

Guia de preparação GH-900

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.

7
capítulos por domínio

40
questões comentadas

70%
nota de corte

6
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-900: 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. Colaboração no GitHub	10–15%	18 conceitos · 14 pegadinhas
2. Comunidade do GitHub	5–10%	14 conceitos · 12 pegadinhas
3. Fundamentos de Git e GitHub	25–30%	18 conceitos · 14 pegadinhas
4. Gerenciamento de Projetos no GitHub	5–10%	16 conceitos · 12 pegadinhas
5. Privacidade, Segurança e Administração	10–15%	16 conceitos · 14 pegadinhas
6. Práticas Modernas de Desenvolvimento	10–15%	17 conceitos · 14 pegadinhas
7. Repositórios do GitHub	10–15%	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 · Colaboração no GitHub

Onde colocar um modelo para que todo novo PR já venha com uma lista de verificação na descrição?

A. Em .github/workflows/pr-template.yml
B. Em docs/README.md
C. Em .github/pull_request_template.md
D. Em .gitignore na raiz

D2 · Colaboração no GitHub

Qual opção de merge junta todos os commits do PR em um único commit no branch de destino?

A. Create a merge commit
B. Revert pull request
C. Squash and merge
D. Rebase and merge

D3 · Colaboração no GitHub

Um revisor quer propor a correção exata de uma linha para que o autor aplique com um clique. O que usar?

A. Sugestão de alteração
B. Nova issue sem referência
C. Gist separado com o trecho
D. Novo fork do repositório

D4 · Comunidade do GitHub

O que caracteriza um software de código aberto?

A. Código gratuito e sem nenhuma licença
B. Código disponível sob licença aberta
C. Código publicado só em repositório privado
D. Código mantido por uma única empresa

D5 · Comunidade do GitHub

Qual é um benefício de segurança frequentemente associado ao código aberto?

A. O código fica oculto de atacantes
B. Os bugs desaparecem sozinhos
C. Mais pessoas podem revisar o código
D. Os testes deixam de ser necessários

D6 · Comunidade do GitHub

Por que contribuir com projetos de código aberto pode ajudar a carreira de um desenvolvedor?

A. Impede críticas ao código
B. Garante emprego imediato
C. Cria um histórico público de trabalho
D. Dispensa estudar novas tecnologias

D7 · Fundamentos de Git e GitHub

Qual é a sequência típica do GitHub Flow para entregar uma mudança?

A. PR, exclusão do branch e commits
B. Commits na main e depois um branch
C. Branch, commits, PR, revisão e merge
D. Merge, PR e só então o branch

D8 · Fundamentos de Git e GitHub

Qual comando volta para o branch main já existente?

A. git switch main
B. git remote main
C. git tag main
D. git branch -d main

D9 · Fundamentos de Git e GitHub

Uma empresa precisa hospedar a plataforma GitHub em sua própria infraestrutura, por exigências de conformidade. Qual opção atende?

A. GitHub Desktop para Windows
B. GitHub Enterprise Server
C. GitHub Free para pessoas
D. GitHub Mobile para Android

D10 · Gerenciamento de Projetos no GitHub

Qual layout do GitHub Projects funciona como planilha, permitindo agrupar, ordenar e filtrar itens por campos?

A. Table
B. Roadmap
C. Board
D. Pulse

D11 · Gerenciamento de Projetos no GitHub

Uma equipe quer que um item do projeto passe para Done automaticamente quando a issue for fechada. O que usar?

A. Uma regra no .gitignore
B. Workflows internos do Projects
C. Um arquivo CODEOWNERS na raiz
D. Um Gist com o status atual

D12 · Gerenciamento de Projetos no GitHub

Uma equipe quer que toda issue nova com o rótulo bug entre automaticamente no projeto. O que usar?

A. Workflow de adição automática
B. Uma tag do Git
C. Um Gist com a lista
D. Uma regra de .gitignore

01

CAPÍTULO 1 · 10–15% DA PROVA

Colaboração no GitHub

Neste domínio eu preciso conhecer o ciclo do pull request (base e compare, rascunho, revisão, opções de merge), o ciclo da issue (templates, fechamento, transferência, sub-issues, tipos) e os canais de conversa (Discussions, Wiki, Gists, Pages). A prova descreve uma situação de colaboração e oferece recursos reais que resolvem outra coisa. As opções de merge são o ponto mais cobrado: squash junta, rebase alinha sem commit de merge e merge commit preserva tudo com um commit extra. Onde mais erro: confundir # com @, base com compare e esquecer que a palavra-chave só fecha a issue no merge no branch padrão.

Neste capítulo
18 tópicos-chave · 14 pegadinhas · 6 macetes · 14 gatilhos · 4 questões comentadas

Tópicos-chave

Base e compare

No PR, o branch base recebe as mudanças e o compare as fornece. O PR também pode comparar branches de forks.

Pull request em rascunho

Mostra que o trabalho está em andamento; PRs em rascunho não podem ser mesclados até ficarem prontos para revisão. Uma issue com rótulo não bloqueia merge de PR.

Revisão de PR

Comente uma linha no diff; agrupe vários comentários com Iniciar uma revisão e conclua com Comment, Approve ou Request changes. Request changes indica problemas a resolver antes do merge e, com proteção de branch, pode bloqueá-lo. Comment comenta sem aprovar nem bloquear; Dismiss review descarta uma revisão.

Sugestão de alteração

O revisor propõe a correção exata de uma linha e o autor aplica com um clique; sugestões podem ser aplicadas em lote como um commit.

Resolver conversa e pedir revisor

Quando um ponto da revisão foi atendido, resolva a conversa; proteções podem exigir todas as conversas resolvidas. Para pedir revisão de alguém específico, adicione a pessoa como revisora no PR; o CODEOWNERS também pede revisores automaticamente.

Opções de merge

Create a merge commit preserva todos os commits e registra a integração com um commit extra (mostra bem quando o PR entrou). Squash and merge junta tudo em um único commit (histórico limpo). Rebase and merge aplica cada commit sem criar commit de merge. O mantenedor escolhe quais opções ficam disponíveis; Revert pull request desfaz um PR.

Checks no PR

Um X vermelho em verificação obrigatória indica que o check falhou; os logs do check ajudam a entender a falha. Não significa branch apagado, PR aprovado ou repositório arquivado.

Vincular PR a issue

Palavras-chave como close, fix e resolve com #número (ex.: Fixes #42) fecham a issue automaticamente quando o PR é mesclado no branch padrão. A seção Development vincula manualmente e também fecha no merge. A issue é fechada, não apagada.

Referências e menções

#12 referencia issue ou PR do mesmo repositório; org/api#12 referencia outro repositório. Um hash de commit em comentário vira link para o commit. @nome notifica uma pessoa e @org/equipe notifica uma equipe inteira (quando visível para quem escreve).

Gestão de issues

Feche como concluída (resolvida) ou não planejada (ideia descartada); o motivo ajuda em filtros e relatórios. Transferir move a issue para outro repositório da mesma organização mantendo comentários. Fixar destaca no topo. Bloquear a conversa impede comentários de quem não é colaborador. Reações substituem comentários +1 e permitem ordenar por reações. Issues não são arquivadas.

Sub-issues, tipos de issue e permalink

Sub-issues dividem uma issue grande em partes acompanhadas e o progresso aparece na principal. Tipos de issue da organização (Bug, Feature, Task) padronizam a classificação em todos os repositórios e convivem com rótulos. O link permanente de uma linha aponta para um commit específico e não muda, ideal para abrir issue a partir do código.

Templates de issue e PR

Templates de issue ficam em .github/ISSUE_TEMPLATE; formulários de issue em YAML permitem campos obrigatórios (versão, passos para reproduzir). O config.yml dessa pasta impede issues em branco e indica links externos. O modelo de PR fica em .github/pull_request_template.md; vários modelos em .github/PULL_REQUEST_TEMPLATE/.

Notificações

Participando e menções reduz o volume; Watch personalizado permite observar só releases; Subscribe na issue acompanha sem participar e Unsubscribe interrompe. Na caixa de notificações: marcar como concluída, salvar para depois ou cancelar a inscrição. Estrela, fork e seguir não controlam notificações de issue.

GitHub Discussions

Espaço para perguntas abertas, ideias e anúncios, separado das issues: issues são itens de trabalho; discussões são conversas sem fim definido. Issue sem tarefa pode virar discussão e discussão pode gerar issue. Na categoria Q&A, o autor marca a resposta como aceita; discussões podem ser fixadas.

Wiki

Documentação longa com várias páginas editáveis ligada ao repositório. Wikis são repositórios Git próprios, editáveis pelo navegador.

Gists

Compartilham trechos de código ou arquivos de configuração com histórico de versões, sem criar repositório. Podem ser públicos ou secretos: secretos não aparecem em buscas, mas quem tem o link vê. Podem ser incorporados em sites, clonados e ter forks.

GitHub Pages

Publica site estático, como documentação, a partir do repositório. Fontes de publicação: branch, pasta ou workflow do Actions; aceita domínio personalizado e HTTPS.

Filtros de busca

is:issue is:open label:bug lista issues abertas com o rótulo bug. Outros filtros: author:, assignee: e is:pr. Sintaxes como type=issue ou tag:bug não são do GitHub.

Macetes para memorizar

💡 Opções de merge

Squash = espremer em um; Rebase = enfileirar sem nó; Merge commit = guardar tudo e dar um nó.

💡 Base e compare

A base é a casa que recebe; o compare é quem traz a mudança.

💡 Símbolos

@ chama gente; # aponta item; org/repo#N aponta item de outra casa.

💡 Issue x Discussion

Issue tem tarefa; Discussion tem conversa.

💡 Templates

Issue: pasta .github/ISSUE_TEMPLATE (+ config.yml para barrar em branco); PR: .github/pull_request_template.md.

💡 Notificações

Menos barulho = Participando e menções; só versões = Watch personalizado em releases; uma issue = Subscribe.

Tabela de memorização: não confunda

Quando é...	...e quando é
Squash and merge Juntar todos os commits do PR em um único commit no destino.	**Rebase and merge** Aplicar cada commit do PR no destino, sem criar commit de merge.
Create a merge commit Preservar todos os commits e registrar a integração com um commit extra.	**Squash and merge** Histórico mais limpo, com um commit só por PR.
Issues Item de trabalho com tarefa definida.	**Discussions** Perguntas abertas, ideias e anúncios, sem fim definido.
Request changes O revisor achou problemas que precisam ser resolvidos antes do merge.	**Comment** Comentar sem aprovar nem bloquear.
Fechar como concluída A issue foi resolvida.	**Fechar como não planejada** A ideia foi descartada e a issue não será feita.
@menção Notificar pessoa (@nome) ou equipe (@org/equipe).	**#referência** Referenciar issue ou PR (#12 ou org/api#12).
Gist Compartilhar rapidamente um trecho ou arquivo, com histórico, sem criar repositório.	**Wiki** Documentação longa em várias páginas ligada a um repositório.

Números que caem

Item	Valor
Peso do domínio na prova	**10–15% (referência de 7 itens em uma prova de 50)**
Opções de merge de PR	**Três: Create a merge commit, Squash and merge e Rebase and merge**
Pareceres ao concluir uma revisão	**Três: Comment, Approve e Request changes**

Alertas de prova (pegadinhas)

⚠ **Alerta de prova:** Escolher Rebase and merge para juntar todos os commits em um só.
Como não cair: Rebase and merge aplica cada commit; quem junta é Squash and merge.

⚠ **Alerta de prova:** Escolher Squash and merge para preservar todos os commits.
Como não cair: Squash junta tudo. Preservar e registrar a integração com commit extra = Create a merge commit.

⚠ **Alerta de prova:** Achar que Rebase and merge cria commit de merge.
Como não cair: Rebase and merge não cria commit de merge; quem cria é Create a merge commit.

⚠ **Alerta de prova:** Inverter base e compare.
Como não cair: Base recebe; compare fornece. As mudanças vão do compare para a base.

⚠ **Alerta de prova:** Usar #nome para avisar uma pessoa ou #seguranca para avisar uma equipe.
Como não cair: # referencia issues e PRs. Pessoa = @nome; equipe = @org/equipe. Rótulo ou negrito não notificam.

⚠ **Alerta de prova:** Referenciar issue de outro repositório com @org/api/12 ou #org-api-12.
Como não cair: O formato é org/api#12; @ é para pessoas e equipes.

⚠ **Alerta de prova:** Fechar como concluída uma issue cuja ideia foi descartada.
Como não cair: Concluída indica resolvida; descartada = não planejada.

⚠ **Alerta de prova:** Achar que Fixes #42 apaga a issue ou que fecha ao mesclar em qualquer branch.
Como não cair: Fecha (não apaga) quando o PR é mesclado no branch padrão.

⚠ **Alerta de prova:** Arquivar uma issue para destacá-la ou bloquear o repositório para chamar atenção.
Como não cair: Issues não são arquivadas e bloquear o repositório impede o trabalho. Destaque no topo = fixar.

⚠ **Alerta de prova:** Copiar e colar o texto ou criar fork para mover uma issue de repositório.
Como não cair: Copiar perde o histórico e fork não move issues. Use transferir a issue.

⚠ **Alerta de prova:** Usar um rótulo bug para indicar PR em andamento que não deve ser mesclado.
Como não cair: Rótulo em issue não bloqueia merge. Use PR em rascunho.

