão aberta, para não ficar trancado do lado de fora.

Exercícios de código — respostas

Q7 Permissões na prática

Num diretório temporário, crie um arquivo e um diretório, mostre as permissões criadas com o `umask` padrão, depois aplique `chmod 640` ao arquivo e `chmod 750` ao diretório, e use a forma simbólica (`u+x`, `g-w`) para mais dois ajustes. Mostre as permissões com `stat` depois de cada passo.

```
cd "$(mktemp -d)"
umask 022
touch relatorio.txt
mkdir publico
mostrar() { stat -c '%A %a %n' relatorio.txt publico; echo "--"; }

echo "umask $(umask):"
mostrar

chmod 640 relatorio.txt
chmod 750 publico
echo "depois de 640 e 750:"
mostrar

chmod u+x relatorio.txt
chmod g-x,o+x publico
echo "depois de u+x e g-x,o+x:"
mostrar
```

Saída

```
umask 0022:
-rw-r--r-- 644 relatorio.txt
drwxr-xr-x 755 publico
--
depois de 640 e 750:
-rw-r----- 640 relatorio.txt
drwxr-x--- 750 publico
--
depois de u+x e g-x,o+x:
-rwxr----- 740 relatorio.txt
drwxr----x 741 publico
--
```

Com `umask 022`, arquivos nascem com `644` (666 menos 022) e diretórios com `755` (777 menos 022): arquivos não recebem `x` por padrão. O resultado final do diretório, `741`, é estranho de propósito: o grupo pode listar os nomes (`r`), mas não entrar nem abrir nada (sem `x`); os outros podem entrar e abrir um arquivo cujo nome já conhecem (`x`), mas não listar. As formas simbólicas alteram só o que é citado, enquanto a octal define tudo de uma vez.

Q8 Processos em segundo plano e sinais

*Escreva um script que inicie um processo filho em segundo plano que trata o SIGTERM (com `trap`) para "limpar" antes de sair; o pai espera um pouco, envia `kill` (SIGTERM), espera o filho com `wait` e mostra o código de saída. Depois repita com um filho que **ignora** o SIGTERM e precisa do `kill -9`.*

```
trabalhador() {
  trap 'echo "  filho: recebi SIGTERM, limpando arquivos temporários"; exit 0' TERM
  while true; do sleep 0.1; done
}

teimoso() {
  trap '' TERM  # ignora o SIGTERM
  while true; do sleep 0.1; done
}

trabalhador &
pid=$!
sleep 0.3
echo "pai: enviando SIGTERM para o filho"
kill "$pid"
wait "$pid"
echo "pai: filho terminou com código $?"

teimoso &
pid=$!
sleep 0.3
kill "$pid"
sleep 0.3
if kill -0 "$pid" 2>/dev/null; then
  echo "pai: o teimoso ignorou o SIGTERM, usando SIGKILL"
  kill -9 "$pid"
fi
wait "$pid" 2>/dev/null  # sem o aviso "Killed" que o Bash imprime com o PID
echo "pai: teimoso terminou com código $? (128 + 9)"
```

Saída

```
pai: enviando SIGTERM para o filho
  filho: recebi SIGTERM, limpando arquivos temporários
pai: filho terminou com código 0
pai: o teimoso ignorou o SIGTERM, usando SIGKILL
pai: teimoso terminou com código 137 (128 + 9)
```

O `kill -0` não envia sinal nenhum: só testa se o processo existe e se temos permissão para sinalizá-lo. O código de saída de um processo morto por sinal é 128 mais o número do sinal: 137 para o SIGKILL, 143 para

um SIGTERM não tratado. É assim que se reconhece, num *log* de contêiner, um processo morto por falta de memória (o OOM killer usa SIGKILL, daí os famosos `exit code 137`). O `trap` também serve para apagar arquivos temporários com `trap 'rm -f "$tmp"' EXIT`.

Q9 Um script de backup com rotação

> *Escreva `backup.sh` com `set -euo pipefail`, que receba um diretório de origem e um de destino, valide os argumentos, crie um `.tar.gz` com a data no nome e apague os backups com mais de 7 dias, mantendo sempre os 3 mais recentes. Simule backups antigos com `touch -d` para testar a rotação.*

`backup.sh`:

```
#!/usr/bin/env bash
set -euo pipefail

uso() { echo "uso: $0 ORIGEM DESTINO" >&2; exit 2; }
[[ $# -eq 2 ]] || uso
origem=$1
destino=$2
[[ -d $origem ]] || { echo "erro: origem '$origem' não existe" >&2; exit 1; }
mkdir -p "$destino"

hoje=${HOJE:-$(date +%F)}  # HOJE permite testar com uma data fixa
arquivo="$destino/backup-$hoje.tar.gz"
tar -czf "$arquivo" -C "$(dirname "$origem")" "$(basename "$origem")"
echo "criado: $(basename "$arquivo")"

# Os 3 mais recentes ficam sempre; dos demais, apaga os com mais de 7 dias.
mapfile -t todos < <(ls -1t "$destino"/backup-*.tar.gz)
for velho in "${todos[@]:3}"; do
  if [[ -n $(find "$velho" -mtime +7) ]]; then
    rm -- "$velho"
    echo "removido: $(basename "$velho")"
  fi
done
echo "restam: $(ls -1 "$destino" | wc -l) backup(s)"
```

```
mkdir -p site/paginas dest
echo "<h1>oi</h1>" > site/paginas/index.html
for dias in 2 5 9 15 30; do
  touch -d "$dias days ago" "dest/backup-antigo-${dias}d.tar.gz"
done

bash backup.sh site dest
echo "---"
bash backup.sh pasta-que-nao-existe dest 2>&1 || echo "(código de saída $?)"
bash backup.sh 2>&1 || echo "(código de saída $?)"
```

Saída

```
criado: backup-2026-10-20.tar.gz
removido: backup-antigo-9d.tar.gz
removido: backup-antigo-15d.tar.gz
removido: backup-antigo-30d.tar.gz
restam: 3 backup(s)
---
erro: origem 'pasta-que-nao-existe' não existe
(código de saída 1)
uso: backup.sh ORIGEM DESTINO
(código de saída 2)
```

Os *backups* de 2 e 5 dias ficaram: estão entre os 3 mais recentes, junto com o de hoje. O de 9 dias é o quarto mais recente e tem mais de 7 dias, então saiu, com os de 15 e 30. O `2>&1` no teste junta a saída de erro à normal, para as mensagens aparecerem na ordem em que acontecem. A regra "manter pelo menos N" protege contra o caso em que o *backup* parou de rodar há semanas: sem ela, a rotação apagaria **todos** os antigos, sem nenhum novo para substituí-los. As mensagens de erro vão para a saída de erro (`>&2`), e os códigos de saída distinguem "uso errado" (2) de "falha" (1). Um *backup* só está pronto quando a **restauração** foi testada: `tar -tzf` lista o conteúdo, e restaurar numa pasta temporária prova que funciona.

Q10 Analisando um log de acesso

Dado um trecho de log de acesso do nginx, use `awk`, `sort`, `uniq` e `grep` para mostrar: (a) as requisições por código de status; (b) os 2 IPs com mais requisições; (c) as URLs que deram erro 5xx; (d) o total de bytes enviados.

```
cat > access.log <<'LOG'
192.168.0.10 - - [20/Oct/2026:10:00:01 -0300] "GET / HTTP/1.1" 200 5120 "-" "Mozilla/5.0"
192.168.0.11 - - [20/Oct/2026:10:00:02 -0300] "GET /livros HTTP/1.1" 200 2048 "-" "Mozilla/5.0"
10.0.0.5 - - [20/Oct/2026:10:00:03 -0300] "POST /login HTTP/1.1" 401 512 "-" "curl/8.5"
10.0.0.5 - - [20/Oct/2026:10:00:04 -0300] "POST /login HTTP/1.1" 401 512 "-" "curl/8.5"
10.0.0.5 - - [20/Oct/2026:10:00:05 -0300] "POST /login HTTP/1.1" 401 512 "-" "curl/8.5"
192.168.0.10 - - [20/Oct/2026:10:00:06 -0300] "GET /checkout HTTP/1.1" 502 157 "-" "Mozilla/5.0"
192.168.0.12 - - [20/Oct/2026:10:00:07 -0300] "GET /livros/7 HTTP/1.1" 404 300 "-" "Mozilla/5.0"
192.168.0.11 - - [20/Oct/2026:10:00:08 -0300] "GET /api/estoque HTTP/1.1" 500 89 "-"
"Mozilla/5.0"
LOG

echo "(a) por status:"
awk '{print $9}' access.log | sort | uniq -c | sort -rn
echo "(b) top 2 IPs:"
awk '{print $1}' access.log | sort | uniq -c | sort -rn | head -2
echo "(c) erros 5xx:"
awk '$9 ~ /^5/ {print $9, $7}' access.log
echo "(d) total de bytes:"
awk '{total += $10} END {printf "%d bytes (%.1f KB)\n", total, total / 1024}' access.log
```

Saída

```
(a) por status:
      3 401
      2 200
      1 502
      1 500
      1 404
(b) top 2 IPs:
      3 10.0.0.5
      2 192.168.0.11
(c) erros 5xx:
502 /checkout
500 /api/estoque
(d) total de bytes:
9250 bytes (9.0 KB)
```

O `awk` divide cada linha por espaços: `$1` é o IP, `$7` a URL, `$9` o status e `$10` os bytes. O `uniq -c` só conta linhas **vizinhas** iguais, por isso o `sort` antes dele é obrigatório. As três tentativas de login com 401 vindas do mesmo IP, com `curl`, são o padrão de uma tentativa de força bruta: é esse tipo de análise que ferramentas como o fail2ban automatizam, bloqueando o IP no *firewall*. Num servidor real, o mesmo vale com `zcat access.log.*.gz` para os *logs* já rotacionados.

MÓDULO 2

Git e colaboração

Git na prática, Git Flow, a primeira automação com GitHub Actions, branches protegidas e pull requests, Semantic Versioning e tags.

Artigos 07 a 11

#07 Git na Prática: Commits, Branches e Merges sem Medo
#08 Trabalhando em Equipe com Git Flow
#09 GitHub Actions: Sua Primeira Automação de CI/CD
#10 Protegendo Branches e Revisando Código com Pull Requests
#11 Semantic Versioning e Tags de Release

Perguntas conceituais — respostas

Q11 As três áreas do Git

> *Explique o diretório de trabalho, a staging area (índice) e o repositório. O que `git add -p` permite, e por que a staging area existe?*

O **diretório de trabalho** são os arquivos como estão no disco, com as alterações em andamento. A **staging area** é o rascunho do próximo *commit*: só entra nele o que foi adicionado com `git add`. O **repositório** (`.git`) guarda os *commits* já feitos. A *staging area* existe para que um *commit* contenha **uma** mudança coerente, mesmo que o diretório de trabalho tenha várias misturadas: você corrigiu um *bug* e, de passagem, ajustou a formatação de outro arquivo; são dois *commits*. O `git add -p` vai além e mostra cada trecho alterado de um arquivo, perguntando se ele entra, o que permite separar duas mudanças feitas **no mesmo arquivo**.

Q12 merge, rebase e o histórico

> *Qual a diferença entre integrar uma branch com `git merge --no-ff` e com `git rebase`? Por que nunca se faz rebase de commits já publicados?*

O `merge --no-ff` cria um **commit de merge** com dois pais, mesmo quando um avanço rápido (*fast-forward*) seria possível: o histórico mostra que houve uma *branch*, quais *commits* eram dela e quando entrou. O `rebase` **reescreve** os *commits* da *branch* como se tivessem sido feitos depois da ponta da outra: o histórico fica linear, mas os *commits* ganham novos *hashes*, porque são *commits* novos. Se alguém já baixou os antigos, os dois históricos divergem, e a pessoa tem de reconciliar *commits* "duplicados", o que exigiria um `push --force` que apaga o trabalho de outros. Daí a regra: *rebase* só em *commits* locais, ainda não enviados; o que já foi publicado se integra com *merge*.

Q13 Branches protegidas

> *Que regras de proteção a branch `main` de um projeto em equipe deve ter no GitHub, e por quê?*

- **Exigir *pull request*** para qualquer mudança: nada vai direto para a `main`.
- **Exigir aprovação** de pelo menos um revisor, e invalidá-la se novos *commits* chegarem depois.

- **Exigir que as verificações do CI passem** (testes, *lint*, *build*) antes do *merge*.
- **Proibir *force push*** e a exclusão da *branch*.
- **Exigir a *branch* atualizada** com a `main` antes do *merge*, para que o CI tenha testado a combinação real.

Juntas, elas garantem que a `main` esteja sempre num estado que passou nos testes e foi vista por mais de uma pessoa, e que o histórico não possa ser reescrito. Também valem para administradores (a opção de não permitir exceções), senão a regra é só uma sugestão.

Q14 **Semantic Versioning**

> *Dada a versão `2.4.1` de uma biblioteca, qual é a próxima versão em cada caso: (a) corrigiu um bug sem mudar a API; (b) acrescentou uma função nova; (c) removeu um parâmetro de uma função existente. O que significa `^2.4.1` num `composer.json` ou `package.json`?*

- (a) **2.4.2**: correção, incrementa o PATCH.
- (b) **2.5.0**: funcionalidade nova compatível, incrementa o MINOR e zera o PATCH.
- (c) **3.0.0**: mudança incompatível, incrementa o MAJOR e zera os outros dois.

O `^2.4.1` aceita qualquer versão **compatível** segundo o SemVer: a partir de 2.4.1 e abaixo de 3.0.0. É uma promessa do autor da biblioteca, não uma garantia: por isso o *lockfile* fixa as versões exatas que foram testadas.

Exercícios de código — respostas

Q15 **Um fluxo com branches e merge --no-ff**

> *Num repositório novo, crie a `main` com um commit, a branch `dev` a partir dela, uma branch `feature/busca` a partir da `dev` com dois commits, e integre-a na `dev` com `git merge --no-ff`. Mostre o grafo com `git log --graph --oneline` (sem os hashes) e liste os commits de merge.*

```
export GIT_AUTHOR_NAME=Ana GIT_AUTHOR_EMAIL=ana@exemplo.com GIT_COMMITTER_NAME=Ana
GIT_COMMITTER_EMAIL=ana@exemplo.com
git init -q -b main loja && cd loja
echo "# Loja" > README.md
git add README.md && git commit -q -m "chore: commit inicial"
git switch -q -c dev

git switch -q -c feature/busca
echo "busca simples" > busca.txt
git add busca.txt && git commit -q -m "feat: busca por título"
echo "busca por autor" >> busca.txt
git commit -q -am "feat: busca por autor"

git switch -q dev
git merge -q --no-ff feature/busca -m "Merge branch 'feature/busca' into dev"
git branch -q -d feature/busca

echo "grafo:"
git log --graph --format='%s' --all
echo "merges:"
git log --merges --format='%s'
```

Saída

```
grafo:
*   Merge branch 'feature/busca' into dev
|\
| * feat: busca por autor
| * feat: busca por título
|/
* chore: commit inicial
merges:
Merge branch 'feature/busca' into dev
```

O grafo mostra a "bolha" da *branch*: os dois *commits* de busca ficam num ramo lateral, unidos à `dev` pelo *commit* de *merge*. Com um *fast-forward* (o padrão quando a `dev` não andou), os dois *commits* entrariam direto na linha da `dev`, sem nenhum registro de que formaram uma unidade. O `git branch -d` só apaga *branches* já integradas; o `-D`, que apaga mesmo as não integradas, perde trabalho.

