

Guia de preparação GH-200

Tudo o que a prova cobra, organizado por domínio: conceitos-chave, comparações que confundem, pegadinhas reais, gatilhos do enunciado, questões comentadas e um plano para chegar pronto ao exame.

5	32	70%	6
capítulos por domínio	questões comentadas	nota de corte	provas completas no curso

ANTES DE COMEÇAR

Como usar este guia

Este guia foi feito para ser estudado junto com as 6 provas completas do curso GH-200: o guia explica o que cai e onde estão as armadilhas; as provas treinam o ritmo e mostram o que você ainda não domina.

Quando	O que fazer
Semana 1	Faça o teste diagnóstico (página seguinte) e leia os capítulos 1 e 2. No fim da semana, faça a Prova 1 do curso no tempo do exame.
Semana 2	Leia os demais capítulos, começando pelos domínios em que você errou mais no diagnóstico. Faça as provas 2 e 3.
Semana 3	Releia os alertas de pegadinha e as tabelas de comparação. Faça as provas 4 e 5 e revise cada erro pela explicação.
Semana 4	Estude a folha de cola, faça a Prova 6 e refaça as provas abaixo de 80%. Agende o exame quando estiver acima de 80% em todas (a nota de corte é 70%).

Mapa do exame

Cada prova do curso tem 50 questões e 100 minutos, no formato do exame. Referência de aprovação: 70%.

Domínio (capítulo)	Peso	Neste guia
1. Automação segura e otimizada	10-15%	18 conceitos · 14 pegadinhas
2. Consumir e solucionar problemas de fluxos de trabalho	15-20%	18 conceitos · 14 pegadinhas
3. Criar e gerenciar fluxos de trabalho	20-25%	19 conceitos · 14 pegadinhas
4. Criar e manter ações	15-20%	18 conceitos · 14 pegadinhas
5. Gerenciar o GitHub Actions na empresa (enterprise)	20-25%	18 conceitos · 14 pegadinhas

> **Como ler os capítulos:** os **tópicos-chave** são o que mais cai; a **tabela de memorização** separa o que a prova costuma confundir; os **alertas de prova** mostram as armadilhas das alternativas erradas; os **gatilhos** ligam palavras do enunciado à resposta; e o **checklist** diz se você está pronto para o domínio.

Estude um pouco todo dia, de graça

Instagram @strecht.simulados
Um post de prova por dia: questão comentada, pegadinhas e o "não confunda" de cada certificação.
Seguir no Instagram →

YouTube Strecht Simulados
Shorts diários com pegadinhas de prova e, aos sábados, um simulado comentado.
Inscrever-se no canal →

DIAGNÓSTICO

Você já sabe isso?

Responda às 12 questões sem consultar nada. As respostas, com a explicação e o capítulo para revisar, estão no fim do guia. Comece o estudo pelos capítulos em que errou.

D1 · Automação segura e otimizada

A Escola Arte e Ciência usa o plano GitHub Team e tentou configurar revisores obrigatórios no ambiente de um repositório privado, sem sucesso. Por quê?

A. O ambiente precisa ter um wait timer antes dos revisores
B. É preciso cadastrar ao menos seis revisores para o ambiente funcionar
C. No Free, Pro e Team, isso só vale em repositórios públicos
D. Em repositórios privados, só owners podem ser revisores

D2 · Automação segura e otimizada

Antes de adotar uma ação do Marketplace com o selo de verified creator, o líder de segurança do Banco Aurora pergunta se ainda precisa revisar algo. Qual orientação segue a documentação?

A. O selo garante que as tags publicadas da ação não serão movidas
B. O selo dispensa revisão porque o GitHub audita cada nova versão
C. O selo comprova que a ação passou em teste de intrusão do GitHub
D. O selo é um bom sinal, mas o código da ação ainda deve ser auditado

D3 · Automação segura e otimizada

Na Escola Técnica Progresso, o job deploy referencia o ambiente production, que tem revisores obrigatórios e o secret DB_SENHA. Quando o job consegue ler DB_SENHA?

A. Assim que o workflow começa, ainda antes da aprovação
B. Apenas em execuções disparadas à mão por administradores
C. Depois que um revisor aprova e o job é enviado ao runner
D. Desde o enfileiramento, junto com os secrets do repositório

D4 · Consumir e solucionar problemas de fluxos de trabalho

A Seguradora Proteja percebeu que cada novo commit gera uma execução com falha imediata de um workflow que foi editado recentemente. Qual é a causa provável?

A. Um arquivo de workflow inválido em .github/workflows
B. O repositório atingiu o limite de 256 jobs por execução
C. Falta o gatilho on: push explícito dentro do arquivo
D. O workflow está desabilitado e registra falha a cada commit

D5 · Consumir e solucionar problemas de fluxos de trabalho

No YAML da Destilaria Alambique, o job lint define env: &comum { NODE_ENV: test, CI_FLAG: '1' } e o job test usa env: *comum. Quais variáveis o job test recebe?

A. As do lint mais variáveis GITHUB_ com o prefixo comum
B. NODE_ENV=test e CI_FLAG=1, as mesmas definidas no lint
C. Nenhuma, já que o alias só funciona dentro do próprio job
D. Só NODE_ENV, porque o alias copia só a primeira chave

D6 · Consumir e solucionar problemas de fluxos de trabalho

Na Bicicletas Pedal Livre, um workflow agendado falha toda noite, mas só o Marcos recebe as notificações de falha, embora não seja o autor original do workflow. Qual é a explicação?

A. Marcos fez o último push no branch padrão do repositório
B. Marcos aprovou o último pull request que mexeu no workflow
C. Marcos foi quem alterou por último a sintaxe cron do arquivo
D. Marcos é o dono do repositório e recebe todos os avisos de falha dele

D7 · Criar e gerenciar fluxos de trabalho

No workflow da Rodoviária Expresso Sol, um step executa echo "RELEASE=1.2" >> "$GITHUB_ENV" e, na linha seguinte do mesmo run, imprime $RELEASE, que sai vazia. O que explica?

A. Quem grava em GITHUB_ENV não vê o valor; só os próximos steps
B. GITHUB_ENV precisa do comando ::set-env:: para ter efeito
C. Valores gravados em GITHUB_ENV só aparecem no próximo job
D. Faltou declarar RELEASE no env do topo do workflow antes do step

D8 · Criar e gerenciar fluxos de trabalho

O Banco Ipiranga Digital quer disparar o workflow Publicar sempre que o workflow Build terminar, com sucesso ou falha, e checar o resultado dentro do job. Como configurar o gatilho?

A. on: check_suite com types: [completed] filtrado por Build
B. on: workflow_call com o nome Build informado em inputs
C. on: workflow_run com workflows: [Build] e types: [completed]
D. needs: Build no primeiro job do workflow Publicar, sem gatilho

D9 · Criar e gerenciar fluxos de trabalho

Um engenheiro da Laticínios Serra Azul escreveu if: !startsWith(github.ref, 'refs/tags/') em um step, e o arquivo de workflow passou a ser rejeitado. O que corrigir?

A. Trocar startsWith por contains, que aceita o operador de negação
B. Remover o prefixo refs/tags/, pois o if só compara nomes curtos
C. Mover a condição para o nível do job, onde o ! é aceito sem aspas
D. Envolver a expressão em ${{ }}, já que ! é reservado no YAML

D10 · Criar e manter ações

Na Refrescos Gelado Bom, a opção Publish this Action to the GitHub Marketplace aparece desabilitada para o repositório da organização. O que explica e o que mais é exigido para publicar?

A. O repositório precisa de ao menos dez estrelas para publicar
B. Falta aceitar o Developer Agreement; publicar ainda exige 2FA
C. A ação precisa de revisão do GitHub antes de liberar a caixa
D. O repositório precisa ser interno para aparecer no Marketplace

D11 · Criar e manter ações

Uma ação JavaScript da Fundição Metal Vivo obtém um token temporário de uma API e vai usá-lo em comandos que podem aparecer no log. O que fazer logo depois de obter o token?

A. Chamar core.setSecret, equivalente a add-mask, com o token
B. Chamar core.exportVariable, que mascara variáveis exportadas
C. Chamar core.debug, que só exibe o token em modo de depuração
D. Chamar core.setOutput com o token para o GitHub ocultá-lo

D12 · Criar e manter ações

O Banco Vale Dourado ativou immutable releases no repositório da sua ação. Como manter a tag major v2 sempre apontando para a versão mais recente?

A. Criar a release v2 e mover a tag dela a cada nova versão lançada
B. Manter v2 como tag Git sem release e movê-la a cada versão nova
C. Excluir a release v2 e recriá-la com o mesmo nome a cada versão
D. Proteger um branch v2 e bloquear force push nele a cada versão

01

CAPÍTULO 1 · 10-15% DA PROVA

Automação segura e otimizada

Domínio de menor peso, mas cheio de pegadinhas: preciso saber blindar workflows contra script injection e pwn request, fixar ações com segurança, reduzir permissões do GITHUB_TOKEN, usar OIDC no lugar de chaves guardadas, proteger implantações com ambientes, gerar e verificar artifact attestations e otimizar custo com cache, artefatos e timeouts. As questões costumam descrever um risco e pedir a recomendação da documentação, ou listar permissões e pedir as necessárias. Eu erro quando a alternativa errada traz uma proteção real, mas de outro problema (wait timer no lugar de reviewers, secret scanning no lugar de CodeQL, cache no lugar de artefato). Também preciso dos números de ambientes, do GITHUB_TOKEN e do cache.

Neste capítulo

18 tópicos-chave · 14 pegadinhas · 7 macetes · 14 gatilhos · 4 questões comentadas

Tópicos-chave

Script injection

Expressões ${{ }} em run são avaliadas e inseridas no script antes de ele rodar. Contextos que terminam em body, title, head_ref, label, message, name e email são entrada não confiável; IDs gerados pelo GitHub (run_id, repository_id) não são. A defesa principal é uma ação que recebe o valor como input; a alternativa para script inline é env intermediário usado como variável entre aspas no shell.

pull_request_target e workflow_run

Rodam em contexto privilegiado: pull_request_target executa o workflow do branch padrão do repositório base com token de leitura e escrita e acesso a secrets. Fazer checkout de github.event.pull_request.head.sha e executar o código é o ataque pwn request. Artefatos de PRs de forks baixados em workflow_run são dados não confiáveis e nunca devem ser executados.

Fixação de ações

Tag pode ser movida ou excluída se o repositório da ação for invadido; SHA completo é a referência mais segura sem lockfile, ao custo de não receber atualizações automáticas. Confira que o SHA é do repositório original, e não de um fork. Use tag só se confiar no criador.

Verified creator e auditoria

O selo indica que o GitHub verificou a identidade da equipe criadora; é um bom sinal, mas não garante que tags não serão movidas nem que o código foi auditado. Ainda é preciso auditar como a ação trata secrets e conteúdo do repositório.

Dependabot e ações

Version updates abrem PRs quando sai nova versão de uma ação referenciada no formato de repositório (tag ou SHA). Alertas do Dependabot dependem de versionamento semântico e não cobrem ações fixadas por SHA. Ações e reusable workflows locais e referências docker:// não são atualizados.

GITHUB_TOKEN e permissions

Criado no início de cada job e expira no fim do job ou na vida máxima: 6 horas em runners hospedados e 24 horas em auto-hospedados (onde o job pode durar até 5 dias). Ao declarar permissions, todo escopo não citado vira none, inclusive contents; aceita read, write, none, read-all, write-all e {}. As permissões se limitam ao repositório do workflow.

Token de GitHub App em vez de PAT

Para acessar outros repositórios sem depender de uma pessoa, gere no workflow um token de instalação de GitHub App (por exemplo, com create-github-app-token). PAT de outra pessoa troca uma dependência por outra; deploy key dá só acesso Git, não autentica a REST API.

OIDC

Elimina chaves de nuvem de longa duração: o job pede um JWT e troca por credencial temporária que expira sozinha, com controle granular no provedor. O job precisa de id-token: write (e contents: read se fizer checkout); id-token: write não dá escrita em outros recursos.

Claims do token OIDC

iss é token.actions.githubusercontent.com; aud padrão é a URL do dono do repositório e pode ser personalizada. Para exigir um reusable workflow específico, inclua job_workflow_ref no sub e exija esse valor na nuvem.

Regras de proteção de ambiente

Required reviewers: até seis usuários ou equipes, uma aprovação basta, com opção de impedir autoaprovação (prevent self-review). Wait timer atrasa o job de 1 a 43.200 minutos sem consumir minutos. Deployment branches and tags restringe quem implanta, comparando com o GITHUB_REF; o curinga não casa com /. O bypass de administradores vem ligado por padrão.

Secrets de ambiente e planos

Os secrets do ambiente só ficam acessíveis depois que todas as regras de proteção passam e o job vai ao runner. Em Free, Pro e Team, required reviewers e wait timer em repositórios privados não estão disponíveis; só em públicos ou com GitHub Enterprise.

Redação de segredos nos logs

A redação procura o valor exato, no runner. Valor derivado de secret (Base64, codificação de URL) deve ser registrado com ::add-mask:: antes do uso. Guarde um secret por valor sensível, nunca um JSON inteiro em um secret.

Forks e Dependabot

Workflows de PRs do Dependabot rodam como se fossem de fork: GITHUB_TOKEN só leitura e nenhum secret, mesmo declarando write-all.

Runners auto-hospedados em repositório público

Devem ser evitados: PRs de forks podem executar código na sua infraestrutura e comprometer a máquina de forma persistente. Runners hospedados usam VMs efêmeras e limpas; tornar o runner efêmero ou exigir aprovação não elimina o risco.

Artifact attestations

Ligam o artefato às instruções de build e dão SLSA Build Level 2; exigir builds em reusable workflows conhecidos e isolados leva ao Level 3. Para binário, o job precisa de id-token: write, contents: read e attestations: write; para imagem, também packages: write. Em repositório privado exigem Enterprise Cloud; atestar o que será lançado, não cada build de teste.

Verificar attestation

Binário: gh attestation verify CAMINHO -R ORG/REPO. Imagem: gh attestation verify oci://ghcr.io/ORG/IMAGEM:tag -R ORG/REPO, depois de autenticar no registro. Sem verificação, a attestation não traz benefício.

Cache: segurança e remoção

Não guarde credenciais no cache: quem consegue abrir PR pode ler os caches do branch base, e o cache não é assinado. Um workflow restaura caches do branch atual, do base do PR e do branch padrão, nunca de irmãos ou filhos. Caches sem acesso há mais de 7 dias saem, e no limite de 10 GB sai primeiro o de acesso mais antigo.

Custo e otimização

Repositório público com runner padrão e qualquer runner auto-hospedado não consomem minutos pagos. timeout-minutes realista evita job travado queimando até 360 minutos. Armazenamento é cobrado por hora acumulada: excluir artefatos só evita cobrança futura; artefatos dividem cota com o GitHub Packages e o cache tem cota própria.

Macetes para memorizar

Entrada não confiável

Se o usuário digita (título, corpo, branch, label, mensagem, nome, email), é suspeito; se o GitHub gera (IDs), não.

Defesa contra injection

Melhor: ação com input. Alternativa: env intermediário. Nunca: ${{ }} direto no run.

Vida do GITHUB_TOKEN

Token: 6 horas no hospedado e 24 horas no self-hosted; o job self-hosted aguenta 5 dias, o token não.

Permissões de attestation

Binário: ID + Contents + Attestations; imagem: soma Packages.

Revisores de ambiente

Até 6 na lista, 1 aprova, admin fura se o bypass estiver ligado.

Cache

7 dias sem uso, 10 GB, sai o mais antigo; restaura do atual, do base e do padrão, nunca do irmão.

Grátis no Actions

Público com runner padrão e qualquer auto-hospedado não pagam minutos.