⚠ **Alerta de prova:** Usar regra de proteção de branch ou .gitignore para exigir campos obrigatórios em issues.
Como não cair: Proteção é de branches. Campos obrigatórios = formulário de issue em YAML; impedir issues em branco = config.yml da pasta de modelos.

⚠ **Alerta de prova:** Achar que estrela, fork ou seguir o dono controlam notificações.
Como não cair: Notificações se ajustam com Watch (incluindo personalizado, só releases), Subscribe e a opção Participando e menções.

⚠ **Alerta de prova:** Achar que Gist secreto é privado ou que Gists são só para download.
Como não cair: Secreto só não aparece em buscas; quem tem o link vê. Gists podem ser incorporados em sites, clonados e ter forks.

Como cai na prova

📝 **Como cai:** Pergunta sobre opção de merge descrita pelo efeito no histórico.
Como resolver: Identifique as duas pistas: quantos commits chegam ao destino (um ou todos) e se há commit de merge. Um = squash; todos sem commit de merge = rebase; todos com commit extra = merge commit.

📝 **Como cai:** Cenário de revisão de PR (comentar linha, sugerir correção, pedir ajustes, encerrar ponto).
Como resolver: Mapeie a ação: linha = comentar no diff; correção pronta = sugestão; vários comentários = iniciar revisão; problemas = Request changes; ponto atendido = resolver conversa.

📝 **Como cai:** Sintaxe de referência ou menção com quatro formatos parecidos.
Como resolver: Pessoa ou equipe começa com @; issue ou PR usa #; de outro repositório, org/repo#N. Descarte $, ! e hifens.

📝 **Como cai:** Escolha do canal certo para um conteúdo (pergunta aberta, documentação longa, trecho de código, site).
Como resolver: Conversa aberta = Discussions; páginas de documentação = Wiki; trecho com histórico sem repositório = Gist; site estático = Pages.

📝 **Como cai:** Ação de manutenção em issue (descartar, mover, destacar, conter discussão).
Como resolver: Descartada = fechar como não planejada; repositório errado = transferir; destaque = fixar; ânimos exaltados = bloquear conversa. Alternativas desproporcionais (apagar repositório) caem fora.

Gatilhos do enunciado

Se o enunciado trouxer...	Pense em
um commit só no branch de destino	**Squash and merge**
cada commit no destino, sem commit de merge	**Rebase and merge**
preservar commits e registrar a integração com commit extra	**Create a merge commit**
propor a correção exata de uma linha para aplicar com um clique	**Sugestão de alteração**
vários comentários enviados juntos com parecer final	**Iniciar uma revisão**
PR ainda em andamento que não deve ser mesclado	**Pull request em rascunho**
Fixes #N, close, resolve na descrição	**Fecha a issue no merge no branch padrão**
vincular manualmente PR a issue	**Seção Development**
dividir issue grande em partes ligadas à principal	**Sub-issues**
classificar issues de todos os repositórios como Bug, Feature ou Task	**Tipos de issue da organização**
issue a partir de uma linha específica do código	**Link permanente (permalink) da linha**
muitos +1 em uma issue popular	**Reações**
discussão acalorada e fora do tema	**Bloquear a conversa**
site estático da documentação a partir do repositório	**GitHub Pages (branch, pasta ou Actions)**

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

- Squash junta em um commit; Rebase and merge não cria commit de merge; merge commit preserva tudo com commit extra.
- Palavra-chave de fechamento só fecha a issue quando o PR é mesclado no branch padrão.
- @ notifica pessoas e equipes; # referencia issues e PRs.
- PR em rascunho não pode ser mesclado até ficar pronto para revisão.

Essenciais para a prova (checklist)

- ☐ Sei diferenciar as três opções de merge pelo efeito no histórico.
- ☐ Sei o papel dos branches base e compare.
- ☐ Sei usar comentário no diff, sugestão de alteração, iniciar revisão, Request changes e resolver conversa.
- ☐ Sei vincular PR a issue por palavra-chave e pela seção Development.
- ☐ Sei referenciar issues de outro repositório e mencionar pessoas e equipes.
- ☐ Sei onde ficam os templates de issue e PR e para que serve o config.yml.
- ☐ Sei fechar, transferir, fixar e bloquear issues e quando usar sub-issues e tipos de issue.
- ☐ Sei configurar notificações (Participando e menções, Watch personalizado, Subscribe).
- ☐ Sei quando usar Discussions, Wiki, Gists e GitHub Pages.

Questões de revisão do capítulo

R1.1 · Colaboração no GitHub

Uma issue foi aberta no repositório errado da mesma organização. Como movê-la mantendo comentários?

A. Criar um fork
B. Copiar e colar o texto
C. Criar uma tag
D. Transferir a issue

R1.2 · Colaboração no GitHub

Uma pessoa quer ser avisada só quando um repositório publicar novas releases, sem receber avisos de issues e PRs. Como configurar?

A. Seguir o dono do repositório
B. Fazer fork do repositório
C. Observar só releases
D. Dar estrela no repositório

R1.3 · Colaboração no GitHub

Como referenciar em um comentário a issue 12 de outro repositório, chamado org/api?

A. #org-api-12
B. @org/api/12
C. !org/api:12
D. org/api#12

R1.4 · Colaboração no GitHub

Uma comunidade quer um espaço para perguntas abertas, ideias e anúncios, separado das issues de trabalho. O que habilitar?

A. GitHub Actions
B. GitHub Discussions
C. GitHub Codespaces
D. GitHub Packages

Respostas comentadas

R1.1. D) Transferir a issue

Por quê: A transferência mantém comentários e responsáveis quando possível.

✗ A) Incorreta. Não move issues.
✗ B) Incorreta. Perde o histórico.
✗ C) Incorreta. Não move issues.

> **Dica de prova:** Issue no lugar errado: transferir.

R1.2. C) Observar só releases

Por quê: Observar tudo, só participações ou tipos personalizados são opções de notificação.

✗ A) Incorreta. Mostra atividade no feed.

✗ B) Incorreta. Não gera notificações.

✗ D) Incorreta. Não gera notificações.

> **Dica de prova:** Só releases: observação personalizada.

R1.3. D) org/api#12

Por quê: Dentro do mesmo repositório, basta #12.

✗ A) Incorreta. Não é o formato.

✗ B) Incorreta. @ é para pessoas e equipes.

✗ C) Incorreta. Não é o formato.

> **Dica de prova:** Issue de outro repositório: org/repo#N.

R1.4. B) GitHub Discussions

Por quê: Issues são itens de trabalho; discussões são conversas sem um fim definido.

✗ A) Incorreta. Automatiza fluxos de trabalho.

✗ C) Incorreta. Oferece ambientes de desenvolvimento.

✗ D) Incorreta. Hospeda pacotes.

> **Dica de prova:** Conversas abertas: Discussions.

02

CAPÍTULO 2 · 5–10% DA PROVA

Comunidade do GitHub

Neste domínio preciso entender como funciona o código aberto no GitHub: licenças, papel dos mantenedores, como contribuir sem permissão de escrita e o que é InnerSource. A prova também cobra os recursos sociais e de descoberta (estrela, Watch, seguir, Explore, Trending, coleções, README de perfil) e programas como Sponsors, Marketplace, Skills e Education. Os enunciados são curtos e os distratores costumam ser outros produtos do GitHub com função diferente. Onde mais erro: estrela x Watch, licença permissiva x copyleft e achar que InnerSource publica o código na internet.

Neste capítulo

14 tópicos-chave · 12 pegadinhas · 5 macetes · 14 gatilhos · 4 questões comentadas

Tópicos-chave

Código aberto

Software com código disponível sob licença aberta; a licença define direitos e obrigações de quem reutiliza. Código sem licença não é aberto: sem licença, nada é permitido. Não depende de ser mantido por uma única empresa.

Licenças permissivas x copyleft

Permissivas, como MIT, não proíbem uso comercial. Copyleft, como GPL, exige que derivados usem a mesma licença; não torna o código privado. Ambas permitem modificar. Uma licença explícita esclarece o uso permitido, e o GitHub ajuda a escolher ao criar o repositório.

Mantenedores

Decidem o que entra no projeto de código aberto; arquivos como GOVERNANCE.md podem explicar as regras de decisão. Nem o primeiro a abrir issue, nem visitantes, nem o GitHub decidem pelo projeto.

Benefícios do código aberto

Mais pessoas podem revisar o código, embora transparência não substitua boas práticas de segurança. Para o desenvolvedor, contribuir cria um histórico público de trabalho e desenvolve colaboração e revisão de código. Para a empresa, contribuir permite influenciar e melhorar o que já usa e reduz o custo de manter mudanças próprias.

Como contribuir

Sem permissão de escrita, o caminho é fazer fork, alterar e abrir um PR, que os mantenedores revisam. Antes, leia o CONTRIBUTING.md e comece por issues marcadas para iniciantes. Antes de abrir issue de bug, busque issues parecidas; um bom relato traz passos para reproduzir, versão e comportamento esperado.

good first issue

Rótulo comum para indicar issues simples para quem está começando; o GitHub destaca issues com esse rótulo para novos contribuidores. needs triage indica item ainda não avaliado; duplicate indica duplicidade.

Sync fork

Atualiza pelo GitHub um fork quando o original recebeu novos commits. Pela linha de comando, usa-se um remoto upstream com fetch e merge. Apagar e recriar o fork é desnecessário.

InnerSource

Aplica práticas de código aberto dentro da empresa (forks, PRs, documentação aberta) sem publicar o código na internet. Benefícios: reuso de código entre equipes, melhor documentação e colaboração. Equipes contribuem com o projeto de outra por pull requests revisados, e a visibilidade interna permite que as equipes encontrem os projetos umas das outras.

GitHub Sponsors e FUNDING.yml

Sponsors permite apoiar financeiramente desenvolvedores e organizações que mantêm projetos de código aberto; patrocinadores escolhem valores e níveis. O arquivo .github/FUNDING.yml exibe o botão Sponsor no repositório e aceita GitHub Sponsors e outras plataformas.

GitHub Marketplace

Reúne actions e aplicativos prontos para integrar ao GitHub. Action é usada em workflows como step; GitHub App integra via API, pode reagir a eventos e tem permissões próprias.

Estrela, Watch e seguir

Estrela marca o repositório como favorito e mostra apreço, ajuda na descoberta e aparece na lista de favoritos do perfil, mas não gera notificações nem dá acesso. Watch (observar) gera notificações. Seguir perfis acompanha a atividade pública de desenvolvedores e organizações.

Descoberta de projetos

A página Explore e Trending mostram repositórios e desenvolvedores em alta; tópicos e coleções (listas temáticas selecionadas pelo GitHub, no Explore) também ajudam a descobrir projetos.

Perfil

O README que aparece no topo do perfil vem de um repositório público com exatamente o nome do usuário. O gráfico de contribuições mostra as contribuições do último ano; contribuições em repositórios privados podem aparecer de forma anônima.

GitHub Skills e GitHub Education

GitHub Skills oferece cursos práticos feitos dentro de repositórios, usando issues e Actions para guiar cada etapa. GitHub Education atende estudantes com ferramentas gratuitas e o Copilot sem custo; o Student Developer Pack reúne ofertas de parceiros.

Macetes para memorizar

💡 Estrela x Watch x seguir

Estrela = curtir; Watch = ser avisado; seguir = acompanhar pessoas.

💡 Copyleft

Copyleft contagia: derivado herda a mesma licença.

💡 InnerSource

Open source da porta para dentro: PRs entre equipes, visibilidade interna.

💡 Programas do GitHub

Sponsors = dinheiro; Skills = cursos em repositório; Education = estudante; Marketplace = loja de actions e apps.

💡 Primeira contribuição

Ler (CONTRIBUTING), buscar (duplicatas), começar pequeno (good first issue), fork e PR.

Tabela de memorização: não confunda

Quando é...	...e quando é
Estrela Marcar favorito e mostrar apreço, sem notificações.	**Watch** Receber notificações da atividade do repositório.
Licença permissiva (MIT) Reuso amplo, inclusive comercial, sem exigir a mesma licença nos derivados.	**Licença copyleft (GPL)** Derivados precisam ser distribuídos na mesma licença.
Action Usada como step em workflows.	**GitHub App** Integração via API, com eventos e permissões próprias.
InnerSource (visibilidade interna) Colaboração aberta entre equipes, com o código protegido dentro da empresa.	**Código aberto (visibilidade pública)** Código disponível a qualquer pessoa sob licença aberta.
GitHub Sponsors Apoiar financeiramente mantenedores e organizações.	**GitHub Education** Benefícios para estudantes, como ferramentas gratuitas e Copilot sem custo.

Números que caem

Item	Valor
Peso do domínio na prova	**5–10% (referência de 5 itens em uma prova de 50)**
Período do gráfico de contribuições do perfil	**Último ano**

Alertas de prova (pegadinhas)

⚠ **Alerta de prova:** Achar que dar estrela gera notificações ou cria um fork.
Como não cair: Estrela é favorito e apreço; notificações vêm do Watch e cópia é o fork.

⚠ **Alerta de prova:** Achar que Watch dá acesso de escrita ao código.
Como não cair: Watch só gera notificações; nem estrela nem Watch dão acesso.

⚠ **Alerta de prova:** Achar que código aberto é código gratuito sem nenhuma licença.
Como não cair: Sem licença nada é permitido. Código aberto = código disponível sob licença aberta.