Q16 Uma tag anotada e o changelog

> *No mesmo estilo, crie três commits seguindo o Conventional Commits (`feat:`, `fix:`, `docs:`), crie a tag anotada `v1.0.0`, mais dois commits (um `feat:` e um `fix:`), e gere, com `git log`, a lista das mudanças desde a tag, agrupada por tipo. Com base nela, diga qual deve ser a próxima versão.*

```
export GIT_AUTHOR_NAME=Ana GIT_AUTHOR_EMAIL=ana@exemplo.com GIT_COMMITTER_NAME=Ana
GIT_COMMITTER_EMAIL=ana@exemplo.com
git init -q -b main lib && cd lib
for msg in "feat: cálculo de frete" "fix: frete negativo para CEP vazio" "docs: README com
exemplos"; do
  git commit -q --allow-empty -m "$msg"
done
git tag -a v1.0.0 -m "primeira versão estável"

git commit -q --allow-empty -m "feat: frete grátis acima de um valor"
git commit -q --allow-empty -m "fix: arredondamento do frete por peso"

ultima=$(git describe --tags --abbrev=0)
echo "mudanças desde $ultima:"
for tipo in feat fix; do
  git log "$ultima..HEAD" --format='%s' | grep "^$tipo:" | sed "s/^$tipo: /  [$tipo] /"
done
if git log "$ultima..HEAD" --format='%s%n%b' | grep -q 'BREAKING CHANGE\|^[a-z]*!:'; then
nivel=major
elif git log "$ultima..HEAD" --format='%s' | grep -q '^feat'; then nivel=minor
else nivel=patch; fi
echo "próxima versão: incremento $nivel"
git tag -n1
```

Saída

```
mudanças desde v1.0.0:
  [feat] frete grátis acima de um valor
  [fix] arredondamento do frete por peso
próxima versão: incremento minor
v1.0.0          primeira versão estável
```

Houve um `feat` e nenhuma mudança incompatível: a próxima versão é **v1.1.0**. É exatamente o que ferramentas como o *semantic-release* e o *release-please* fazem, lendo as mensagens: a convenção dos *commits* vira versão e *changelog* automáticos. A *tag* anotada (`-a`) guarda autor, data e mensagem, como um *commit*; a leve (sem `-a`) é só um nome para um *hash*, e o `git describe` a ignora por padrão.

Q17 Uma primeira pipeline do GitHub Actions

Escreva `.github/workflows/ci.yml` que rode em todo push e pull request para `main` e `dev`: faça o checkout, configure o PHP 8.5 com cache das dependências do Composer, rode `composer install`, o lint e os testes. Depois confira, com Python, que o arquivo é YAML válido e liste os gatilhos e os passos.

`ci.yml`:

```
name: CI

on:
  push:
    branches: [main, dev]
  pull_request:
    branches: [main, dev]

jobs:
  testes:
    runs-on: ubuntu-latest
    timeout-minutes: 10
    steps:
      - uses: actions/checkout@v4

      - uses: shivammathur/setup-php@v2
        with:
          php-version: "8.5"
          coverage: none

      - name: Cache do Composer
        uses: actions/cache@v4
        with:
          path: vendor
          key: composer-${{ hashFiles('composer.lock') }}

      - name: Dependências
        run: composer install --no-interaction --prefer-dist

      - name: Lint
        run: find src tests -name '*.php' -print0 | xargs -0 -n1 php -l

      - name: Testes
        run: vendor/bin/phpunit
```

```
import yaml

with open("ci.yml", encoding="utf-8") as f:
    wf = yaml.safe_load(f)

gatilhos = wf[True]  # o YAML 1.1 lê a chave "on" como o booleano True
print("gatilhos:", ", ".join(f"{ev} -> {cfg['branches']}" for ev, cfg in gatilhos.items()))
for nome, job in wf["jobs"].items():
    print(f"job {nome} em {job['runs-on']} (até {job['timeout-minutes']} min):")
    for passo in job["steps"]:
        print("  -", passo.get("name") or passo["uses"])
```

Saída

```
gatilhos: push -> ['main', 'dev'], pull_request -> ['main', 'dev']
job testes em ubuntu-latest (até 10 min):
  - actions/checkout@v4
  - shivammathur/setup-php@v2
  - Cache do Composer
  - Dependências
  - Lint
  - Testes
```

A curiosidade do `wf[True]` é real: no YAML 1.1, que o PyYAML segue, `on`, `yes` e `off` são booleanos, e a chave `on:` vira `True`. O GitHub trata o arquivo do jeito certo, mas ferramentas genéricas de YAML tropeçam nisso. A chave do cache inclui o *hash* do `composer.lock`: o cache só é reaproveitado enquanto as dependências não mudam. O `timeout-minutes` impede que um teste travado consuma horas de execução. Com a proteção de *branch* da pergunta anterior, este *job* vira o portão do *merge*.

MÓDULO 3

Containers

O que é um container, os primeiros containers, o Dockerfile, volumes e redes, Docker Compose, boas práticas de imagens e a publicação em registries.

Artigos 12 a 18

#12 O Que é um Container e Por Que Isso Mudou Tudo
#13 Instalando e Rodando Seus Primeiros Containers
#14 Escrevendo um Dockerfile do Zero
#15 Docker Volumes e Redes: Persistência e Comunicação
#16 Docker Compose: Orquestrando Múltiplos Serviços Localmente
#17 Boas Práticas de Imagens: Leveza, Segurança e Camadas
#18 Publicando Imagens no Docker Hub e GitHub Container Registry

Perguntas conceituais — respostas

Q18 Container não é máquina virtual

Qual a diferença entre um container e uma máquina virtual? Que recursos do kernel Linux tornam o container possível?

Uma **máquina virtual** emula um computador inteiro, com o próprio kernel e o próprio sistema operacional, sobre um *hypervisor*: isola muito bem, mas cada VM ocupa gigabytes e leva dezenas de segundos para subir. Um **container** é um processo comum do Linux, que **compartilha o kernel** do hospedeiro, mas enxerga uma visão isolada do sistema. Quem cria esse isolamento são os **namespaces** (PID, rede, pontos de montagem, usuários, *hostname*: o processo vê só os seus processos, a sua rede e o seu sistema de arquivos) e os **cgroups** (limites e contabilidade de CPU, memória e E/S). O sistema de arquivos vem de uma **imagem** em camadas. Resultado: sobe em milissegundos e ocupa megabytes, com um isolamento mais fraco que o de uma VM, já que um *bug* no kernel afeta todos os containers.

Q19 Imagem, container e camadas

Qual a diferença entre imagem e container? Por que a ordem das instruções no Dockerfile afeta o tempo de build?

A **imagem** é um modelo somente leitura: um conjunto de **camadas**, uma por instrução que altera arquivos (`RUN`, `COPY`, `ADD`), mais os metadados (comando inicial, variáveis, porta). O **container** é uma instância em execução de uma imagem, com uma camada gravável por cima, que desaparece com ele. No *build*, cada instrução vira uma camada guardada em cache; quando uma instrução muda (ou os arquivos que ela copia mudam), **ela e todas as seguintes** são refeitas. Por isso o que muda pouco vem primeiro (instalar pacotes do sistema, copiar o arquivo de dependências e instalá-las) e o código da aplicação, que muda a cada *commit*, vem por último.

Q20 Volume ou bind mount

> *Para o banco de dados de um container MySQL, por que usar um volume nomeado? E quando um bind mount (uma pasta do hospedeiro) é a escolha certa?*

Os dados gravados na camada do container **somem** quando ele é removido, e atualizar a imagem do MySQL exige recriar o container. Um **volume nomeado** (`-v dados_mysql:/var/lib/mysql`) é administrado pelo Docker, sobrevive à remoção do container, tem bom desempenho e permissões corretas. O ***bind mount*** (`-v ./src:/var/www/html`) liga uma pasta específica do hospedeiro: é o certo em **desenvolvimento**, para editar o código no seu editor e ver a mudança no container na hora, e para entregar arquivos de configuração. Para dados de produção, o volume; para código em desenvolvimento, o *bind mount*.

Q21 Uma imagem mínima e segura

> *Cite quatro práticas que deixam uma imagem menor e mais segura, e por que rodar o processo como root dentro do container é um risco.*

- **Imagem base pequena** e com versão fixa (`php:8.5-fpm-alpine`, *distroless*), nunca `latest`: menos pacotes significam menos vulnerabilidades e *downloads* menores.
- ***Build* em múltiplos estágios**: compiladores, ferramentas de *build* e o cache do Composer ficam no estágio de *build*, e a imagem final recebe só o resultado.
- ** `.dockerignore` **: `.git`, `.env`, `node_modules` e *logs* nunca entram no contexto, o que evita vazar segredos e invalidar o cache.
- **Usuário sem privilégios** (`USER app`) e nenhum segredo em `ENV` ou `ARG` (eles ficam gravados nas camadas e aparecem no `docker history`).

Como *root*, o processo tem o *root* do kernel compartilhado; com uma falha de escape de container, ou um volume montado por engano, o atacante ganha o *root* do hospedeiro. Um usuário comum limita o estrago.

Exercícios de código — respostas

Q22 Um Dockerfile em múltiplos estágios para PHP

> *Escreva um `Dockerfile` para uma aplicação PHP com Composer: um estágio que instala as dependências sem as de desenvolvimento, e um estágio final com `php:8.5-apache`, a extensão `pdo_mysql`, o OPcache, o código, um usuário sem privilégios e o `DocumentRoot` apontando para `public/`. Escreva também o `.dockerignore`. Mostre o caminho equivalente sem Docker, num servidor Debian.*

```
# Estágio 1: dependências (o Composer não vai para a imagem final)
FROM composer:2 AS deps
WORKDIR /app
COPY composer.json composer.lock ./
RUN composer install --no-dev --no-scripts --prefer-dist --no-interaction --optimize-autoloader

# Estágio 2: aplicação
FROM php:8.5-apache
RUN docker-php-ext-install pdo_mysql opcache \
 && a2enmod rewrite \
 && sed -ri 's!/var/www/html!/var/www/html/public!g' /etc/apache2/sites-available/000-
default.conf
COPY docker/opcache.ini /usr/local/etc/php/conf.d/opcache.ini
WORKDIR /var/www/html
COPY --from=deps /app/vendor ./vendor
COPY . .
RUN chown -R www-data:www-data storage
USER www-data
```

```
# .dockerignore
.git
.env
vendor
storage/logs/*
tests
node_modules
*.md
```

O `COPY composer.json composer.lock` vem antes do `COPY . .` para que o `composer install` só rode de novo quando as dependências mudarem. O `vendor` local fica fora do contexto (`.dockerignore`) porque a imagem tem o seu, montado no estágio 1; e o `.env` nunca entra na imagem: as variáveis chegam na execução (`docker run --env-file .env`). Com `USER www-data`, o Apache não consegue escutar na porta 80, que exige privilégio: num arranjo real, mude a porta para 8080 em `ports.conf` e no *virtual host*, ou deixe o Apache iniciar como *root* (ele próprio troca para `www-data` nos processos que atendem as requisições) e retire o `USER`.

Sem Docker, num Debian, o mesmo resultado sai de: `sudo apt install apache2 libapache2-mod-php8.5 php8.5-mysql php8.5-opcache composer` (o PHP 8.5 pode exigir o repositório de pacotes de Ondřej Surý, que traz as versões novas), `sudo a2enmod rewrite`, o código em `/var/www/livraria` com `composer install --no-dev -o`, o `DocumentRoot /var/www/livraria/public` no *virtual host* em `/etc/apache2/sites-available/`, `sudo chown -R www-data: /var/www/livraria/storage` e `sudo systemctl reload apache2`.

Q23 Um docker-compose para a aplicação

> *Escreva um `compose.yaml` com três serviços: a aplicação PHP (construída do Dockerfile anterior, porta 8080), um MySQL 8.4 com volume nomeado e um healthcheck, e um Redis. A aplicação só deve subir depois que o MySQL estiver saudável, e a senha do banco deve vir de um arquivo `.env`. Confira com Python a estrutura: a ordem de dependência e os volumes declarados.*

`compose.yaml`:

```
services:
  app:
    build: .
    ports:
      - "8080:80"
    environment:
      DB_HOST: db
      DB_NAME: livraria
      DB_USER: livraria
      DB_PASS: ${DB_PASS:?defina DB_PASS no .env}
      REDIS_HOST: cache
    depends_on:
      db:
        condition: service_healthy
      cache:
        condition: service_started

  db:
    image: mysql:8.4
    environment:
      MYSQL_DATABASE: livraria
      MYSQL_USER: livraria
      MYSQL_PASSWORD: ${DB_PASS:?defina DB_PASS no .env}
      MYSQL_RANDOM_ROOT_PASSWORD: "yes"
    volumes:
      - dados_mysql:/var/lib/mysql
    healthcheck:
      test: ["CMD", "mysqladmin", "ping", "-h", "127.0.0.1"]
      interval: 5s
      timeout: 3s
      retries: 10

  cache:
    image: redis:7-alpine

volumes:
  dados_mysql:
```

```
import yaml

with open("compose.yaml", encoding="utf-8") as f:
    c = yaml.safe_load(f)

servicos = c["services"]
for nome, s in servicos.items():
    origem = s.get("image") or "build " + s["build"]
    deps = s.get("depends_on", {})
    condicoes = ", ".join(f"{d} ({cfg['condition']})" for d, cfg in deps.items()) or "nenhuma"
    print(f"{nome:5} {origem:15} depende de: {condicoes}")

usados = {v.split(":")[0] for s in servicos.values() for v in s.get("volumes", [])}
print("volumes usados e declarados batem?", usados == set(c["volumes"]))
print("porta publicada só na app?", [n for n, s in servicos.items() if "ports" in s] == ["app"])
```

Saída

```
app   build .         depende de: db (service_healthy), cache (service_started)
db    mysql:8.4       depende de: nenhuma
cache redis:7-alpine  depende de: nenhuma
volumes usados e declarados batem? True
porta publicada só na app? True
```

O `depends_on` com `service_healthy` espera o *healthcheck* do MySQL passar, o que leva alguns segundos depois de o container subir; sem isso, a aplicação tentaria conectar a um banco que ainda está inicializando. Só a aplicação publica porta: o MySQL e o Redis ficam acessíveis apenas na rede interna do Compose, pelos nomes `db` e `cache`. O `${DB_PASS:?...}` faz o `docker compose up` falhar com uma mensagem clara se a variável não estiver definida, em vez de subir o banco com senha vazia. Sem Docker, o equivalente é instalar `mysql-server` (ou `mariadb-server`) e `redis-server` com o `apt`, criar o banco e o usuário com `CREATE DATABASE` e `CREATE USER`, e apontar o `.env` da aplicação para `127.0.0.1`.

MÓDULO 4

CI/CD

O que é CI/CD, pipelines de build, teste e deploy, testes no pipeline, variáveis e secrets, deploy automático e monitoramento do pipeline.