Tabela de memorização: não confunda

Quando é...	...e quando é
pull_request CI de código de fork sem privilégios: roda no contexto do PR.	**pull_request_target** Workflow do branch base com token de escrita e secrets; nunca fazer checkout do head e executar.
Cache Reaproveitar dependências entre execuções, restauradas por chave.	**Artefato** Guardar resultado do job para download ou para outros jobs.
Tag (@v3) Quando confia no criador e quer receber correções; a tag pode ser movida.	**SHA completo** Máxima segurança sem lockfile, conferindo que o commit é do repositório original.
GITHUB_TOKEN Acesso ao próprio repositório do workflow durante o job.	**Token de instalação de GitHub App** Acessar outros repositórios ou disparar workflows sem depender de usuário.
Wait timer Atrasar o início do deploy por um tempo, sem consumir minutos.	**timeout-minutes** Encerrar job que passa de uma duração; não atrasa o início.
Code scanning com CodeQL e OpenSSF Scorecards Encontrar padrões vulneráveis em workflows, como script injection e permissões amplas.	**Secret scanning** Encontrar segredos expostos no código.
Required reviewers Exigir aprovação humana antes do job ir ao runner.	**Deployment branches and tags** Restringir de quais branches ou tags é possível implantar.

Números que caem

Item	Valor
Vida máxima do GITHUB_TOKEN	**6 horas em runners hospedados; 24 horas em auto-hospedados**
Limite de execução de job em runner auto-hospedado	**5 dias**
Required reviewers	**Até 6 usuários ou equipes; 1 aprovação basta**
Wait timer	**1 a 43.200 minutos, sem tempo cobrado**
Cache: remoção padrão	**Sem acesso há mais de 7 dias; limite de 10 GB por repositório**
Artifact attestations	**SLSA Build Level 2; com reusable workflow isolado, Level 3**
Timeout padrão de job	**360 minutos**

Alertas de prova (pegadinhas)

⚠ **Alerta de prova:** Trocar aspas duplas por simples ou usar toJSON para mitigar injection no run.
Como não cair: A expressão continua inserida no script; use ação com input ou env intermediário.

⚠ **Alerta de prova:** Validar o título com regex ou limitar caracteres no próprio run.
Como não cair: A injeção acontece antes da validação; texto curto também pode conter comandos.

⚠ **Alerta de prova:** Achar que pull_request_target com checkout do head é seguro porque o token seria só leitura.
Como não cair: Em pull_request_target o token tem leitura e escrita no base e há secrets: é pwn request.

⚠ **Alerta de prova:** Trocar workflow_run por pull_request_target para ganhar segurança.
Como não cair: Os dois são privilegiados; o cuidado é tratar artefatos e código de fork como não confiáveis.

⚠ **Alerta de prova:** Supor que o selo verified creator garante tags fixas ou código revisado.
Como não cair: O selo trata da identidade do criador; tags podem ser movidas e o código deve ser auditado.

⚠ **Alerta de prova:** Esperar alerta do Dependabot para ação fixada por SHA.
Como não cair: Alertas não cobrem SHA; version updates sim abrem PRs.

⚠ **Alerta de prova:** Declarar só permissions: { issues: write } e esperar que o checkout continue.
Como não cair: Escopos não citados viram none; inclua contents: read.

⚠ **Alerta de prova:** Atribuir a falha de um job de 30 horas ao limite de execução do self-hosted.
Como não cair: O job pode durar até 5 dias; o que acaba é o GITHUB_TOKEN, com vida máxima de 24 horas em auto-hospedado (6 horas em hospedado).

⚠ **Alerta de prova:** Achar que derivados de secret já são mascarados.
Como não cair: A redação procura o valor exato; registre o derivado com ::add-mask::.

⚠ **Alerta de prova:** Usar wait timer ou remover admins da lista de revisores para impedir que admins furem a aprovação.
Como não cair: Desligue Allow administrators to bypass configured protection rules no ambiente.

⚠ **Alerta de prova:** Achar que são precisas todas as aprovações, ou a maioria, dos revisores do ambiente.
Como não cair: Uma aprovação de qualquer revisor listado libera o job; seis é o máximo de revisores, não o mínimo.

⚠ **Alerta de prova:** Atestar todos os builds de teste porque attestation garante artefato seguro.
Como não cair: Attestation liga artefato à origem, não garante segurança, e só vale se verificada; ateste o que será lançado.

⚠ **Alerta de prova:** Exigir que o aud seja o repositório caller para amarrar o OIDC a um reusable workflow.
Como não cair: aud padrão é a URL do dono; o reusable é identificado por job_workflow_ref no sub.

⚠ **Alerta de prova:** Achar que cache e artefatos dividem a mesma cota de 10 GB, ou que excluir artefato estorna o mês.
Como não cair: Cache tem 10 GB próprios; artefatos dividem cota com Packages; armazenamento acumulado não é estornado.

Como cai na prova

📝 **Como cai:** Trecho de workflow com expressão em run e pergunta sobre mitigação.
Como resolver: Identifique se o contexto é texto controlado pelo usuário; descarte trocas de aspas, regex, shell e toJSON; escolha ação com input ou env intermediário.

📝 **Como cai:** Escolha DUAS ou TRÊS permissões necessárias (OIDC ou attestation).
Como resolver: OIDC: id-token: write (+ contents: read se houver checkout). Attestation de binário: id-token: write, contents: read, attestations: write; packages: write só para imagem.

📝 **Como cai:** Cenário de ambiente de produção com uma exigência de controle.
Como resolver: Mapeie: aprovação humana = required reviewers; atraso = wait timer; origem do deploy = deployment branches and tags; admin furando = desligar bypass; privado em Team = precisa Enterprise.

📝 **Como cai:** Pergunta sobre custo ou desempenho.
Como resolver: Dependências: cache com hashFiles; resultado: artefato; job travando: timeout-minutes; armazenamento: reduzir retention-days e excluir o desnecessário; lembrar que público com runner padrão e self-hosted não pagam minutos.

📝 **Como cai:** Risco de uma prática de supply chain (tag, selo, Dependabot).
Como resolver: Tag é móvel; SHA completo do repositório original é o mais seguro; selo não substitui auditoria; Dependabot atualiza mas não alerta SHA nem cobre ações locais e docker://.

Gatilhos do enunciado

Se o enunciado trouxer...	Pense em
github.event.issue.title ou pull_request.body direto em run	**Script injection: ação com input ou env intermediário**
pull_request_target + checkout de pull_request.head.sha + make test	**Pwn request: código do fork com token e secrets do base**
Ação de terceiros referenciada por tag	**Tag pode ser movida ou excluída; SHA completo do repositório original**
Chaves AWS ou Azure guardadas como secret e rotação manual	**OIDC com id-token: write e credencial curta por job**
Só o reusable deploy.yml aprovado pode assumir o papel na nuvem	**job_workflow_ref no sub do token OIDC**
PAT de engenheiro que vai sair da empresa	**Token de instalação de GitHub App**
Deploy deve esperar 45 minutos após disparo	**Wait timer do ambiente (1 a 43.200 minutos)**
Só main e release/* podem implantar em production	**Deployment branches and tags com padrões selecionados**

Mudança em arquivos de workflow precisa de revisão do time de plataforma	**.github/workflows no CODEOWNERS**
Detectar padrões vulneráveis nos workflows	**Code scanning com CodeQL e OpenSSF Scorecards**
Workflow falha só nos PRs do Dependabot	**Roda como fork: token só leitura e sem secrets**
Repositório open source público com runner na rede interna	**Evitar runner auto-hospedado: forks executam código**
Apagar caches de uma chave para build limpo	**DELETE /repos/{owner}/{repo}/actions/caches?key=CHAVE**
Nível SLSA de attestations	**Build Level 2; reusable isolado eleva a Level 3**

🎯 Se você lembrar de só uma coisa deste capítulo...

- Expressão com texto do usuário nunca vai direto em run: ação com input ou env intermediário.
- pull_request_target e workflow_run são privilegiados: nada de checkout e execução de código ou artefato de fork.
- Ao declarar permissions, o que não foi citado vira none; OIDC precisa de id-token: write.
- Secrets de ambiente só existem depois que as regras de proteção passam.
- Cache não é cofre: nada de credencial em caminho de cache.

Essenciais para a prova (checklist)

- ☐ Sei reconhecer contextos não confiáveis e as duas mitigações de script injection.
- ☐ Sei explicar o pwn request com pull_request_target e o cuidado com artefatos em workflow_run.
- ☐ Sei por que SHA completo é mais seguro que tag e o que conferir no SHA.
- ☐ Sei o que o Dependabot faz e não faz com ações.
- ☐ Sei as permissões de OIDC e de artifact attestations e as claims aud, iss e sub.
- ☐ Sei configurar required reviewers, wait timer, deployment branches and tags e bypass de admins.
- ☐ Sei a vida máxima do GITHUB_TOKEN em hospedado e auto-hospedado.
- ☐ Sei quando usar token de GitHub App no lugar de PAT ou GITHUB_TOKEN.
- ☐ Sei as regras de acesso, remoção e segurança do cache e a diferença para artefatos.
- ☐ Sei o que é gratuito em minutos e como cortar custo de armazenamento e jobs travados.

Questões de revisão do capítulo

R1.1 · Automação segura e otimizada

A Turismo Trilhas quer que só jobs executados pelo reusable workflow aprovado deploy.yml, do repositório plataforma, consigam assumir o papel de deploy na nuvem via OIDC. O que configurar?

A. Exigir que o aud seja igual ao nome do repositório caller
B. Exigir environment: production em todos os workflows callers
C. Guardar a chave da nuvem só como secret do repositório plataforma
D. Incluir job_workflow_ref no sub e exigir esse valor na nuvem

R1.2 · Automação segura e otimizada

Um workflow da Cooperativa Café Sul converte um secret para Base64 antes de passá-lo a uma ferramenta, que pode imprimir o valor em mensagens de erro. O que a documentação recomenda?

A. Trocar o Base64 por codificação de URL, que fica fora do log
B. Nada, pois derivados de secrets já são mascarados sozinhos
C. Salvar o valor em GITHUB_OUTPUT, que o GitHub esconde dos logs
D. Registrar o valor em Base64 com ::add-mask:: antes de usá-lo

R1.3 · Automação segura e otimizada

A Farma Saúde Já quer que o deploy em production espere 45 minutos depois de disparado, para dar tempo de monitorar o staging. Qual recurso usar?

A. Grupo de concurrency que segura o job por 45 minutos na fila
B. timeout-minutes: 45 no job de deploy, que espera antes de começar
C. Step com sleep 2700 no início do job, até o limite de 360 minutos
D. Wait timer do ambiente, de 1 a 43.200 min, sem tempo cobrado

R1.4 · Automação segura e otimizada

Um job da Tecelagem Fio Fino roda em runner auto-hospedado por cerca de 30 horas e começa a falhar nas chamadas à API perto do fim. Qual é a explicação?

A. Jobs em runners self-hosted são encerrados após 24 horas
B. A API bloqueia tokens que passam de mil requisições por hora
C. O GITHUB_TOKEN vence após 6 horas também em self-hosted
D. O GITHUB_TOKEN só pode ser renovado por até 24 horas

Respostas comentadas

R1.1. D) Incluir job_workflow_ref no sub e exigir esse valor na nuvem

Por quê: A documentação mostra modelos de sub com job_workflow_ref para exigir um reusable workflow específico, padronizando implantações entre repositórios.

✗ A) Incorreta. O aud padrão é a URL do dono do repositório e não identifica o reusable workflow.
✗ B) Incorreta. O ambiente sozinho não garante qual workflow executou o job.
✗ C) Incorreta. Isso volta a usar segredo de longa duração, o que o OIDC evita.

> **Dica de prova:** Exigir reusable específico no OIDC: job_workflow_ref no sub.

R1.2. D) Registrar o valor em Base64 com ::add-mask:: antes de usá-lo

Por quê: Como a redação automática não é garantida para valores transformados, registre cada valor sensível derivado de um secret.

✗ A) Incorreta. Qualquer transformação gera um novo valor que também precisa ser registrado.
✗ B) Incorreta. A redação procura o valor exato; um valor transformado pode escapar.
✗ C) Incorreta. GITHUB_OUTPUT não mascara valores; outputs com segredo podem até ser omitidos.

> **Dica de prova:** Transformou o segredo? Mascare o resultado também.

R1.3. D) Wait timer do ambiente, de 1 a 43.200 min, sem tempo cobrado

Por quê: Wait timer é uma regra de proteção do ambiente; em planos Free, Pro e Team ela só está disponível para repositórios públicos.

✗ A) Incorreta. concurrency controla execuções simultâneas, não um atraso programado.
✗ B) Incorreta. timeout-minutes limita a duração do job; não atrasa o início.
✗ C) Incorreta. Um sleep consome minutos do runner enquanto espera.

> **Dica de prova:** Atraso antes do deploy: wait timer do ambiente.

R1.4. D) O GITHUB_TOKEN só pode ser renovado por até 24 horas

Por quê: O GITHUB_TOKEN é criado no início de cada job e expira no fim do job ou na vida máxima efetiva: 6 horas em runners hospedados e 24 horas em auto-hospedados.

✗ A) Incorreta. Em runners auto-hospedados, o limite de execução de job é de 5 dias.
✗ B) Incorreta. O limite de requisições não explica a falha ligada ao tempo de execução.
✗ C) Incorreta. 6 horas é a vida máxima em runners hospedados, por causa do limite de execução de job.

> **Dica de prova:** Job de mais de 24 h em self-hosted: não confie no GITHUB_TOKEN.

02

CAPÍTULO 2 · 15-20% DA PROVA

Consumir e solucionar problemas de fluxos de trabalho

Neste domínio eu preciso interpretar execuções e resolver falhas: reexecutar jobs, ligar logs de depuração, achar a informação certa no log, baixar e gerenciar artefatos pela interface, CLI e REST API, desabilitar workflows, usar workflow templates e consumir reusable workflows e composite actions. As questões descrevem um sintoma (job skipped, output inconsistente, log faltando, agendamento parado) e pedem a causa ou o próximo passo, muitas vezes com comando gh ou endpoint. Onde mais erro: misturar ACTIONS_STEP_DEBUG com ACTIONS_RUNNER_DEBUG, achar que reexecução usa o commit mais novo ou os privilégios de quem reexecutou, e esquecer que reusable workflow enxerga tudo pelo contexto do caller. No guia oficial os templates aparecem como starter workflows; o nome atual é workflow templates.

Neste capítulo

18 tópicos-chave · 14 pegadinhas · 7 macetes · 14 gatilhos · 4 questões comentadas

Tópicos-chave

Reexecução

Dá para reexecutar todos os jobs, só os que falharam ou um job específico (inclusive uma combinação da matriz), até 30 dias depois e no máximo 50 vezes. Usa o mesmo GITHUB_SHA e GITHUB_REF e os privilégios de quem disparou a execução original; github.triggering_actor mostra quem reexecutou. Só github.run_attempt muda; run_id e run_number ficam iguais.

Reexecução e reusable workflow

Ao reexecutar todos os jobs, o reusable é resolvido de novo pela referência indicada (ex.: @main). Ao reexecutar só os falhados ou um job, mantém-se o SHA da primeira tentativa.

Logs de depuração

ACTIONS_STEP_DEBUG = true (secret ou variável) detalha os steps; ACTIONS_RUNNER_DEBUG = true gera logs de diagnóstico do runner, na pasta runner-diagnostic-logs do arquivo de logs. Se secret e variável existirem, vale o secret. Para depuração pontual, reexecute com Enable debug logging ou gh run rerun --debug.

runner.debug e RUNNER_DEBUG

São preenchidos pelo runner com 1 quando a depuração está ligada; não são o interruptor. Um step só para modo debug usa if: ${{ runner.debug == '1' }}.

Lendo logs na interface

A busca (Search logs) só considera steps expandidos. Set up job > Runner Image > Included Software mostra o software do runner hospedado que rodou. O grafo de visualização mostra jobs, status e dependências em tempo real.

Por que o job foi pulado

A avaliação do if de job aparece no arquivo system.txt do job, dentro do log baixado em zip, inclusive em runners hospedados. Para if de step, ligue os logs de depuração.

Logs pela API e tentativas

GET /repos/{owner}/{repo}/actions/runs/{run_id}/logs baixa os logs da execução; há variantes por tentativa (attempts/{attempt_number}/logs) e por job (jobs/{job_id}/logs); DELETE no mesmo caminho exclui. O arquivo de uma tentativa parcial só traz os jobs reexecutados.

GitHub CLI para execuções

gh run list --workflow ARQUIVO lista execuções; gh run view RUN_ID detalha; gh run download RUN_ID -n NOME baixa um artefato; gh run rerun --debug reexecuta com depuração; gh workflow disable e enable pausam e reativam; gh run delete exclui.