⚠ **Alerta de prova:** Achar que a copyleft torna o código privado ou que a permissiva proíbe uso comercial.
Como não cair: Copyleft exige a mesma licença nos derivados; a permissiva não proíbe uso comercial; ambas permitem modificar.

⚠ **Alerta de prova:** Achar que InnerSource publica o código na internet ou isola as equipes.
Como não cair: O código fica interno e o objetivo é o oposto do isolamento: reuso e colaboração entre equipes.

⚠ **Alerta de prova:** Escolher visibilidade pública para InnerSource em uma enterprise.
Como não cair: Pública expõe fora da empresa e privada esconde o projeto. Use interna.

⚠ **Alerta de prova:** Contribuir sem permissão pedindo a senha do mantenedor ou anexando código em issue.
Como não cair: O fluxo padrão é fork, alteração e PR.

⚠ **Alerta de prova:** Enviar de imediato um PR grande ou mencionar todos os mantenedores na primeira contribuição.
Como não cair: Leia o CONTRIBUTING.md e comece por issues para iniciantes; menções em massa geram ruído.

⚠ **Alerta de prova:** Usar CODEOWNERS ou .gitattributes para exibir o botão Sponsor.
Como não cair: O botão vem do .github/FUNDING.yml.

⚠ **Alerta de prova:** Criar o README de perfil em um Gist ou na Wiki.
Como não cair: Precisa ser um repositório público com exatamente o nome do usuário.

⚠ **Alerta de prova:** Achar que action e GitHub App são iguais ou que a diferença é preço.
Como não cair: Action roda em workflows; App integra via API.

⚠ **Alerta de prova:** Procurar projetos em alta no log de auditoria ou em Settings.
Como não cair: Descoberta é em Explore, Trending, tópicos e coleções.

Como cai na prova

📝 **Como cai:** Qual programa ou produto do GitHub atende ao cenário (apoio financeiro, cursos, estudantes, integrações).
Como resolver: Identifique o público e o objetivo: dinheiro para mantenedores = Sponsors; aprender = Skills; estudante = Education; actions e apps = Marketplace. Pages, Gists e Mobile são distratores frequentes.

📝 **Como cai:** Diferença entre recursos sociais (estrela, Watch, seguir, fork).
Como resolver: Pergunte o efeito: notificação = Watch; favorito = estrela; atividade de pessoas = seguir; cópia = fork. Nenhum deles dá acesso de escrita.

📝 **Como cai:** Conceito de código aberto ou licença.
Como resolver: Código aberto exige licença aberta; copyleft exige a mesma licença nos derivados; sem licença nada é permitido. Descarte afirmações absolutas como 'proíbe modificações' ou 'garante emprego'.

📝 **Como cai:** InnerSource (definição, benefício, visibilidade, forma de contribuir).
Como resolver: Lembre: interno, aberto entre equipes, PRs revisados, visibilidade interna, reuso de código. Qualquer alternativa que publique na internet ou isole equipes está errada.

Gatilhos do enunciado

Se o enunciado trouxer...	Pense em
contribuir sem permissão de escrita	**Fork, alterar e abrir um PR**
fork desatualizado em relação ao original	**Sync fork**
apoiar financeiramente mantenedores	**GitHub Sponsors**
botão Sponsor no repositório	**.github/FUNDING.yml**
práticas de código aberto dentro da empresa	**InnerSource**
equipes encontram projetos umas das outras na enterprise	**Visibilidade interna**
issues simples para iniciantes	**Rótulo good first issue**
actions e aplicativos prontos	**GitHub Marketplace**
repositórios e desenvolvedores em alta	**Explore e Trending**
listas temáticas selecionadas pelo GitHub	**Coleções do Explore**
README no topo do perfil	**Repositório público com o nome do usuário**
cursos práticos dentro de repositórios	**GitHub Skills**
estudante quer ferramentas gratuitas e Copilot sem custo	**GitHub Education**
quem decide o que entra no projeto	**Mantenedores**

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

- Estrela não gera notificação; Watch gera.
- Sem licença, nada é permitido; copyleft exige a mesma licença nos derivados.
- InnerSource é dentro da empresa, com visibilidade interna e PRs revisados.
- Contribuir sem permissão de escrita = fork e PR.

Essenciais para a prova (checklist)

- ☐ Sei o que caracteriza código aberto e por que uma licença explícita importa.
- ☐ Sei diferenciar licença permissiva de copyleft.
- ☐ Sei o fluxo de contribuição por fork e PR e as boas práticas antes da primeira contribuição.
- ☐ Sei atualizar um fork pelo GitHub e pela linha de comando.
- ☐ Sei definir InnerSource, seus benefícios e a visibilidade que o favorece.
- ☐ Sei diferenciar estrela, Watch e seguir.
- ☐ Sei para que servem Sponsors, FUNDING.yml, Marketplace, Skills e Education.
- ☐ Sei criar o README de perfil e onde descobrir projetos (Explore, Trending, tópicos, coleções).

Questões de revisão do capítulo

R2.1 · Comunidade do GitHub

Como acompanhar a atividade pública de desenvolvedores e organizações de interesse no GitHub?

A. Seguindo os perfis
B. Clonando os perfis
C. Criando forks dos perfis
D. Abrindo issues nos perfis

R2.2 · Comunidade do GitHub

Onde descobrir repositórios e desenvolvedores em alta na comunidade do GitHub?

A. Na página Explore e em Trending
B. No log de auditoria
C. Na aba Settings do repositório
D. No arquivo .gitignore

R2.3 · Comunidade do GitHub

Qual é um benefício típico do InnerSource para uma empresa?

A. Fim das revisões de código
B. Isolamento total das equipes
C. Reuso de código entre equipes
D. Publicação do código na internet

R2.4 · Comunidade do GitHub

O repositório original recebeu novos commits depois que uma pessoa fez o fork. Como atualizar o fork pelo GitHub?

A. Usar a opção Sync fork
B. Criar uma release no fork
C. Abrir uma issue no original
D. Apagar e recriar o fork

Respostas comentadas

R2.1. A) Seguindo os perfis

Por quê: Estrelas marcam repositórios favoritos e ajudam na descoberta.

✗ B) Incorreta. Não é o mecanismo.
✗ C) Incorreta. Perfis não são forkados.
✗ D) Incorreta. Não é o mecanismo.

> **Dica de prova:** Acompanhar pessoas: seguir.

R2.2. A) Na página Explore e em Trending

Por quê: Tópicos e coleções também ajudam na descoberta.

✗ B) Incorreta. Registra ações administrativas.
✗ C) Incorreta. Configura o repositório.
✗ D) Incorreta. Lista arquivos ignorados.

> **Dica de prova:** Descobrir projetos: Explore e Trending.

R2.3. C) Reuso de código entre equipes

Por quê: InnerSource também melhora a documentação e a colaboração.

✗ A) Incorreta. Revisões continuam.
✗ B) Incorreta. É o oposto.
✗ D) Incorreta. O código fica interno.

> **Dica de prova:** InnerSource: reuso interno.

R2.4. A) Usar a opção Sync fork

Por quê: Pela linha de comando, usa-se um remoto upstream com fetch e merge.

✗ B) Incorreta. Não atualiza o fork.

✗ C) Incorreta. Não atualiza o fork.

✗ D) Incorreta. É desnecessário.

> **Dica de prova:** Atualizar fork: Sync fork.

03

CAPÍTULO 3 · 25–30% DA PROVA

Fundamentos de Git e GitHub

Este é o domínio mais pesado da prova (25–30%), então preciso dominar os comandos do Git do dia a dia e saber exatamente o que cada um faz e o que não faz. A prova dá um cenário curto (desfazer um commit, trazer mudanças do remoto, criar um branch, marcar uma versão) e coloca como alternativas erradas comandos reais que fazem outra coisa, ou opções que não existem para aquele comando. Também cai a diferença entre Git e GitHub, entre controle de versão distribuído e centralizado, os tipos de conta e produtos (Enterprise Cloud x Server, Desktop, Mobile, CLI) e sintaxe de Markdown. Onde mais erro: confundir fetch com pull, reset com revert, fork com clone e esquecer que tags não sobem no push comum.

Neste capítulo

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

Tópicos-chave

Controle de versão distribuído x centralizado

O Git é distribuído porque cada clone tem todo o histórico; Mercurial também é distribuído. Subversion (SVN) e CVS são centralizados: o histórico fica no servidor e o trabalho offline é limitado. Benefícios do controle de versão: voltar a uma versão anterior funcional, trabalhar em paralelo com branches, rastreabilidade e colaboração.

Git x GitHub

Git é a ferramenta que versiona o código e funciona localmente e offline; GitHub é a plataforma que hospeda repositórios Git e adiciona colaboração, automação e gestão. Commits, branches e merges são do Git; issues e pull requests são do GitHub. Git não depende do site do GitHub para funcionar.

Três áreas do Git

Diretório de trabalho, área de preparação (staging) e repositório. git add move a alteração para a preparação e git commit registra no repositório. No git status, Untracked files são arquivos que nunca foram adicionados; Changes to be committed são os já preparados; um arquivo removido aparece como deleted.

Commit

Cada commit é identificado por um hash do conteúdo (os primeiros caracteres costumam bastar nos comandos), registra autor, data e mensagem e aponta para o commit anterior. Senha, issues e estrelas não fazem parte do commit. Boa mensagem é clara e específica (ex.: Corrige validação de CPF no cadastro), não asdf nem Alterações diversas.

Configuração inicial

Antes do primeiro commit numa máquina nova: git config --global user.name e git config --global user.email. git config --list mostra todas as configurações em vigor; com --show-origin, mostra de qual arquivo vem cada valor.

git init x git clone

git init cria a pasta .git local (apagá-la remove o histórico local) e não cria nada no GitHub. git clone URL cria a cópia local completa e já configura o remoto origin; git clone -b develop URL clona já no branch develop, baixando também o histórico dos outros branches.

Remotos e upstream

git remote add origin URL liga um repositório criado com git init a um repositório vazio no GitHub; git remote -v lista os remotos e endereços. No primeiro push de um branch novo, git push -u origin feature define o upstream, e depois basta git push e git pull. Em um fork clonado, git remote add upstream URL-original e git fetch upstream trazem as novidades do projeto original.

fetch, pull e origin/main

git fetch só baixa e atualiza os branches remotos de acompanhamento, como origin/main (cópia local da main do remoto); a main local só muda depois do merge. git pull baixa e integra. git pull --rebase reaplica os commits locais sobre os remotos, deixando o histórico linear.

Branch

Internamente, um branch é um ponteiro móvel para um commit, por isso criar branches é rápido e barato. Criar: git branch nome ou git switch -c nome (cria e muda; git checkout -b faz o mesmo). Mudar: git switch main (ou git checkout main). Apagar local: git branch -d (só se integrado; -D força); renomear: -m; apagar remoto: git push origin --delete nome.

HEAD

HEAD normalmente representa o commit atual do branch. HEAD~1 é o pai do commit atual e HEAD~2 é dois commits antes. Em detached HEAD, ele aponta direto para um commit. Não confundir com origin (remoto padrão) nem com .gitignore.

Merge, fast-forward e conflito

Estando no branch de destino, git merge feature-x integra o outro branch. Se a main não recebeu commits novos, o Git só avança o ponteiro (fast-forward), sem commit de merge. Se duas pessoas alteraram a mesma linha, ocorre conflito: edita o arquivo, marca com git add e conclui o commit; git merge --abort desiste e volta ao estado anterior.

Rebase e cherry-pick

Rebase reaplica os commits sobre outro ponto e deixa o histórico linear (merge comum cria commit de merge com dois pais). Rebase e amend reescrevem o histórico e devem ser evitados em commits já compartilhados. git cherry-pick hash aplica um commit específico ao branch atual criando um novo commit com as mesmas alterações.

Desfazer alterações

git restore arquivo descarta mudanças não preparadas; git restore --staged tira da preparação. git reset --soft HEAD~1 desfaz o último commit local mantendo as alterações preparadas; --hard descarta. git revert hash desfaz criando um commit novo, a forma segura em branch compartilhado. git commit --amend corrige o último commit ainda não enviado. git reflog encontra o hash de um commit perdido localmente.

Comandos de inspeção

git status (branch atual, modificados e preparados), git log (--oneline um por linha, --graph desenho de branches, --all todos os branches, --author filtra autor, --grep filtra mensagem, --since espera data), git diff (não preparado), git diff --staged (preparado), git show hash (detalhes de um commit), git blame (quem alterou cada linha). status e log só leem.

Arquivos rastreados e .gitignore

git mv renomeia e já prepara; git rm remove do repositório e do diretório (fica no histórico); git rm --cached para de rastrear sem apagar o arquivo local. O .gitignore só vale para arquivos ainda não rastreados: ignore .env e pastas de dependências como node_modules/, mas versione README, LICENSE e .github/workflows/. git commit -a prepara só os rastreados modificados; arquivos novos ainda precisam de git add. git stash guarda mudanças temporárias e git stash pop recupera.

Tags

git tag v1.0 cria uma tag leve (só ponteiro); git tag -a v2.0 -m 'Versão 2.0' cria tag anotada, com mensagem e autor. O git push comum não envia tags: use git push origin v1.0 ou git push --tags. git tag -d apaga a tag local.