Artigos 19 a 24

#19 O Que é CI/CD e Por Que Sua Empresa Precisa Disso
#20 Pipeline com GitHub Actions: Build, Test e Deploy Automático
#21 Testes Automatizados no Pipeline: Qualidade sem Atrito
#22 Variáveis de Ambiente e Secrets em Pipelines
#23 Deploy Automático para Servidores com GitHub Actions
#24 Notificações, Logs e Monitoramento do Pipeline

Perguntas conceituais — respostas

Q24 Integração, entrega e implantação contínuas

> *Diferencie integração contínua, entrega contínua e implantação contínua. O que precisa ser verdade sobre os testes para que a implantação contínua seja segura?*

- **Integração contínua (CI)**: todo *push* é integrado à *branch* principal com frequência (pelo menos diariamente) e verificado automaticamente: *build*, testes e *lint*. Os problemas de integração aparecem em minutos, e não na véspera da entrega.
- **Entrega contínua**: além disso, todo *commit* que passa no pipeline gera um artefato **pronto para produção**; o deploy é um botão, apertado por uma pessoa quando o negócio quiser.
- **Implantação contínua**: o botão some; todo *commit* que passa vai para produção sozinho.

Para a implantação contínua ser segura, a suíte de testes precisa ser **confiável** (sem testes intermitentes que todos ignoram), cobrir os fluxos críticos e rodar rápido; e o deploy precisa de rede de proteção: monitoramento, *rollback* automático e, de preferência, liberação gradual (*canary*) ou *feature flags*.

Q25 Secrets no pipeline

> *Como o GitHub Actions protege os secrets, e por que um pull request vindo de um fork não recebe os secrets do repositório? Que erro comum ainda vaza um segredo?*

Os *secrets* ficam cifrados nas configurações do repositório ou do ambiente e só são entregues ao *job* em execução, como variáveis; o GitHub mascara os valores conhecidos nos *logs* (aparecem como `***`). Um PR de *fork* roda código escrito por **qualquer pessoa**: se recebesse os *secrets*, bastaria um `curl` no *workflow* alterado para enviá-los ao atacante; por isso o evento `pull_request` de *forks* roda sem eles (e o `pull_request_target`, que os tem, exige muito cuidado para nunca executar o código do PR). Erros que ainda vazam: imprimir o segredo **transformado** (codificado em Base64, cortado, dentro de uma URL), que a máscara não reconhece; gravá-lo num artefato ou numa imagem Docker; e usar *actions* de terceiros sem fixar a versão pelo *hash* do *commit*.

Q26 O pipeline como portão

Num pipeline com as etapas lint, testes unitários, build da imagem, testes de integração, deploy em homologação e deploy em produção, em que ordem e com que dependências elas devem rodar? O que deve acontecer se os testes de integração falharem?

Do mais **barato e rápido** para o mais caro: *lint* e testes unitários primeiro (podem rodar em paralelo, levam segundos e pegam a maioria dos erros); depois o *build* da imagem, **uma vez só**; os testes de integração usam essa imagem; o deploy em homologação usa a **mesma** imagem; e a produção, também a mesma, depois de uma aprovação manual ou de verificações automáticas em homologação. Se os testes de integração falharem, o pipeline **para**: nada é implantado, o autor é notificado e a imagem não é promovida. "Construir uma vez, promover a mesma" garante que o que foi testado é exatamente o que chega à produção.

Exercícios de código — respostas

Q27 Um deploy por SSH com rollback

Escreva `deploy.sh`, que simule localmente a estratégia de releases com symlink: cria `releases/<versão>`, copia o código, troca o symlink `current` de forma atômica, roda uma verificação de saúde e, se ela falhar, volta o symlink para a versão anterior. Mantenha só as 3 últimas releases. Teste um deploy bom, um ruim e mais três bons.

`deploy.sh`:

```
#!/usr/bin/env bash
set -euo pipefail

base=$(realpath -m "$1") versao=$2 codigo=$3  # absoluto: o symlink não pode depender de onde se
está
releases="$base/releases"
nova="$releases/$versao"
mkdir -p "$releases"
anterior=$(readlink "$base/current" 2>/dev/null || true)

cp -r "$codigo" "$nova"
ln -sfn "$nova" "$base/current.tmp"
mv -T "$base/current.tmp" "$base/current"  # rename é atômico: nunca existe um "current" pela
metade
echo "[$versao] ativa"

if ! grep -q "OK" "$base/current/health.txt"; then
  echo "[$versao] verificação de saúde falhou"
  if [[ -n $anterior ]]; then
    ln -sfn "$anterior" "$base/current.tmp" && mv -T "$base/current.tmp" "$base/current"
    echo "[$versao] rollback para $(basename "$anterior")"
  fi
  rm -rf "$nova"
  exit 1
fi

ls -1d "$releases"/* | sort | head -n -3 | xargs -r rm -rf  # mantém as 3 mais recentes
```

```
mkdir -p codigo
for v in v1.0.0 v1.1.0 v1.2.0 v1.3.0 v1.4.0; do
  if [[ $v == v1.1.0 ]]; then echo "ERRO" > codigo/health.txt; else echo "OK" >
codigo/health.txt; fi
  bash deploy.sh srv "$v" codigo || echo "  (deploy de $v falhou com código $?)"
  echo "  current -> $(basename "$(readlink srv/current)")"
done
echo "releases guardadas: $(ls srv/releases | tr '\n' ' ')"
```

Saída

```
[v1.0.0] ativa
  current -> v1.0.0
[v1.1.0] ativa
[v1.1.0] verificação de saúde falhou
[v1.1.0] rollback para v1.0.0
  (deploy de v1.1.0 falhou com código 1)
  current -> v1.0.0
[v1.2.0] ativa
  current -> v1.2.0
[v1.3.0] ativa
  current -> v1.3.0
[v1.4.0] ativa
  current -> v1.4.0
releases guardadas: v1.2.0 v1.3.0 v1.4.0
```

A troca em dois passos (`ln -sfn` num nome temporário e `mv -T` por cima do `current`) é o que torna a mudança atômica: o `mv` é um *rename*, uma operação única do sistema de arquivos, e o servidor web nunca encontra o *link* inexistente ou apontando para uma pasta incompleta. A *release* com falha nem fica guardada, e o *rollback* é só trocar o *symlink* de volta. No servidor de verdade, o *script* roda via `ssh deploy@servidor 'bash -s' < deploy.sh`, a verificação é um `curl -fsS http://localhost/health`, e o código chega por `rsync` ou `git archive`. As versões têm um só dígito em cada parte; com `v1.10.0`, o `sort` simples erraria a ordem, e o certo seria `sort -V`.

Q28 Testes com serviços no pipeline

Escreva um job do GitHub Actions que rode os testes de integração contra um MySQL 8.4 de verdade, usando `services` com healthcheck, e passe as credenciais por variáveis de ambiente, com a senha vinda de um secret. Confira com Python que o serviço tem healthcheck e que nenhuma senha está escrita em texto no arquivo.

`integracao.yml`:

```
name: Integração

on:
  pull_request:
    branches: [main]

jobs:
  integracao:
    runs-on: ubuntu-latest
    services:
      mysql:
        image: mysql:8.4
        env:
          MYSQL_DATABASE: livraria_teste
          MYSQL_ROOT_PASSWORD: ${{ secrets.MYSQL_TESTE_PASS }}
        ports:
          - 3306:3306
        options: >-
          --health-cmd "mysqladmin ping -h 127.0.0.1"
          --health-interval 5s
          --health-timeout 3s
          --health-retries 10
    env:
      DB_HOST: 127.0.0.1
      DB_NAME: livraria_teste
      DB_USER: root
      DB_PASS: ${{ secrets.MYSQL_TESTE_PASS }}
    steps:
      - uses: actions/checkout@v4
      - uses: shivammathur/setup-php@v2
        with:
          php-version: "8.5"
      - run: composer install --no-interaction
      - run: mysql -h 127.0.0.1 -uroot -p"$DB_PASS" livraria_teste < schema.sql
      - run: vendor/bin/phpunit --testsuite integracao
```

```
import re
import yaml

texto = open("integracao.yml", encoding="utf-8").read()
job = yaml.safe_load(texto)["jobs"]["integracao"]
mysql = job["services"]["mysql"]

print("tem healthcheck?", "--health-cmd" in mysql["options"])
print("variáveis do job:", ", ".join(job["env"]))
senhas = [v for k, v in {**mysql["env"], **job["env"]}.items() if "PASS" in k]
print("todas as senhas vêm de secrets?", all(re.fullmatch(r"\$\{\{ secrets\.\w+ \}\}", s) for s
in senhas))
```

Saída

```
tem healthcheck? True
variáveis do job: DB_HOST, DB_NAME, DB_USER, DB_PASS
todas as senhas vêm de secrets? True
```

O contêiner de serviço sobe antes dos passos, e o runner espera o *healthcheck* passar. O MySQL de teste é descartado ao fim do *job*: cada execução começa com um banco limpo, carregado do `schema.sql`. Para um banco que só existe durante o teste, a senha poderia até estar no arquivo, mas manter o hábito de usar *secrets* evita que o mesmo arquivo seja copiado um dia para algo que importa. A verificação em Python é o embrião de uma política automática: ferramentas como o `gitleaks` e o *secret scanning* do GitHub fazem o mesmo em todo o repositório.

MÓDULO 5

Infraestrutura como código

Terraform (recursos na AWS, módulos e o state), Ansible e a combinação dos dois.

Artigos 25 a 30

#25 Introdução ao Terraform: Infraestrutura que Você Pode Versionar
#26 Seus Primeiros Recursos na AWS com Terraform
#27 Módulos no Terraform: Reusabilidade e Organização
#28 State do Terraform: Entendendo o Arquivo Mais Crítico do Projeto
#29 Ansible: Configurando Servidores de Forma Declarativa
#30 Terraform + Ansible: Do Provisionamento à Configuração

Perguntas conceituais — respostas

Q29 Declarativo ou imperativo

> *O Terraform é **declarativo**; um script Bash que chama a CLI da AWS é **imperativo**. Explique a diferença com o exemplo "quero 3 servidores", e o que acontece em cada caso quando já existem 2.*

No **imperativo**, você descreve os **passos**: "crie 3 servidores". Rodando o *script* com 2 já existentes, ele cria mais 3, e passam a existir 5, a menos que você escreva a lógica de verificar o que existe, contar e criar só a diferença. No **declarativo**, você descreve o **estado desejado**: "devem existir 3 servidores". O Terraform compara esse desejo com o que existe (registrado no *state* e consultado na nuvem), calcula a diferença e cria **1**. Rodar de novo não faz nada. É essa propriedade (convergir para o estado desejado, de forma idempotente) que permite versionar a infraestrutura no Git e aplicá-la quantas vezes for preciso.

Q30 O arquivo de state

> *O que o `terraform.tfstate` guarda? Por que ele não deve ir para o Git, e por que um backend remoto com lock é obrigatório numa equipe?*

O *state* é o mapa entre os recursos do código (`aws_instance.web`) e os objetos reais na nuvem (o id `i-0abc...`), com os atributos conhecidos de cada um. Sem ele, o Terraform não sabe o que já criou. Ele não vai para o Git porque guarda **segredos em texto puro** (senhas de banco geradas, chaves) e porque versões diferentes em *branches* diferentes levariam a aplicar sobre um mapa desatualizado, criando duplicatas ou destruindo recursos. O *backend* remoto (um *bucket* S3, com o *lock* nativo do S3 ou uma tabela do DynamoDB nas versões antigas; o Terraform Cloud; o GitLab) guarda uma cópia única, cifrada e versionada, e o **lock** impede que duas pessoas rodem `apply` ao mesmo tempo e corrompam o *state*.

Q31 Ler o plano antes de aplicar

> *Num `terraform plan`, o que significam os símbolos `+`, `-`, `~` e `-/+`? Por que um `-/+` num banco de dados deve parar tudo?*

- `+`: o recurso será **criado**.
- `-`: será **destruído**.

- `~`: será **alterado no lugar**, sem recriar.
- `-/+`: será **destruído e recriado**, porque um atributo que não pode ser alterado no lugar mudou (o nome do banco, a zona de disponibilidade, o tipo de disco em alguns casos).

Num banco de dados, `-/+` significa **apagar o banco e criar um vazio**: todos os dados se perdem, salvo *snapshot*. O plano mostra qual atributo força a recriação (`# forces replacement`). A defesa é ler todo plano antes do `apply` e, para recursos com estado, usar `lifecycle { prevent_destroy = true }`, que faz o Terraform recusar qualquer plano que destrua o recurso.

Q32 Terraform e Ansible juntos

> *Por que o curso combina Terraform e Ansible, em vez de usar só um deles? Qual é a fronteira entre os dois?*

Cada um é melhor numa camada. O **Terraform** provisiona a **infraestrutura**: rede, máquinas, banco gerenciado, DNS, permissões, recursos que a API da nuvem cria e destrói, e cujo ciclo de vida ele acompanha pelo *state*. O **Ansible** configura **o que roda dentro** das máquinas: pacotes, arquivos de configuração, usuários, serviços. Ele se conecta por SSH, não precisa de agente nem de *state* e é idempotente por módulo. A fronteira típica: o Terraform cria as VMs e gera, como saída, o inventário (os IPs) que o Ansible usa para instalá-las e configurá-las. Quando a aplicação roda em containers ou em serviços gerenciados, a parte do Ansible encolhe ou desaparece: a configuração vai para a imagem.

Exercícios de código — respostas

Q33 Uma VPC e um servidor com Terraform

> *Escreva o Terraform para criar, na AWS, uma VPC `10.0.0.0/16` com uma sub-rede pública, um security group que libere a porta 443 para a internet e a 22 só para o seu IP (recebido como variável), e uma instância `t3.micro` com Debian. Use variáveis, tags comuns e uma saída com o IP público.*

```
terraform {
  required_version = ">= 1.9"
  required_providers {
    aws = { source = "hashicorp/aws", version = "~> 6.0" }
  }
}

provider "aws" {
  region = var.regiao
  default_tags {
    tags = { projeto = "livraria", ambiente = var.ambiente, gerenciado_por = "terraform" }
  }
}

variable "regiao" {
  type    = string
  default = "sa-east-1"
}
variable "ambiente" {
  type    = string
  default = "homologacao"
}
variable "meu_ip" {
  type        = string
  description = "IP liberado para SSH, no formato 203.0.113.10/32"
  validation {
    condition     = can(cidrhost(var.meu_ip, 0)) && endswith(var.meu_ip, "/32")
    error_message = "Informe um único IP, com /32."
  }
}

resource "aws_vpc" "principal" {
  cidr_block           = "10.0.0.0/16"
  enable_dns_hostnames = true
}

resource "aws_internet_gateway" "gw" {
  vpc_id = aws_vpc.principal.id
}

resource "aws_subnet" "publica" {
  vpc_id                  = aws_vpc.principal.id
  cidr_block              = cidrsubnet(aws_vpc.principal.cidr_block, 8, 1) # 10.0.1.0/24
  map_public_ip_on_launch = true
}

resource "aws_route_table" "publica" {
  vpc_id = aws_vpc.principal.id
  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.gw.id
  }
}
```

```
resource "aws_route_table_association" "publica" {
  subnet_id      = aws_subnet.publica.id
  route_table_id = aws_route_table.publica.id
}

resource "aws_security_group" "web" {
  vpc_id = aws_vpc.principal.id
  ingress {
    from_port   = 443
    to_port     = 443
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = [var.meu_ip]
  }
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

data "aws_ami" "debian" {
  most_recent = true
  owners      = ["136693071363"] # conta oficial do Debian na AWS
  filter {
    name   = "name"
    values = ["debian-13-amd64-*"]
  }
}

resource "aws_instance" "web" {
  ami                    = data.aws_ami.debian.id
  instance_type          = "t3.micro"
  subnet_id              = aws_subnet.publica.id
  vpc_security_group_ids = [aws_security_group.web.id]
}

output "ip_publico" {
  value = aws_instance.web.public_ip
}
```

Fluxo: `terraform init` (baixa o *provider*), `terraform plan -var meu_ip=203.0.113.10/32 -out plano` (mostra o que vai acontecer), `terraform apply plano` (aplica exatamente o plano revisado) e, ao fim do estudo, `terraform destroy`, para não pagar pelo que ficou ligado. Sem uma sub-rede pública com rota para o *internet gateway*, a instância teria IP público e mesmo assim seria inalcançável: é o erro mais comum nos primeiros passos. O `validation` recusa um `0.0.0.0/0` na porta 22 antes de chegar à AWS. O código não foi

aplicado numa conta real para este caderno; o `terraform validate` confere a sintaxe e as referências sem criar nada.