Artefatos: acesso e ciclo de vida

Baixar exige leitura (seção Artifacts, CLI ou API); remover exige escrita e não pode ser desfeito. A data de expiração está em expires_at na REST API. Excluir uma execução apaga também os artefatos dela.

Artefatos: upload e download

O upload gera digest SHA-256 que o download confere, avisando se diferir. Na v4 os artefatos são imutáveis: conteúdo novo pede nome novo. download-artifact sem name baixa todos, um por pasta; de outra execução, informe token e run id.

API de artefatos

GET /repos/{owner}/{repo}/actions/runs/{run_id}/artifacts lista os artefatos de uma execução; archive_download_url leva ao download. A listagem do repositório filtra por name, não por run_id.

Excluir execução

Exige escrita e só é possível com a execução concluída (ou com mais de 2 semanas); uma execução em andamento precisa ser cancelada antes. Para dados expostos, também dá para excluir só os logs.

Desabilitar versus excluir workflow

Desabilitar (interface, PUT .../workflows/{workflow_id}/disable ou gh workflow disable) evita novos disparos, preserva arquivo e histórico e não cancela execuções em andamento, mas impede reexecuções até reativar. Excluir é remover o arquivo de .github/workflows; não há endpoint para excluir workflow.

Agendamentos que não rodam

Em forks de repositórios públicos os agendados vêm desabilitados (podem ser habilitados); em repositório público, param após 60 dias sem atividade. As notificações de falha de agendamento vão para quem alterou a sintaxe cron por último.

Workflow templates

Ficam no repositório .github da organização, pasta workflow-templates, com YAML e .properties.json (name e description obrigatórios; iconName, categories e filePatterns opcionais). $default-branch é trocado pelo branch padrão na criação. São usados em Actions > New workflow > Configure e geram cópia independente; a visibilidade do .github define quem os vê. Categorias de alto nível: CI, Deployment, Security e Automation.

Consumindo reusable workflows

No chamado, o contexto github é o do caller, e actions/checkout sem parâmetros baixa o repositório do caller. env do caller não chega ao chamado; use inputs ou vars. O job que chama aceita name, uses, with, secrets, strategy, needs, if, concurrency e permissions, mas não runs-on nem steps. Reusable em repositório privado exige liberação em Access.

Cancelamento e propagação

No cancelamento, os if são reavaliados: jobs e steps com always() continuam; os demais recebem sinal e, após 5 minutos, são encerrados à força. Falha em um job de needs propaga skipped pela cadeia.

Pular execuções e checks

[skip ci], [ci skip], [no ci], [skip actions], [actions skip] ou o trailer skip-checks: true evitam só workflows de push e pull_request. Workflow pulado por filtro de paths não gera resultado, e um check obrigatório fica Pending, bloqueando o merge.

Macetes para memorizar

Reexecução

30 dias, 50 vezes, mesmo commit, privilégios do primeiro; só run_attempt sobe.

Depuração

STEP detalha steps; RUNNER diagnostica o runner; runner.debug só avisa que está ligado.

Artefatos

Ler baixa, escrever apaga (sem volta); v4 não sobrescreve; sem name traz tudo.

Templates

Template copia, reusable centraliza, composite encapsula steps.

Reusable

O chamado vê o mundo pelos olhos do caller: github, checkout e runners; o env fica para trás.

Skip

[skip ci] só pula push e pull_request.

Prazos de 30

30 dias para reexecutar e 30 dias para a aprovação de fork expirar.

Tabela de memorização: não confunda

Quando é...	...e quando é
ACTIONS_STEP_DEBUG Mais detalhe nos logs dos steps (e para ver core.debug).	**ACTIONS_RUNNER_DEBUG** Logs de diagnóstico do processo do runner e do worker.
Workflow template Ponto de partida copiado uma vez; correções futuras não chegam aos repositórios.	**Reusable workflow** Definição central chamada com uses: em um job; corrigir uma vez vale para todos.
Composite action Agrupar steps no runner do job atual; aparece como um step no log.	**Reusable workflow** Jobs com runs-on próprio (ex.: etapa em macOS); loga cada job e step.
Desabilitar workflow Pausa reversível, preserva arquivo e histórico.	**Excluir workflow** Remover o arquivo de .github/workflows.
Reexecutar todos os jobs Reusable chamado por branch é resolvido de novo.	**Reexecutar só falhados ou um job** Menor custo; mantém o SHA do reusable da primeira tentativa.
gh run list Listar execuções, filtrando com --workflow e --limit.	**gh run view** Detalhar uma execução pelo ID, inclusive o log.
Baixar artefato Basta leitura.	**Remover artefato** Exige escrita e é irreversível.

Números que caem

Item	Valor
Prazo para reexecutar	**Até 30 dias após a execução inicial**
Máximo de reexecuções	**50 vezes por execução**
Aprovação pendente de execução de fork	**Expira após 30 dias como falha**
Cancelamento forçado	**Após 5 minutos, o servidor encerra o que ainda estiver marcado para cancelar**
Excluir execução	**Concluída ou com mais de 2 semanas**
Agendamento em repositório público sem atividade	**Desabilitado após 60 dias**
Contexto strategy	**job-index começa em 0; job-total é a contagem real de jobs**

Alertas de prova (pegadinhas)

⚠ **Alerta de prova:** Reexecutar o workflow inteiro porque a matriz não reexecutaria por job.
Como não cair: Jobs de matriz podem ser reexecutados individualmente; é o menor custo.

⚠ **Alerta de prova:** Achar que a reexecução usa o commit mais novo do branch.
Como não cair: Usa o mesmo GITHUB_SHA e GITHUB_REF do evento original.

⚠ **Alerta de prova:** Achar que a reexecução feita por um admin roda com privilégios dele.
Como não cair: Os privilégios são de quem disparou a execução original.

⚠ **Alerta de prova:** Dizer que run_id ou run_number mudam ao reexecutar.
Como não cair: Só run_attempt aumenta.

⚠ **Alerta de prova:** Ligar RUNNER_DEBUG = 1 no env para ter logs de depuração.
Como não cair: O interruptor é ACTIONS_STEP_DEBUG ou ACTIONS_RUNNER_DEBUG = true como secret ou variável.

⚠ **Alerta de prova:** Usar gh run rerun --verbose ou criar secret quando o pedido é depuração pontual.
Como não cair: Reexecute com Enable debug logging ou gh run rerun --debug.

⚠ **Alerta de prova:** Esperar que o arquivo de logs da última tentativa traga todos os jobs.
Como não cair: A tentativa parcial só traz os jobs reexecutados; baixe também as anteriores.

⚠ **Alerta de prova:** Tentar sobrescrever artefato v4 com o mesmo nome concedendo actions: write.
Como não cair: Artefatos v4 são imutáveis; envie com outro nome.

⚠ **Alerta de prova:** Achar que artefatos de execuções excluídas ficam até a data de expiração.
Como não cair: Excluir a execução apaga os artefatos junto.

⚠ **Alerta de prova:** Esperar que correções no template cheguem aos workflows já criados.
Como não cair: Template é cópia independente; para centralizar, use reusable workflow.

⚠ **Alerta de prova:** Achar que o reusable de outro repositório faz checkout do próprio repositório ou enxerga o env do caller.
Como não cair: Contexto github e checkout são do caller; env não atravessa a chamada.

⚠ **Alerta de prova:** Colocar runs-on ou steps no job que chama um reusable workflow.
Como não cair: O runner é definido pelos jobs do chamado; o job caller não tem steps.

⚠ **Alerta de prova:** Achar que [skip ci] impede workflows de schedule, pull_request_target ou workflow_dispatch.
Como não cair: Só push e pull_request são afetados.

⚠ **Alerta de prova:** Achar que workflow desabilitado ainda permite reexecutar execuções antigas.
Como não cair: É preciso habilitar de novo antes; nem admin nem CLI contornam.

Como cai na prova

📝 **Como cai:** Sintoma de execução (job pulado, valor errado, log faltando) e pergunta pela causa.
Como resolver: Localize em que nível ocorre (execução, tentativa, job, step); confira propagação de needs, regras de reexecução, step recolhido e tentativas parciais antes de escolher.

📝 **Como cai:** Qual comando gh ou endpoint REST atende o pedido.
Como resolver: Separe recurso e verbo: runs/{run_id}/logs (logs), runs/{run_id}/artifacts (artefatos), workflows/{id}/disable (pausar), gh run list/view/download/rerun, gh workflow disable/enable. Descarte caminhos inventados e verbos trocados.

📝 **Como cai:** Escolher entre template, reusable workflow e composite action.
Como resolver: Cópia inicial independente = template; lógica central com jobs e runner próprio = reusable; só agrupar steps no job atual = composite.

📝 **Como cai:** Reusable workflow de outro repositório se comporta de forma inesperada.
Como resolver: Lembre que contexto github, checkout, runners e cobrança são do caller, env não atravessa e o acesso a repositório privado depende de Access.

📝 **Como cai:** Precisa de mais informação de log.
Como resolver: Pontual e sem criar nada: reexecutar com debug. Permanente: ACTIONS_STEP_DEBUG para steps ou ACTIONS_RUNNER_DEBUG para o runner, como secret ou variável.

Gatilhos do enunciado

Se o enunciado trouxer...	Pense em
Uma combinação da matriz falhou por instabilidade	**Reexecutar só aquele job**
Job skipped sem explicação	**system.txt do job no zip de logs; ou falha propagada por needs**
Busca no log não encontra mensagem impressa	**O step estava recolhido; expanda antes de buscar**
Quais ferramentas estavam no runner hospedado ontem	**Set up job > Runner Image > Included Software**
Matriz com output de mesmo nome e valores inconsistentes	**Prevalece o último job a terminar; use nome único por combinação**
Integridade do artefato baixado	**Digest SHA-256 gerado no upload e conferido no download**
Data em que um artefato será apagado	**expires_at na REST API de artefatos**
Execução de fork aguardando aprovação há mais de um mês	**Expira após 30 dias como falha, com anotação**
Fork de repositório público sem agendamento rodando	**Agendados vêm desabilitados em forks públicos**
Só uma pessoa recebe notificação de falha do agendamento	**Foi quem alterou o cron por último**
PR de documentação bloqueado por check obrigatório com filtro paths	**Workflow pulado deixa o check Pending**
SHA publicado em pull_request diferente do último commit do branch	**GITHUB_SHA é o merge commit; use github.event.pull_request.head.sha**

Nome da execução manual com ambiente e quem disparou	**run-name com inputs e github.actor**
Toda execução nova falha logo após editar um workflow	**Arquivo de workflow inválido em .github/workflows**

🎯 Se você lembrar de só uma coisa deste capítulo...

- Reexecução usa o mesmo commit e os privilégios de quem disparou; só run_attempt muda.
- ACTIONS_STEP_DEBUG e ACTIONS_RUNNER_DEBUG são os interruptores; RUNNER_DEBUG e runner.debug só refletem o estado.
- Desabilitar não cancela o que está rodando, mas bloqueia reexecuções até reativar.
- Template é cópia; reusable roda no contexto do caller e o env não atravessa.
- Excluir execução apaga artefatos; remover artefato exige escrita e não tem volta.

Essenciais para a prova (checklist)

- ☐ Sei as opções, o prazo, o limite e os privilégios de uma reexecução.
- ☐ Sei ligar logs de depuração de step e de runner e reexecutar com debug.
- ☐ Sei onde ver por que um job foi pulado e qual software o runner tinha.
- ☐ Sei baixar, listar, validar e remover artefatos pela interface, CLI e REST API.
- ☐ Sei as regras de imutabilidade de artefatos v4 e do download sem name.
- ☐ Sei desabilitar e reativar workflows e a diferença para excluir.
- ☐ Sei criar e usar workflow templates com .properties.json e $default-branch.
- ☐ Sei como contexto, checkout, env e chaves do job caller funcionam em reusable workflows.
- ☐ Sei o efeito de [skip ci] e de filtros de paths em checks obrigatórios.
- ☐ Sei por que agendamentos param em forks e repositórios públicos inativos.

Questões de revisão do capítulo

R2.1 · Consumir e solucionar problemas de fluxos de trabalho

Na Chopperia Barril, uma matriz com quatro jobs usa strategy.job-index e strategy.job-total para nomear relatórios. Quais valores o primeiro job da matriz recebe?

A. Índice 1 e job-total igual a 3, contando só a partir do segundo job
B. Índice 1, porque job-index começa em um, e job-total igual a 4
C. Índice 0 e job-total igual a 3, porque os dois começam em zero
D. Índice 0, porque job-index começa em zero, e job-total igual a 4

R2.2 · Consumir e solucionar problemas de fluxos de trabalho

Um analista do Atacadão Preço Bom usa a busca do log de uma execução e não encontra uma mensagem que ele sabe que foi impressa por um step. Qual é a causa provável?

A. A busca desconsidera as mensagens de steps concluídos com sucesso
B. A busca só funciona nos logs baixados no arquivo compactado
C. A busca exige ACTIONS_STEP_DEBUG ligado para indexar o texto
D. A busca só considera steps expandidos, e esse estava recolhido

R2.3 · Consumir e solucionar problemas de fluxos de trabalho

A Locadora Roda Livre chama, a partir do repositório app-frota, um reusable workflow guardado no repositório ci-central. O workflow chamado usa actions/checkout sem parâmetros. Qual repositório é baixado?

A. Nenhum, porque o checkout exige o input repository nesse caso
B. O app-frota, que é o repositório do workflow que fez a chamada
C. Os dois repositórios, cada um em uma pasta separada
D. O ci-central, onde o arquivo do reusable workflow fica

R2.4 · Consumir e solucionar problemas de fluxos de trabalho

Um script da Loja Moda Jovem precisa baixar os logs completos de uma execução de workflow para arquivamento. Qual endpoint da REST API usar?

A. DELETE /repos/{owner}/{repo}/actions/runs/{run_id}/logs
B. GET /repos/{owner}/{repo}/actions/runs/{run_id}/logs
C. GET /repos/{owner}/{repo}/actions/workflows/{workflow_id}/logs
D. GET /repos/{owner}/{repo}/actions/runs/{run_id}/artifacts

Respostas comentadas

R2.1. D) Índice 0, porque job-index começa em zero, e job-total igual a 4

Por quê: O contexto strategy traz fail-fast, job-index (base zero), job-total e max-parallel.

✗ A) Incorreta. Os dois valores estão errados para o primeiro job.
✗ B) Incorreta. O primeiro índice é 0, não 1.
✗ C) Incorreta. job-total não começa em zero; com quatro jobs, vale 4.

> **Dica de prova:** job-index começa em 0; job-total é a contagem real.

R2.2. D) A busca só considera steps expandidos, e esse estava recolhido

Por quê: Para buscar no log, expanda os steps relevantes e use a caixa Search logs no canto superior direito da saída.

✗ A) Incorreta. O resultado do step não altera a busca; o que importa é estar expandido.
✗ B) Incorreta. A busca funciona na interface da execução.
✗ C) Incorreta. Logs de depuração não são requisito para buscar.

> **Dica de prova:** Busca no log: expanda o step antes de procurar.

R2.3. B) O app-frota, que é o repositório do workflow que fez a chamada

Por quê: Quando você reutiliza um workflow de outro repositório, as ações do workflow chamado agem sobre o contexto do caller, inclusive o actions/checkout.

✗ A) Incorreta. O input repository é opcional; sem ele, vale o repositório do caller.
✗ C) Incorreta. Apenas o repositório do caller é baixado por padrão.
✗ D) Incorreta. O checkout sem parâmetros não baixa o repositório que guarda o reusable workflow.

> **Dica de prova:** Reusable workflow de outro repo faz checkout do repo que chamou.

R2.4. B) GET /repos/{owner}/{repo}/actions/runs/{run_id}/logs

Por quê: Também há endpoints para logs de uma tentativa (.../attempts/{attempt_number}/logs) e de um job (.../jobs/{job_id}/logs).

✗ A) Incorreta. Esse método exclui os logs da execução.
✗ C) Incorreta. Esse endpoint não existe.
✗ D) Incorreta. Esse endpoint lista artefatos da execução.

> **Dica de prova:** Logs da execução por API: GET .../runs/{run_id}/logs.