Autenticação e push forçado

Para operações Git com o GitHub, use token de acesso pessoal via HTTPS ou chave SSH (endereço de clonagem começa com git@); a senha da conta não é mais aceita no Git e o código 2FA não é credencial do Git. Gerenciadores de credenciais e o GitHub CLI simplificam o uso de tokens. git push --force em branch compartilhado pode apagar commits alheios; --force-with-lease reduz o risco.

Contas, produtos e ferramentas

Tipos de conta: pessoal, organização e enterprise (várias organizações com políticas e cobrança centralizadas). Organização gerencia acesso por equipes e papéis e os repositórios ficam com ela se o funcionário sai. Enterprise Cloud é hospedado pelo GitHub; Enterprise Server, pela empresa. GitHub Desktop é a interface gráfica; GitHub Mobile permite ler, comentar, revisar e mesclar pelo celular; GitHub CLI (gh) faz gh pr create, gh issue list e gh repo clone. GitHub Flow: branch, commits, PR, revisão e merge, com a main sempre pronta para implantar. Markdown: ## é título nível 2 (há seis níveis), [texto](url) é link, ! antes do link é imagem, tabela usa barras verticais e linha de hífens, três crases com o nome da linguagem destacam código e - [ ] é tarefa aberta.

Macetes para memorizar

💡 fetch x pull

fetch = espiar (só baixa); pull = puxar para dentro (baixa e integra).

💡 Desfazer

Local? reset. Compartilhado? revert. Perdi? reflog. Só a mensagem? amend.

💡 Fork x clone

Fork fica no GitHub (na nuvem); clone vem para casa (local).

💡 Reescrevem histórico

R e A reescrevem: Rebase e Amend. Merge, fetch e revert não.

💡 Três áreas

Trabalho → add → Preparação → commit → Repositório → push → GitHub.

💡 git switch

switch muda; switch -c cria e muda (c de create).

💡 Tags no push

Tag não pega carona: precisa de push próprio (origin v1.0 ou --tags).

💡 Markdown

título, > citação, - lista, ! imagem, [ ] tarefa aberta, [x] feita, três crases = código.

Tabela de memorização: não confunda

Quando é...	...e quando é

git fetch Quero só baixar e revisar o que chegou do remoto antes de integrar; atualiza origin/main, não a main local.	**git pull** Quero baixar e já integrar no branch atual (com --rebase, reaplica meus commits locais sobre os remotos).
Fork Quero uma cópia do repositório na minha conta do GitHub, por exemplo para contribuir sem permissão de escrita.	**Clone** Quero uma cópia local completa, com histórico, para trabalhar na máquina; é comum clonar o próprio fork.
git reset --soft HEAD~1 O commit está só no meu repositório local e quero desfazê-lo mantendo as alterações preparadas.	**git revert hash** O commit já foi enviado para um branch compartilhado: desfaz criando um commit novo, sem reescrever o histórico.
git reset --soft Desfazer o commit e manter as mudanças preparadas.	**git reset --hard** Desfazer e descartar as mudanças (se perdeu algo importante, git reflog ajuda a achar o hash).
Merge Integrar preservando a história; sem divergência vira fast-forward, com divergência cria commit de merge com dois pais. Não reescreve commits existentes.	**Rebase** Deixar o histórico linear reaplicando os commits; nunca em commits já compartilhados.
git diff Ver linhas alteradas que ainda não foram preparadas com git add.	**git diff --staged** Ver o que já foi preparado e ainda não foi comitado.
GitHub Enterprise Cloud A empresa quer o plano Enterprise hospedado pelo GitHub (com organizações, repositórios privados e PRs).	**GitHub Enterprise Server** A empresa precisa hospedar a plataforma na própria infraestrutura, por conformidade.

Números que caem

Item	Valor
Peso do domínio na prova	**25–30% (referência de 14 itens em uma prova de 50)**
HEAD~1 / HEAD~2	**Um commit antes do atual (pai) / dois commits antes (avô)**
Níveis de título em Markdown	**Seis, de # a ######**
Áreas do fluxo do Git	**Três: trabalho, preparação e repositório**

Alertas de prova (pegadinhas)

⚠ **Alerta de prova:** Achar que git fetch já atualiza a main local (ou que pull só baixa).
Como não cair: fetch só baixa e atualiza origin/main; a main local só muda após o merge. pull = fetch + integração.

⚠ **Alerta de prova:** Escolher git reset --hard com push forçado para desfazer um commit que já está na main compartilhada.
Como não cair: Isso reescreve o histórico compartilhado. Em branch compartilhado, a resposta é git revert, que cria um commit novo.

⚠ **Alerta de prova:** Marcar git revert --soft ou git push --soft para desfazer o último commit local.
Como não cair: --soft é opção do git reset. revert e push não têm essa opção.

⚠ **Alerta de prova:** Achar que git commit -a inclui arquivos novos.
Como não cair: -a prepara só arquivos rastreados modificados; os novos ainda precisam de git add. Assinar commit é -S, e -a não faz push.

⚠ **Alerta de prova:** Achar que colocar um .env já rastreado no .gitignore resolve.
Como não cair: O .gitignore só vale para arquivos não rastreados. Use git rm --cached .env; segredos já enviados devem ser trocados. git add --force faz o contrário: força o rastreamento.

⚠ **Alerta de prova:** Achar que a tag criada localmente sobe com o git push comum.
Como não cair: Tags vão com git push origin v1.0 ou git push --tags. git fetch baixa, não envia; git tag -d apaga.

⚠ **Alerta de prova:** Marcar 'Senha da conta via HTTPS' ou 'Código 2FA como senha' como forma de autenticar o Git.
Como não cair: A senha não é mais aceita no Git e o código 2FA não é credencial do Git. As duas válidas: token de acesso pessoal via HTTPS e chave SSH.

⚠ **Alerta de prova:** Inverter fork e clone (clone na conta do GitHub, fork local).
Como não cair: Fork = cópia no GitHub; clone = cópia local. Nenhum dos dois apaga o original nem exige ser dono.

⚠ **Alerta de prova:** Incluir Mercurial ou GitHub Desktop entre os sistemas centralizados.
Como não cair: Centralizados: SVN e CVS. Git e Mercurial são distribuídos; GitHub Desktop é só um cliente do Git.

⚠ **Alerta de prova:** Marcar commits, branches ou merges como recursos exclusivos do GitHub.
Como não cair: Esses são do Git. O que só existe no GitHub, entre as opções típicas, são issues e pull requests.

⚠ **Alerta de prova:** Achar que, sem commits novos na main, o merge gera conflito ou é recusado.
Como não cair: Sem divergência é fast-forward: o Git só move o ponteiro, sem commit de merge e sem conflito.

⚠ **Alerta de prova:** Achar que trabalhar em branches elimina conflitos de merge.
Como não cair: Branches isolam o trabalho e permitem revisar em PR, mas conflitos ainda podem ocorrer quando a mesma linha é alterada; o Git não escolhe sozinho.

⚠ **Alerta de prova:** Confundir git log --grep com --author para achar os commits de uma pessoa.
Como não cair: --author filtra por autor; --grep busca no texto da mensagem; --since espera uma data; --stat mostra estatísticas.

⚠ **Alerta de prova:** Confundir git branch -d (apaga) com -m (renomeia), ou achar que git branch -d apaga o branch remoto.
Como não cair: -d apaga só o local (e só se integrado; -D força); -m renomeia. Remoto: git push origin --delete nome.

Como cai na prova

📝 **Como cai:** Cenário de desfazer algo ('mantendo as alterações', 'já enviado', 'sem reescrever o histórico').
Como resolver: 1) O commit já foi compartilhado? Sim → git revert. 2) Não → reset (--soft mantém preparado, --hard descarta) ou --amend se é só corrigir o último. 3) Perdeu algo → reflog.

📝 **Como cai:** Comando certo com opções trocadas entre comandos (git log --staged, git remote --list, git push --soft).
Como resolver: Associe cada opção ao comando dono: --staged é do diff, --list é do config, --soft é do reset, -v lista remotos no git remote. Opção no comando errado é alternativa inválida.

📝 **Como cai:** Múltipla resposta 'Escolha DUAS' (autenticação, comandos que trazem do remoto, centralizados, recursos só do GitHub).
Como resolver: Marque exatamente duas. Elimine primeiro o que é de outra camada (Git x GitHub, envio x recebimento, leitura x escrita) e confira que as duas restantes respondem ao mesmo critério.

📝 **Como cai:** Pergunta conceitual 'o que é X internamente / o que representa' (branch, HEAD, origin/main).
Como resolver: Branch = ponteiro móvel; HEAD = commit atual do branch; origin/main = cópia local da main remota. Descarte 'cópia completa dos arquivos' ou 'repositório separado'.

📝 **Como cai:** Produto ou tipo de conta para um cenário de empresa.
Como resolver: Quem hospeda? Empresa → Enterprise Server; GitHub → Enterprise Cloud. Várias organizações → conta enterprise. Repositórios da empresa → organização, não conta pessoal.

📝 **Como cai:** Sintaxe de Markdown com quatro símbolos parecidos.
Como resolver: Lembre o papel de cada símbolo: # título, > citação, - lista, --- linha horizontal, ! imagem. Link é [texto](url), nessa ordem.

Gatilhos do enunciado

Se o enunciado trouxer...	Pense em
trocar de branch com urgência sem comitar o trabalho pela metade	**git stash (depois git stash pop)**
commit perdido depois de git reset --hard	**git reflog para achar o hash**
commit com erro já enviado à main compartilhada	**git revert com o hash**
aplicar só um commit de outro branch	**git cherry-pick**
quem alterou cada linha e em qual commit	**git blame (ou a visão Blame no GitHub)**
corrigir a mensagem do último commit antes do push	**git commit --amend**
desistir de um merge com conflitos	**git merge --abort**
primeiro push de branch novo e lembrar o remoto	**git push -u origin branch**
buscar novidades do projeto original a partir do fork pela CLI	**git remote add upstream URL-original e git fetch upstream**
hospedar a plataforma na própria infraestrutura	**GitHub Enterprise Server**
várias organizações com políticas e cobrança centralizadas	**Conta enterprise**
interface gráfica no computador, sem linha de comando	**GitHub Desktop**
aprovar PRs e ver notificações pelo celular	**GitHub Mobile**
criar PRs e ver issues pelo terminal	**GitHub CLI (gh)**

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

- Em branch compartilhado, desfazer é com git revert; reset --hard com push forçado reescreve o histórico de todos.
- O .gitignore não afeta arquivos já rastreados: use git rm --cached.
- Tags não sobem no git push comum.
- Git com GitHub autentica por token via HTTPS ou por chave SSH; a senha da conta não serve.
- Commits, branches e merges são do Git; issues e PRs são do GitHub.

Essenciais para a prova (checklist)

- ☐ Sei explicar a diferença entre fetch, pull e pull --rebase.
- ☐ Sei escolher entre reset --soft, reset --hard, revert, amend e reflog conforme o cenário.
- ☐ Sei criar, mudar, renomear e apagar branches local e remoto.
- ☐ Sei definir upstream com git push -u e configurar o remoto upstream de um fork.
- ☐ Sei o que é fast-forward, quando surge conflito e como abortar um merge.
- ☐ Sei criar tag leve e anotada e enviá-las ao GitHub.
- ☐ Sei a diferença entre Enterprise Cloud e Enterprise Server e entre conta pessoal, organização e enterprise.
- ☐ Sei identificar GitHub Desktop, GitHub Mobile e GitHub CLI pelos cenários.
- ☐ Sei escrever títulos, links, imagens, tabelas, blocos de código e listas de tarefas em Markdown.

Questões de revisão do capítulo

R3.1 · Fundamentos de Git e GitHub

O que faz a opção -a em git commit -a -m "mensagem"?

A. Envia o commit ao remoto em seguida
B. Prepara os arquivos rastreados modificados
C. Adiciona também os arquivos não rastreados
D. Assina o commit com a chave GPG

R3.2 · Fundamentos de Git e GitHub

Após um git fetch, o que representa origin/main?

A. Um novo branch criado no GitHub
B. A main local já atualizada
C. Um fork da organização
D. A cópia local da main do remoto

R3.3 · Fundamentos de Git e GitHub

Dois desenvolvedores alteraram a mesma linha de um arquivo em branches diferentes. O que acontece no merge?

A. Ocorre um conflito a ser resolvido
B. O repositório é bloqueado para sempre
C. O merge escolhe a alteração mais antiga
D. O Git apaga as duas alterações

R3.4 · Fundamentos de Git e GitHub

Qual comando mostra os detalhes e as alterações de um commit específico?

A. git clean a1b2c3
B. git show a1b2c3
C. git init a1b2c3
D. git stash a1b2c3

Respostas comentadas

R3.1. B) Prepara os arquivos rastreados modificados

Por quê: Arquivos novos ainda precisam de git add na primeira vez.

✗ A) Incorreta. Não faz push.
✗ C) Incorreta. Arquivos novos ficam de fora.
✗ D) Incorreta. Assinatura usa -S.

> **Dica de prova:** commit -a: rastreados modificados.

R3.2. D) A cópia local da main do remoto

Por quê: Branches remotos de acompanhamento são atualizados pelo fetch.

✗ A) Incorreta. Não é criado no GitHub.
✗ B) Incorreta. A main local só muda após merge.
✗ C) Incorreta. Não é um fork.

> **Dica de prova:** origin/main: referência remota local.

R3.3. A) Ocorre um conflito a ser resolvido

Por quê: Conflitos são resolvidos editando o arquivo, marcando com git add e concluindo o commit.