Q34 Uma política contra destruição no plano

*O `terraform show -json plano` gera o plano em JSON. Escreva um script Python que leia esse JSON e **reprove** o plano (código de saída 1) se ele destruir ou recriar qualquer recurso dos tipos `aws_db_instance` ou `aws_s3_bucket`, listando as ações de todos os recursos. Teste com um plano de exemplo.*

```
import json
import sys

# Trecho no formato de "terraform show -json plano" (só os campos usados aqui).
plano = json.loads("""
{"resource_changes": [
  {"address": "aws_instance.web",         "type": "aws_instance",    "change": {"actions":
["update"]}},
  {"address": "aws_db_instance.principal","type": "aws_db_instance", "change": {"actions":
["delete", "create"]}},
  {"address": "aws_s3_bucket.capas",      "type": "aws_s3_bucket",   "change": {"actions": ["no-
op"]}},
  {"address": "aws_security_group.web",   "type": "aws_security_group", "change": {"actions":
["create"]}}
]}
""")

PROTEGIDOS = {"aws_db_instance", "aws_s3_bucket"}
SIMBOLO = {("create",): "+", ("delete",): "-", ("update",): "~", ("no-op",): " ",
           ("delete", "create"): "-/+", ("create", "delete"): "+/-"}

violacoes = []
for rc in plano["resource_changes"]:
    acoes = tuple(rc["change"]["actions"])
    print(f"{SIMBOLO.get(acoes, '?'):>3} {rc['address']}")
    if rc["type"] in PROTEGIDOS and "delete" in acoes:
        violacoes.append(rc["address"])

if violacoes:
    print("REPROVADO: o plano destrói recursos protegidos:", ", ".join(violacoes))
    sys.exit(1)
print("aprovado")
```

Saída

```
  ~ aws_instance.web
-/+ aws_db_instance.principal
    aws_s3_bucket.capas
  + aws_security_group.web
REPROVADO: o plano destrói recursos protegidos: aws_db_instance.principal
```

Num pipeline, essa verificação roda entre o `plan` e o `apply`: um código de saída diferente de zero para tudo antes que alguém aprove sem ler. É uma "política como código" mínima; ferramentas como o OPA (com o Conftest) e o Sentinel do Terraform Cloud fazem o mesmo com uma linguagem própria. A sequência `["delete", "create"]` é o `-/+` do plano em texto; com `create_before_destroy` no recurso, ela vira `["create", "delete"]`.

Q35 Idempotência como no Ansible

> *O módulo `lineinfile` do Ansible garante que uma linha exista num arquivo, e só altera o arquivo se precisar, informando `changed` ou `ok`. Escreva em Bash a função `garantir_linha ARQUIVO REGEX LINHA`, com o mesmo comportamento: se uma linha casar com a expressão regular, ela é substituída pela nova (se for diferente); senão, a linha é acrescentada. Rode-a duas vezes sobre um `sshd_config` de exemplo e mostre que a segunda não muda nada.*

```
garantir_linha() {
  local arquivo=$1 regex=$2 linha=$3
  if grep -qxF -- "$linha" "$arquivo"; then
    echo "ok:      $linha"
  elif grep -qE -- "$regex" "$arquivo"; then
    sed -i -E "s|$regex.*|$linha|" "$arquivo"
    echo "changed: $linha (substituída)"
  else
    echo "$linha" >> "$arquivo"
    echo "changed: $linha (acrescentada)"
  fi
}

cat > sshd_config <<'CFG'
Port 22
#PermitRootLogin prohibit-password
PasswordAuthentication yes
CFG

for rodada in 1 2; do
  echo "rodada $rodada:"
  garantir_linha sshd_config '^#?PermitRootLogin' 'PermitRootLogin no'
  garantir_linha sshd_config '^#?PasswordAuthentication' 'PasswordAuthentication no'
  garantir_linha sshd_config '^#?MaxAuthTries' 'MaxAuthTries 3'
done
echo "--- resultado:"
cat sshd_config
```

Saída

```
rodada 1:
changed: PermitRootLogin no (substituída)
changed: PasswordAuthentication no (substituída)
changed: MaxAuthTries 3 (acrescentada)
rodada 2:
ok:      PermitRootLogin no
ok:      PasswordAuthentication no
ok:      MaxAuthTries 3
--- resultado:
Port 22
PermitRootLogin no
PasswordAuthentication no
MaxAuthTries 3
```

A segunda rodada só diz `ok`: o arquivo já está no estado desejado. Isso permite rodar o *playbook* inteiro a cada mudança, sem medo, e o relatório de `changed` vira um detector de desvios: se alguém editou o servidor à mão, a próxima execução mostra e corrige. No Ansible, a mesma tarefa é `ansible.builtin.lineinfile` com `regexp`, `line` e um `notify` para reiniciar o `sshd` **só** quando algo mudou. Antes de reiniciar, `sshd -t` valida a configuração; um erro aqui tranca todos do lado de fora.

MÓDULO 6

Observabilidade

Métricas, logs e traces; Grafana, dashboards e alertas; rastreamento distribuído com OpenTelemetry; alertas inteligentes e resposta a incidentes.

Artigos 31 a 34

#31 Introdução ao Monitoramento: Métricas, Logs e Traces
#32 Grafana: Dashboards e Alertas que Fazem Sentido
#33 Rastreamento Distribuído com OpenTelemetry
#34 Alertas Inteligentes e Cultura de Resposta a Incidentes

Perguntas conceituais — respostas

Q36 Os três pilares

Qual pergunta cada pilar da observabilidade responde melhor: métricas, logs e traces? Dê um exemplo de uso de cada um durante um incidente de lentidão no checkout.

- **Métricas** respondem **"o quê e quanto"**: números agregados ao longo do tempo, baratos de guardar. "A latência p95 do *checkout* subiu de 300 ms para 4 s às 14h10, e a taxa de erros foi para 8%."
- ***Traces*** respondem **"onde"**: o caminho de uma requisição pelos serviços, com o tempo de cada etapa. "Das 4 s, 3,7 s foram gastos na chamada ao serviço de frete."
- ***Logs*** respondem **"por quê"**: o detalhe de um evento. "O serviço de frete registrou *timeout* ao consultar a transportadora, com o id da requisição."

O fluxo típico vai da métrica (o alerta disparou) ao *trace* (onde está a demora) e ao *log* (qual foi o erro), e o identificador de *trace* presente nos *logs* é o que liga um ao outro.

Q37 Por que percentis, e não a média

Por que um painel deve mostrar a latência p95 ou p99, e não a média? Dê um exemplo numérico.

A média esconde a cauda. Com 100 requisições, 95 em 100 ms e 5 em 5 s, a média é 345 ms, um número que não descreve a experiência de **ninguém**: a maioria teve 100 ms, e 5 usuários esperaram 5 segundos. O p95 diz que "95% das requisições levaram até X", e o p99 mostra a cauda que afeta os usuários mais azarados, que muitas vezes são os mais valiosos (os carrinhos maiores fazem mais consultas). E um usuário que carrega uma página com 20 requisições quase certamente encontra pelo menos uma lenta da cauda. Percentis também não se somam nem se tiram médias entre servidores: calcule-os a partir de histogramas.

Q38 SLO e orçamento de erro

O que são SLI, SLO e orçamento de erro? Se o SLO de disponibilidade do checkout é 99,9% por mês, quanto tempo de indisponibilidade o mês permite, e o que a equipe faz quando o orçamento acaba?

O **SLI** (*indicador*) é a medida: a proporção de requisições do *checkout* que responderam com sucesso em menos de 1 s. O **SLO** (*objetivo*) é a meta para o SLI: 99,9% num mês. O **orçamento de erro** é o que sobra:

0,1%, que num mês de 30 dias são 43,2 minutos de indisponibilidade total (ou o equivalente em requisições com falha). Enquanto há orçamento, a equipe pode arriscar (implantar com frequência, fazer experimentos). Quando ele acaba, a prioridade muda por acordo prévio: congelam-se as funcionalidades novas e o esforço vai para a confiabilidade até o orçamento se recompor. O SLO transforma "o sistema precisa ser estável" numa regra objetiva que produto e engenharia aceitam juntos.

Q39 Alertas que acordam alguém

> *Por que alertar sobre "CPU acima de 80%" costuma ser um alerta ruim, e "taxa de erros do checkout consumindo o orçamento de erro 14 vezes mais rápido que o normal por 1 hora" é um bom? O que é fadiga de alertas?*

CPU alta é uma **causa possível**, não um sintoma que o usuário sente: o sistema pode estar ótimo com 90% de CPU (é para isso que ela existe) e péssimo com 20% (travado esperando o banco). Um alerta deve disparar quando **o usuário está sendo afetado** e alguém precisa agir agora. O alerta por **taxa de consumo do orçamento** (*burn rate*) mede justamente isso: a esse ritmo, o orçamento do mês acaba em uns 2 dias, e não é ruído passageiro, porque se sustentou por uma hora. A **fadiga de alertas** acontece quando chegam tantos alertas sem ação que as pessoas passam a ignorá-los, inclusive o que importa. Regra: todo alerta que acorda alguém deve ser urgente, real e acionável; o resto vira painel ou *ticket*.

Exercícios de código — respostas

Q40 Percentis de latência

> *Dado um conjunto de 20 latências (em ms) de um endpoint, calcule a média, a mediana, o p95 e o p99 com Python, e mostre como uma única requisição muito lenta afeta a média e o p99, mas não a mediana.*

```
import statistics

def percentil(dados, p):
    """Percentil pelo método do "posto mais próximo": o menor valor com p% dos dados até ele."""
    ordenados = sorted(dados)
    posto = max(1, -(-p * len(ordenados) // 100))  # teto de p*n/100
    return ordenados[posto - 1]

latencias = [92, 101, 95, 110, 98, 105, 97, 120, 99, 102,
             96, 103, 100, 94, 108, 115, 97, 101, 450, 99]

def resumo(rotulo, dados):
    print(f"{rotulo:<16} média {statistics.mean(dados):6.1f} | mediana
{statistics.median(dados):5.1f} | "
          f"p95 {percentil(dados, 95):5} | p99 {percentil(dados, 99):5}")

resumo("normal", latencias)
resumo("com 1 de 8 s", latencias[:-1] + [8000])
```

Saída

```
normal           média  119.1 | mediana 100.5 | p95   120 | p99   450
com 1 de 8 s     média  514.1 | mediana 101.0 | p95   450 | p99  8000
```

Uma única requisição de 8 s levou a média de 120 para mais de 500 ms, quando 19 das 20 requisições levaram cerca de 100 ms: a média "acusa" uma lentidão que quase ninguém viveu e, no caso normal, esconde a de 450 ms. A mediana nem se mexe. O p95 mostra o que 1 em cada 20 usuários sente; o p99, com só 20 amostras, é o próprio pior caso. Percentis exigem volume para fazer sentido, e é por isso que os sistemas de monitoramento os calculam sobre milhares de requisições, a partir de histogramas.

Q41 Orçamento de erro e burn rate

> *Escreva um script Python que, para um SLO de 99,9% num mês de 30 dias, calcule o orçamento de erro em minutos e em requisições (para 10 milhões de requisições no mês), e, dada a taxa de erros da última hora, a burn rate e em quantos dias o orçamento acabaria. Classifique em "página alguém" (burn rate ≥ 14,4), "abre ticket" (≥ 6) ou "normal".*

```
SLO = 0.999
DIAS = 30
REQUISICOES_MES = 10_000_000

orcamento = 1 - SLO
print(f"orçamento: {orcamento * DIAS * 24 * 60:.1f} minutos ou {orcamento * REQUISICOES_MES:,.0f}
requisições com erro por mês")

for taxa_erros in [0.0005, 0.008, 0.02]:
    burn = taxa_erros / orcamento  # 1,0 = gasta o orçamento exatamente em 30 dias
    dias = DIAS / burn
    acao = "página alguém" if burn >= 14.4 else "abre ticket" if burn >= 6 else "normal"
    print(f"erros {taxa_erros:6.2%} -> burn rate {burn:5.1f}x, orçamento acaba em {dias:5.1f}
dias: {acao}")
```

Saída

```
orçamento: 43.2 minutos ou 10,000 requisições com erro por mês
erros  0.05% -> burn rate   0.5x, orçamento acaba em  60.0 dias: normal
erros  0.80% -> burn rate   8.0x, orçamento acaba em   3.8 dias: abre ticket
erros  2.00% -> burn rate  20.0x, orçamento acaba em   1.5 dias: página alguém
```

Os limiares 14,4 e 6 vêm do livro *The Site Reliability Workbook*, do Google: 14,4 vezes por uma hora consome 2% do orçamento mensal, o que justifica acordar alguém; 6 vezes por seis horas consome 5%. Na prática, o alerta combina uma janela longa (a taxa se sustentou) com uma curta (e ainda está acontecendo), para não disparar por um pico que já passou nem demorar a detectar. Uma taxa de 0,05% de erros parece ruim, mas está dentro do orçamento: o SLO dá à equipe um critério para **não** reagir a tudo.

Q42 Logs estruturados com correlação

Configure o `logging` do Python para emitir uma linha JSON por evento, com data em ISO 8601 UTC, nível, mensagem, o nome do serviço e um `trace_id` que acompanha todos os logs de uma mesma requisição. Simule duas requisições, uma com erro, e mostre como filtrar os logs de uma delas.

```
import contextvars
import json
import logging
from datetime import datetime, timezone

trace_atual = contextvars.ContextVar("trace_id", default="-")

class FormatoJson(logging.Formatter):
    def format(self, r):
        evento = {
            "ts": datetime.fromtimestamp(r.created,
timezone.utc).isoformat(timespec="milliseconds"),
            "nivel": r.levelname,
            "servico": "checkout",
            "trace_id": trace_atual.get(),
            "msg": r.getMessage(),
        }
        evento.update(getattr(r, "campos", {}))
        return json.dumps(evento, ensure_ascii=False)

saida = logging.StreamHandler()
saida.setFormatter(FormatoJson())
log = logging.getLogger("checkout")
log.addHandler(saida)
log.setLevel(logging.INFO)

def finalizar(pedido, trace_id, falha=False):
    trace_atual.set(trace_id)
    log.info("finalizando pedido", extra={"campos": {"pedido": pedido}})
    if falha:
        log.error("frete indisponível", extra={"campos": {"pedido": pedido, "dependencia":
"frete"}})
    else:
        log.info("pedido finalizado", extra={"campos": {"pedido": pedido, "total": 149.7}})

import io, sys
buffer = io.StringIO()
saida.setStream(buffer)  # captura para filtrar em seguida
finalizar(1041, "a1b2")
finalizar(1042, "c3d4", falha=True)

linhas = [json.loads(l) for l in buffer.getvalue().splitlines()]
print("eventos da requisição c3d4:")
for e in linhas:
    if e["trace_id"] == "c3d4":
        sem_data = {k: v for k, v in e.items() if k != "ts"}
        print(" ", json.dumps(sem_data, ensure_ascii=False))
print("formato da data:", "ok" if linhas[0]["ts"].endswith("+00:00") else "?")
```

Saída

```
eventos da requisição c3d4:
  {"nivel": "INFO", "servico": "checkout", "trace_id": "c3d4", "msg": "finalizando pedido",
"pedido": 1042}
  {"nivel": "ERROR", "servico": "checkout", "trace_id": "c3d4", "msg": "frete indisponível",
"pedido": 1042, "dependencia": "frete"}
formato da data: ok
```

Com uma linha JSON por evento, a ferramenta de *logs* (Loki, Elasticsearch, CloudWatch) indexa cada campo, e a busca vira `trace_id = "c3d4"` ou `nivel = "ERROR" and dependencia = "frete"`, em vez de expressões regulares sobre texto livre. O `ContextVar` mantém o `trace_id` da requisição em andamento, inclusive em código assíncrono, sem passá-lo de função em função. Com o OpenTelemetry, esse id é o mesmo do *trace*, e o Grafana salta do *log* para o *trace* com um clique. A data em UTC evita a confusão de horários entre servidores em fusos diferentes e no horário de verão.