03

CAPÍTULO 3 · 20-25% DA PROVA

Criar e gerenciar fluxos de trabalho

Este é o domínio mais pesado junto com o de enterprise e mora no YAML: gatilhos e filtros, tipos de atividade, workflow_dispatch e workflow_call, condições com funções de status, contextos e expressões, variáveis de ambiente, arquivos de ambiente (GITHUB_ENV, GITHUB_OUTPUT, GITHUB_PATH, GITHUB_STEP_SUMMARY), matriz, service containers, cache e concorrência. A prova mostra um trecho de workflow e pergunta se roda, quantos jobs gera, que valor aparece ou o que corrigir. Eu erro quando confundo chaves que existem mas fazem outra coisa: GITHUB_ENV com GITHUB_OUTPUT, always() com !cancelled(), toJSON com fromJSON, branches-ignore com branches e !. Contar jobs de matriz com include e exclude também exige atenção.

Neste capítulo

19 tópicos-chave · 14 pegadinhas · 8 macetes · 14 gatilhos · 4 questões comentadas

Tópicos-chave

Filtros de branches e paths

branches e branches-ignore não podem ser usados juntos no mesmo evento; para incluir e excluir, use branches com ! depois do padrão positivo, porque a ordem importa. Com branches e paths definidos, os dois precisam ser satisfeitos. Em paths basta um arquivo casar; em paths-ignore o workflow só deixa de rodar se todos os arquivos alterados casarem.

Filtro só de tags ou só de branches

Se o push define só tags (ou tags-ignore), pushes de branch não disparam; se define só branches, pushes de tag não disparam. Sem nenhum dos dois, roda para branches e tags.

Tipos de atividade e múltiplos eventos

pull_request e pull_request_target sem types disparam só em opened, synchronize e reopened; labeled e closed precisam ser declarados. Cada atividade gera uma execução, mesmo que ocorram juntas (issue aberta com dois labels e types [opened, labeled] = três execuções). Vários eventos em on funcionam como OU.

workflow_dispatch

Inputs aceitam boolean, choice, number, environment e string. O botão Run workflow só aparece com o arquivo no branch padrão; inputs são opcionais. Use inputs.NOME para manter booleanos, porque github.event.inputs converte para string.

workflow_call (reusable workflow)

Inputs exigem type (boolean, number ou string); default é opcional. Secrets usam required: true para exigir que o caller os passe; sem isso, secret ausente vira string vazia. Outputs usam value apontando para jobs.JOB.outputs.NOME, nunca direto para steps.

schedule

Cron POSIX de cinco campos, interpretado em UTC por padrão, executado no último commit do branch padrão, com intervalo mínimo de 5 minutos. Sob alta carga, agendamentos no início da hora podem atrasar ou ser descartados, então escolha outro minuto. Em repositório público, agendamentos são desabilitados após 60 dias sem atividade.

workflow_run e repository_dispatch

workflow_run com workflows: [Nome] e types: [completed] roda depois de outro workflow, com o resultado em github.event.workflow_run.conclusion. repository_dispatch exige event_type, filtrável com types, e entrega o client_payload em github.event.client_payload.

Funções de status no if

Sem função de status, o if aplica success(). Para reagir a falha, comece com failure() (ex.: failure() && steps.testes.conclusion == 'failure'). always() roda até no cancelamento; a recomendação para sucesso ou falha, sem cancelamento, é if: ${{ !cancelled() }}.

Sintaxe de expressões

Literais de string usam aspas simples; aspas duplas geram erro. Não existem os operadores in nem ===; para lista de valores use contains(fromJSON('[...]'), github.event_name). No if pode-se omitir ${{ }}, exceto quando a expressão começa com !, reservado no YAML.

Contextos versus variáveis padrão

Contextos como github.ref são avaliados antes de o job ir ao runner; variáveis padrão como $GITHUB_REF só existem no runner e não estão no contexto env. github.ref é o ref completo (refs/heads/release/2.4) e github.ref_name o nome curto (release/2.4). vars inexistente vira string vazia.

Restrições de contexto no if

Secrets não podem ser usados diretamente em if: mapeie para env do job e teste env.NOME != ''. O if de job não aceita o contexto matrix; para pular combinações use exclude ou condição nos steps.

env e defaults

env de step vence o de job, que vence o de workflow. Uma variável não pode referenciar outra do mesmo mapa env. defaults.run define shell e working-directory dos steps run e, no nível do workflow, não aceita contextos nem expressões; defaults não aceita env.

Arquivos de ambiente e comandos

GITHUB_ENV cria variável para os próximos steps do job; GITHUB_OUTPUT cria output lido por steps.ID.outputs.NOME; GITHUB_PATH acrescenta diretório ao PATH dos próximos steps; GITHUB_STEP_SUMMARY gera Markdown no resumo do job. O step que grava não enxerga a mudança. ::group::/::endgroup:: recolhe log e ::warning file=...,line=...:: cria anotação no arquivo.

Outputs entre jobs

Mapeie steps.ID.outputs para jobs.JOB.outputs e, no job dependente, declare needs e leia needs.JOB.outputs.NOME. Output que contém secret é omitido (Skip output since it may contain secret); para isso, use um cofre e passe só um identificador.

Matriz

Cada combinação vira um job (multiplique os tamanhos das listas), com limite de 256 jobs. include que casa acrescenta chaves sem sobrescrever; que não casa cria job novo. exclude precisa casar só parcialmente e remove todas as combinações correspondentes; include é processado depois. fail-fast padrão é true; max-parallel limita jobs simultâneos; matriz dinâmica usa fromJSON(needs.JOB.outputs.NOME).

needs e propagação

O job espera todos os jobs listados em needs. Se uma dependência falha ou é pulada, o dependente é pulado (skipped, não failure), a menos que use if: always().

Shells padrão

Linux e macOS sem shell: bash -e {0}, sem pipefail. shell: bash aplica bash --noprofile --norc -eo pipefail {0}. Windows sem shell: pwsh (PowerShell Core). Em job container, o shell padrão de run é sh.

Service containers e job container

services, container e ações Docker exigem runner Linux. Job em contêiner fala com o service pelo rótulo como hostname, sem mapear portas; job direto no runner mapeia portas, usa localhost e lê a porta em job.services.ID.ports. Para esperar o serviço, use options com --health-cmd.

Cache e concurrency

Chave boa de cache usa hashFiles do lockfile; se a key muda, restore-keys restaura o cache mais recente pelo prefixo e um cache novo é salvo se o job passar (cache existente não é sobrescrito). Em concurrency, use ${{ github.workflow }}-${{ github.ref }} para não cancelar outros workflows; concurrency existe no workflow e no job.

Macetes para memorizar

Arquivos de ambiente

ENV = variável, OUTPUT = saída, PATH = caminho, STEP_SUMMARY = relatório; todos valem do próximo step em diante.

Filtros de ref

Só tags: branch fica de fora. Só branches: tag fica de fora. Nenhum: tudo dispara.

Cron

5 campos, UTC, branch padrão, mínimo 5 minutos, e fuja do minuto zero.

outcome x conclusion

outcome = o que aconteceu; conclusion = o que ficou registrado depois do continue-on-error.

Matriz

include que casa soma chave; que não casa vira job. exclude basta casar uma parte.

Shells

Linux sem shell: bash -e. bash explícito: + pipefail. Windows: pwsh. Contêiner: sh.

Status

!cancelled() = sucesso ou falha; always() = até cancelado; failure() = abre a porta para pós-falha.

Expressão

Aspas simples sempre; ! no começo pede ${{ }}.

Tabela de memorização: não confunda

Quando é...	...e quando é
GITHUB_ENV Variável de ambiente para os próximos steps do mesmo job.	**GITHUB_OUTPUT** Output de step, lido por steps.ID.outputs.NOME e exposto em jobs.JOB.outputs para outros jobs.
always() Tarefa que precisa rodar até se a execução for cancelada.	**!cancelled()** Rodar com sucesso ou falha, mas não quando alguém cancelar (recomendação da documentação).
inputs.NOME Comparar input booleano de workflow_dispatch como booleano.	**github.event.inputs.NOME** Mesma informação, mas convertida em string; a comparação com true falha.
Job em contêiner (container:) Conecta ao service pelo rótulo (ex.: db), sem mapear portas; shell padrão sh.	**Job direto no runner** Mapeia portas, usa localhost e lê a porta em job.services.NOME.ports.
steps.ID.outcome Resultado bruto do step (failure se falhou).	**steps.ID.conclusion** Resultado depois de continue-on-error (success se o erro foi tolerado).
workflow_run Disparar um workflow depois que outro workflow termina.	**needs** Ordenar jobs dentro do mesmo workflow.
strategy.max-parallel Limitar quantos jobs da matriz rodam ao mesmo tempo.	**strategy.fail-fast** Decidir se uma falha cancela os demais jobs da matriz (padrão true).

Números que caem

Item	Valor
Jobs por matriz	**Máximo de 256, em runners hospedados e auto-hospedados**
timeout-minutes padrão do job	**360 minutos**
Cron	**5 campos, UTC por padrão, intervalo mínimo de 5 minutos**

Agendamento em repositório público sem atividade	**Desabilitado após 60 dias**
Resumo de job	**Até 1 MiB por step; até 20 resumos de steps exibidos por job**
Outputs de job	**Até 1 MB por job; 50 MB no total da execução**
Encadeamento com workflow_run	**Até 3 níveis**
repository_dispatch	**event_type até 100 caracteres; client_payload até 10 propriedades de primeiro nível**

Alertas de prova (pegadinhas)

⚠ **Alerta de prova:** Usar branches e branches-ignore juntos no mesmo evento.
Como não cair: É rejeitado; use só branches com padrões ! para excluir.

⚠ **Alerta de prova:** Colocar o padrão negativo antes do positivo, como ['!releases/*-alpha', 'releases/*'].
Como não cair: A ordem importa: positivo depois do negativo inclui de novo. O ! vem depois do padrão positivo.

⚠ **Alerta de prova:** Usar types: [merged] em pull_request.
Como não cair: Não existe merged; use types: [closed] com if: github.event.pull_request.merged == true.

⚠ **Alerta de prova:** Usar types: [released, prereleased] para pegar toda release publicada.
Como não cair: prereleased não dispara para pré-release publicada a partir de rascunho; use types: [published]. created não dispara para rascunhos.

⚠ **Alerta de prova:** Escrever if: steps.testes.conclusion == 'failure' e esperar que rode após a falha.
Como não cair: Sem função de status vale success(); combine failure() && com a checagem.

⚠ **Alerta de prova:** Usar always() para um relatório que não deve rodar em cancelamento.
Como não cair: always() roda até cancelado; use !cancelled().

⚠ **Alerta de prova:** Gravar em GITHUB_ENV e ler a variável na linha seguinte do mesmo step.
Como não cair: O valor só vale do próximo step em diante; não existe ::set-env:: nem efeito em outro job.

⚠ **Alerta de prova:** Achar que include troca a versão de uma combinação existente.
Como não cair: include nunca sobrescreve valores originais: se não encaixa, cria job extra (9 combinações + include novo = 10).

⚠ **Alerta de prova:** Contar que cada entrada de exclude remove um único job.
Como não cair: exclude casa parcialmente e remove todas as combinações correspondentes.

⚠ **Alerta de prova:** Filtrar a matriz com if: matrix.os == ... no job.
Como não cair: O if de job é avaliado antes da matriz; use exclude ou condição nos steps.

⚠ **Alerta de prova:** Usar toJSON ou passar o output sem conversão para montar matriz dinâmica.
Como não cair: Outputs são strings; use fromJSON(needs.JOB.outputs.NOME).

⚠ **Alerta de prova:** Esperar que um push feito com GITHUB_TOKEN dispare o CI on: push.
Como não cair: Eventos criados com GITHUB_TOKEN não iniciam workflows (exceto workflow_dispatch e repository_dispatch); use token de GitHub App ou PAT.

⚠ **Alerta de prova:** Achar que paths-ignore descarta qualquer push que toque em docs.
Como não cair: Basta um arquivo fora dos padrões ignorados para o workflow rodar.

⚠ **Alerta de prova:** Declarar shell: bash achando que remove o -e, ou que o padrão Linux já tem pipefail.
Como não cair: Padrão Linux é bash -e sem pipefail; shell: bash mantém -e e acrescenta -o pipefail.

Como cai na prova

📝 **Como cai:** Matriz com listas, include e exclude: quantos jobs rodam?
Como resolver: Multiplique os tamanhos das listas; aplique cada exclude removendo todas as combinações que casam parcialmente; depois aplique include: se encaixa sem sobrescrever, só acrescenta chave; se não, soma um job. Confira o limite de 256.

📝 **Como cai:** Push ou PR com filtros de branches, paths ou tags: o workflow roda?
Como resolver: Cheque cada filtro separado: branches com ! respeitando a ordem; paths (basta um) ou paths-ignore (só bloqueia se todos casarem); com branches e paths, os dois precisam passar; filtro só de tags não pega branch.

📝 **Como cai:** Condição if que não funciona como esperado.
Como resolver: Pergunte: falta função de status? começa com ! sem ${{ }}? usa aspas duplas? usa secrets direto ou matrix no if de job? usa variável de ambiente em vez de contexto? vars inexistente virou string vazia?

📝 **Como cai:** Valor que chega vazio entre steps, jobs ou reusable workflow.
Como resolver: Entre steps: GITHUB_OUTPUT + id (ou GITHUB_ENV para variável). Entre jobs: jobs.JOB.outputs + needs. Reusable: workflow_call.outputs.value aponta para jobs.JOB.outputs. Segredo: cofre + identificador.

📝 **Como cai:** Escolher o gatilho certo para um cenário.
Como resolver: Mapeie o verbo: manual (workflow_dispatch), chamado por outro (workflow_call), depois de outro workflow (workflow_run), sistema externo (repository_dispatch), horário (schedule), release publicada (release published), merge (closed + merged).

Gatilhos do enunciado

Se o enunciado trouxer...	Pense em
Rodar sempre que outro workflow terminar, com sucesso ou falha	**on: workflow_run com workflows: [Nome] e types: [completed]**

Job só quando o PR for mesclado	**pull_request types: [closed] + if github.event.pull_request.merged == true**
issue_comment só quando o comentário é em pull request	**if: ${{ github.event.issue.pull_request }}**
Saída longa recolhida em seção expansível no log	**::group::título e ::endgroup::**
Tabela Markdown na página de resumo da execução	**Gravar no arquivo GITHUB_STEP_SUMMARY**
CLI instalada em pasta própria encontrada pelos próximos steps	**echo da pasta >> $GITHUB_PATH**
API externa aceita só duas conexões simultâneas para a matriz	**strategy.max-parallel: 2**
Versão experimental da matriz que pode falhar sem reprovar	**continue-on-error: ${{ matrix.experimental }} no job**
Dois workflows do mesmo repositório cancelando um ao outro	**group: ${{ github.workflow }}-${{ github.ref }}**
Testes começam antes de o banco do service aceitar conexões	**options com --health-cmd (ex.: pg_isready) e intervalos**
Repetir um bloco (env ou job inteiro) no mesmo arquivo	**Âncora &nome e alias *nome do YAML (não cria needs)**
Badge de status de um branch específico no README	**actions/workflows/ARQUIVO/badge.svg?branch=NOME**
Link do site publicado na página de implantações	**environment com url: ${{ steps.ID.outputs.NOME }}**
Job sem timeout travou esperando uma API	**Padrão de timeout-minutes é 360 minutos**

🎯 **Se você lembrar de só uma coisa deste capítulo...**

- Sem função de status o if vira success(); pós-falha começa com failure(), e relatório sem cancelamento usa !cancelled().
- include nunca sobrescreve; exclude casa parcialmente; matriz tem limite de 256 jobs.
- Quem grava em GITHUB_ENV, GITHUB_OUTPUT ou GITHUB_PATH não vê o efeito; só os próximos steps.
- Push feito com GITHUB_TOKEN não dispara novo workflow (salvo workflow_dispatch e repository_dispatch).
- Contêiner no workflow (services, container ou ação Docker) exige runner Linux.