✗ B) Incorreta. Não é o comportamento.
✗ C) Incorreta. O Git não escolhe sozinho.
✗ D) Incorreta. Não é o comportamento.

> **Dica de prova:** Mesma linha alterada: conflito.

R3.4. B) git show a1b2c3

Por quê: git show também exibe tags e outros objetos.

✗ A) Incorreta. Remove arquivos.

✗ C) Incorreta. Cria repositório.

✗ D) Incorreta. Guarda mudanças.

> **Dica de prova:** Detalhes de um commit: git show.

04

CAPÍTULO 4 · 5–10% DA PROVA

Gerenciamento de Projetos no GitHub

É o domínio de menor peso (5–10%), mas tem pontos fáceis se eu souber os layouts e os tipos de campo do GitHub Projects. A prova descreve uma necessidade de planejamento (sprint, prioridade, estimativa, linha do tempo, Kanban) e espera o recurso exato do Projects; os distratores são recursos de código, como tags do Git, forks e Gists, que não organizam trabalho. Também cai milestone x rótulo, automações internas e itens de rascunho. Onde mais erro: confundir campo de iteração com campo de data e Table com Board.

Neste capítulo

16 tópicos-chave · 12 pegadinhas · 5 macetes · 14 gatilhos · 4 questões comentadas

Tópicos-chave

GitHub Projects da organização

Um projeto da organização acompanha issues de vários repositórios em um só lugar. Para aparecer na aba Projects de um repositório, vincule o projeto ao repositório; um projeto pode ser vinculado a vários repositórios.

Layout Table

Funciona como planilha e permite agrupar, ordenar e filtrar itens por campos. Exemplo: agrupar pela coluna Assignees para ver os itens por responsável; também dá para agrupar por Status, Iteration ou campos próprios.

Layout Board

Mostra os itens em colunas no estilo Kanban. As colunas costumam ser definidas por um campo de seleção, como Status, e arrastar o cartão entre colunas atualiza o valor do campo.

Layout Roadmap

Mostra os itens em uma linha do tempo com datas; usa campos de data ou de iteração.

Visões salvas

Guardam filtros, agrupamentos e ordenações e podem usar layouts Table, Board ou Roadmap. Um mesmo projeto pode ter várias visões, por exemplo uma só com o sprint atual e outra com todos os bugs.

Campos personalizados

Tipos: texto, número, data, seleção única e iteração. Rótulo e milestone não são tipos de campo personalizado.

Campo de iteração

Planeja o trabalho em ciclos de tempo, como sprints, e pode alimentar o Roadmap.

Campo numérico

Registra estimativas de esforço, como pontos; visões agrupadas podem mostrar a soma de campos numéricos. Campo de texto não soma.

Campo de seleção única e campo de data

Seleção única classifica com opções fixas, como prioridade Alta, Média ou Baixa, Status e Tamanho. Campo de data registra a data prevista de entrega de cada item e alimenta o Roadmap; campo de texto não valida datas.

Itens de rascunho

Registram ideias rápidas no projeto sem criar issue nem escolher repositório. Depois, o rascunho aprovado é convertido em issue e continua no projeto, agora ligado à issue.

Workflows internos do Projects

Automatizam o projeto: mudar o item para Done quando a issue é fechada, adicionar automaticamente itens novos que batem com filtros (rótulo bug, tipo) e arquivar itens automaticamente.

Arquivar itens

Tira da visão os itens concluídos sem apagá-los. Pode ser feito manualmente ou por workflow interno.

Insights do Projects e atualizações de status

Insights do Projects mostram gráficos da evolução do trabalho, com gráficos padrão como burn-up e personalizados. Atualizações de status informam o andamento geral (No prazo, Em risco) com um resumo visível para quem acompanha o projeto.

Modelos de projeto

Fazem novos projetos já começarem com os mesmos campos, visões e automações, agilizando a padronização entre equipes.

Milestones

Agrupam issues até uma meta (ex.: versão 2.0) com data prevista e barra de progresso; o progresso considera a proporção de itens fechados. Rótulo de versão classifica, mas não tem prazo nem progresso.

Rótulos, atribuição e respostas salvas

Rótulos personalizados classificam issues (bug, melhoria, documentação) com cores próprias; os repositórios vêm com rótulos padrão editáveis. Atribuir a issue deixa claro o responsável, e uma issue pode ter mais de um. Respostas salvas guardam textos repetidos e ficam nas configurações da conta pessoal.

Macetes para memorizar

💡 Layouts

Table = tabela/planilha; Board = quadro Kanban; Roadmap = rota no tempo.

💡 Tipos de campo

TNDSI: Texto, Número, Data, Seleção única, Iteração.

💡 Iteração x data

Iteração repete (sprint); data marca um dia (entrega).

💡 Milestone x rótulo

Milestone tem meta, prazo e barra; rótulo só tem cor e nome.

💡 Automação

Projects se cuida sozinho: workflows internos movem para Done, adicionam e arquivam.

Tabela de memorização: não confunda

Quando é...	...e quando é
Milestone Acompanhar um grupo de issues até uma meta com data prevista e barra de progresso.	**Rótulo** Classificar itens (bug, melhoria, documentação) com cores, sem prazo nem progresso.
Layout Table Visão de planilha para agrupar, ordenar e filtrar por campos.	**Layout Board** Visão Kanban em colunas definidas por um campo como Status.

Layout Roadmap Linha do tempo com datas ou iterações.	**Layout Board** Colunas de status.
Campo de iteração Ciclos de tempo, como sprints.	**Campo de data** Data prevista de entrega de cada item.
Campo numérico Estimativas que precisam ser somadas.	**Campo de seleção única** Opções fixas, como prioridade Alta, Média ou Baixa.
Item de rascunho Ideia rápida ainda sem repositório definido.	**Issue** Trabalho aprovado: converta o rascunho e ele continua no projeto.

Números que caem

Item	Valor
Peso do domínio na prova	**5–10% (referência de 4 itens em uma prova de 50)**
Layouts do GitHub Projects	**Três: Table, Board e Roadmap**
Tipos de campo personalizado	**Cinco: texto, número, data, seleção única e iteração**

Alertas de prova (pegadinhas)

⚠ **Alerta de prova:** Usar um rótulo de versão para acompanhar entrega com prazo e progresso.
Como não cair: Rótulo só classifica. Prazo e barra de progresso = milestone.

⚠ **Alerta de prova:** Escolher Roadmap ou Board para a visão de planilha.
Como não cair: Roadmap é linha do tempo e Board é colunas. Planilha = Table.

⚠ **Alerta de prova:** Escolher Table ou Insights para a visão Kanban.
Como não cair: Kanban = Board, com colunas definidas por um campo como Status.

⚠ **Alerta de prova:** Escolher Pulse como layout do Projects.
Como não cair: Pulse é página de atividade do repositório (Insights), não layout do Projects.

⚠ **Alerta de prova:** Marcar campo de rótulo ou campo de milestone como tipo de campo personalizado.
Como não cair: Os tipos são texto, número, data, seleção única e iteração.

⚠ **Alerta de prova:** Usar campo de data ou de número para sprints.
Como não cair: Ciclos de tempo = campo de iteração.

⚠ **Alerta de prova:** Usar campo de texto para estimativas que precisam ser somadas.
Como não cair: Texto não soma; use campo numérico.

⚠ **Alerta de prova:** Achar que cada projeto aceita só um repositório ou que é preciso copiar issues à mão.
Como não cair: Um projeto da organização reúne issues de vários repositórios.

⚠ **Alerta de prova:** Criar branches, tags ou forks para separar itens por pessoa ou por sprint.
Como não cair: Isso é recurso de código. No Projects, agrupe (Assignees) ou crie visões salvas com filtros.

⚠ **Alerta de prova:** Apagar e recriar o rascunho como milestone para virar issue.
Como não cair: Milestone não é issue. Converta o rascunho em issue.

⚠ **Alerta de prova:** Usar CODEOWNERS, .gitignore ou Gist para mudar o status do item automaticamente.
Como não cair: Status automático e adição automática = workflows internos do Projects.

⚠ **Alerta de prova:** Usar rótulos em todas as issues para comunicar se o projeto está no prazo.
Como não cair: Rótulos classificam itens, não o projeto. Use atualizações de status do projeto.

Como cai na prova

📝 **Como cai:** 'Qual tipo de campo...' para uma necessidade de planejamento.
Como resolver: Ciclo = iteração; dia = data; opções fixas = seleção única; somar = número; texto livre = texto. Descarte 'campo de rótulo' e 'campo de milestone'.

📝 **Como cai:** 'Qual layout...' descrevendo a aparência.
Como resolver: Planilha = Table; colunas = Board; linha do tempo = Roadmap. Pulse e Insights não são layouts.

📝 **Como cai:** Automação no projeto (status, adição, arquivamento).
Como resolver: Sempre workflows internos do Projects; arquivos do repositório (.gitignore, CODEOWNERS) e Gists não automatizam projetos.

📝 **Como cai:** Organizar e acompanhar trabalho com distratores de código (tags, forks, branches, Gists). **Como resolver:** Elimine o que é recurso de código; fique com o recurso de gestão: milestone, rótulo, atribuição, visão salva, rascunho.

Gatilhos do enunciado

Se o enunciado trouxer...	Pense em
sprints	**Campo de iteração**
prioridade Alta, Média ou Baixa	**Campo de seleção única**
estimativa de esforço somada na visão	**Campo numérico**
data prevista de entrega por item	**Campo de data**
linha do tempo	**Layout Roadmap**
planilha, agrupar, ordenar e filtrar	**Layout Table**
colunas estilo Kanban	**Layout Board**
grupo de issues com data prevista e barra de progresso	**Milestone**
item vai para Done quando a issue fecha	**Workflows internos do Projects**
toda issue nova com rótulo bug entra no projeto	**Workflow de adição automática**
ideias antes de decidir o repositório	**Itens de rascunho**
esconder concluídos sem apagar	**Arquivar os itens**
gráfico da evolução do trabalho concluído	**Insights do Projects (burn-up)**
novos projetos com os mesmos campos, visões e automações	**Modelos de projeto**

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

- Sprint = campo de iteração; entrega = campo de data.
- Table é planilha, Board é Kanban, Roadmap é linha do tempo.
- Milestone tem prazo e barra de progresso; rótulo só classifica.
- Projeto da organização reúne issues de vários repositórios e aceita rascunhos.

Essenciais para a prova (checklist)

- ☐ Sei identificar os layouts Table, Board e Roadmap pela descrição.

- ☐ Sei os cinco tipos de campo personalizado e quando usar cada um.
- ☐ Sei usar visões salvas, agrupamento e soma de campos numéricos.
- ☐ Sei criar e converter itens de rascunho em issues.
- ☐ Sei o que os workflows internos do Projects automatizam.
- ☐ Sei diferenciar milestone de rótulo.
- ☐ Sei para que servem Insights do Projects, atualizações de status e modelos de projeto.
- ☐ Sei vincular um projeto da organização a um repositório.

Questões de revisão do capítulo

R4.1 · Gerenciamento de Projetos no GitHub

Qual layout do GitHub Projects mostra os itens em uma linha do tempo com datas?

A. Layout Board
B. Layout Table
C. Layout Roadmap
D. Página Pulse

R4.2 · Gerenciamento de Projetos no GitHub

Uma equipe quer registrar a estimativa de esforço de cada item, como pontos, e somá-los na visão. Qual campo usar?

A. Campo de data
B. Campo de texto
C. Campo de iteração
D. Campo numérico

R4.3 · Gerenciamento de Projetos no GitHub

Uma equipe quer ver no mesmo projeto só os itens do sprint atual, e outra visão com todos os bugs. O que usar?

A. Visões salvas com filtros
B. Tags do Git
C. Forks do projeto
D. Repositórios separados

R4.4 · Gerenciamento de Projetos no GitHub

Um gestor quer registrar ideias rápidas no projeto antes de decidir em qual repositório viram issues. O que usar?

A. Releases de teste marcadas
B. Branches temporários de ideias
C. Commits vazios na main
D. Itens de rascunho no Projects

Respostas comentadas

R4.1. C) Layout Roadmap

Por quê: O roadmap usa campos de data ou de iteração.

✗ A) Incorreta. Exibe colunas de status.
✗ B) Incorreta. Exibe planilha.
✗ D) Incorreta. Mostra atividade do repositório.

> **Dica de prova:** Linha do tempo: Roadmap.

R4.2. D) Campo numérico

Por quê: Visões agrupadas podem mostrar a soma de campos numéricos.

✗ A) Incorreta. Registra datas.
✗ B) Incorreta. Não soma valores.
✗ C) Incorreta. Define ciclos.

> **Dica de prova:** Estimativas somáveis: campo numérico.

R4.3. A) Visões salvas com filtros

Por quê: Visões podem usar layouts Table, Board ou Roadmap.

✗ B) Incorreta. Marcam versões.
✗ C) Incorreta. Projetos não usam forks.
✗ D) Incorreta. É desnecessário.

> **Dica de prova:** Recortes do mesmo projeto: visões salvas.

R4.4. D) Itens de rascunho no Projects

Por quê: Rascunhos podem ser convertidos em issues depois.

✗ A) Incorreta. Não registram ideias.
✗ B) Incorreta. Não registram ideias de forma organizada.
✗ C) Incorreta. Não registram ideias.

> **Dica de prova:** Ideias sem repositório: rascunhos.