MÓDULO 7

AWS na prática

EC2, VPC e IAM; ECS e Lambda; RDS, ElastiCache e estratégias de dados; Route 53, CloudFront e ACM.

Artigos 35 a 38

#35 AWS na Prática: EC2, VPC e IAM em Profundidade
#36 ECS e Lambda: Containers e Serverless na AWS
#37 RDS, ElastiCache e Estratégias de Dados na AWS
#38 Route53, CloudFront e ACM: Rede e Entrega de Conteúdo

Perguntas conceituais — respostas

Q43 Sub-rede pública e privada

Numa VPC, o que torna uma sub-rede "pública" ou "privada"? Em qual delas ficam o load balancer, os servidores da aplicação e o banco RDS, e como os servidores privados baixam atualizações da internet?

Não é uma propriedade da sub-rede em si, mas da **tabela de rotas** associada a ela: uma sub-rede é pública se a rota `0.0.0.0/0` aponta para um *internet gateway*; é privada se não aponta. O **load balancer** fica nas públicas (recebe o tráfego da internet); os **servidores da aplicação** e o **RDS**, nas privadas, sem IP público, alcançáveis só de dentro da VPC. Para sair para a internet (atualizações, APIs externas), as privadas apontam `0.0.0.0/0` para um **NAT gateway** que fica numa sub-rede pública: a conexão de saída funciona, mas ninguém de fora consegue iniciar uma conexão de entrada. O NAT gateway é cobrado por hora e por GB, um custo que surpreende em ambientes de estudo.

Q44 IAM e o menor privilégio

*Por que a aplicação na EC2 deve usar uma **role** do IAM, e não chaves de acesso (`AWS_ACCESS_KEY_ID`) gravadas no servidor? O que é o princípio do menor privilégio numa política?*

Chaves de acesso são credenciais **permanentes**: gravadas num arquivo ou no código, vazam (num *commit*, num *backup*, numa imagem) e continuam valendo até alguém as revogar. Uma **role** associada à instância fornece credenciais **temporárias**, renovadas automaticamente pelo serviço de metadados, que nunca aparecem em arquivo nenhum; o SDK as encontra sozinho. O **menor privilégio** é conceder só as ações e os recursos necessários: a aplicação que grava capas no S3 recebe `s3:PutObject` e `s3:GetObject` no *bucket* `livraria-capas`, e não `s3:*` em `*`. Se a aplicação for comprometida, o estrago fica limitado ao que ela podia fazer.

Q45 ECS ou Lambda

Para cada parte da livraria, escolha entre ECS Fargate e Lambda: (a) a API principal, com tráfego constante; (b) o redimensionamento das capas enviadas; (c) um relatório que roda uma vez por noite e leva 40 minutos.

- (a) **ECS Fargate**: um serviço com containers sempre ligados atende o tráfego constante com latência previsível, sem *cold start*, e o custo por hora é menor que o da Lambda com carga contínua.
- (b) **Lambda**: disparada pelo evento de *upload* no S3, roda alguns segundos por imagem, escala sozinha com o volume e não custa nada quando não há *uploads*.
- (c) **ECS** (uma tarefa agendada pelo EventBridge): a Lambda tem limite de **15 minutos** por execução, então 40 minutos não cabem nela.

A regra prática: Lambda para trabalho curto, orientado a eventos e intermitente; containers para serviços de longa duração ou carga contínua.

Q46 CDN e cache

> *O que o CloudFront faz por um site com usuários no Brasil todo e servidor em São Paulo? Por que as URLs de arquivos estáticos costumam ter um hash no nome (`app.3f9a2c.css`)?*

O CloudFront é uma **CDN**: guarda cópias do conteúdo em pontos de presença espalhados pelo mundo, perto dos usuários. A primeira requisição numa região vai à origem; as seguintes são servidas pela cópia local, com menos latência e sem carregar o servidor. Também termina o TLS perto do usuário e absorve picos e ataques de volume. O *hash* no nome resolve o problema da **invalidação**: o arquivo pode ser servido com cache de um ano (`Cache-Control: max-age=31536000, immutable`), porque, quando o conteúdo muda, o nome muda junto e o HTML passa a apontar para um arquivo novo, sem esperar o cache expirar nem pagar por invalidações manuais. O HTML, que aponta para eles, tem cache curto.

Exercícios de código — respostas

Q47 Planejando sub-redes

> *Com o módulo `ipaddress` do Python, divida a VPC `10.0.0.0/16` em sub-redes `/20` e reserve as seis primeiras: pública, aplicação e dados, em duas zonas de disponibilidade. Mostre, para cada uma, o intervalo, o número de endereços utilizáveis na AWS (que reserva 5 por sub-rede) e verifique que não há sobreposição.*

```
import ipaddress
from itertools import combinations

vpc = ipaddress.ip_network("10.0.0.0/16")
subredes = vpc.subnets(new_prefix=20)
camadas = ["publica", "aplicacao", "dados"]
zonas = ["sa-east-1a", "sa-east-1b"]

plano = {}
for camada in camadas:
    for zona in zonas:
        plano[f"{camada}-{zona[-1]}"] = next(subredes)

for nome, rede in plano.items():
    print(f"{nome:12} {str(rede):15} {rede[0]} - {rede[-1]}  ({rede.num_addresses - 5}
utilizáveis)")

sobrepostas = [(a, b) for (a, ra), (b, rb) in combinations(plano.items(), 2) if ra.overlaps(rb)]
print("sobreposições:", sobrepostas or "nenhuma")
print("sub-redes /20 livres na VPC:", sum(1 for _ in subredes))
```

Saída

```
publica-a    10.0.0.0/20     10.0.0.0 - 10.0.15.255  (4091 utilizáveis)
publica-b    10.0.16.0/20    10.0.16.0 - 10.0.31.255  (4091 utilizáveis)
aplicacao-a  10.0.32.0/20    10.0.32.0 - 10.0.47.255  (4091 utilizáveis)
aplicacao-b  10.0.48.0/20    10.0.48.0 - 10.0.63.255  (4091 utilizáveis)
dados-a      10.0.64.0/20    10.0.64.0 - 10.0.79.255  (4091 utilizáveis)
dados-b      10.0.80.0/20    10.0.80.0 - 10.0.95.255  (4091 utilizáveis)
sobreposições: nenhuma
sub-redes /20 livres na VPC: 10
```

Os 5 endereços reservados pela AWS em cada sub-rede são o da rede, o do roteador da VPC, o do DNS, um reservado para uso futuro e o de *broadcast*. Deixar 10 blocos `/20` livres é proposital: acrescentar uma zona ou uma camada depois é fácil, e mudar o CIDR de uma VPC em uso não é. Evite também sobrepor a faixa da VPC à da rede do escritório ou de outras VPCs, senão um *peering* ou uma VPN entre elas fica impossível.

Q48 Auditando uma política do IAM

> *Escreva um script Python que analise uma política do IAM (JSON) e aponte: declarações com `"Action": "*"` ou serviço inteiro (`"s3:*"`), `"Resource": "*"` em ações de escrita, e ausência de condição em `iam:PassRole`. Teste com uma política ruim e uma boa.*

```
import json

def auditar(nome, politica):
    problemas = []
    for i, d in enumerate(politica["Statement"], 1):
        if d.get("Effect") != "Allow":
            continue
        acoes = d["Action"] if isinstance(d["Action"], list) else [d["Action"]]
        recursos = d["Resource"] if isinstance(d["Resource"], list) else [d["Resource"]]
        for a in acoes:
            if a == "*" or a.endswith(":*"):
                problemas.append(f"#{i}: ação ampla demais '{a}'")
            escrita = any(p in a for p in ("Put", "Delete", "Create", "Update", "*"))
            if escrita and "*" in recursos:
                problemas.append(f"#{i}: '{a}' em qualquer recurso")
            if a == "iam:PassRole" and "Condition" not in d:
                problemas.append(f"#{i}: iam:PassRole sem condição")
    print(f"{nome}: {'aprovada' if not problemas else 'reprovada'}")
    for p in problemas:
        print("   -", p)

ruim = json.loads("""{"Version": "2012-10-17", "Statement": [
  {"Effect": "Allow", "Action": "s3:*", "Resource": "*"},
  {"Effect": "Allow", "Action": ["iam:PassRole"], "Resource": "*"}
]}""")

boa = json.loads("""{"Version": "2012-10-17", "Statement": [
  {"Effect": "Allow", "Action": ["s3:GetObject", "s3:PutObject"],
   "Resource": "arn:aws:s3:::livraria-capas/*"}
]}""")

auditar("política ruim", ruim)
auditar("política boa", boa)
```

Saída

```
política ruim: reprovada
   - #1: ação ampla demais 's3:*'
   - #1: 's3:*' em qualquer recurso
   - #2: iam:PassRole sem condição
política boa: aprovada
```

O `iam:PassRole` sem restrição é especialmente perigoso: permite entregar **qualquer** *role* (inclusive uma de administrador) a um serviço, como uma Lambda que a própria aplicação cria, uma escalada de privilégio clássica. A versão boa restringe as ações e o recurso a um prefixo de um *bucket*. Na AWS, o IAM Access Analyzer faz essa análise com muito mais regras e ainda sugere políticas a partir do uso real registrado no CloudTrail.

MÓDULO 8

Kubernetes

Orquestrando containers em escala, EKS e Kubernetes em produção: segurança, GitOps e deploys avançados.

Artigos 39 a 41

#39 Introdução ao Kubernetes: Orquestrando Containers em Escala
#40 EKS: Kubernetes Gerenciado na AWS
#41 Kubernetes em Produção: Segurança, GitOps e Deploys Avançados

Perguntas conceituais — respostas

Q49 Pod, Deployment e Service

Explique o papel de um Pod, de um Deployment e de um Service. Por que nunca se cria um Pod diretamente em produção?

O **Pod** é a menor unidade: um ou mais containers que compartilham rede e volumes, agendados juntos num nó. É **efêmero**: se o nó cai, o Pod morre e ninguém o recria. O **Deployment** declara o estado desejado ("3 réplicas desta imagem") e, por meio de um ReplicaSet, mantém esse número de Pods, recriando os que morrem e fazendo a troca gradual de versão (*rolling update*). O **Service** dá um nome de DNS e um IP estáveis a um conjunto de Pods selecionados por *labels*, distribuindo o tráfego entre eles, mesmo que os Pods mudem de IP a cada recriação. Um Pod criado diretamente não tem quem o recrie, não escala e não faz atualização gradual: é o equivalente a rodar um processo sem nenhum supervisor.

Q50 Liveness e readiness

Qual a diferença entre a liveness probe e a readiness probe? Por que usar a mesma verificação, que testa o banco de dados, nas duas pode derrubar o sistema inteiro?

A ***readiness*** responde "este Pod pode receber tráfego agora?". Se falha, o Pod é **retirado do Service** (não recebe requisições), mas continua vivo, por exemplo enquanto aquece o cache ou espera uma dependência. A ***liveness*** responde "este processo travou e precisa ser reiniciado?". Se falha, o kubelet **mata e reinicia** o container. Se a *liveness* testar o banco, uma lentidão do banco faz **todos** os Pods falharem na verificação ao mesmo tempo, e o Kubernetes reinicia todos, e de novo, e de novo: uma falha parcial vira indisponibilidade total, e os reinícios ainda sobrecarregam o banco que estava se recuperando. A *liveness* deve testar só o próprio processo (o servidor HTTP responde?); as dependências ficam, com cuidado, na *readiness*.

Q51 requests e limits

O que são `requests` e `limits` de CPU e memória num container? O que acontece quando um container passa do limite de memória, e quando passa do de CPU?

O **request** é a reserva: o agendador só coloca o Pod num nó que tenha esse tanto livre, e ele é a base para o autoscaler calcular a utilização. O **limit** é o teto. Passando do limite de **memória**, o container é **morto** pelo kernel (OOMKilled, o código de saída 137 do módulo 1) e reiniciado. Passando do de **CPU**, ele não morre: é **estrangulado** (*throttled*), recebendo menos tempo de processador e ficando lento. Sem `requests`, o

agendador amontoa Pods num nó que não os comporta; sem limite de memória, um vazamento num Pod derruba os vizinhos do mesmo nó. Muitas equipes definem limite de memória igual ao *request* e deixam a CPU sem limite, para não sofrer *throttling* desnecessário.

Q52 **GitOps**

> *O que é GitOps, e qual a diferença entre o pipeline "empurrar" (o CI roda `kubectl apply` no cluster) e o modelo "puxar" do Argo CD ou do Flux?*

GitOps é usar um repositório Git como **fonte única da verdade** do estado do cluster: os manifestos declarados ali são o que deve estar rodando, e toda mudança passa por *commit* e *pull request*. No modelo **empurrar**, o CI tem credenciais de administrador do cluster e aplica as mudanças; se alguém alterar o cluster à mão, nada percebe a divergência, e as credenciais poderosas ficam no CI. No modelo **puxar**, um agente **dentro** do cluster (Argo CD, Flux) observa o repositório e reconcilia continuamente: aplica o que mudou no Git e **desfaz** alterações manuais (*drift*). O CI só precisa de permissão para escrever no repositório, a auditoria é o próprio histórico do Git, e o *rollback* é um `git revert`.