Essenciais para a prova (checklist)

- ☐ Sei prever se um push ou PR dispara o workflow com branches, paths, tags e seus -ignore.
- ☐ Sei os tipos padrão de pull_request e quantas execuções eventos simultâneos geram.
- ☐ Sei declarar inputs de workflow_dispatch e de workflow_call e seus tipos aceitos.
- ☐ Sei escrever condições com failure(), always(), !cancelled() e contains(fromJSON(...)).
- ☐ Sei passar dados entre steps, entre jobs e de reusable workflow para o caller.
- ☐ Sei contar jobs de matriz com include e exclude e usar max-parallel, fail-fast e continue-on-error.
- ☐ Sei como um job em contêiner e um job no runner se conectam a service containers.
- ☐ Sei o shell padrão em Linux, Windows e job container e o efeito de shell: bash.

- [ ] Sei montar chave de cache com hashFiles e grupo de concurrency por workflow.
- [ ] Sei as regras do schedule: cinco campos, UTC, branch padrão, 5 minutos e 60 dias.

Questões de revisão do capítulo

R3.1 · Criar e gerenciar fluxos de trabalho

Na Cargas Brasil Norte, um job usa if: ${{ vars.DEPLOY_HABILITADO == 'true' }}. Em um repositório onde essa variável de configuração nunca foi criada, o que acontece?

A. O valor true passa a ser assumido para variáveis ausentes
B. A referência vira string vazia e o job acaba pulado
C. O GitHub busca a variável em secrets e usa aquele valor
D. O workflow falha com erro de variável não definida

R3.2 · Criar e gerenciar fluxos de trabalho

Na Laticínios Vale do Leite, um step de notificação deve rodar só quando o step com id testes falhar. O time escreveu if: steps.testes.conclusion == 'failure', mas o step nunca roda. Qual condição corrige?

A. always() || steps.testes.outcome == 'failure'
B. cancelled() && steps.testes.conclusion == 'failure'
C. steps.testes.outcome != 'success'
D. failure() && steps.testes.conclusion == 'failure'

R3.3 · Criar e gerenciar fluxos de trabalho

No workflow da Transportadora Via Norte, o job test recebe a âncora &base_job com runs-on, timeout-minutes e steps, e o job alt-test é definido apenas como *base_job. O que alt-test executa?

A. Os steps do job test, só depois que o job test terminar
B. Os mesmos runs-on, timeout-minutes e steps do job test
C. Os steps do job test, mas com um runner padrão do GitHub
D. Nada, pois aliases YAML não são aceitos em workflows

R3.4 · Criar e gerenciar fluxos de trabalho

No workflow da Viação Litorânea, AMBIENTE=geral é definido no topo do arquivo, AMBIENTE=job no job deploy e AMBIENTE=step em um step desse job. Qual valor esse step enxerga?

A. step, pois o nível mais específico prevalece
B. geral, pois o nível de workflow sobrepõe os outros
C. job, pois o step não sobrescreve variável do job
D. Erro de validação, pois o nome foi repetido

Respostas comentadas

R3.1. B) A referência vira string vazia e o job acaba pulado

Por quê: O contexto vars traz variáveis de configuração de organização, repositório e ambiente; quando a variável não foi definida, retorna string vazia.

✗ A) Incorreta. Não existe valor padrão true para variáveis ausentes.
✗ C) Incorreta. vars e secrets são contextos separados.
✗ D) Incorreta. Não há erro; o valor ausente é tratado como string vazia.

> **Dica de prova:** vars inexistente = string vazia.

R3.2. D) failure() && steps.testes.conclusion == 'failure'

Por quê: Se o if não contém função de status, o GitHub aplica success(); para reagir a uma falha com condição extra, combine failure() com a checagem do step.

✗ A) Incorreta. always() com || faria o step rodar em todos os casos.
✗ B) Incorreta. cancelled() só é verdadeiro quando o workflow é cancelado.
✗ C) Incorreta. Sem função de status, o success() padrão continua impedindo o step após a falha.

> **Dica de prova:** Condição pós-falha sempre começa com failure().

R3.3. B) Os mesmos runs-on, timeout-minutes e steps do job test

Por quê: Uma âncora (&) marca um trecho e o alias (*) o repete; usar a âncora em um job inteiro reaproveita runs-on, timeout e steps no outro job.

✗ A) Incorreta. O alias não cria dependência; sem needs, os jobs rodam em paralelo.
✗ C) Incorreta. O runs-on faz parte da configuração repetida pelo alias.
✗ D) Incorreta. Workflows aceitam âncoras e aliases YAML para reduzir repetição.

> **Dica de prova:** &nome define, *nome repete o bloco inteiro; não cria needs.

R3.4. A) step, pois o nível mais específico prevalece

Por quê: Em env, a variável definida no step sobrepõe as de job e workflow durante o step; a do job sobrepõe a do workflow durante o job.

✗ B) Incorreta. O nível de workflow é o menos específico e perde para job e step.
✗ C) Incorreta. Variáveis de step sobrescrevem as de job enquanto o step executa.
✗ D) Incorreta. Repetir o nome em níveis diferentes é permitido.

> **Dica de prova:** env: step vence job, que vence workflow.

04

CAPÍTULO 4 · 15-20% DA PROVA

Criar e manter ações

Aqui eu preciso saber construir ações dos três tipos (JavaScript, contêiner Docker e composite), escrever o action.yml, usar o Actions Toolkit, tratar inputs, outputs e códigos de saída, e distribuir, versionar e publicar a ação no Marketplace. As questões trazem uma ação quebrada (input vazio, output vazio, script não encontrado, variável não expandida) e pedem a correção, ou pedem o tipo de ação certo para uma restrição de sistema operacional e velocidade. Onde mais erro: achar que composite cria INPUT_, esquecer que Docker só roda em Linux, confundir args com CMD e GITHUB_STATE com GITHUB_ENV, e versionar errado (mover major para versão incompatível ou mexer em release imutável).

Neste capítulo
18 tópicos-chave · 14 pegadinhas · 7 macetes · 14 gatilhos · 4 questões comentadas

Tópicos-chave

Tipos de ação

JavaScript: separa código do ambiente, é a mais rápida e roda em Linux, macOS e Windows se for JavaScript puro. Contêiner Docker: empacota sistema e ferramentas, é mais lenta por causa da imagem e só roda em runner Linux. Composite: agrupa steps run e uses e roda nos três sistemas.

Arquivo de metadados

Toda ação precisa de action.yml ou action.yaml (preferência por action.yml). Obrigatórios: name, description e runs; author, inputs, outputs e branding são opcionais. Trocar action.yml por action.yaml em ação publicada tira as versões antigas do Marketplace, embora sigam no repositório.

runs de ação JavaScript

using: node20 ou node24 e main apontando para o arquivo (ex.: dist/index.js); opcionais pre, post, pre-if e post-if. pre roda no início do job e post no fim, com padrão always(); em pre-if e post-if as funções de status avaliam o job, e o contexto steps não existe no pre. runs.pre não funciona quando a ação é usada como ação local.

runs de ação Docker

using: docker e image: Dockerfile (caminho relativo ao metadata, arquivo chamado Dockerfile) ou docker://imagem pública. args no action.yml substitui o CMD do Dockerfile e vai para o ENTRYPOINT; entrypoint no action.yml substitui o ENTRYPOINT. Inputs só chegam ao contêiner como argumento se passados em args; pre-entrypoint e post-entrypoint fazem o papel de pre e post.

Regras do Dockerfile de ação

Não use USER: ações Docker rodam como root. Evite WORKDIR, porque o GitHub monta o GITHUB_WORKSPACE como diretório de trabalho; use caminho absoluto no ENTRYPOINT. A forma exec do ENTRYPOINT não expande variáveis: chame um shell (sh -c) ou um entrypoint.sh, que precisa de chmod +x.

Composite action

runs.using: composite com runs.steps; shell é obrigatório em todo step com run e não há defaults no action.yml. Inputs são lidos com ${{ inputs.nome }} (composite não cria INPUT_); outputs precisam de value ligado a steps.ID.outputs.NOME; arquivos da própria ação via ${{ github.action_path }} ou $GITHUB_ACTION_PATH. No log, aparece como um único step.

Inputs de ação

Em JavaScript e Docker, cada input vira a variável INPUT_NOME em maiúsculas. required: true documenta, mas não gera erro automático se o input não for passado: valide no código. deprecationMessage registra aviso quando um input em desuso é usado.

Outputs de ação

Ações JavaScript e Docker podem gravar outputs não declarados via GITHUB_OUTPUT e eles funcionam; declarar no action.yml serve para documentar a interface. Valor mascarado não pode virar output.

Actions Toolkit

Para ações em Node, a documentação indica @actions/core e @actions/github. Mapeamento: getInput = INPUT_, setOutput = GITHUB_OUTPUT, exportVariable = GITHUB_ENV, saveState = GITHUB_STATE (lido como STATE_ no pre e post da mesma ação), setSecret = add-mask, debug = ::debug:: (visível com ACTIONS_STEP_DEBUG), startGroup e endGroup = group.

Códigos de saída

0 é sucesso; qualquer outro valor é falha (não existe neutral). Em JavaScript use core.setFailed(mensagem); em Docker, exit 1 ou outro não zero no entrypoint. Quando uma ação falha, ações concorrentes são canceladas e as seguintes são puladas.

Empacotamento de ação JavaScript

O GitHub baixa e executa a ação como pacote completo do commit referenciado: inclua as dependências empacotadas em dist com rollup ou ncc, sem commitar node_modules. No processo de exemplo, as dependências compiladas ficam só nos commits das tags de release, montados por workflow.

Onde guardar a ação

Ação usada só pelo próprio repositório: qualquer pasta, de preferência .github/actions. Ação para outros usarem: repositório dedicado, com versionamento e releases. Ação em subpasta de outro repositório: uses: owner/repo/pasta@ref.

Formas de referência em uses

owner/repo@ref, owner/repo/caminho@ref, ./caminho (relativo ao workspace, exige actions/checkout antes e não leva @) e docker://imagem:tag para rodar imagem pública sem criar ação.

Compartilhar ação privada

Guarde em repositório privado e libere os repositórios da organização na seção Access das settings. O runner baixa com token de instalação só leitura que expira em uma hora. Risco: colaboradores externos dos repositórios liberados podem ver logs de execuções que usam a ação.

Versionamento semântico

Patch corrige, minor acrescenta recurso compatível, major traz mudança incompatível (como renomear input obrigatório). Versão compatível: nova release (v1.4.2) e mover a tag major v1 para ela; incompatível: publicar v3 e manter v2. Usuários referenciam @v1 ou SHA completo, nunca @main; o GitHub não mantém tag latest.

Immutable releases

Travam a tag e os assets da release (não podem ser movidos, modificados nem excluídos); título e notas continuam editáveis, e gera-se uma release attestation. Major móvel fica como tag Git sem release. Excluída uma release imutável, o mesmo nome de tag não pode ser reutilizado; boa prática é criar como rascunho, anexar assets e publicar.

Publicação no Marketplace

Requisitos: repositório público, um único arquivo de metadados na raiz e nome único. Exige aceitar o GitHub Marketplace Developer Agreement (em organização, um owner aceita) e 2FA. Publica-se em Draft a release marcando a opção, com categoria e tag, sem revisão do GitHub; para despublicar, desmarque em cada release. branding.icon (Feather) e branding.color formam o badge.

Portabilidade e documentação

Para funcionar em GitHub.com, GHE.com e GitHub Enterprise Server, use GITHUB_API_URL, GITHUB_GRAPHQL_URL ou @actions/github, nunca URL fixa. O README deve ter descrição, inputs e outputs obrigatórios e opcionais, secrets e variáveis usados e um exemplo de uso.

Macetes para memorizar

Escolha do tipo

Rápida e em todo lugar: JavaScript. Ambiente fechado: Docker (só Linux). Juntar steps: composite.

action.yml mínimo

N-D-R: name, description, runs.

Dockerfile de ação

Sem USER, sem WORKDIR, ENTRYPOINT absoluto, chmod +x, shell para expandir variável.

Toolkit

getInput lê INPUT_, setOutput grava OUTPUT, exportVariable grava ENV, saveState grava STATE, setSecret mascara.

Composite

Composite pede três coisas: shell em cada run, inputs pelo contexto e value no output.

Versionamento

Compatível move a major; quebrou cria nova major; usuário usa @v1 ou SHA, nunca @main.

Marketplace

Público, um action.yml na raiz, nome único, acordo e 2FA; sai desmarcando release por release.

Tabela de memorização: não confunda

Quando é...	...e quando é
Ação JavaScript Precisa ser rápida e rodar em Linux, macOS e Windows com JavaScript puro.	**Ação de contêiner Docker** Precisa de sistema e ferramentas em versão específica, aceitando ser mais lenta e só Linux.
Composite action Agrupar steps para usar como um único step no runner do job atual.	**Reusable workflow** Precisa de jobs próprios com runs-on (outro runner), chamado no nível do job.
GITHUB_STATE Passar valor (ex.: PID) entre pre, main e post da mesma ação, lido como STATE_.	**GITHUB_ENV / GITHUB_OUTPUT** Variáveis para steps seguintes do job / outputs para steps e jobs seguintes.
args no action.yml Definidos, substituem o CMD e chegam ao ENTRYPOINT; levam inputs ao contêiner.	**CMD do Dockerfile** Valem só sem args; devem ter padrões sensatos como --help.
Mover a tag major (v1) Nova versão compatível, como v1.4.2.	**Publicar nova major (v3)** Mudança incompatível, mantendo a anterior para quem não migrou.
image: Dockerfile Usar o Dockerfile da pasta do action.yml.	**image: docker://imagem** Usar imagem já publicada em registro.
core.setFailed Registrar a mensagem e marcar o step como falho.	**core.warning / core.debug** Só aviso ou mensagem de depuração; a ação segue com sucesso.

Números que caem

Item	Valor
Token de download de ação privada compartilhada	**Só leitura, expira em 1 hora**
Runtimes de ação JavaScript	**node20 ou node24**
Pin por commit	**SHA completo de 40 caracteres**
Código de saída	**0 = sucesso; qualquer outro = falha**

Alertas de prova (pegadinhas)

⚠ **Alerta de prova:** Ler INPUT_AMBIENTE dentro de uma composite action.
Como não cair: Composite não cria INPUT_; use ${{ inputs.ambiente }}.

⚠ **Alerta de prova:** Declarar output de composite só com description.
Como não cair: Falta value com ${{ steps.ID.outputs.NOME }}.

⚠ **Alerta de prova:** Step run em composite sem shell, ou tentar defaults.run.shell no action.yml.
Como não cair: shell é obrigatório em cada step run; não há defaults no metadata.

⚠ **Alerta de prova:** Usar ação Docker em job windows-latest via docker:// ou com shell: bash.
Como não cair: Contêiner Docker exige runner Linux, qualquer que seja a forma de chamada.

⚠ **Alerta de prova:** Colocar USER app ou USER root no Dockerfile para resolver permissão.
Como não cair: Ações Docker já rodam como root e USER não deve ser usado; permission denied no entrypoint se resolve com chmod +x.

⚠ **Alerta de prova:** Adicionar WORKDIR para corrigir script não encontrado.
Como não cair: O GitHub monta GITHUB_WORKSPACE como diretório de trabalho; use caminho absoluto no ENTRYPOINT.

⚠ **Alerta de prova:** Trocar $GITHUB_SHA por ${{ github.sha }} no Dockerfile.
Como não cair: Expressões não são avaliadas no Dockerfile; na forma exec, chame um shell para expandir variáveis.

⚠ **Alerta de prova:** Achar que required: true faz a ação falhar quando o input falta.
Como não cair: Não há validação automática; valide no código da ação.

⚠ **Alerta de prova:** Achar que output não declarado no action.yml é descartado ou falha.
Como não cair: Funciona normalmente em JavaScript e Docker; declarar só documenta.

⚠ **Alerta de prova:** Usar GITHUB_ENV ou GITHUB_OUTPUT para o post ler um valor do main.
Como não cair: pre, main e post conversam por GITHUB_STATE (core.saveState) e STATE_.

⚠ **Alerta de prova:** Esperar revisão do GitHub, mínimo de estrelas ou repositório interno para publicar.
Como não cair: Basta repositório público, um metadata na raiz, nome único, Developer Agreement e 2FA.

⚠ **Alerta de prova:** Mover a tag de uma release imutável ou editar seus assets.
Como não cair: Tag e assets ficam travados; só título e notas mudam. Major móvel é tag Git sem release.