05

CAPÍTULO 5 · 10–15% DA PROVA

Privacidade, Segurança e Administração

Aqui preciso saber proteger contas (2FA, passkeys, SSO, EMU), escolher a credencial certa (token refinado, deploy key, SSH), dar o acesso mínimo (papéis de repositório e de organização) e reconhecer cada recurso de segurança (secret scanning, push protection, code scanning, Dependabot, advisories). A prova costuma pedir o recurso que resolve exatamente o cenário e coloca outros recursos de segurança reais como distratores, então a diferença entre detectar e prevenir, código e dependências, é decisiva. Os papéis Read, Triage, Write, Maintain e Admin caem com frequência. Onde mais erro: secret scanning x push protection, SSO x EMU e Triage x Maintain.

Neste capítulo

16 tópicos-chave · 14 pegadinhas · 6 macetes · 14 gatilhos · 4 questões comentadas

Tópicos-chave

Autenticação de dois fatores (2FA)

Adiciona uma camada extra de proteção ao login além da senha. Os códigos de recuperação permitem entrar sem o segundo fator e devem ser guardados em local seguro, fora do dispositivo de 2FA. A organização pode exigir 2FA de todos, o que pode remover membros que não ativarem.

Passkeys

Usam chaves criptográficas no dispositivo, dispensam digitar senha e resistem a phishing; também podem servir como segundo fator. Senhas curtas, perguntas secretas e códigos por e-mail não substituem.

SSO com SAML x Enterprise Managed Users

SSO com SAML faz os membros autenticarem pelo provedor de identidade corporativo antes de acessar recursos da organização. Com Enterprise Managed Users (EMU), as próprias contas são criadas e controladas pelo IdP (ex.: Entra ID), que cuida do provisionamento e do acesso.

Credenciais de acesso

Token de acesso refinado (fine-grained) escolhe repositórios e permissões, ideal para script com acesso mínimo; token clássico com tudo liberado dá acesso amplo demais. Defina data de expiração e escopo mínimo. Chave de implantação (deploy key) dá acesso SSH a um único repositório, só leitura ou leitura e escrita, sem usar conta de pessoa. Chave SSH evita digitar credencial a cada push.

Token vazado

A primeira ação é revogar e substituir o token; depois, se preciso, limpar o histórico com ferramentas próprias. Apagar só o arquivo atual não resolve, porque o histórico mantém o token, e renomear o repositório não o invalida.

Papéis de repositório

Read, Triage, Write, Maintain e Admin. Read lê e comenta; Triage gerencia issues e PRs (rótulos, atribuições) sem escrever código; Write grava no repositório; Maintain gerencia o repositório (configurações de issues e wiki) sem ações sensíveis como excluir; Admin tem controle total, inclusive exclusão e configurações de segurança. Papéis de repositório personalizados (entre Write e Maintain, por exemplo) estão no Enterprise Cloud.

Papéis de organização

Proprietário tem controle administrativo total, incluindo cobrança e configurações (boa prática: poucos). Gerente de segurança gerencia alertas e configurações de segurança sem ser proprietário, aplicando menor privilégio. Gerente de cobrança cuida só de pagamentos. Colaborador externo não é membro e acessa repositórios específicos, sempre com um papel. Em repositório de conta pessoal, o colaborador convidado recebe leitura e escrita.

Permissões base, equipes e privilégios de membros

Permissões base definem o acesso mínimo de todos os membros aos repositórios; equipes e papéis ampliam por repositório. Equipes agrupam pessoas, podem ser aninhadas e mencionadas como @org/equipe. Nos privilégios de membros, o proprietário define, por exemplo, se membros podem criar repositórios públicos. A política de forks privados impede cópias de repositórios privados para contas pessoais.

Visibilidade e exposição

Pública (qualquer pessoa), privada (só quem tem acesso) e interna (todos os membros da empresa, não o público; existe em contas enterprise). Arquivado não é visibilidade. Ao tornar público um repositório privado, código e histórico ficam visíveis, mas os segredos do Actions continuam protegidos; revise o histórico antes.

Proteção de branch e rulesets

Regras de proteção impedem merge na main sem aprovação e sem testes passando: exigir revisões, status checks, histórico linear (sem commits de merge), commits assinados (GPG, SSH ou S/MIME), descartar aprovações obsoletas quando chegam novos commits e conversas resolvidas. Rulesets aplicam regras a vários branches e repositórios de forma centralizada, no repositório ou na organização.

Secret scanning e push protection

A varredura de segredos detecta credenciais, como tokens de API, enviadas ao repositório. A proteção de push bloqueia o envio do segredo antes que ele chegue ao repositório. Os alertas detectam; a proteção de push previne.

Code scanning e Copilot Autofix

Code scanning analisa o código-fonte em busca de vulnerabilidades e erros usando o CodeQL; os alertas aparecem na aba Security. Copilot Autofix usa IA para sugerir correções para esses alertas, que devem ser revisadas antes do merge.

Dependabot e grafo de dependências

O grafo de dependências lista os pacotes de que o projeto depende, montado a partir de manifestos e arquivos de lock, e serve de base para os alertas. Dependabot alerts e security updates avisam de vulnerabilidades conhecidas e abrem PRs, usando o GitHub Advisory Database. Atualizações de versão periódicas são configuradas em .github/dependabot.yml e são diferentes das atualizações de segurança.

Relato privado e aviso de segurança

O relato privado de vulnerabilidade permite a um pesquisador reportar a falha aos mantenedores sem expô-la. Corrigida a falha, os mantenedores publicam um aviso de segurança (security advisory) com versões afetadas e corrigidas, que pode receber um identificador CVE.

Logs e visão geral de segurança

O log de segurança da conta mostra eventos da própria conta, como logins e alterações de 2FA. O log de auditoria da organização mostra quem alterou configurações e quando, filtrável por ator, ação e data. A visão geral de segurança (security overview) consolida os alertas de todos os repositórios da organização e ajuda a priorizar riscos.

Políticas do Copilot

A organização habilita o Copilot só para algumas equipes e controla quais recursos ficam disponíveis nas políticas do Copilot; empresas também definem políticas no nível enterprise.

Macetes para memorizar

Papéis de repositório

R-T-W-M-A em ordem crescente: Read lê, Triage tria, Write escreve, Maintain mantém, Admin apaga.

Detectar x prevenir

Scanning acha depois; push protection barra antes.

O que cada scanner olha

Code scanning = meu código; Dependabot = código dos outros (dependências); secret scanning = senhas e tokens.

SSO x EMU

SSO: a pessoa entra pela porta da empresa; EMU: a empresa é dona da conta.

Token vazado

Revoga primeiro, limpa depois.

Credenciais mínimas

Refinado para API, deploy key para um repositório por SSH, expiração sempre.

Tabela de memorização: não confunda

Quando é...	...e quando é
Secret scanning Detectar um token que já foi enviado em um commit.	**Push protection** Bloquear o push que contém o token antes de chegar ao repositório.
Code scanning Vulnerabilidades e erros no próprio código-fonte (CodeQL).	**Dependabot** Vulnerabilidades conhecidas em dependências, com PRs de atualização.
SSO com SAML Membros autenticam pelo IdP corporativo antes de acessar a organização.	**Enterprise Managed Users** As próprias contas são criadas e controladas pelo IdP.
Triage Organizar issues e PRs (rótulos, atribuições) sem escrever código.	**Maintain** Gerenciar o repositório (issues, wiki) sem ações destrutivas como excluir.
Token de acesso refinado Script acessando a API com repositórios e permissões mínimas escolhidos.	**Chave de implantação** Servidor lendo um único repositório por SSH sem conta de pessoa.
Log de segurança da conta Eventos da própria conta: logins, alterações de 2FA.	**Log de auditoria da organização** Quem alterou configurações da organização e quando.
Interna Todos os membros da empresa veem, o público não.	**Privada** Só quem tem acesso concedido vê.

Números que caem

Item	Valor
Peso do domínio na prova	**10–15% (referência de 7 itens em uma prova de 50)**
Papéis de repositório padrão	**Cinco: Read, Triage, Write, Maintain e Admin**
Formas de assinar commits	**GPG, SSH ou S/MIME**

Alertas de prova (pegadinhas)

⚠ **Alerta de prova:** Escolher secret scanning para bloquear o token antes do push.
Como não cair: Secret scanning detecta; quem bloqueia antes é a proteção de push.

⚠ **Alerta de prova:** Escolher Dependabot ou secret scanning para analisar vulnerabilidades no código com CodeQL.
Como não cair: Dependabot trata de dependências e secret scanning de credenciais. Código-fonte = code scanning.

⚠ **Alerta de prova:** Escolher Write ou Admin para quem só deve organizar issues e PRs.
Como não cair: Write grava código e Admin tem controle total. Organizar sem escrever código = Triage.

⚠ **Alerta de prova:** Escolher Triage ou Write para gerenciar o repositório sem poder excluí-lo.
Como não cair: Triage só issues e PRs; Write grava código. Gerenciar sem ações destrutivas = Maintain.

⚠ **Alerta de prova:** Adicionar o consultor como proprietário ou membro de todas as equipes para acessar um repositório.
Como não cair: Isso daria acesso total ou amplo. Acesso pontual sem ser membro = colaborador externo.

⚠ **Alerta de prova:** Usar token clássico com tudo liberado ou conta compartilhada para automação.
Como não cair: Token clássico total dá acesso amplo demais e conta compartilhada perde rastreabilidade. Use token refinado (API) ou deploy key (SSH em um repositório).

⚠ **Alerta de prova:** Apagar só o arquivo atual ou renomear o repositório após vazar um token.
Como não cair: O histórico mantém o token e renomear não o invalida. Primeiro revogue e substitua.

⚠ **Alerta de prova:** Achar que tornar público um repositório privado expõe os segredos do Actions ou só o README.
Como não cair: Código e histórico ficam visíveis; os segredos do Actions continuam protegidos.

⚠ **Alerta de prova:** Marcar Arquivada como visibilidade ou Privada como visível à empresa inteira.
Como não cair: Arquivado não é visibilidade. Visível a toda a empresa, não ao público = interna.

⚠ **Alerta de prova:** Confundir SSO com SAML com EMU.
Como não cair: SSO autentica membros pelo IdP; com EMU, as contas em si são criadas e controladas pelo IdP.

⚠ **Alerta de prova:** Escolher Exigir histórico linear ou commits assinados para invalidar aprovação após novos commits.
Como não cair: Histórico linear trata de commits de merge e assinatura trata de identidade. Novos commits, nova revisão = descartar aprovações obsoletas.

⚠ **Alerta de prova:** Consultar Insights ou histórico de commits para saber quem mudou configurações da organização.
Como não cair: Insights mostra métricas e commits só registram código. Quem mudou o quê = log de auditoria.

⚠ **Alerta de prova:** Abrir issue pública ou discussão Q&A para reportar uma falha de segurança.
Como não cair: Isso expõe a falha antes da correção. Use o relato privado de vulnerabilidade.

⚠ **Alerta de prova:** Usar CODEOWNERS ou labeler.yml para configurar atualizações periódicas de dependências.
Como não cair: CODEOWNERS define revisores e labeler.yml rótulos. Atualizações de versão = .github/dependabot.yml.

Como cai na prova

📝 **Como cai:** Cenário de segurança com quatro recursos de segurança reais.
Como resolver: Pergunte: é código, dependência ou credencial? É detectar ou bloquear antes? Código = code scanning; dependência = Dependabot; credencial detectada = secret scanning; bloqueio no push = push protection.

📝 **Como cai:** Qual papel dar a alguém com uma restrição ('sem escrever código', 'sem excluir', 'sem ser membro').
Como resolver: Suba a escada Read, Triage, Write, Maintain, Admin até o primeiro papel que cumpre a necessidade sem ultrapassar a restrição. Sem ser membro = colaborador externo; segurança sem ser dono = gerente de segurança.

📝 **Como cai:** Credencial certa para automação.
Como resolver: API com acesso mínimo = token refinado com expiração; servidor em um repositório por SSH = deploy key. Descarte senha, conta compartilhada e token clássico total.

📝 **Como cai:** Regra de proteção específica descrita pelo efeito.
Como resolver: Sem commits de merge = histórico linear; autoria verificada = commits assinados; aprovação perde validade com novos commits = descartar aprovações obsoletas; várias regras centralizadas = rulesets.

📝 **Como cai:** Identidade corporativa (login pelo IdP, contas gerenciadas, 2FA para todos).
Como resolver: Login pelo IdP = SSO com SAML; contas criadas pelo IdP = EMU; todos com 2FA = exigência de 2FA na organização.

Gatilhos do enunciado

Se o enunciado trouxer...	Pense em
bloquear token antes de chegar ao repositório	**Proteção de push**
token de API enviado por engano em commit	**Varredura de segredos (e revogar o token)**
CodeQL, vulnerabilidades e erros no código	**Code scanning**
IA sugere correção para alerta do code scanning	**Copilot Autofix**
dependência com vulnerabilidade conhecida e PR de atualização	**Dependabot**
contas criadas e controladas pelo provedor de identidade	**Enterprise Managed Users**
autenticar pelo IdP corporativo antes de acessar a organização	**SSO com SAML**
sem senha e resistente a phishing	**Passkeys**
servidor lê um único repositório por SSH	**Deploy key**
script com permissões mínimas pela API	**Token de acesso refinado**
regras para vários branches e repositórios de forma centralizada	**Rulesets**
alertas consolidados de todos os repositórios	**Visão geral de segurança**
aviso formal com versões afetadas e corrigidas, CVE	**Aviso de segurança (security advisory)**
segurança sem ser proprietário	**Gerente de segurança**

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

- Secret scanning detecta; push protection bloqueia antes.
- Token vazado: revogar e substituir primeiro.
- Triage organiza issues e PRs sem código; Maintain gerencia sem excluir; Admin pode tudo.
- Interna é visível para toda a empresa e não para o público; arquivado não é visibilidade.
- EMU: o IdP controla as contas; SSO com SAML: o IdP autentica os membros.