Exercícios de código — respostas

Q53 **Um Deployment pronto para produção**

> *Escreva os manifestos de um Deployment da API da livraria (3 réplicas, imagem com tag fixa, `requests` e `limits`, readiness e liveness em rotas diferentes, usuário sem privilégios e sistema de arquivos somente leitura) e do Service. Depois valide com Python: as probes usam rotas diferentes, todos os containers têm limite de memória e nenhuma imagem usa `Latest`.*

`api.yaml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: livraria-api
  labels: {app: livraria-api}
spec:
  replicas: 3
  selector:
    matchLabels: {app: livraria-api}
  strategy:
    rollingUpdate: {maxUnavailable: 0, maxSurge: 1}
  template:
    metadata:
      labels: {app: livraria-api}
    spec:
      securityContext:
        runAsNonRoot: true
        runAsUser: 10001
      containers:
        - name: api
          image: ghcr.io/livraria/api:1.4.2
          ports:
            - containerPort: 8080
          resources:
            requests: {cpu: 250m, memory: 256Mi}
            limits: {memory: 256Mi}
          readinessProbe:
            httpGet: {path: /ready, port: 8080}
            periodSeconds: 5
          livenessProbe:
            httpGet: {path: /health, port: 8080}
            initialDelaySeconds: 10
            periodSeconds: 10
            failureThreshold: 3
          securityContext:
            allowPrivilegeEscalation: false
            readOnlyRootFilesystem: true
          envFrom:
            - secretRef: {name: livraria-api-env}
---
apiVersion: v1
kind: Service
metadata:
  name: livraria-api
spec:
  selector: {app: livraria-api}
  ports:
    - port: 80
      targetPort: 8080
```

```
import yaml

docs = list(yaml.safe_load_all(open("api.yaml", encoding="utf-8")))
deploy = next(d for d in docs if d["kind"] == "Deployment")
service = next(d for d in docs if d["kind"] == "Service")
containers = deploy["spec"]["template"]["spec"]["containers"]

for c in containers:
    rd, lv = c["readinessProbe"]["httpGet"]["path"], c["livenessProbe"]["httpGet"]["path"]
    print(f"{c['name']}: readiness {rd}, liveness {lv} -> {'ok' if rd != lv else 'mesma rota!'}")
    print(f"  limite de memória: {c['resources'].get('limits', {}).get('memory', 'AUSENTE')}")
    tag = c["image"].rsplit(":", 1)[-1] if ":" in c["image"].split("/")[-1] else "latest"
    print(f"  imagem {c['image']} -> tag {'proibida' if tag == 'latest' else tag}")

seletor = service["spec"]["selector"]
rotulos = deploy["spec"]["template"]["metadata"]["labels"]
print("o Service encontra os Pods?", all(rotulos.get(k) == v for k, v in seletor.items()))
```

Saída

```
api: readiness /ready, liveness /health -> ok
  limite de memória: 256Mi
  imagem ghcr.io/livraria/api:1.4.2 -> tag 1.4.2
o Service encontra os Pods? True
```

O `maxUnavailable: 0` com `maxSurge: 1` faz a atualização subir um Pod novo, esperar a *readiness* dele e só então desligar um antigo: a capacidade nunca cai durante o deploy. O sistema de arquivos somente leitura impede que um invasor grave ferramentas no container; se a aplicação precisar de uma pasta temporária, monta-se um volume `emptyDir` em `/tmp`. O erro mais comum de iniciante, o seletor do Service que não casa com os *labels* dos Pods, faz o Service existir sem nenhum destino; a última verificação pega isso. No cluster, `kubectl apply -f api.yaml` e `kubectl rollout status deployment/livraria-api` acompanham a troca.

Q54 A conta do autoscaler

O HPA (Horizontal Pod Autoscaler) calcula as réplicas desejadas com `desejadas = teto(atuais × métrica_atual / métrica_alvo)`, ignora variações dentro de uma tolerância de 10% e respeita os limites mínimo e máximo. Implemente o cálculo em Python e simule uma rampa de carga com alvo de 60% de CPU, mínimo de 2 e máximo de 10 réplicas.

```
import math

def desejadas(atuais, uso, alvo=60, minimo=2, maximo=10, tolerancia=0.10):
    razao = uso / alvo
    if abs(razao - 1) <= tolerancia:
        return atuais  # dentro da tolerância: não mexe
    return max(minimo, min(maximo, math.ceil(atuais * razao)))

replicas = 2
print("minuto  uso médio  réplicas")
for minuto, uso in enumerate([30, 64, 90, 120, 95, 70, 40, 20], 1):
    nova = desejadas(replicas, uso)
    seta = "=" if nova == replicas else ("↑" if nova > replicas else "↓")
    print(f"{minuto:6}  {uso:8}%  {replicas:2} {seta} {nova}")
    replicas = nova
```

Saída

```
minuto  uso médio  réplicas
     1        30%   2 = 2
     2        64%   2 = 2
     3        90%   2 ↑ 3
     4       120%   3 ↑ 6
     5        95%   6 ↑ 10
     6        70%  10 = 10
     7        40%  10 ↓ 7
     8        20%   7 ↓ 3
```

No minuto 2, 64% está a menos de 10% do alvo, e nada muda. No minuto 6, o cálculo pediria 12 réplicas (10 × 70/60, arredondado para cima), e o máximo segurou em 10. Depois que a carga cai, o uso médio **por réplica** despenca (a mesma carga dividida por mais Pods), e o HPA reduz em saltos. Na prática, a redução é amortecida por uma janela de estabilização (5 minutos por padrão), para não desligar Pods numa queda momentânea e religá-los em seguida. O uso é medido em relação ao **request** de CPU, por isso um Deployment sem `requests` não pode ser escalado pelo HPA por CPU.

MÓDULO 9

Segurança, custos, resiliência e cultura

DevSecOps, compliance e auditoria, FinOps, chaos engineering, platform engineering e a cultura DevOps.

Artigos 42 a 47

#42 DevSecOps: Integrando Segurança em Todo o Pipeline
#43 Compliance e Auditoria: LGPD, SOC2 e Evidências Automatizadas
#44 Performance e FinOps: Otimizando Custo e Velocidade na Cloud
#45 Resiliência e Chaos Engineering
#46 Platform Engineering: Construindo a Plataforma Interna de Desenvolvimento
#47 Cultura DevOps: Maturidade, Postmortems e Melhoria Contínua

Perguntas conceituais — respostas

Q55 Shift left

> *O que significa "levar a segurança para a esquerda" num pipeline? Cite quatro verificações automáticas e em que etapa cada uma roda.*

Significa encontrar os problemas de segurança **o mais cedo possível** no ciclo (à esquerda, numa linha do tempo que vai do código à produção), quando corrigir custa minutos, e não depois do deploy, quando custa um incidente. Verificações:

- **Varredura de segredos** (gitleaks, *secret scanning*): no *pre-commit* e em todo *push*, antes que a chave chegue ao histórico.
- **SAST** (análise estática do código, como o Semgrep e o CodeQL): no *pull request*, aponta injeção de SQL, XSS e uso inseguro de APIs.
- **SCA** (análise de dependências, como `composer audit`, `npm audit` e o Dependabot): no PR e diariamente, porque vulnerabilidades novas surgem em versões antigas.
- **Varredura da imagem** (Trivy, Grype): depois do *build*, nos pacotes do sistema dentro do container; e **DAST** (OWASP ZAP) contra o ambiente de homologação.

O cuidado é calibrar: um pipeline que bloqueia por qualquer alerta de baixa gravidade ensina a equipe a ignorá-lo.

Q56 LGPD na infraestrutura

> *Que obrigações da LGPD afetam diretamente decisões de infraestrutura e de operação? Cite três exemplos práticos.*

- **Segurança e prevenção** (art. 46): dados pessoais cifrados em trânsito (TLS) e em repouso (discos e *backups*), controle de acesso com menor privilégio e registro de quem acessou o quê.
- **Minimização e retenção**: não guardar dado pessoal sem finalidade, e apagá-lo quando a finalidade acaba, inclusive nos *logs* (CPF e e-mail não vão para o *log* de aplicação) e nos *backups*, com política de expiração.

- **Comunicação de incidentes** (art. 48): avisar a ANPD e os titulares quando um incidente puder causar risco ou dano relevante. Isso exige saber **o que** vazou, o que depende de *logs* de auditoria e de um inventário dos dados.

Também pesa a **transferência internacional**: hospedar dados de brasileiros fora do país exige base legal e garantias contratuais, o que influencia a escolha da região da nuvem.

Q57 Onde o dinheiro vaza

Cite quatro fontes comuns de desperdício numa conta de nuvem e a correção de cada uma.

- **Recursos ociosos**: ambientes de homologação ligados de noite e no fim de semana, volumes e IPs órfãos de máquinas já apagadas, *snapshots* antigos. Correção: desligar por agendamento, *tags* obrigatórias com dono e ambiente, e limpeza periódica.
- **Superdimensionamento**: instâncias e bancos escolhidos "por garantia", usando 10% da CPU. Correção: *right-sizing* com base nas métricas reais.
- **Preço sob demanda para carga constante**: o que roda 24 horas por dia, o ano todo, custa bem menos com *Savings Plans* ou instâncias reservadas.
- **Transferência de dados**: NAT gateway processando tráfego que poderia ir por um *VPC endpoint* (para o S3, por exemplo), tráfego entre zonas e saída para a internet sem CDN.

O primeiro passo de FinOps não é cortar: é tornar o custo **visível** por equipe e por produto, para que quem decide a arquitetura veja o preço da decisão.

Q58 Chaos engineering

O que é chaos engineering, e por que ele começa com uma hipótese e um "raio de explosão" pequeno? Dê um experimento para a livraria.

É provocar falhas **de propósito**, de forma controlada, para descobrir as fraquezas antes que elas apareçam sozinhas numa sexta-feira à noite. Começa com uma **hipótese** sobre o estado estável ("se o Redis cair, o site continua respondendo, mais lento, com p95 abaixo de 1 s, porque a aplicação cai para o banco"), mede o sistema antes, injeta a falha e compara. O **raio de explosão** pequeno (um Pod, uma fração do tráfego, primeiro em homologação) e um botão de abortar garantem que o experimento não vire o incidente que ele queria prevenir. Experimento para a livraria: derrubar o serviço de frete por 5 minutos e verificar se o *checkout* continua aceitando pedidos com frete estimado, em vez de falhar.

Q59 Postmortem sem culpados

Por que um postmortem deve ser "sem culpados" (blameless)? Que seções ele precisa ter para gerar aprendizado, e não só um relato?

Porque, se o resultado de um erro for a punição de quem errou, as pessoas passam a **esconder** erros e informações, e a organização perde exatamente o que precisava saber. A pergunta útil não é "quem errou", mas "por que o sistema permitiu que um erro humano previsível causasse o incidente": um comando destrutivo sem confirmação, um alerta que não existia, um *runbook* desatualizado. Seções: resumo e impacto (quem foi afetado, por quanto tempo, quanto custou); linha do tempo com os horários (detecção, mitigação,

resolução); causas contribuintes (geralmente mais de uma); o que funcionou e o que atrapalhou; e **ações** com dono e prazo, acompanhadas até o fim. Um postmortem cujas ações ninguém executa é só um relato.

Exercícios de código — respostas

Q60 Varredura de segredos

Escreva um script Python que procure segredos em arquivos de texto com expressões regulares: chaves de acesso da AWS (`AKIA` seguido de 16 caracteres), chaves privadas PEM, tokens do GitHub (`ghp_`) e atribuições do tipo `senha = "..."` ou `password: ...`. Mostre arquivo e linha, mascarando o valor encontrado, e devolva código 1 se achar algo.

```
import re
import sys

REGRAS = {
    "chave AWS": re.compile(r"\bAKIA[0-9A-Z]{16}\b"),
    "chave privada": re.compile(r"-----BEGIN (?:RSA |EC |OPENSSH )?PRIVATE KEY-----"),
    "token GitHub": re.compile(r"\bghp_[A-Za-z0-9]{36}\b"),
    "senha no código": re.compile(r"(?i)\b(?:senha|password|pass|secret)\s*[:=]\s*['\"]?([^'\"\s]
{6,})"),
}

arquivos = {
    "config.php": "<?php\n$db = 'livraria';\n$senha = \"Sup3rS3cret0!\";\n",
    ".env.example": "DB_PASS=troque-aqui\nAWS_KEY=\n",
    "deploy.sh": "export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE\n",
    "README.md": "Rode `composer install` e configure o .env.\n",
}

def mascarar(s):
    return s[:4] + "*" * (len(s) - 4)

achados = 0
for nome, conteudo in arquivos.items():
    for n, linha in enumerate(conteudo.splitlines(), 1):
        for regra, rx in REGRAS.items():
            if m := rx.search(linha):
                valor = m.group(1) if m.groups() else m.group(0)
                if regra == "senha no código" and valor in {"troque-aqui", "changeme"}:
                    continue  # placeholder conhecido do .env.example
                print(f"{nome}:{n}: {regra}: {mascarar(valor)}")
                achados += 1

print(f"{achados} possível(is) segredo(s)")
sys.exit(1 if achados else 0)
```

Saída

```
config.php:3: senha no código: Sup3*********
deploy.sh:1: chave AWS: AKIA****************
2 possível(is) segredo(s)
```

O `.env.example` não acusou nada porque `troque-aqui` está na lista de valores de exemplo: sem essa exceção, a ferramenta reclamaria do próprio modelo de configuração, e a equipe aprenderia a ignorá-la. A chave do `deploy.sh` é a chave de exemplo da documentação da AWS (`AKIAIOSFODNN7EXAMPLE`); uma real deveria ser **revogada** imediatamente, porque apagar o arquivo não a tira do histórico do Git. Ferramentas como o gitleaks usam centenas de regras e a entropia do texto para achar segredos sem formato conhecido.

Q61 Comparando preços de instâncias

> *Escreva um script Python que compare o custo mensal de 4 servidores ligados o mês inteiro, em três modelos: sob demanda (US$ 0,0832 por hora cada), plano de economia de 1 ano (37% de desconto) e uma combinação de 2 reservados com 2 spot (70% de desconto, mas podem ser interrompidos). Calcule também o custo de um ambiente de homologação de 2 servidores sob demanda ligado só em horário comercial (10 h por dia, 22 dias por mês).*

```
HORA = 0.0832
HORAS_MES = 730  # média de horas num mês
DESC_PLANO, DESC_SPOT = 0.37, 0.70

def mensal(qtd, preco_hora, horas=HORAS_MES):
    return qtd * preco_hora * horas

cenarios = {
    "sob demanda": mensal(4, HORA),
    "plano de 1 ano": mensal(4, HORA * (1 - DESC_PLANO)),
    "2 plano + 2 spot": mensal(2, HORA * (1 - DESC_PLANO)) + mensal(2, HORA * (1 - DESC_SPOT)),
}
base = cenarios["sob demanda"]
for nome, custo in cenarios.items():
    print(f"{nome:17} US$ {custo:7.2f}/mês  ({1 - custo / base:5.1%} de economia)")

sempre_ligada = mensal(2, HORA)
comercial = mensal(2, HORA, horas=10 * 22)
print(f"homologação 24x7:      US$ {sempre_ligada:6.2f}/mês")
print(f"homologação comercial: US$ {comercial:6.2f}/mês ({1 - comercial / sempre_ligada:.0%} a
menos)")
```

Saída