⚠ **Alerta de prova:** Fixar por SHA abreviado ou pelos sete primeiros caracteres.
Como não cair: A documentação exige o SHA completo do commit.

⚠ **Alerta de prova:** Renomear input obrigatório como v2.5.0 e mover v2.
Como não cair: Mudança incompatível pede nova major; mover v2 quebraria quem usa @v2.

Como cai na prova

📝 **Como cai:** Ação quebrada: input vazio, output vazio, script não achado ou variável literal no log.
Como resolver: Identifique o tipo da ação primeiro; aplique a regra do tipo: composite (inputs pelo contexto, value no output, shell, action_path), Docker (args, caminho absoluto, shell no exec, chmod), JavaScript (INPUT_, GITHUB_OUTPUT, runs.pre não vale local).

📝 **Como cai:** Escolher o tipo de ação para uma restrição.
Como resolver: Windows ou macOS no requisito elimina Docker; ambiente específico de sistema aponta Docker; velocidade aponta JavaScript; agrupar steps aponta composite; outro runner aponta reusable workflow.

📝 **Como cai:** Como versionar uma nova versão da ação.
Como resolver: Pergunte se quebra compatibilidade: não quebra = release semântica + mover major; quebra = nova major. Com immutable releases, a major móvel é tag Git sem release.

📝 **Como cai:** Publicar, despublicar ou compartilhar a ação.
Como resolver: Marketplace: requisitos (público, metadata na raiz, nome único), acordo e 2FA, Draft a release; despublicar: desmarcar em cada release. Compartilhar privado: Access no repositório.

📝 **Como cai:** Qual chamada do toolkit ou arquivo de ambiente usar.
Como resolver: Mapeie a intenção: falhar = setFailed; mascarar = setSecret; passar para post = saveState; output = setOutput; variável para steps = exportVariable; depurar = debug com ACTIONS_STEP_DEBUG.

Gatilhos do enunciado

Se o enunciado trouxer...	Pense em
Ação que precisa rodar em Ubuntu, Windows e macOS	**JavaScript ou composite, nunca Docker**
Versão específica de sistema e ferramentas independente do runner	**Ação de contêiner Docker (mais lenta, só Linux)**
Mesmos quatro steps repetidos em vários workflows no mesmo runner	**Composite action**
Token obtido em tempo de execução pode aparecer no log	**core.setSecret (add-mask) logo depois de obter**
Input antigo será removido e quem usa deve ser avisado	**deprecationMessage no input**
core.debug não aparece no log	**Habilitar ACTIONS_STEP_DEBUG = true (secret ou variável)**
Ação JavaScript sem commitar node_modules	**Empacotar em dist com rollup ou ncc**
Composite precisa executar script do próprio repositório da ação	**${{ github.action_path }}/scripts/arquivo.sh com shell**
uses: ./.github/actions/x não encontra o action.yml	**Falta actions/checkout antes**
Rodar comando em imagem alpine:3.8 sem criar ação	**uses: docker://alpine:3.8**
Opção de publicar no Marketplace desabilitada	**Developer Agreement não aceito; publicar ainda exige 2FA**
Ação chama endereço fixo da API do GitHub.com e falha em GHE.com	**GITHUB_API_URL ou @actions/github**
Ícone e cor no badge do Marketplace	**branding com icon (Feather) e color**

Compartilhar ação privada com a organização sem publicar	**Access no repositório da ação; token de leitura de uma hora**

🎯 Se você lembrar de só uma coisa deste capítulo...

- Docker só roda em runner Linux; JavaScript e composite rodam nos três sistemas.
- Composite não cria INPUT_, exige shell em cada run e value nos outputs.
- args do action.yml substituem o CMD; ação Docker roda como root, sem USER e sem WORKDIR.
- required: true não valida nada sozinho; output não declarado funciona.
- Immutable release trava tag e assets; major móvel fica em tag Git sem release.

Essenciais para a prova (checklist)

- ☐ Sei escolher entre JavaScript, Docker e composite por sistema operacional, velocidade e ambiente.
- ☐ Sei as chaves obrigatórias do action.yml e as opções de runs para cada tipo.
- ☐ Sei as regras de Dockerfile de ação: USER, WORKDIR, ENTRYPOINT, args e CMD.
- ☐ Sei ler inputs e definir outputs em composite e em JavaScript ou Docker.
- ☐ Sei o mapeamento das funções do Actions Toolkit para variáveis e arquivos de ambiente.
- ☐ Sei falhar uma ação em JavaScript e em Docker.
- ☐ Sei empacotar uma ação JavaScript sem node_modules.
- ☐ Sei versionar com tags semânticas, major móvel e immutable releases.
- ☐ Sei os requisitos e o fluxo de publicação e despublicação no Marketplace.
- ☐ Sei compartilhar ação privada com a organização e o risco envolvido.

Questões de revisão do capítulo

R4.1 · Criar e manter ações

Uma composite action da Startup Conta Fácil tem um step run: ./scripts/build.sh sem a chave shell, e a ação é rejeitada. O que falta?

A. Declarar shell no step, exigido quando o step usa run
B. Declarar runs.main apontando para o script de build
C. Trocar run por uses: ./scripts/build.sh dentro do mesmo step
D. Definir defaults.run.shell no topo do action.yml

R4.2 · Criar e manter ações

A Livraria Página Nova precisa de uma ação que use uma versão específica do sistema operacional e de ferramentas, sem depender do que está instalado no runner. Qual tipo atende e qual é a contrapartida?

A. Docker, que empacota o ambiente, mas é mais lenta por causa da imagem
B. Composite, que instala o sistema desejado antes de cada step
C. Docker, que roda em Linux, macOS e Windows com a mesma imagem
D. JavaScript, que leva o sistema operacional junto no pacote dist

R4.3 · Criar e manter ações

Uma composite action da Escola Saber Digital tenta ler o input ambiente pela variável INPUT_AMBIENTE em um step run, mas o valor chega vazio. Qual é a causa?

A. A variável correta mantém a caixa do nome: INPUT_ambiente
B. Composite não cria INPUT_ sozinha; ler com ${{ inputs.ambiente }}
C. A variável só é criada quando o input tem required: true
D. O input só vira variável quando a ação declara outputs

R4.4 · Criar e manter ações

Uma ação JavaScript do Restaurante Panela Cheia grava o output tempo com core.setOutput, mas esse output não está declarado no action.yml. O que acontece?

A. O valor vira a variável de ambiente OUTPUT_TEMPO no job seguinte
B. A ação falha porque o output não foi declarado no metadata
C. O valor é descartado e o step seguinte recebe texto vazio
D. O output funciona normalmente, ainda que não esteja declarado

Respostas comentadas

R4.1. A) Declarar shell no step, exigido quando o step usa run

Por quê: runs.steps[*].shell é obrigatório quando run está definido; os shells aceitos são os mesmos da sintaxe de workflow.

✗ B) Incorreta. runs.main é de ações JavaScript.
✗ C) Incorreta. uses chama ações, não scripts de shell.
✗ D) Incorreta. A sintaxe de metadados de composite define shell por step; não há defaults no action.yml.

> **Dica de prova:** Composite com run: shell em cada step.

R4.2. A) Docker, que empacota o ambiente, mas é mais lenta por causa da imagem

Por quê: Docker container actions empacotam o ambiente com o código; são ideais para configuração específica, mas mais lentas que JavaScript e restritas a Linux.

✗ B) Incorreta. Composite agrupa steps; não troca o sistema operacional do runner.
✗ C) Incorreta. Ações Docker só rodam em runners Linux.
✗ D) Incorreta. Ações JavaScript usam os binários do runner e não carregam o sistema operacional.

> **Dica de prova:** Ambiente controlado: Docker; custo: velocidade e só Linux.

R4.3. B) Composite não cria INPUT_ sozinha; ler com ${{ inputs.ambiente }}

Por quê: O GitHub cria INPUT_NOME para inputs de ações JavaScript e Docker, mas ações compostas devem acessar os valores pelo contexto inputs.

✗ A) Incorreta. Quando a variável é criada (JavaScript e Docker), o nome fica em maiúsculas; em composite ela simplesmente não é criada.
✗ C) Incorreta. required não faz o runner criar INPUT_ em uma ação composta.
✗ D) Incorreta. Declarar outputs não tem relação com a criação de variáveis para os inputs.

> **Dica de prova:** Composite action lê inputs com ${{ inputs.nome }}.

R4.4. D) O output funciona normalmente, ainda que não esteja declarado

Por quê: Declarar outputs no action.yml documenta a interface da ação, mas JavaScript e Docker actions podem definir outputs não declarados via GITHUB_OUTPUT.

✗ A) Incorreta. Outputs não viram variáveis de ambiente com esse prefixo.

✗ B) Incorreta. Não declarar o output não gera falha.

✗ C) Incorreta. O valor não é descartado.

> **Dica de prova:** Output não declarado ainda funciona; declare para documentar.

05

CAPÍTULO 5 · 20-25% DA PROVA

Gerenciar o GitHub Actions na empresa (enterprise)

Aqui eu preciso dominar a governança do Actions em três níveis (enterprise, organização e repositório): políticas de ações permitidas, permissões padrão do GITHUB_TOKEN, runners padrão, larger runners e auto-hospedados, grupos de runners, secrets e variáveis por nível e a REST API de permissões. A prova traz um cenário de empresa com uma restrição concreta e pede a configuração exata, a sintaxe de uma lista de padrões, o comando do config.sh ou o endpoint certo. Muita cobrança é de número: limites de secrets e variáveis, retenção, cache e prazos de runner. Onde mais erro: confundir larger runner com runner padrão (IP estático, VNET e imagem personalizada só em larger), achar que a organização vence a enterprise e esquecer que, em secrets e variáveis, o nível mais específico vence.

Neste capítulo

18 tópicos-chave · 14 pegadinhas · 8 macetes · 14 gatilhos · 4 questões comentadas

Tópicos-chave

Política de ações permitidas

Na organização ou na enterprise, define se roda qualquer ação, só ações da organização ou ações selecionadas. Na opção de ações selecionadas combinam-se: Allow actions created by GitHub (organizações actions e github), ações de verified creators do Marketplace e uma lista de padrões permitidos ou bloqueados. Permitir só ações da organização bloqueia até actions/checkout.

Sintaxe da lista de padrões

Ações usam OWNER/REPOSITORY@TAG-OU-SHA; reusable workflows usam OWNER/REPOSITORY/.github/workflows/ARQUIVO@REF, com o caminho antes do @ em uma única entrada. Curinga *, vírgula entre padrões, org/* libera a organização inteira e o prefixo ! bloqueia (ex.: *, !fibra-legado/deploy@*). Sem o *, só o que está listado é permitido.

Ações locais e a política

A política controla ações e reusable workflows referenciados por repositório. Uma ação local chamada com uses: ./.github/actions/validar roda mesmo fora da lista, e não usa referência por SHA.

Require actions to be pinned to a full-length commit SHA

Exige SHA completo em toda ação, seja de terceiros, do GitHub ou da própria organização; SHA abreviado e tag não atendem. Reusable workflows continuam podendo ser referenciados por tag.

Hierarquia de políticas

Política imposta pela enterprise limita as opções da organização; sem política da enterprise, owners da organização têm controle total. O que a organização desliga (por exemplo, as políticas de fork privado) um repositório não consegue religar.

Workflow permissions (GITHUB_TOKEN padrão)

A organização escolhe entre o padrão permissivo (leitura e escrita) e o restrito (leitura em contents e packages). Organizações novas e enterprises criadas desde 2/2/2023 vêm com o restrito. A opção Allow GitHub Actions to create and approve pull requests vem desligada por padrão, e os workflows ainda ajustam o token com a chave permissions.

Políticas de fork em repositórios privados

Opções: Run workflows from fork pull requests, Send write tokens, Send secrets to workflows from pull requests e Require approval. Com Run workflows ligado e Send write tokens desligado, o token é só leitura; Send write tokens dá escrita ao token, mas secrets dependem de Send secrets. Em repositório público, quem contribui pela primeira vez via fork depende, por padrão, de aprovação de alguém com escrita.

Runners hospedados padrão

Imagens mantidas pelo GitHub e atualizadas toda semana (anúncios no repositório actions/runner-images); para versão fixa de ferramenta, use ações setup-* como actions/setup-node. Software extra: apt-get no Ubuntu (depois de sudo apt-get update), brew no macOS e choco no Windows. As faixas de IP ficam em GET /meta, chave actions, mas não são indicadas como allowlist.

Larger runners

Mais CPU, RAM e disco, grupos, autoscaling e opções de GPU, nos planos Team e Enterprise Cloud. São sempre cobrados por minuto, sem usar os minutos incluídos, e só pelo tempo em que workflows rodam. IP estático, imagens personalizadas (imagens de VM) e rede privada do Azure valem só em Linux e Windows.

Runner auto-hospedado

Você cuida do sistema operacional e dos softwares; só o aplicativo do runner se atualiza sozinho. É gratuito no Actions (o custo da máquina é seu). Comunica-se com o GitHub só por HTTPS de saída na porta 443, sem porta de entrada; para ações Docker e service containers, precisa ser Linux com Docker.

Runners efêmeros, JIT e ARC

./config.sh --ephemeral faz o runner receber um único job, padrão recomendado para autoscaling. Runners just-in-time são criados pelo endpoint generate-jitconfig e iniciados com ./run.sh --jitconfig. Em Kubernetes, a recomendação é o Actions Runner Controller com runner scale sets via Helm; autoscaling próprio reage ao webhook workflow_job (queued, in_progress, completed).

--disableupdate

Desliga a atualização automática do runner, útil quando a versão vem na própria imagem do contêiner. Em troca, é obrigatório atualizar em até 30 dias após uma nova versão, senão o runner para de receber jobs.

Grupos de runners

Criados na organização ou enterprise, aceitam larger runners e auto-hospedados, nunca runners padrão. Toda organização tem o grupo default; cada runner fica em um grupo por vez; --runnergroup no config.sh registra no grupo e falha se ele não existir. Por padrão o grupo só atende repositórios privados; públicos exigem liberação.

runs-on com rótulos e grupo

Um array de rótulos exige um runner com todos eles (E lógico); group + labels exige runner do grupo que também tenha os rótulos. Rótulos não diferenciam maiúsculas e não são validados contra o hardware; vão no registro com --labels ou depois pela interface ou REST API. Sem runner compatível, o job espera até 24 horas antes de falhar.

Secrets por nível

Secret de repositório: quem tem escrita cria. Secret de ambiente: admin do repositório. Secret de organização: owners, com acesso para todos, só privados ou lista; no GitHub Free não chega a repositórios privados, e gh secret set --org sem --visibility vale só para privados. Precedência: ambiente vence repositório, que vence organização.

Variáveis de configuração

Lidas pelo contexto vars e exibidas sem máscara nos logs, então nada sensível nelas. Mesma regra de precedência dos secrets (o nível mais baixo vence). Pela REST API: POST cria e PATCH atualiza, com valor em texto claro.

Secrets pela REST API

Fluxo obrigatório: obter a public key, cifrar com libsodium (sealed box) e enviar encrypted_value em um PUT que cria ou atualiza. Secrets de ambiente ficam em /repos/{owner}/{repo}/environments/{environment_name}/secrets.

Retenção de artefatos, logs e cache

Artefatos e logs: padrão de 90 dias, configurável de 1 a 90 em repositório público e de 1 a 400 em privado, valendo só para objetos novos; por API, actions/permissions/artifact-and-log-retention. Cache: padrão de 7 dias e 10 GB por repositório, configurável até 365 dias e 10.000 GB, sem passar do limite definido no nível acima.

Macetes para memorizar

Limites de secrets e variáveis

Secret é mil-cem-cem; variável é mil-quinhentos-cem. Os dois: 48 KB cada.

Precedência

Quem está mais perto do job vence: ambiente > repositório > organização.

Retenção

Artefato e log: 90 padrão, público 90, privado 400. Cache: 7 dias e 10 GB, sobe até 365 e 10.000.

Prazos de runner

14 dias some o persistente, 1 dia o efêmero, 30 dias para atualizar com --disableupdate, 24 horas na fila sem runner.

Larger runner e rede

Larger Linux e Windows têm rede (IP estático, VNET, imagem própria); macOS não.

Status do runner

Idle = pronto; Active = trabalhando; Offline = desconectado.