Essenciais para a prova (checklist)

- ☐ Sei a função de 2FA, códigos de recuperação e passkeys.
- ☐ Sei diferenciar SSO com SAML de Enterprise Managed Users.
- ☐ Sei escolher entre token refinado, token clássico, deploy key e chave SSH.
- ☐ Sei o que fazer primeiro quando um token vaza.
- ☐ Sei as permissões de Read, Triage, Write, Maintain e Admin.
- ☐ Sei os papéis de organização: proprietário, membro, gerente de segurança, gerente de cobrança e colaborador externo.
- ☐ Sei as visibilidades pública, privada e interna e o efeito de tornar público um repositório privado.
- ☐ Sei as regras de proteção de branch e quando usar rulesets.
- ☐ Sei diferenciar secret scanning, push protection, code scanning, Copilot Autofix e Dependabot.
- ☐ Sei onde consultar log de segurança, log de auditoria e visão geral de segurança.

Questões de revisão do capítulo

R5.1 · Privacidade, Segurança e Administração

Onde um proprietário define se os membros da organização podem criar repositórios públicos?

A. Nas notas de cada release
B. No arquivo CODEOWNERS da raiz
C. No README da organização
D. Nos privilégios de membros

R5.2 · Privacidade, Segurança e Administração

Qual medida adiciona uma camada extra de proteção ao login da conta, além da senha?

A. Repositório público no perfil
B. Autenticação de dois fatores
C. Arquivo README no perfil
D. Branch padrão chamado main

R5.3 · Privacidade, Segurança e Administração

Uma empresa quer que as contas dos funcionários no GitHub sejam criadas e controladas pelo provedor de identidade corporativo. Qual recurso atende?

A. Enterprise Managed Users
B. GitHub Sponsors
C. Contas pessoais gratuitas
D. Colaboradores externos

R5.4 · Privacidade, Segurança e Administração

Uma organização precisa de um papel entre Write e Maintain, com permissões escolhidas a dedo. O que usar?

A. O papel de proprietário da organização
B. Um colaborador externo sem papel
C. Um token clássico com todos os escopos
D. Um papel de repositório personalizado

Respostas comentadas

R5.1. D) Nos privilégios de membros

Por quê: Restringir repositórios públicos reduz o risco de exposição.

✗ A) Incorreta. Descrevem versões.
✗ B) Incorreta. Define revisores por caminho.
✗ C) Incorreta. É só apresentação.

> **Dica de prova:** Quem cria repositórios: privilégios dos membros.

R5.2. B) Autenticação de dois fatores

Por quê: Passkeys e 2FA reduzem o risco de invasão de contas.

✗ A) Incorreta. Não protege a conta.
✗ C) Incorreta. Não protege a conta.
✗ D) Incorreta. Não protege a conta.

> **Dica de prova:** Proteger login: 2FA.

R5.3. A) Enterprise Managed Users

Por quê: Com EMU, o provisionamento e o acesso vêm do IdP, como o Entra ID.

✗ B) Incorreta. Não gerencia contas.
✗ C) Incorreta. São controladas pelas pessoas.
✗ D) Incorreta. Não são contas gerenciadas.

> **Dica de prova:** Contas controladas pela empresa: EMU.

R5.4. D) Um papel de repositório personalizado

Por quê: Papéis personalizados estão disponíveis no plano Enterprise Cloud.

✗ A) Incorreta. Dá acesso total.
✗ B) Incorreta. Sempre tem um papel.
✗ C) Incorreta. Não é papel de repositório.

> **Dica de prova:** Permissões sob medida: papel personalizado.

06

CAPÍTULO 6 · 10–15% DA PROVA

Práticas Modernas de Desenvolvimento

Este domínio cobre GitHub Actions, Codespaces, github.dev e GitHub Copilot, então preciso saber as chaves do YAML de workflow, os eventos que disparam execuções e onde guardar segredos. A prova traz um cenário de automação ou de ambiente de desenvolvimento e coloca como distratores chaves reais do YAML com outra função (uses, env, runs-on no lugar de needs) ou recursos de outro domínio. No Copilot, cai muito a diferença entre os recursos (inline, Chat, modo agente, agente de codificação, code review, CLI) e entre os planos Free, Pro, Business e Enterprise. Onde mais erro: workflow_dispatch x repository_dispatch, github.dev x Codespaces e achar que variáveis são secretas.

Neste capítulo

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

Tópicos-chave

Workflow do GitHub Actions

Arquivos YAML em .github/workflows/; cada arquivo define eventos (on), jobs e steps. .git/workflows/ não serve (é o diretório interno do Git). Um workflow pode ter vários eventos na seção on.

Eventos de disparo

push (commits), pull_request (PR aberto ou com novos commits; comum para testes), schedule com cron (agendado, horários em UTC e pode atrasar em alta demanda), workflow_dispatch (manual pela interface), repository_dispatch (chamada da API) e release. Padrão comum: testes em pull_request e deploy em push na main.

Jobs, needs e runs-on

Sem needs, os jobs rodam em paralelo por padrão; needs: test faz o job deploy esperar o test terminar com sucesso. runs-on define o tipo de máquina, como ubuntu-latest, windows-latest e macos-latest. strategy define matriz e estratégia; steps lista os passos.

uses x run

Em um step, uses reutiliza uma action (ex.: uses: actions/checkout@v4) e run executa comandos de shell. Fixar a versão ou o hash da action aumenta a previsibilidade e a segurança.

Matriz

strategy.matrix combina versões, sistemas e outras variáveis, por exemplo para rodar os testes em três versões do Node.js no mesmo workflow, sem repositórios separados.

Executores hospedados x auto-hospedados

Executores hospedados pelo GitHub oferecem Linux, Windows e macOS gerenciados (não ChromeOS, Solaris nem FreeBSD). Executores auto-hospedados rodam em máquinas próprias, com acesso à rede interna, e não consomem minutos do GitHub. Minutos dos executores hospedados são grátis em repositórios públicos e cobrados em privados além do uso gratuito.

Variáveis, segredos e GITHUB_TOKEN

Variáveis guardam configurações; segredos guardam credenciais, são cifrados e mascarados nos logs e são lidos com a sintaxe secrets.NOME. Chave de API para deploy vai em segredos do repositório ou ambiente, nunca no YAML, README ou comentário. GITHUB_TOKEN é criado automaticamente para autenticar no próprio repositório e suas permissões podem ser restringidas no workflow.

Artefatos e badge

Arquivos gerados por um job, como relatório de testes, ficam disponíveis para download como artefato do workflow; artefatos também passam arquivos entre jobs do mesmo workflow. Para mostrar no README se o último workflow passou, use o badge de status, cujo Markdown a página do workflow oferece pronto.

Ambientes com revisores exigidos

Fazem o deploy para produção esperar a aprovação de um responsável. Ambientes também guardam segredos específicos, como os de produção. Variável de ambiente, cache ou matriz não pausam jobs.

Workflow reutilizável x action composta

Um workflow reutilizável central (workflow_call) compartilha um workflow inteiro entre repositórios, evitando copiar o YAML. Actions compostas reúnem steps; workflows reutilizáveis reúnem jobs. Falha temporária de rede: reexecutar os jobs com falha pela interface; os logs de cada step mostram onde falhou.

GitHub Codespaces

Ambiente de desenvolvimento na nuvem com terminal e computação; pode ser criado a partir de branches, commits ou PRs (ótimo para testar um PR sem instalar nada). Cobra por computação e armazenamento além da cota incluída e há tempo limite de inatividade; parar codespaces ociosos evita custo e eles podem ser retomados. Mais memória e CPU: mudar o tipo de máquina, que consome mais da cota.

Dev container

O arquivo .devcontainer/devcontainer.json define imagem, extensões, configurações e comandos iniciais, padronizando o ambiente de todos. Depois de alterá-lo, reconstrua o contêiner (rebuild). Dev containers também funcionam localmente com o VS Code.

Prebuilds, dotfiles e segredos do Codespaces

Prebuilds preparam o ambiente com antecedência e reduzem o tempo de abertura, mas consomem armazenamento e computação ao serem gerados. Um repositório de dotfiles, escolhido nas configurações pessoais do Codespaces, aplica aliases e configurações de shell em todo codespace. Chaves de API vão em segredos do Codespaces, definidos por usuário, repositório ou organização.

github.dev

Editor leve no navegador aberto com a tecla . (ponto) no repositório; é gratuito, mas não tem terminal nem computação para builds e testes. Para executar, use Codespaces.

Recursos do Copilot

Sugestões de código inline (texto cinza aceito com Tab); Copilot Chat (explica código em linguagem natural e sugere testes e correções); modo agente (planeja, edita vários arquivos e executa comandos); agente de codificação (recebe uma issue e propõe a solução em um PR para revisão); revisão de código do Copilot (primeira revisão de PR, complementa a humana); Copilot na linha de comando (sintaxe de comandos no terminal); suporte a vários modelos (escolher o modelo de IA, conforme plano e políticas). O desenvolvedor continua responsável pelo que a IA produz.

Instruções personalizadas e exclusão de conteúdo

Instruções personalizadas do repositório, comumente em .github/copilot-instructions.md, fazem o Copilot seguir padrões do projeto (estilo de testes, bibliotecas). Exclusão de conteúdo impede o Copilot de usar arquivos sensíveis como contexto e é configurada por administradores nos planos Business e Enterprise; .gitignore, rótulos e proteção de branch não controlam o Copilot.

Planos do Copilot

Copilot Free: uso individual com limites mensais. Copilot Pro: individual sem limites de completações e chat; estudantes e professores verificados podem recebê-lo sem custo. Copilot Business: para organizações, com políticas e licenças gerenciadas de forma central. Copilot Enterprise: acrescenta recursos para empresas com GitHub Enterprise Cloud.

Macetes para memorizar

Chaves do YAML

on = quando; runs-on = onde; needs = depois de quem; uses = pega pronta; run = roda comando; strategy = matriz.

Eventos manuais

workflow_dispatch = Dedo na interface; repository_dispatch = API.

github.dev x Codespaces

Ponto (.) = só edita; Codespaces = edita e executa.

Segredos

Segredo é Sigiloso: cifrado e mascarado; variável é visível.

Planos do Copilot

Free tem teto, Pro é pessoa sem teto, Business é organização, Enterprise é Enterprise Cloud.

Arquivos de configuração

Workflows em .github/workflows/; ambiente em .devcontainer/devcontainer.json; padrões do Copilot em .github/copilot-instructions.md.

Mudou o ambiente

Mexeu no devcontainer.json? Rebuild.

Tabela de memorização: não confunda

Quando é...	...e quando é
needs Definir dependência e ordem entre jobs (deploy espera test).	**uses** Chamar uma action em um step (actions/checkout@v4).
workflow_dispatch Iniciar o workflow manualmente pela interface.	**repository_dispatch** Disparar o workflow por chamada da API.
github.dev Editar rápido no navegador (tecla ponto), gratuito e sem terminal.	**Codespaces** Precisa de terminal, computação, build e testes em ambiente completo.

Variáveis Configurações não sensíveis, que não são mascaradas.	**Segredos** Credenciais: cifradas e mascaradas nos logs, lidas com secrets.NOME.
Executores hospedados pelo GitHub Máquinas gerenciadas Linux, Windows e macOS do GitHub.	**Executores auto-hospedados** Máquinas próprias com acesso à rede interna; não consomem minutos do GitHub.
Workflow reutilizável Compartilhar jobs inteiros entre repositórios (workflow_call).	**Action composta** Reunir uma sequência de steps.
Copilot Business Empresa que quer controlar políticas e licenças de forma central na organização.	**Copilot Pro** Desenvolvedor individual que quer completações e chat sem limites.

Números que caem

Item	Valor
Peso do domínio na prova	**10–15% (referência de 7 itens em uma prova de 50)**
Sistemas dos executores hospedados pelo GitHub	**Três: Linux, Windows e macOS**
Fuso dos horários do schedule	**UTC**

Alertas de prova (pegadinhas)

⚠ **Alerta de prova:** Usar uses: test, env: test ou runs-on: test para o deploy esperar o test.
Como não cair: uses chama actions, env define variáveis e runs-on define o executor. Ordem entre jobs = needs.

⚠ **Alerta de prova:** Escolher repository_dispatch para execução manual pela interface.
Como não cair: repository_dispatch é por API. Botão manual na interface = workflow_dispatch.

⚠ **Alerta de prova:** Achar que o github.dev executa builds e testes, ou que é idêntico ao Codespaces.
Como não cair: github.dev não tem computação nem terminal; quem tem é o Codespaces.

⚠ **Alerta de prova:** Achar que variáveis são cifradas ou que segredos aparecem em texto nos logs.
Como não cair: Segredos são cifrados e mascarados; variáveis não.