```
sob demanda       US$  242.94/mês  ( 0.0% de economia)
plano de 1 ano    US$  153.05/mês  (37.0% de economia)
2 plano + 2 spot  US$  112.97/mês  (53.5% de economia)
homologação 24x7:      US$ 121.47/mês
homologação comercial: US$  36.61/mês (70% a menos)
```

Desligar a homologação fora do expediente economiza mais, em proporção, que qualquer desconto: o servidor desligado não custa nada (os discos continuam cobrados). As instâncias *spot* só servem a cargas que toleram interrupção com 2 minutos de aviso: *workers* de fila, processamento em lote, réplicas extras atrás de um *load balancer*, nunca o único servidor do banco. Os preços são ilustrativos: a tabela real muda por região e por geração de instância, e a calculadora da AWS é a fonte certa.

MÓDULO 10

Capstone: o sistema completo

Arquitetura, provisionamento, microsserviços, o pipeline completo e as operações em produção.

Artigos 48 a 52

#48 Projeto Capstone: Arquitetura do Sistema Completo
#49 Capstone: Provisionando a Infraestrutura Completa
#50 Capstone: Implementando os Microsserviços
#51 Capstone: Pipeline Completo de CI/CD
#52 Capstone: Operações em Produção e Retrospectiva da Jornada

Perguntas conceituais — respostas

Q62 Microsserviços: quando vale a pena

O capstone divide o sistema em microsserviços. Que problemas eles resolvem, que custos trazem, e por que muitas equipes deveriam começar com um monólito bem modularizado?

Resolvem problemas de **organização e escala**: equipes diferentes implantam serviços diferentes sem coordenar cada deploy, cada parte escala conforme a sua carga (o catálogo recebe 100 vezes mais leituras que o *checkout*), e uma falha fica contida num serviço. Os custos são reais: chamadas de rede no lugar de chamadas de função (latência e falhas parciais), consistência eventual entre bancos separados, pipelines, monitoramento e *tracing* distribuído para cada serviço, versionamento de contratos e muito mais infraestrutura para operar. Uma equipe pequena paga todos esses custos sem colher os benefícios. Um **monólito modular** (módulos com fronteiras claras e sem acessar as tabelas uns dos outros) é rápido de desenvolver e simples de operar, e permite extrair um serviço depois, quando houver um motivo concreto: carga, equipe ou ciclo de deploy diferentes.

Q63 Implantação gradual

Explique os deploys blue-green e canary. Que métricas decidem se o canary avança ou volta?

No ***blue-green***, a versão nova (verde) sobe completa ao lado da atual (azul); depois dos testes, o tráfego troca **de uma vez** para a verde, e a azul fica de prontidão para um retorno instantâneo. Exige o dobro de capacidade durante a troca. No ***canary***, a versão nova recebe uma **fração** do tráfego (1%, 5%, 25%...), e cada etapa só avança se ela se comportar tão bem quanto a atual: o estrago de um defeito fica limitado à fração. Métricas de decisão: **taxa de erros** e **latência** (p95, p99) do *canary* comparadas com as da versão estável **no mesmo período**, além de métricas de negócio (taxa de conversão do *checkout*). Ferramentas como o Argo Rollouts e o Flagger automatizam a análise e o retorno.

Exercícios de código — respostas

Q64 Decidindo um canary

*Escreva uma função Python que receba as contagens de requisições e erros e a latência p95 da versão estável e do canary num intervalo, e decida entre "avançar", "aguardar" (amostra pequena demais, menos de 500 requisições no canary) e "reverter" (taxa de erros do canary acima de 1,5 vez a da estável **e** acima de 0,5% em termos absolutos, ou p95 mais de 20% pior). Teste quatro cenários.*

```
def decidir(estavel, canary, min_amostra=500):
    if canary["req"] < min_amostra:
        return "aguardar", f"só {canary['req']} requisições no canary"
    erro_e = estavel["erros"] / estavel["req"]
    erro_c = canary["erros"] / canary["req"]
    if erro_c > 1.5 * erro_e and erro_c > 0.005:
        return "reverter", f"erros {erro_c:.2%} contra {erro_e:.2%}"
    if canary["p95"] > 1.2 * estavel["p95"]:
        return "reverter", f"p95 {canary['p95']} ms contra {estavel['p95']} ms"
    return "avançar", f"erros {erro_c:.2%}, p95 {canary['p95']} ms"

estavel = {"req": 95_000, "erros": 190, "p95": 310}
cenarios = {
    "saudável":       {"req": 5_000, "erros": 9, "p95": 318},
    "começou agora":  {"req": 120, "erros": 3, "p95": 900},
    "mais erros":     {"req": 5_000, "erros": 60, "p95": 305},
    "mais lento":     {"req": 5_000, "erros": 10, "p95": 420},
}
for nome, canary in cenarios.items():
    decisao, motivo = decidir(estavel, canary)
    print(f"{nome:14} -> {decisao:8} ({motivo})")
```

Saída

```
saudável       -> avançar  (erros 0.18%, p95 318 ms)
começou agora  -> aguardar (só 120 requisições no canary)
mais erros     -> reverter (erros 1.20% contra 0.20%)
mais lento     -> reverter (p95 420 ms contra 310 ms)
```

O cenário "começou agora" tem 2,5% de erros e p95 de 900 ms, e mesmo assim a decisão é aguardar: com 120 requisições, 3 erros podem ser acaso, e reverter por isso tornaria todo deploy instável. A exigência dupla para os erros (relativa **e** absoluta) evita reverter um *canary* que passou de 0,01% para 0,02%: "o dobro de quase nada" ainda é quase nada. Uma análise rigorosa usa testes estatísticos, como fazem o Kayenta (do Spinnaker) e o Argo Rollouts com consultas ao Prometheus.

MÓDULO 11

Outras nuvens e plataformas

Azure para quem conhece AWS, compute, storage e redes no Azure, AKS, Azure DevOps, GitLab CI/CD, GitLab self-hosted, Bitbucket e Gitea.

Artigos 53 a 60

#53 Azure para Quem Já Conhece AWS
#54 Compute, Storage e Redes no Azure
#55 Azure Kubernetes Service (AKS)
#56 Azure DevOps: Pipelines e Repositórios
#57 GitLab CI/CD: A Alternativa Enterprise ao GitHub Actions
#58 GitLab Self-Hosted: Soberania Total sobre Código e Pipelines
#59 Bitbucket e o Ecossistema Atlassian
#60 Gitea: Self-Hosted Leve para Times Menores

Perguntas conceituais — respostas

Q65 AWS para Azure

> *Para quem conhece a AWS, qual o equivalente no Azure de: VPC, EC2, S3, RDS para PostgreSQL, IAM roles para aplicações, EKS e CloudWatch? Que conceito do Azure não tem equivalente direto?*

- VPC → **Virtual Network (VNet)**.
- EC2 → **Virtual Machines**.
- S3 → **Blob Storage** (dentro de uma *Storage Account*).
- RDS para PostgreSQL → **Azure Database for PostgreSQL (Flexible Server)**.
- *Roles* do IAM para aplicações → **Managed Identities**, com permissões dadas pelo RBAC do Azure.
- EKS → **AKS**.
- CloudWatch → **Azure Monitor**, com o Log Analytics e o Application Insights.

O conceito sem equivalente direto é o **Resource Group**: todo recurso do Azure pertence a um, que agrupa o ciclo de vida (apagar o grupo apaga tudo dentro), as permissões e a contabilidade de custos. A organização acima dele (*subscriptions* e *management groups*) também é diferente da estrutura de contas da AWS.

Q66 GitLab CI e GitHub Actions

> *Compare a estrutura de um `.gitlab-ci.yml` com a de um workflow do GitHub Actions. O que são stages e o que a palavra-chave `needs` muda?*

No GitHub Actions, um *workflow* tem *jobs* que rodam **em paralelo** por padrão, e a ordem vem do `needs`. No GitLab CI, o arquivo declara uma lista de ***stages*** (`build`, `test`, `deploy`), e cada *job* pertence a um *stage*: os *jobs* de um *stage* rodam em paralelo, e o *stage* seguinte só começa quando **todos** os do anterior terminam. O `needs` no GitLab quebra essa barreira: um *job* passa a depender só dos *jobs* listados e pode começar assim que eles terminam, formando um grafo (DAG) e encurtando o pipeline. O GitLab também tem *runners* próprios fáceis de hospedar, registro de containers e *pipelines* de *merge request* integrados na mesma plataforma.

Q67 Hospedar a própria forja

Quando faz sentido hospedar o próprio GitLab ou Gitea, em vez de usar o GitHub ou o GitLab.com? Que responsabilidades a equipe assume?

Faz sentido quando há exigência de **soberania** (o código não pode sair da infraestrutura da empresa, por contrato ou regulação), rede isolada da internet, integração com sistemas internos, ou custo, quando há muitos usuários e as funcionalidades pagas do serviço hospedado não compensam. O Gitea é leve (um binário, roda numa VM pequena) e serve bem a times menores; o GitLab traz tudo (CI, registro, *issues*), mas pede bem mais recursos. A equipe assume **tudo** o que o serviço hospedado fazia: atualizações de segurança em dia (uma forja exposta e desatualizada é um alvo valioso), *backups* testados do banco e dos repositórios, alta disponibilidade, monitoramento, capacidade de disco e dos *runners*, e o suporte aos usuários.

Exercícios de código — respostas

Q68 Um pipeline do GitLab com needs

Escreva um `.gitlab-ci.yml` com os stages `build`, `test` e `deploy`: um job de build, dois de teste (unitários e lint), em que o lint não depende do build, e um deploy manual só na branch `main`. Use Python para calcular a ordem real de execução a partir do `needs`.

`gitlab-ci.yml`:

```
stages: [build, test, deploy]

default:
  image: php:8.5-cli

build:
  stage: build
  script:
    - composer install --no-dev -o
  artifacts:
    paths: [vendor/]

lint:
  stage: test
  needs: []  # não espera o build: começa junto com ele
  script:
    - find src -name '*.php' -print0 | xargs -0 -n1 php -l

unitarios:
  stage: test
  needs: [build]
  script:
    - vendor/bin/phpunit

deploy_producao:
  stage: deploy
  needs: [unitarios, lint]
  script:
    - ./deploy.sh
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
      when: manual
```

```
import yaml

ci = yaml.safe_load(open("gitlab-ci.yml", encoding="utf-8"))
stages = ci["stages"]
jobs = {k: v for k, v in ci.items() if isinstance(v, dict) and "script" in v}

def dependencias(nome):
    j = jobs[nome]
    if "needs" in j:
        return j["needs"]
    anteriores = stages[: stages.index(j["stage"])]  # sem needs: espera todos os stages
anteriores
    return [n for n, o in jobs.items() if o["stage"] in anteriores]

onda, feitos = 1, set()
while len(feitos) < len(jobs):
    prontos = sorted(n for n in jobs if n not in feitos and set(dependencias(n)) <= feitos)
    print(f"onda {onda}: {', '.join(prontos)}")
    feitos |= set(prontos)
    onda += 1

regra = jobs["deploy_producao"]["rules"][0]
print(f"deploy: {regra['when']} quando {regra['if']}")
```

Saída

```
onda 1: build, lint
onda 2: unitarios
onda 3: deploy_producao
deploy: manual quando $CI_COMMIT_BRANCH == "main"
```

Sem o `needs: []`, o *lint* esperaria o *stage* `build` inteiro, embora não use nada dele: o pipeline ficaria uma etapa mais longo à toa. O *artifact* `vendor/` é o que leva as dependências do *job* de *build* para o de testes, que rodam em containers diferentes. O `when: manual` cria um botão no pipeline: a entrega é contínua, a implantação não. O arquivo real se chama `.gitlab-ci.yml`, na raiz do repositório, e o GitLab oferece um validador (*CI Lint*) no próprio painel.

MÓDULO 12

IA em operações e extras

AIOps, pipelines e infraestrutura gerados com IA, detecção de anomalias em logs, geradores de senhas seguras e a instalação manual do MySQL.

Artigos 61 a 66

#61 AIOps na Prática: O Que a IA Já Faz em Operações Hoje
#62 Gerando e Revisando Pipelines com IA
#63 Detecção de Anomalias e Análise Inteligente de Logs com IA
#64 Geração de Infraestrutura com IA
#65 Geradores de Senhas Seguras: Implementações em 5 Linguagens de Programação
#66 Instalação Manual do MySQL no Debian, Arch, Fedora e openSUSE

Perguntas conceituais — respostas

Q69 Revisando o que a IA gerou

> *Um assistente de IA gerou um workflow do GitHub Actions e um módulo Terraform para você. Que verificações são obrigatórias antes de aceitá-los, e por que "funcionou no primeiro teste" não basta?*

- **Ler linha a linha**, como se fosse o PR de um colega novo: a IA produz código plausível, que pode usar versões de *actions* inexistentes ou antigas, opções que não existem, ou portas e permissões abertas demais (`0.0.0.0/0` no SSH, `permissions: write-all`, `s3:*`).
- **Fixar versões e conferir as fontes**: *actions* de terceiros pela *tag* ou pelo *hash* do *commit*, *providers* com versão restrita, nomes de pacotes conferidos no registro oficial. Um nome de pacote "inventado" pela IA pode ter sido registrado por um atacante (*slopsquatting*).
- **Rodar as ferramentas de sempre**: `terraform validate` e `plan` lido com atenção, `actionlint`, as verificações de segredos e de políticas do pipeline.
- **Nunca colar segredos** no *prompt*.

"Funcionou no primeiro teste" só prova o caminho feliz: as falhas de segurança e de custo aparecem quando alguém as explora ou quando chega a fatura.

Q70 Anomalia não é incidente

> *Um sistema de AIOps detecta uma "anomalia" no volume de logs de erro. Por que nem toda anomalia deve virar alerta, e o que diferencia a detecção por limiar fixo da detecção por desvio do padrão?*

Uma anomalia é algo **estatisticamente incomum**, não necessariamente ruim: o pico de acessos na Black Friday, um *deploy* que muda o formato dos *logs*, um robô de busca. Transformar toda anomalia em alerta reproduz a fadiga de alertas do módulo 6. O **limiar fixo** ("mais de 100 erros por minuto") é simples e explicável, mas não acompanha a sazonalidade: alto demais de madrugada, baixo demais no horário de pico. A **detecção por desvio** aprende o comportamento normal (a média e a dispersão de uma janela recente, ou por hora do dia) e sinaliza o que se afasta muito dele. O uso sensato é tratar anomalias como **contexto** para a investigação e como sinal fraco, combinado com os alertas de sintomas baseados em SLO.

Q71 Instalando um banco sem contêiner

O artigo #66 instala o MySQL manualmente em quatro famílias de distribuições. Quais são os pacotes e os comandos de serviço no Debian e no Fedora, e que passo de segurança nenhuma instalação deve pular?

- **Debian/Ubuntu**: `sudo apt install mariadb-server` (o Debian traz o MariaDB nos repositórios oficiais; o MySQL da Oracle vem do repositório APT dela, `mysql-apt-config`), depois `sudo systemctl enable --now mariadb`.
- **Fedora**: `sudo dnf install community-mysql-server` (ou `mariadb-server`), depois `sudo systemctl enable --now mysqld` (ou `mariadb`).

No Arch, o pacote oficial é o `mariadb`, que exige `mariadb-install-db` antes de iniciar o serviço; no openSUSE, `zypper install mariadb`. O passo que não se pula é o **endurecimento inicial**, com `mysql_secure_installation` (ou `mariadb-secure-installation`): remove os usuários anônimos e o banco `test` e impede o *root* remoto. Depois, cria-se um usuário próprio para a aplicação, com permissões só no banco dela, e o servidor escuta apenas em `127.0.0.1` (`bind-address`), a menos que outra máquina precise acessá-lo.

Exercícios de código — respostas

Q72 Detectando anomalias com z-score

*Dada a contagem de erros por minuto ao longo de uma hora, calcule, para cada minuto, o z-score em relação à média e ao desvio padrão dos 15 minutos **anteriores**, e sinalize os minutos com z acima de 3. Compare com a hora inteira como referência, e com uma versão robusta, baseada na mediana e no desvio absoluto mediano (MAD).*

```
import statistics

erros = [4, 5, 3, 6, 4, 5, 4, 3, 5, 6, 4, 5, 4, 3, 5,   # minutos 0-14: normal
         4, 6, 5, 4, 30, 5, 4, 3, 5, 4, 6, 5, 4, 5, 4,   # minuto 19: pico isolado
         12, 14, 13, 15, 14, 13, 15, 14, 16, 15, 14, 13]  # a partir do 30: novo patamar