Lista de padrões

* libera tudo, org/* libera a organização, ! bloqueia, vírgula separa.

Hierarquia

Enterprise impõe, organização obedece, repositório não religa o que a organização desligou.

Tabela de memorização: não confunda

Quando é...	...e quando é
Larger runner Quer máquina hospedada pelo GitHub com mais recursos, GPU, IP estático, imagem de VM própria ou acesso privado à VNET do Azure, aceitando cobrança por minuto.	**Runner auto-hospedado** Precisa de controle total de hardware, rede e software, inclusive em Kubernetes com ARC; o Actions não cobra, mas a manutenção do sistema é sua.
--ephemeral Cada runner deve receber um único job e ser desregistrado em seguida (autoscaling).	**--disableupdate** A versão do runner é controlada na imagem; exige atualizar em até 30 dias após nova versão.
Secrets Dado sensível: mascarado nos logs, limites de 1.000 na organização, 100 por repositório e 100 por ambiente, enviado cifrado pela API.	**Variáveis de configuração** Configuração não sensível: sem máscara, 500 por repositório, POST e PATCH em texto claro.
Allow actions created by GitHub Liberar de uma vez as ações das organizações actions e github, como actions/checkout e github/codeql-action.	**Allow actions by verified creators** Liberar ações do Marketplace cujo criador foi verificado pelo GitHub, sem listar uma a uma.
ACTIONS_RUNNER_HOOK_JOB_STARTED / COMPLETED Rodar script antes ou depois de cada job no runner auto-hospedado sem tocar nos workflows (variável no sistema ou no .env do runner).	**Step de limpeza no workflow** Quando alterar cada workflow é aceitável; não atende cenários que proíbem mexer nos arquivos.
_diag/Runner_* Log do aplicativo do runner, criado a cada inicialização.	**_diag/Worker_*** Log detalhado da execução de cada job na máquina.
Webhook workflow_job Escalar runners reagindo a jobs que entram na fila, começam e terminam.	**Evento workflow_run** Encadear um workflow depois de outro; não informa a fila de cada job.

Números que caem

Item	Valor
Secrets: quantidade	**1.000 na organização, 100 por repositório, 100 por ambiente**
Tamanho de secret e de variável	**48 KB cada**
Variáveis: quantidade	**1.000 na organização, 500 por repositório, 100 por ambiente; 256 KB somando organização e repositório por execução**
Repositório com acesso a mais de 100 secrets de organização	**Usa os 100 primeiros em ordem alfabética**

Retenção de artefatos e logs	**Padrão 90 dias; 1 a 90 em público; 1 a 400 em privado**
Cache por repositório	**Padrão 7 dias e 10 GB; até 365 dias e 10.000 GB**
Remoção de runner sem conexão	**14 dias (persistente); 1 dia (efêmero)**
Runner com --disableupdate	**Atualizar em até 30 dias após nova versão**
Job sem runner compatível	**Até 24 horas na fila antes de falhar; runner que não pega o job em 60 segundos o devolve à fila**
Reusable workflows encadeados	**Até 10 níveis e 50 reusable workflows únicos; workflow_run: 3 níveis**
Rede do runner auto-hospedado	**HTTPS de saída na porta 443, com pelo menos 70 kbps**
GITHUB_TOKEN só leitura por padrão	**Enterprises criadas desde 2/2/2023 e organizações novas**

Alertas de prova (pegadinhas)

⚠ **Alerta de prova:** Achar que larger runners consomem os minutos incluídos do plano em repositório privado.
Como não cair: Larger runners são sempre cobrados por minuto, sem cota incluída; e não há custo por runner criado e parado.

⚠ **Alerta de prova:** Marcar que IP estático também vale para larger runners macOS.
Como não cair: IP estático, imagens personalizadas e rede privada do Azure são só para larger runners Linux e Windows.

⚠ **Alerta de prova:** Supor que a variável ou o secret da organização prevalece sobre o do repositório.
Como não cair: Vence o nível mais específico: ambiente, depois repositório, depois organização. Nomes repetidos entre níveis não dão erro.

⚠ **Alerta de prova:** Usar 500 por repositório e 256 KB como limites de secrets.
Como não cair: Esses números são de variáveis. Secrets: 1.000 na organização, 100 por repositório, 100 por ambiente e 48 KB cada.

⚠ **Alerta de prova:** Abrir a porta 443 de entrada para o GitHub enviar jobs ao runner auto-hospedado.
Como não cair: O runner busca jobs por conexão HTTPS de saída na porta 443; não precisa de porta de entrada, webhook nem túnel SSH.

⚠ **Alerta de prova:** Com IP allow list ligada, trocar o GITHUB_TOKEN por PAT para escapar da restrição.
Como não cair: A allow list vale para web, API e Git, inclusive PATs, apps e GITHUB_TOKEN; a solução é incluir os IPs dos runners auto-hospedados.

⚠ **Alerta de prova:** Com runners de repositório desativados, usar o runner registrado em outro repositório.
Como não cair: Runners de repositório não são compartilhados; use runner de organização ou enterprise liberado ao repositório. Os runners de repositório existentes ficam Disabled, sem exclusão automática.

⚠ **Alerta de prova:** Responder que o máximo de retenção em repositório privado é 90 dias.
Como não cair: 90 é o padrão e o teto para públicos; privados vão até 400 dias, e retention-days por artefato não passa do teto configurado.

⚠ **Alerta de prova:** Achar que um runner desligado fica Offline para sempre.
Como não cair: Runner persistente sem conexão é removido após 14 dias; efêmero, após 1 dia. O critério é conexão, não jobs recebidos.

⚠ **Alerta de prova:** Incluir reusable workflows na exigência de SHA completo, ou poupar as ações do GitHub dela.
Como não cair: A exigência pega ações de qualquer origem (GitHub, organização, terceiros) e não pega reusable workflows.

⚠ **Alerta de prova:** Colocar runners padrão ubuntu-latest em um grupo de runners.
Como não cair: Grupos só aceitam larger runners e auto-hospedados; ao desabilitar runners padrão, os jobs passam a mirar grupos.

⚠ **Alerta de prova:** Aceitar nomes de secret como GITHUB_DEPLOY_KEY, 1PASSWORD_TOKEN ou DEPLOY-TOKEN-PROD.
Como não cair: Só [a-zA-Z0-9_], sem prefixo GITHUB_, sem começar com número e sem hífen ou espaço.

⚠ **Alerta de prova:** Para secret de 120 KB, dividir em vários secrets ou guardar como variável.
Como não cair: A documentação indica cifrar com gpg, versionar o .gpg e guardar a passphrase como secret; o conteúdo decifrado não é mascarado, então não o imprima.

⚠ **Alerta de prova:** Achar que só owners da organização criam secrets de ambiente, ou que escrita basta.
Como não cair: Secret de ambiente exige admin do repositório; escrita cria apenas secret de repositório.

Como cai na prova

📝 **Como cai:** Cenário pede para escrever a entrada da política de ações selecionadas.
Como resolver: Identifique se é ação (OWNER/REPO@REF) ou reusable workflow (OWNER/REPO/.github/workflows/ARQUIVO@REF); para liberar tudo menos algo, comece com * e use ! no padrão bloqueado; descarte ponto e vírgula como separador (o certo é vírgula), hífen no lugar de ! e @ antes do nome da organização.

📝 **Como cai:** Mesmo secret ou variável definido em vários níveis: qual valor o job recebe?
Como resolver: Veja se o job declara environment; se sim, vale o do ambiente; senão, o do repositório; a organização só vale quando não há outro. Nomes repetidos nunca geram erro.

📝 **Como cai:** Escolha DUAS sobre larger runners, grupos ou rótulos.
Como resolver: Julgue cada alternativa sozinha contra as regras: cobrança por minuto, rede só Linux/Windows, grupo default automático, um grupo por runner, rótulos sem distinção de maiúsculas e sem validação de hardware, --labels no registro e, em runner já existente, rótulo pela interface ou REST API.

📝 **Como cai:** Script precisa de um endpoint da REST API.
Como resolver: Ache o recurso no caminho: permissions/selected-actions (lista permitida), permissions/workflow (token padrão e criação de PR), permissions/repositories (onde o Actions está habilitado), permissions/artifact-and-log-retention, environments/{nome}/secrets. Caminhos como retention-settings, secret-list ou policies/allowed-actions-list não existem.

📝 **Como cai:** Runner auto-hospedado com problema de rede, proxy ou status.
Como resolver: Lembre: só saída HTTPS 443; teste com config.sh --check; proxy exige reiniciar o runner; Offline significa aplicativo sem conexão; logs em _diag (Runner_ do aplicativo, Worker_ do job).

Gatilhos do enunciado

Se o enunciado trouxer...	Pense em
Kubernetes e runners que escalam conforme a quantidade de jobs	**Actions Runner Controller com runner scale sets via Helm**
Runner que processa um único job e é desregistrado	**./config.sh --ephemeral, ou runner JIT pelo endpoint generate-jitconfig**
Limpar o runner antes de cada job sem alterar os workflows	**ACTIONS_RUNNER_HOOK_JOB_STARTED com o caminho do script**
IP allow list ativada e runners auto-hospedados falhando	**Adicionar os IPs dos runners à allow list**
Runner hospedado acessando banco em VNET do Azure sem expor portas	**Rede privada do Azure com larger runners**
Allowlist estável no firewall para runners do GitHub	**Larger runner com IP estático, não as faixas de GET /meta**
Versão da ferramenta mudou de uma semana para outra no ubuntu-latest	**Imagens atualizadas semanalmente; fixar a versão com setup-***
Testar se o runner alcança todos os serviços do GitHub	**./config.sh --check --url URL --pat TOKEN, detalhes na pasta _diag**
Mudou https_proxy e no_proxy e o runner usa o proxy antigo	**Reiniciar o aplicativo do runner (no_proxy não aceita IP ou CIDR)**
Política só da organização e actions/checkout parou	**Ações do GitHub também foram bloqueadas; usar ações selecionadas com Allow actions created by GitHub**
Workflow com pull-requests: write não consegue abrir PR com GITHUB_TOKEN	**Opção Allow GitHub Actions to create and approve pull requests, desligada por padrão**
Repositório público não enxerga secret de organização criado pelo gh	**gh secret set --org sem --visibility vale só para privados**
Custom image de VM com ferramentas pré-instaladas em runner hospedado	**Só com larger runners**
Reusable workflow chamado por 30 repositórios: quem paga e de quem são os runners	**Do caller: reusable usa os runners e a cobrança do repositório que chama**

🎯 Se você lembrar de só uma coisa deste capítulo...

- Ação local (./) passa por qualquer política; a exigência de SHA completo não atinge reusable workflows.
- Grupos de runners só têm larger runners e auto-hospedados, e repositórios públicos ficam de fora por padrão.
- Em reusable workflow, quem chama paga e empresta os runners; caller público só acessa reusable público.
- Variáveis nunca são mascaradas; secret pela API é public key + libsodium + encrypted_value em PUT.
- Contêiner (ação Docker ou service container) em runner auto-hospedado exige Linux com Docker.

Essenciais para a prova (checklist)

- ☐ Sei escrever a lista de ações selecionadas com *, org/*, ! e o caminho completo de reusable workflow.
- ☐ Sei o padrão do GITHUB_TOKEN em organizações novas e onde liberar a criação de PRs pelo Actions.
- ☐ Sei os limites de quantidade e tamanho de secrets e de variáveis e a ordem de precedência.
- ☐ Sei diferenciar larger runner, runner padrão e auto-hospedado em custo, rede e manutenção.
- ☐ Sei o efeito de --ephemeral, --disableupdate, --runnergroup, --labels e --check no config.sh.
- ☐ Sei os prazos de 14 dias, 1 dia, 30 dias e 24 horas ligados a runners.
- ☐ Sei configurar retenção de artefatos, logs e cache e seus limites por visibilidade.
- ☐ Sei quem cria secrets de repositório, de ambiente e de organização e o efeito do plano Free.
- ☐ Sei como rótulos e grupo em runs-on selecionam o runner.
- ☐ Sei os endpoints de permissions, secrets de ambiente e variáveis da REST API.

Questões de revisão do capítulo

R5.1 · Gerenciar o GitHub Actions na empresa (enterprise)

Um administrador da Contábil Ativo criou um secret de organização com gh secret set --org contabil API_KEY, sem outras opções, e repositórios públicos não enxergam o secret. Por quê?

A. Repositórios públicos exigem a opção --env no comando
B. Sem --visibility, ele vale só para privados
C. gh secret set cria somente secrets de repositório
D. O secret só passa a valer depois de rodar gh secret sync

R5.2 · Gerenciar o GitHub Actions na empresa (enterprise)

A equipe do Restaurante Sabor Caseiro quer limpar diretórios temporários no runner auto-hospedado antes de cada job, sem alterar os arquivos de workflow. Como configurar?

A. Incluir um step de limpeza no começo de cada workflow do repo
B. Usar runs.pre em uma ação local registrada no próprio runner
C. Definir ACTIONS_RUNNER_HOOK_JOB_STARTED com o caminho do script
D. Criar um service container que execute a limpeza antes de cada job

R5.3 · Gerenciar o GitHub Actions na empresa (enterprise)

A Telefonia Voz Clara quer permitir todas as ações das organizações octo-tools e octo-ci na lista da política de ações selecionadas. Como escrever a lista?

A. octo-tools/@*, octo-ci/@*
B. octo-tools; octo-ci
C. @octo-tools, @octo-ci
D. octo-tools/*, octo-ci/*

R5.4 · Gerenciar o GitHub Actions na empresa (enterprise)

A Biotech Célula Viva está planejando seus secrets do GitHub Actions. Quais limites de quantidade e tamanho valem?

A. 1.000 na organização, 100 por repositório e 100 por ambiente; 48 KB
B. 500 por repositório e 100 por ambiente, somando até 256 KB
C. Quantidade sem limite, desde que cada secret tenha até 48 KB
D. 100 na organização, 1.000 por repositório e 10 por ambiente; 64 KB

Respostas comentadas

R5.1. B) Sem --visibility, ele vale só para privados

Por quê: Para gerenciar secrets de organização pelo CLI, autentique também com o escopo admin:org.

✗ A) Incorreta. --env cria secrets de ambiente, não amplia o acesso.
✗ C) Incorreta. Com --org, o comando cria secret de organização.
✗ D) Incorreta. Não existe etapa de sincronização.

> **Dica de prova:** gh secret set --org: padrão só privados.

R5.2. C) Definir ACTIONS_RUNNER_HOOK_JOB_STARTED com o caminho do script

Por quê: Scripts de pré e pós-job são disparados por ACTIONS_RUNNER_HOOK_JOB_STARTED e ACTIONS_RUNNER_HOOK_JOB_COMPLETED, definidas no sistema ou no arquivo .env do runner; código de saída diferente de 0 no pré-job falha o job.

✗ A) Incorreta. Isso exigiria alterar os workflows, o que o cenário não permite.
✗ B) Incorreta. Não existe registro de ação local no runner; runs.pre é recurso do metadata de ações JavaScript.
✗ D) Incorreta. Service containers servem a jobs que os declaram no workflow.

> **Dica de prova:** Limpeza automática no runner auto-hospedado: hooks de job started/completed.

R5.3. D) octo-tools/*, octo-ci/*

Por quê: Na lista, use OWNER/REPOSITORY@TAG-OU-SHA para ações e OWNER/REPOSITORY/PATH/FILENAME@REF para reusable workflows, com * como curinga e vírgula entre padrões.

✗ A) Incorreta. A sintaxe da lista segue OWNER/REPOSITORY@REF; @* sem repositório não é o formato documentado.
✗ B) Incorreta. O separador é vírgula e é preciso indicar os repositórios com /*.
✗ C) Incorreta. Não se usa @ antes do nome da organização.

> **Dica de prova:** Liberar organização inteira: org/*.

R5.4. A) 1.000 na organização, 100 por repositório e 100 por ambiente; 48 KB

Por quê: Se o repositório tiver acesso a mais de 100 secrets de organização, o workflow usa só os 100 primeiros em ordem alfabética; segredos maiores que 48 KB pedem o contorno com gpg.

✗ B) Incorreta. 500 por repositório e 256 KB são limites de variáveis de configuração, não de secrets.

✗ C) Incorreta. Há limites de quantidade por nível.