JANELA, LIMITE = 15, 3.0
for minuto in range(JANELA, len(erros)):
    anteriores = erros[minuto - JANELA:minuto]
    media, desvio = statistics.mean(anteriores), statistics.stdev(anteriores)
    z = (erros[minuto] - media) / desvio
    if z > LIMITE:
        print(f"minuto {minuto:2}: {erros[minuto]:2} erros, média {media:4.1f}, z = {z:5.1f}  <-
anomalia")

print("versão robusta (mediana e MAD):")
for minuto in range(JANELA, len(erros)):
    anteriores = erros[minuto - JANELA:minuto]
    mediana = statistics.median(anteriores)
    mad = statistics.median(abs(x - mediana) for x in anteriores) or 1
    z_robusto = 0.6745 * (erros[minuto] - mediana) / mad
    if z_robusto > 3.5:
        print(f"  minuto {minuto:2}: {erros[minuto]:2} erros, mediana {mediana}, z robusto =
{z_robusto:5.1f}")

media_hora = statistics.mean(erros)
desvio_hora = statistics.stdev(erros)
print(f"com a hora inteira: média {media_hora:.1f}, desvio {desvio_hora:.1f}, "
      f"z do minuto 19 = {(30 - media_hora) / desvio_hora:.1f}")
```

Saída

```
minuto 19: 30 erros, média  4.5, z =  27.9  <- anomalia
versão robusta (mediana e MAD):
  minuto 19: 30 erros, mediana 4, z robusto =  17.5
  minuto 30: 12 erros, mediana 5, z robusto =   4.7
  minuto 31: 14 erros, mediana 5, z robusto =   6.1
  minuto 32: 13 erros, mediana 5, z robusto =   5.4
  minuto 33: 15 erros, mediana 5, z robusto =   6.7
  minuto 34: 14 erros, mediana 5, z robusto =   6.1
  minuto 35: 13 erros, mediana 5, z robusto =   5.4
  minuto 36: 15 erros, mediana 5, z robusto =   6.7
com a hora inteira: média 7.8, desvio 5.7, z do minuto 19 = 3.9
```

A versão com média e desvio padrão detectou o pico do minuto 19, mas **não** a mudança de patamar do minuto 30, que é o problema real: o próprio pico, ainda dentro da janela, inflou o desvio padrão, e um salto de 4 para 12 erros por minuto ficou com z abaixo de 3. Com a hora inteira como referência é pior ainda: até o pico fica com z de só 3,9. A versão robusta usa a mediana e o MAD, que um valor extremo quase não altera, e acusa o novo patamar do minuto 30 ao 36; a partir do 37, quando a maior parte da janela já está no patamar novo, ele passa a ser o "normal" e deixa de ser sinalizado, o que é bom (um alerta tocando por horas pela mesma causa não ajuda) e perigoso (se ninguém olhou, a degradação foi aceita em silêncio). O fator 0,6745 torna o MAD comparável ao desvio padrão, e o limite 3,5 é o usual para esse z robusto.

Contagens de erros são pequenas e assimétricas: em produção, os modelos também consideram a hora do dia e o dia da semana.

Q73 Um gerador de senhas no terminal

> *O artigo #65 gera senhas seguras em cinco linguagens. Escreva em Bash `gerar_senha TAMANHO`, que use `/dev/urandom` e garanta pelo menos uma minúscula, uma maiúscula, um dígito e um símbolo, recusando tamanhos menores que 12. Gere 1000 senhas e confira (sem imprimi-las) que todas cumprem as regras e que não há repetidas.*

```
gerar_senha() {
  local tamanho=${1:-16}
  if (( tamanho < 12 )); then
    echo "tamanho mínimo é 12" >&2
    return 1
  fi
  local senha
  while true; do
    senha=$(LC_ALL=C tr -dc 'A-Za-z0-9!@#%^&*_+=-' < /dev/urandom | head -c "$tamanho")
    # sorteia de novo até ter as quatro classes: não enfraquece a senha, só descarta as que
faltam
    [[ $senha =~ [a-z] && $senha =~ [A-Z] && $senha =~ [0-9] && $senha =~ [^A-Za-z0-9] ]] &&
break
  done
  printf '%s\n' "$senha"
}

gerar_senha 8 2>&1 || echo "(recusado, código $?)"

for i in $(seq 1000); do gerar_senha 16; done > senhas.txt
total=$(wc -l < senhas.txt)
unicas=$(sort -u senhas.txt | wc -l)
validas=$(grep -c -E '^.{16}$' senhas.txt)
com_classes=$(grep '[a-z]' senhas.txt | grep '[A-Z]' | grep '[0-9]' | grep -c '[^A-Za-z0-9]')
echo "geradas: $total | únicas: $unicas | com 16 caracteres: $validas | com as 4 classes:
$com_classes"
```

Saída

```
tamanho mínimo é 12
(recusado, código 1)
geradas: 1000 | únicas: 1000 | com 16 caracteres: 1000 | com as 4 classes: 1000
```

O `/dev/urandom` é a fonte de aleatoriedade criptográfica do kernel; o `$RANDOM` do Bash não serve para senhas, porque é previsível e só tem 15 bits. O `LC_ALL=C` faz o `tr` tratar a entrada como bytes, e não como caracteres de um *locale* UTF-8, onde bytes inválidos poderiam causar erro. Com 75 símbolos possíveis e 16 posições, cada senha tem cerca de 99 bits de entropia: adivinhá-la por força bruta está fora de alcance. O `printf '%s\n'` em vez de `echo` evita que uma senha começando com `-n` seja interpretada como opção.

PARA IR ALÉM

Desafios integradores

Projetos maiores, que juntam vários módulos. Não há uma única resposta certa: o gabarito mostra um caminho.

D1

Uma calculadora de terminal em Bash

*Escreva uma calculadora de terminal em Bash com sete operações: soma, subtração, multiplicação, divisão, potência (expoente inteiro), porcentagem (`15 % 200` é 15% de 200) e raiz quadrada (`raiz 2`). Ela deve: (a) aceitar vírgula ou ponto como separador decimal; (b) guardar o último resultado em `ans`, utilizável na conta seguinte; (c) recusar divisão por zero, raiz de negativo, expoente fracionário, resultado infinito e texto que não é número com uma mensagem clara, **sem encerrar o programa**; (d) ter os comandos `hist` (histórico) e `sair`. Separe o núcleo (`calc_nucleo.sh`, só funções, sem leitura de teclado) da interface (`calc.sh`). Como o Bash só faz contas com inteiros, use o `awk` para as contas decimais.*

Caminho da solução

- **Núcleo**: a função `calc_avaliar` recebe a linha e preenche as variáveis `RESULTADO` ou `ERRO`, devolvendo 0 ou 1, como um comando comum. Ela não lê nada nem imprime nada, por isso pode ser testada com um *script* de testes ou usada por outra interface.
- **Números**: a validação é uma expressão regular do próprio Bash (`=~`); a vírgula vira ponto antes de ir para o `awk`, que faz a conta com `double` e devolve o resultado com 15 algarismos significativos, ou `inf` e `nan`, que o núcleo recusa.
- **Interface**: lê com `read -r` (o `-r` preserva barras invertidas), guarda o histórico num array e troca o ponto pela vírgula na exibição.

`calc_nucleo.sh`:

```
# Núcleo da calculadora: só funções, sem entrada nem saída.
ANS=0

_numero() {  # converte "12,5" ou "ans" para "12.5"; falha se não for número
  local t=$1
  [[ $t == ans ]] && { REPLY=$ANS; return 0; }
  [[ $t =~ ^-?[0-9]+([.,][0-9]+)?$ ]] || return 1
  REPLY=${t/,/.}
}

_conta() {  # executa a expressão awk com a e b e confere o resultado
  local r
  r=$(awk -v a="$1" -v b="$2" "BEGIN { printf \"%.15g\", $3 }")
  [[ $r == *inf* || $r == *nan* ]] && { ERRO="resultado fora do alcance"; return 1; }
  RESULTADO=$r
}

calc_avaliar() {
  RESULTADO="" ERRO=""
  local -a p
  read -r -a p <<< "$1"
  local a b
  if [[ ${#p[@]} -eq 2 && ${p[0]} == raiz ]]; then
    _numero "${p[1]}" || { ERRO="\"${p[1]}\" não é um número"; return 1; }
    a=$REPLY
    awk -v a="$a" 'BEGIN { exit !(a < 0) }' && { ERRO="raiz de número negativo"; return 1; }
    _conta "$a" 0 'sqrt(a)' || return 1
  elif [[ ${#p[@]} -eq 3 ]]; then
    _numero "${p[0]}" || { ERRO="\"${p[0]}\" não é um número"; return 1; }
    a=$REPLY
    _numero "${p[2]}" || { ERRO="\"${p[2]}\" não é um número"; return 1; }
    b=$REPLY
    case ${p[1]} in
      +) _conta "$a" "$b" 'a + b' ;;
      -) _conta "$a" "$b" 'a - b' ;;
      '*') _conta "$a" "$b" 'a * b' ;;
      /) awk -v b="$b" 'BEGIN { exit !(b == 0) }' && { ERRO="divisão por zero"; return 1; }
         _conta "$a" "$b" 'a / b' ;;
      %) _conta "$a" "$b" 'a * b / 100' ;;
      ^) [[ $b =~ ^-?[0-9]+$ ]] || { ERRO="o expoente precisa ser inteiro"; return 1; }
         _conta "$a" "$b" 'a ^ b' ;;
      *) ERRO="operação desconhecida \"${p[1]}\" (use + - * / ^ % ou raiz)"; return 1 ;;
    esac || return 1
  else
    ERRO='use "a op b" ou "raiz a"'
    return 1
  fi
  ANS=$RESULTADO  # só um resultado válido substitui o ans
}
```

`calc.sh`:

```
#!/usr/bin/env bash
source "$(dirname "$0")/calc_nucleo.sh"

historico=()
echo "Calculadora: + - * / ^ % e raiz. Comandos: hist, sair."
while printf '> ' && read -r linha; do
  case $linha in
    sair) break ;;
    hist) for i in "${!historico[@]}"; do echo "  $((i + 1)). ${historico[i]}"; done; continue ;;
    '') continue ;;
  esac
  if calc_avaliar "$linha"; then
    historico+=("$linha = ${RESULTADO//./,}")
    echo "${historico[-1]}"
  else
    echo "erro: $ERRO"
  fi
done
echo
```

Entrada digitada:

```
2 + 3
0,1 + 0,2
ans * 10
10 / 0
2 ^ 0,5
raiz -4
raiz 2
ans * ans
10 / 3
15 % 200
abc + 1
2 ^ 10000
2 $ 3
hist
sair
```

Saída

```
Calculadora: + - * / ^ % e raiz. Comandos: hist, sair.
> 2 + 3 = 5
> 0,1 + 0,2 = 0,3
> ans * 10 = 3
> erro: divisão por zero
> erro: o expoente precisa ser inteiro
> erro: raiz de número negativo
> raiz 2 = 1,4142135623731
> ans * ans = 2,00000000000001
> 10 / 3 = 3,33333333333333
> 15 % 200 = 30
> erro: "abc" não é um número
> erro: resultado fora do alcance
> erro: operação desconhecida "$" (use + - * / ^ % ou raiz)
>   1. 2 + 3 = 5
  2. 0,1 + 0,2 = 0,3
  3. ans * 10 = 3
  4. raiz 2 = 1,4142135623731
  5. ans * ans = 2,00000000000001
  6. 10 / 3 = 3,33333333333333
  7. 15 % 200 = 30
>
```

Com a entrada redirecionada, o terminal não ecoa o que foi digitado, e cada resposta aparece logo depois do `>`.

O `%.15g` do `awk` esconde o mesmo ruído do `double` visto nos cadernos de C e de C++: `0,1 + 0,2` mostra 0,3, mas guarda 0,30000000000000004. A operação `*` precisa de aspas no `case`, senão seria o padrão "qualquer coisa", e todas as operações desconhecidas virariam multiplicação. O `2 ^ 10000` estoura o `double` e vira `inf`, que o núcleo recusa. O `bc -l`, quando instalado, faz contas com precisão arbitrária e seria a alternativa para resultados exatos.

Uma versão "gráfica" no próprio terminal fica como extensão, com o `whiptail` (presente na maioria das distribuições): um laço com `whiptail --inputbox "Conta:" 8 50 3>&1 1>&2 2>&3` para ler a conta e `whiptail --msgbox` para mostrar o resultado ou o erro, chamando o mesmo `calc_avaliar`. O núcleo não muda uma linha.

D2

Da máquina vazia ao deploy automatizado

Monte, de ponta a ponta, a entrega de uma aplicação web simples (a API de livros do caderno de PHP serve): o código no Git com branches protegidas; um pipeline que roda lint, testes e varredura de segredos, constrói a imagem e a publica num registry com a tag do commit; a infraestrutura de uma VM criada com Terraform e configurada com Ansible (Docker, usuário de deploy, firewall); o deploy automático com verificação de saúde e rollback; e um painel com latência, taxa de erros e um alerta baseado em SLO.

Caminho da solução

- **Repositório**: `main` protegida (PR, revisão, CI obrigatório), `dev` para integração, Conventional Commits e *tags* SemVer gerando *releases* (módulo 2).
- **CI**: *jobs* de *lint* e testes em paralelo; varredura de segredos e de dependências; *build* da imagem em múltiplos estágios com *cache*; publicação no GHCR com duas *tags*, o *hash* curto do *commit* (imutável, usado no deploy) e a versão da *release* (módulos 2 a 4 e 9).
- **Infraestrutura**: o Terraform cria a VPC, a sub-rede, o *security group* (443 aberto, 22 só do IP do *runner* ou de uma VPN) e a VM, com o *state* num *backend* remoto com *lock*; a saída com o IP alimenta o inventário do Ansible, que instala o Docker, cria o usuário `deploy` sem senha e só com chave, configura o *firewall* e o `sshd` endurecido (módulos 5 e 1).
- **Deploy**: o *job* de deploy, só na `main` e com aprovação do ambiente `producao`, entra por SSH com uma chave guardada como *secret*, faz o `docker pull` da imagem pela *tag* do *commit*, troca o container com o `compose`, roda o `curl -fsS /health` e, se falhar, volta para a *tag* anterior, guardada num arquivo no servidor (módulo 4).
- **Sem Docker no servidor**: a mesma estrutura funciona com o PHP e o nginx instalados pelo Ansible, e o deploy por *releases* com *symlink* do módulo 4.
- **Observabilidade**: o nginx com *log* em JSON; o Prometheus (com o *node exporter* e as métricas da aplicação) e o Grafana no mesmo servidor ou num serviço gerenciado; um painel com taxa de requisições, erros e p95; um SLO de 99,5% de disponibilidade e alertas por *burn rate* enviados ao canal da equipe (módulo 6).
- **Documentação**: um `README` com a arquitetura e os comandos, um *runbook* para "o deploy falhou" e "o site caiu", e um *postmortem* de um incidente simulado, por exemplo parando o container de propósito (módulo 9).

O critério de pronto é **reprodutibilidade**: apagar tudo com `terraform destroy` e recriar o ambiente inteiro com um `apply`, um *playbook* e um *push*, sem nenhum passo manual que só existe na memória de alguém.

Fim do gabarito.

Estudar e realmente Aprender DevOps & Cloud · theknowledgeisle.com