✗ D) Incorreta. Os números estão trocados e o tamanho máximo é 48 KB.

> **Dica de prova:** Secrets: 1.000/100/100 e 48 KB. Variáveis: 1.000/500/100.

DIA DA PROVA

Estratégia de prova

O GH-200 é uma prova supervisionada de 100 minutos, com questões de múltipla escolha e múltipla resposta e possíveis componentes interativos; a aprovação é com 700 de 1000, e não há acesso ao Microsoft Learn durante exames GitHub. Quase todo enunciado é um cenário de empresa com um sintoma concreto (workflow que não roda, valor vazio, job pulado, runner offline, ação rejeitada) e pede a causa, a chave YAML, o comando ou o endpoint exato. As alternativas erradas costumam ser recursos reais que fazem outra coisa (GITHUB_ENV no lugar de GITHUB_OUTPUT, wait timer no lugar de reviewers, cache no lugar de artefato) ou números trocados entre limites parecidos. Os pesos oficiais são 20-25% para criar e gerenciar fluxos e para enterprise, 15-20% para consumo e solução de problemas e para ações, e 10-15% para automação segura. A maioria cobre recursos em disponibilidade geral, então a resposta não depende de recurso em preview.

Como responder

✓ Em Escolha DUAS ou TRÊS, julgue cada alternativa como verdadeira ou falsa isoladamente antes de marcar, sem tentar formar um par que pareça combinar.

✓ Sublinhe a restrição do enunciado (sem alterar os workflows, sem PAT, sem criar secrets, menor custo, sem publicar): ela costuma eliminar duas alternativas sozinha.

✓ Diante de YAML, identifique primeiro o nível (workflow, job ou step) e o tipo de runner ou de ação; muitas regras só valem em um nível.

✓ Em matriz, escreva a conta: produto das listas, menos exclude parcial, mais include que não encaixa; confira o teto de 256.

✓ Desconfie de sintaxe inventada: types: [merged], ::add-path::, runs-on: ubuntu-latest@20, operador in, ===, cache.yml, marketplace.yml, @latest.

✓ Em endpoint REST, ache o recurso no caminho (permissions, secrets, variables, caches, runs, workflows, environments) e depois o verbo; secrets usam PUT cifrado, variáveis POST e PATCH.

✓ Use o nome atual dos recursos: starter workflows do guia oficial são os workflow templates; para versionar e proteger ações, pense em SHA completo e immutable releases.

✓ Marque para revisão as questões de número e volte a elas no fim, comparando com os limites que você memorizou.

Erros que derrubam candidatos

✗ Confundir os limites de secrets (1.000/100/100) com os de variáveis (1.000/500/100) e retenção pública (90) com privada (400).

✗ Achar que a reexecução usa o commit mais recente ou os privilégios de quem reexecutou.

✗ Usar always() quando o pedido exclui cancelamento, ou esquecer failure() em condição pós-falha.

✗ Escolher ação Docker para cenário que exige Windows ou macOS.

✗ Achar que o reusable workflow usa o próprio repositório para checkout, runners e cobrança, ou que recebe o env do caller.

✗ Colocar ${{ }} com texto do usuário direto em run, ou fazer checkout do head em pull_request_target.

✗ Inverter a ordem de padrões com ! em branches ou usar branches e branches-ignore juntos.

Na semana da prova

- ☐ Revisar as tabelas de números: limites de secrets e variáveis, retenção, cache, prazos de runner, GITHUB_TOKEN, reexecução e matriz.
- ☐ Refazer as provas práticas cronometrando 100 minutos e anotar cada erro pela chave confundida, não pelo tema.
- ☐ Escrever de memória um workflow com gatilhos filtrados, matriz com include e exclude, outputs entre jobs e um reusable workflow com inputs, secrets e outputs.
- ☐ Escrever de memória um action.yml de cada tipo (JavaScript, Docker e composite) e um Dockerfile de ação correto.
- ☐ Repassar o mapa de segurança: script injection, pull_request_target, SHA completo, OIDC, permissions e regras de ambiente.
- ☐ Na véspera, revisar só as pegadinhas e os macetes de cada domínio e conferir os requisitos da prova supervisionada.

REVISÃO FINAL

Folha de cola

Leia na véspera e na manhã da prova: os gatilhos e as pegadinhas mais importantes de cada domínio.

1. Automação segura e otimizada

‣ github.event.issue.title ou pull_request.body direto em run → **Script injection: ação com input ou env intermediário**

‣ pull_request_target + checkout de pull_request.head.sha + make test → **Pwn request: código do fork com token e secrets do base**

‣ Ação de terceiros referenciada por tag → **Tag pode ser movida ou excluída; SHA completo do repositório original**

‣ Chaves AWS ou Azure guardadas como secret e rotação manual → **OIDC com id-token: write e credencial curta por job**

‣ Só o reusable deploy.yml aprovado pode assumir o papel na nuvem → **job_workflow_ref no sub do token OIDC**

⚠ A expressão continua inserida no script; use ação com input ou env intermediário.

⚠ A injeção acontece antes da validação; texto curto também pode conter comandos.

⚠ Em pull_request_target o token tem leitura e escrita no base e há secrets: é pwn request.

2. Consumir e solucionar problemas de fluxos de trabalho

‣ Uma combinação da matriz falhou por instabilidade → **Reexecutar só aquele job**

‣ Job skipped sem explicação → **system.txt do job no zip de logs; ou falha propagada por needs**

‣ Busca no log não encontra mensagem impressa → **O step estava recolhido; expanda antes de buscar**

‣ Quais ferramentas estavam no runner hospedado ontem → **Set up job > Runner Image > Included Software**

‣ Matriz com output de mesmo nome e valores inconsistentes → **Prevalece o último job a terminar; use nome único por combinação**

⚠ Jobs de matriz podem ser reexecutados individualmente; é o menor custo.

⚠ Usa o mesmo GITHUB_SHA e GITHUB_REF do evento original.

⚠ Os privilégios são de quem disparou a execução original.

3. Criar e gerenciar fluxos de trabalho

‣ Rodar sempre que outro workflow terminar, com sucesso ou falha → **on: workflow_run com workflows: [Nome] e types: [completed]**

‣ Job só quando o PR for mesclado → **pull_request types: [closed] + if github.event.pull_request.merged == true**

‣ issue_comment só quando o comentário é em pull request → **if: ${{ github.event.issue.pull_request }}**

‣ Saída longa recolhida em seção expansível no log → **::group::título e ::endgroup::**

‣ Tabela Markdown na página de resumo da execução → **Gravar no arquivo GITHUB_STEP_SUMMARY**

⚠ É rejeitado; use só branches com padrões ! para excluir.

⚠ A ordem importa: positivo depois do negativo inclui de novo. O ! vem depois do padrão positivo.

⚠ Não existe merged; use types: [closed] com if: github.event.pull_request.merged == true.

4. Criar e manter ações

‣ Ação que precisa rodar em Ubuntu, Windows e macOS → **JavaScript ou composite, nunca Docker**

‣ Versão específica de sistema e ferramentas independente do runner → **Ação de contêiner Docker (mais lenta, só Linux)**

‣ Mesmos quatro steps repetidos em vários workflows no mesmo runner → **Composite action**

‣ Token obtido em tempo de execução pode aparecer no log → **core.setSecret (add-mask) logo depois de obter**

‣ Input antigo será removido e quem usa deve ser avisado → **deprecationMessage no input**

⚠ Composite não cria INPUT_; use ${{ inputs.ambiente }}.

⚠ Falta value com ${{ steps.ID.outputs.NOME }}.

⚠ shell é obrigatório em cada step run; não há defaults no metadata.

5. Gerenciar o GitHub Actions na empresa (enterprise)

‣ Kubernetes e runners que escalam conforme a quantidade de jobs → **Actions Runner Controller com runner scale sets via Helm**

‣ Runner que processa um único job e é desregistrado → **./config.sh --ephemeral, ou runner JIT pelo endpoint generate-jitconfig**

‣ Limpar o runner antes de cada job sem alterar os workflows → **ACTIONS_RUNNER_HOOK_JOB_STARTED com o caminho do script**

‣ IP allow list ativada e runners auto-hospedados falhando → **Adicionar os IPs dos runners à allow list**

‣ Runner hospedado acessando banco em VNET do Azure sem expor portas → **Rede privada do Azure com larger runners**

⚠ Larger runners são sempre cobrados por minuto, sem cota incluída; e não há custo por runner criado e parado.

⚠ IP estático, imagens personalizadas e rede privada do Azure são só para larger runners Linux e Windows.

⚠ Vence o nível mais específico: ambiente, depois repositório, depois organização. Nomes repetidos entre níveis não dão erro.

GABARITO

Respostas do diagnóstico

D1. C) No Free, Pro e Team, isso só vale em repositórios públicos → revise o capítulo 1

Por quê: Várias regras de proteção de ambiente têm restrição de plano em repositórios privados; no GitHub Enterprise elas ficam disponíveis também para privados.

✗ A) Incorreta. As regras são independentes.

✗ B) Incorreta. Seis é o máximo de revisores, não o mínimo.

✗ D) Incorreta. Revisores precisam ter pelo menos leitura; não precisam ser owners.

> **Dica de prova:** Revisores em ambiente privado: precisa de plano Enterprise.

D2. D) O selo é um bom sinal, mas o código da ação ainda deve ser auditado → revise o capítulo 1

Por quê: O verified creator indica que a ação foi escrita por uma equipe cuja identidade o GitHub verificou; mesmo assim, audite o código para garantir que secrets e conteúdo do repositório são tratados como esperado.

✗ A) Incorreta. Tags podem ser movidas ou excluídas mesmo em ações de criadores verificados.

✗ B) Incorreta. O GitHub não audita o código de cada versão das ações publicadas.

✗ C) Incorreta. O selo trata da verificação do criador, não de testes de segurança da ação.

> **Dica de prova:** Verified creator reduz o risco, mas não substitui a auditoria do código.

D3. C) Depois que um revisor aprova e o job é enviado ao runner → revise o capítulo 1

Por quê: Todas as regras de proteção do ambiente precisam passar antes de o job ir para um runner; só então os secrets do ambiente ficam acessíveis.

✗ A) Incorreta. Antes da aprovação, o job nem é enviado ao runner.

✗ B) Incorreta. Não há restrição por forma de disparo.

✗ D) Incorreta. Secrets de ambiente são lidos quando o job que referencia o ambiente começa.

> **Dica de prova:** Secret de ambiente protegido: só após aprovação.

D4. A) Um arquivo de workflow inválido em .github/workflows → revise o capítulo 2

Por quê: Faça commit apenas de arquivos de workflow válidos; corrija a sintaxe para parar as execuções com falha.

✗ B) Incorreta. 256 jobs é o limite da matriz, não explica falha a cada commit.

✗ C) Incorreta. Sem gatilho, o workflow simplesmente não rodaria.

✗ D) Incorreta. Workflow desabilitado não é disparado.

> **Dica de prova:** Falha em todo commit: YAML de workflow inválido.

D5. B) NODE_ENV=test e CI_FLAG=1, as mesmas definidas no lint → revise o capítulo 2

Por quê: Ao analisar um workflow com âncoras, expanda mentalmente cada alias: o resultado é o mesmo que escrever o bloco por extenso.

✗ A) Incorreta. O nome da âncora não cria variáveis novas.

✗ C) Incorreta. Âncoras e aliases funcionam em qualquer ponto do mesmo arquivo de workflow.

✗ D) Incorreta. O alias repete o mapa inteiro.

> **Dica de prova:** Alias = cópia integral do bloco ancorado.

D6. C) Marcos foi quem alterou por último a sintaxe cron do arquivo → revise o capítulo 2

Por quê: O actor de workflows agendados pode mudar com eventos do repositório, e as notificações seguem quem alterou o cron por último.

✗ A) Incorreta. O último push no branch não define o destinatário.
✗ B) Incorreta. Aprovar pull request não é o critério.
✗ D) Incorreta. Ser dono do repositório não define o destinatário das notificações de agendamento.

> **Dica de prova:** Notificação de schedule: quem mexeu no cron por último.

D7. A) Quem grava em GITHUB_ENV não vê o valor; só os próximos steps → revise o capítulo 3

Por quê: GITHUB_ENV define variáveis para os steps seguintes do job; para usar no próprio step, use a variável do shell diretamente.

✗ B) Incorreta. Basta gravar no arquivo de ambiente GITHUB_ENV; não é preciso outro comando.
✗ C) Incorreta. O valor fica disponível para os steps seguintes do mesmo job, não para outro job.
✗ D) Incorreta. Não é preciso pré-declarar a variável.

> **Dica de prova:** GITHUB_ENV vale do próximo step em diante.

D8. C) on: workflow_run com workflows: [Build] e types: [completed] → revise o capítulo 3

Por quê: workflow_run permite encadear workflows (até três níveis) e filtrar por nome e branch; o resultado do workflow anterior fica em github.event.workflow_run.conclusion.

✗ A) Incorreta. check_suite não dispara para suítes criadas pelo GitHub Actions, para evitar recursão.
✗ B) Incorreta. workflow_call define um reusable workflow, chamado por outro, não reage ao fim de um workflow.
✗ D) Incorreta. needs cria dependência entre jobs do mesmo workflow.

> **Dica de prova:** Rodar depois de outro workflow: workflow_run + completed.

D9. D) Envolver a expressão em ${{ }}, já que ! é reservado no YAML → revise o capítulo 3

Por quê: No if é possível omitir ${{ }}, exceto quando a expressão começa com !, que o YAML interpreta como notação reservada.

✗ A) Incorreta. A troca de função não resolve; o problema é o ! no início do valor YAML.
✗ B) Incorreta. github.ref é o ref completo, e o prefixo refs/tags/ está correto na comparação.
✗ C) Incorreta. A mesma regra vale para if de job e de step.

> **Dica de prova:** if começando com ! exige ${{ }} (ou aspas/parênteses).

D10. B) Falta aceitar o Developer Agreement; publicar ainda exige 2FA → revise o capítulo 4

Por quê: Em organizações, um owner precisa aceitar o acordo; depois, a publicação é feita em Draft a release, com categoria e tag de versão.

✗ A) Incorreta. Não há mínimo de estrelas.
✗ C) Incorreta. Ações que atendem aos requisitos são publicadas sem revisão.
✗ D) Incorreta. O repositório precisa ser público.

> **Dica de prova:** Marketplace bloqueado: acordo de desenvolvedor e 2FA.

D11. A) Chamar core.setSecret, equivalente a add-mask, com o token → revise o capítulo 4

Por quê: Mascare qualquer valor sensível que não seja secret do GitHub com add-mask (core.setSecret) antes de usá-lo; o registro vale para o job.

✗ B) Incorreta. exportVariable grava em GITHUB_ENV e não mascara o valor.
✗ C) Incorreta. Com depuração ligada, o token apareceria no log.
✗ D) Incorreta. setOutput não mascara o valor; um valor mascarado nem pode virar output.

> **Dica de prova:** Segredo gerado em tempo de execução: core.setSecret.

D12. B) Manter v2 como tag Git sem release e movê-la a cada versão nova → revise o capítulo 4

Por quê: Immutable releases travam tag e assets da release; para versões que precisam se mover (major e minor), use tags Git que não estejam ligadas a uma release.

✗ A) Incorreta. A tag de uma release imutável fica travada e não pode ser movida.
✗ C) Incorreta. Depois de excluir uma release imutável, o mesmo nome de tag não pode ser reutilizado.
✗ D) Incorreta. Um branch protegido não resolve a referência de versão major por tag.

> **Dica de prova:** Imutável: release com tag fixa; major móvel sem release.

Agora, treine no formato real

Faça as 6 provas completas do curso GH-200: 300 questões inéditas, tempo do exame, explicação de cada alternativa e link da documentação para revisar.

Ver as 6 provas completas e o cupom do mês →

Até o dia da prova, uma revisão por dia no seu feed

Instagram @strecht.simulados
Um post de prova por dia: questão comentada, pegadinhas e o "não confunda" de cada certificação.
Seguir no Instagram →

YouTube Strecht Simulados
Shorts diários com pegadinhas de prova e, aos sábados, um simulado comentado.
Inscrever-se no canal →

Material independente de preparação, sem afiliação ou endosso de GitHub. As questões são inéditas e não reproduzem a prova real.