⚠ **Alerta de prova:** Guardar o relatório de testes em um segredo ou no GITHUB_TOKEN.
Como não cair: Segredos não guardam arquivos e GITHUB_TOKEN é um token. Arquivo gerado para baixar = artefato.

⚠ **Alerta de prova:** Usar executores hospedados pelo GitHub para acessar a rede interna da empresa.
Como não cair: Eles rodam em máquinas do GitHub. Rede interna = executores auto-hospedados.

⚠ **Alerta de prova:** Marcar ChromeOS, Solaris ou FreeBSD como sistemas dos executores hospedados.
Como não cair: São Linux, Windows e macOS.

⚠ **Alerta de prova:** Achar que os minutos do Actions nunca são cobrados, ou cobrados em todo uso.
Como não cair: Grátis em repositórios públicos; há cobrança em privados além do uso gratuito.

⚠ **Alerta de prova:** Achar que codespaces ativos bloqueiam merges ou que parados deixam de existir.
Como não cair: O motivo para parar é custo de computação; parados podem ser retomados e o repositório não é afetado.

⚠ **Alerta de prova:** Achar que o agente de codificação do Copilot mescla direto na main.
Como não cair: Ele propõe a solução em um PR; quem revisa e aprova continua responsável.

⚠ **Alerta de prova:** Achar que o Copilot Free inclui gestão para organizações ou uso sem limites.
Como não cair: Free tem limites mensais; gestão central é do Business; sem limites individual é o Pro.

⚠ **Alerta de prova:** Usar .gitignore, rótulo ou proteção de branch para impedir o Copilot de ler arquivos sensíveis.
Como não cair: Nenhum deles controla o Copilot. Use exclusão de conteúdo (Business e Enterprise).

⚠ **Alerta de prova:** Usar variável de ambiente para o deploy esperar aprovação.
Como não cair: Variável não pausa jobs. Aprovação antes do deploy = ambiente com revisores exigidos.

⚠ **Alerta de prova:** Copiar o YAML em cada repositório ou guardá-lo em Gist para compartilhar o build.
Como não cair: Copiar gera duplicação e Gist não é executado. Use workflow reutilizável central.

Como cai na prova

📝 **Como cai:** Cenário de workflow pedindo a chave do YAML (ordem, máquina, action, matriz).
Como resolver: Traduza o cenário em uma pergunta: esperar outro job (needs), escolher máquina (runs-on), reaproveitar action (uses), combinar versões (strategy.matrix). As outras alternativas são chaves reais com outra função.

📝 **Como cai:** Qual evento dispara o workflow em determinada situação.
Como resolver: Manual pela interface = workflow_dispatch; agenda = schedule; PR = pull_request; API = repository_dispatch; versão publicada = release.

📝 **Como cai:** Onde guardar credencial (Actions ou Codespaces).
Como resolver: Sempre segredos: do repositório ou ambiente no Actions; do Codespaces (usuário, repositório ou organização) no codespace. README, YAML em texto claro, comentário e nome de branch expõem a chave.

📝 **Como cai:** Qual recurso do Copilot atende a uma necessidade.
Como resolver: Completar enquanto digita = inline; explicar código = Chat; tarefa em várias etapas = modo agente; issue atribuída = agente de codificação abre PR; revisar PR = code review; terminal = Copilot na CLI.

📝 **Como cai:** Qual plano do Copilot para um perfil de cliente.
Como resolver: Pessoa com limites = Free; pessoa sem limites = Pro; organização com gestão central = Business; empresa com Enterprise Cloud = Enterprise. GitHub Sponsors não é plano do Copilot.

📝 **Como cai:** Problema em codespace (lento, caro, configuração nova, mais recursos).
Como resolver: Lento para abrir = prebuilds; custo = parar os ociosos; mudou devcontainer.json = rebuild; falta CPU/memória = tipo de máquina.

Gatilhos do enunciado

Se o enunciado trouxer...	Pense em
job deploy só depois do job test	**needs: test**
rodar todo dia em um horário, sem depender de commits	**schedule com cron (UTC)**
iniciar manualmente pela interface	**workflow_dispatch**
testes quando um PR é aberto ou recebe commits	**pull_request**
várias versões do runtime no mesmo workflow	**strategy.matrix**
aprovação de um responsável antes do deploy em produção	**Ambiente com revisores exigidos**

token criado automaticamente no workflow	**GITHUB_TOKEN**
relatório gerado para baixar depois da execução	**Artefato do workflow**
tecla . (ponto) no repositório	**Editor github.dev**
testar o PR em ambiente completo sem instalar nada	**Abrir um codespace pelo PR**
codespace demora para abrir	**Prebuilds**
aliases e configurações de shell em todo codespace	**Repositório de dotfiles**
IA planeja, edita vários arquivos e executa comandos	**Modo agente do Copilot**
texto cinza aceito com Tab	**Sugestões de código inline**

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

- Sem needs, os jobs rodam em paralelo.
- github.dev não executa nada; Codespaces tem terminal e computação e cobra por uso.
- Credenciais sempre em segredos, que são cifrados e mascarados.
- O Copilot propõe e o humano revisa: agente de codificação abre PR, não mescla.
- Exclusão de conteúdo é o controle para tirar arquivos do contexto do Copilot.

Essenciais para a prova (checklist)

☐ Sei onde ficam os workflows e o papel de on, jobs, steps, needs, runs-on, uses, run e strategy.

☐ Sei qual evento usar para PR, push, agenda, disparo manual, API e release.

☐ Sei a diferença entre executores hospedados e auto-hospedados e como os minutos são cobrados.

☐ Sei diferenciar variáveis, segredos, GITHUB_TOKEN e artefatos.

☐ Sei usar ambientes com revisores exigidos e workflows reutilizáveis.

☐ Sei a diferença entre github.dev e Codespaces e configurar dev container, prebuilds, dotfiles e segredos do Codespaces.

☐ Sei identificar cada recurso do Copilot pelo cenário.

☐ Sei escolher entre Copilot Free, Pro, Business e Enterprise.

☐ Sei para que servem instruções personalizadas e exclusão de conteúdo no Copilot.

Questões de revisão do capítulo

R6.1 · Práticas Modernas de Desenvolvimento

Em um workflow do Actions, como fazer o job deploy esperar o job test terminar com sucesso?

A. Usar uses: test no job deploy
B. Usar env: test no job deploy
C. Usar runs-on: test no job deploy
D. Usar needs: test no job deploy

R6.2 · Práticas Modernas de Desenvolvimento

Qual evento faz um workflow rodar quando um pull request é aberto ou recebe novos commits?

A. pull_request
B. schedule
C. workflow_dispatch
D. release

R6.3 · Práticas Modernas de Desenvolvimento

Um workflow do Actions deve rodar todos os dias às 3h UTC, sem depender de commits. Qual evento usar?

A. push na main
B. release publicada
C. pull_request aberto
D. schedule com cron

R6.4 · Práticas Modernas de Desenvolvimento

Uma equipe quer que a IA faça uma primeira revisão dos PRs, apontando possíveis problemas. Qual recurso usar?

A. Respostas salvas
B. Dependabot version updates
C. GitHub Pages com Jekyll
D. Revisão de código do Copilot

Respostas comentadas

R6.1. D) Usar needs: test no job deploy

Por quê: Sem needs, os jobs rodam em paralelo por padrão.

✗ A) Incorreta. Chama actions.
✗ B) Incorreta. Define variáveis.
✗ C) Incorreta. Define o executor.

> **Dica de prova:** Ordem entre jobs: needs.

R6.2. A) pull_request

Por quê: É comum rodar testes em pull_request e deploy em push na main.

✗ B) Incorreta. É disparo agendado.
✗ C) Incorreta. É disparo manual.
✗ D) Incorreta. Dispara em releases.

> **Dica de prova:** Testes no PR: evento pull_request.

R6.3. D) schedule com cron

Por quê: Horários do schedule usam UTC e podem atrasar em momentos de alta demanda.

✗ A) Incorreta. Depende de commits.
✗ B) Incorreta. Depende de releases.
✗ C) Incorreta. Depende de PRs.

> **Dica de prova:** Rodar diariamente: schedule.

R6.4. D) Revisão de código do Copilot

Por quê: A revisão da IA complementa, e não substitui, a revisão humana.

✗ A) Incorreta. Guardam textos.
✗ B) Incorreta. Atualiza dependências.
✗ C) Incorreta. Publica sites.

> **Dica de prova:** IA revisando PR: Copilot code review.

07

CAPÍTULO 7 · 10–15% DA PROVA

Repositórios do GitHub

Aqui eu preciso reconhecer cada arquivo especial do repositório pelo propósito e saber onde fica cada configuração na interface. A prova descreve uma necessidade (orientar contribuições, reportar falha, chamar revisores por pasta, mostrar como citar o software) e coloca outros arquivos especiais como distratores. Também cobra manutenção do repositório (arquivar, transferir, renomear, template) e as páginas de Insights. Onde mais erro: CONTRIBUTING x CODE_OF_CONDUCT, Pulse x Traffic e achar que repositório sem LICENSE é domínio público.

Neste capítulo

18 tópicos-chave · 14 pegadinhas · 6 macetes · 14 gatilhos · 4 questões comentadas

Tópicos-chave

README.md

É a página inicial do repositório e explica o que é o projeto e como usá-lo; costuma ter descrição, instalação, uso e contribuição. Começar com README facilita clonar e documentar desde o início.

LICENSE

Define os termos de uso. Sem LICENSE em repositório público valem os direitos autorais padrão: o código não vira domínio público, o GitHub não escolhe MIT e o uso comercial não fica liberado. Exemplos de licenças: MIT, Apache 2.0 e GPL.

CONTRIBUTING.md

Guia técnico que orienta novos colaboradores sobre padrões de código e o processo de envio de mudanças. O GitHub mostra um link para ele ao abrir issues e PRs.

CODE_OF_CONDUCT.md

Define as regras de convivência e o comportamento esperado dos participantes. O GitHub oferece modelos, como o Contributor Covenant. Não descreve o processo de contribuição.

SECURITY.md e SUPPORT.md

SECURITY.md explica como relatar vulnerabilidades de forma responsável e aparece na aba de segurança do repositório. SUPPORT.md indica onde buscar ajuda (fórum, canal de suporte) e é um dos arquivos de saúde da comunidade.

CODEOWNERS

Define responsáveis por caminho, chamando automaticamente revisores para PRs que alteram, por exemplo, a pasta /api. Pode ficar na raiz, em .github/ ou em docs/.

CITATION.cff

Mostra como citar o software, com um botão na página do repositório; o GitHub gera citações em formatos como APA e BibTeX. Não confundir com FUNDING.yml (patrocínio).

.gitignore e parentes

.gitignore lista padrões que o Git não deve rastrear (build, .env) e evita enviar arquivos gerados e segredos por engano. .gitkeep é só convenção para manter pastas vazias; .gitconfig configura o Git do usuário; .gitattributes define atributos por caminho.

Repositório .github da organização

Aplica arquivos de saúde da comunidade padrão, como CODE_OF_CONDUCT, a todos os repositórios da organização sem copiá-los. Repositórios que têm arquivos próprios continuam usando os seus.

Criação do repositório

Pela interface, o primeiro commit pode já incluir README, .gitignore (com modelos por linguagem) e licença. Milestones, rótulos, workflows, segredos e releases não fazem parte da criação.

Visibilidade privada

Só pessoas com acesso concedido veem o conteúdo; ter o link ou seguir o dono não dá acesso. Na organização, o acesso vem de papéis, equipes e permissões base.

Template de repositório

Marcar um repositório como template permite criar novos repositórios a partir da sua estrutura, sem o histórico. Fork, ao contrário, copia todo o conteúdo e histórico e mantém o vínculo com o original.

Arquivar, transferir e renomear

Arquivar deixa o repositório visível e só leitura, e pode ser desarquivado. Transferir muda o dono mantendo issues e histórico, com redirecionamento do endereço antigo. Renomear redireciona os links antigos, mas é boa prática atualizar os remotos locais.

Branch padrão

É a base padrão de PRs e do clone, e as palavras-chave de fechamento de issues dependem dele. Renomear de master para main é feito nas configurações; o GitHub redireciona o nome antigo e orienta atualizar clones locais. Não é o único branch que aceita commits.

Exclusão automática de branches

Apaga automaticamente o branch depois que o PR é mesclado. Branches excluídos podem ser restaurados pela página do PR.

Insights

Aba de métricas: Pulse (atividade recente, como PRs mesclados e issues abertas; período ajustável), Contributors (commits de cada colaborador ao longo do tempo), Traffic (visitas e clones, período recente limitado), Community (checklist de padrões: descrição, README, licença, código de conduta e templates), Network (gráfico de branches), Forks, Code frequency e Commits.

Apresentação e descoberta

A seção About define descrição curta e endereço do site; tópicos (ex.: kubernetes, devops) ajudam na descoberta e aparecem na exploração do GitHub; a prévia social define a imagem ao compartilhar o link. Issues, Wiki, Projects e Discussions podem ser desativados para limpar a interface.

Arquivos pela web, LFS e releases

Editar e salvar um arquivo na web cria um commit (dá para criar branch e abrir PR); sem Git local, envie o arquivo por upload na interface. Arquivos muito grandes, como vídeos, pedem Git Large File Storage, pois o GitHub limita o tamanho de arquivos comuns. History mostra o histórico de um arquivo e Blame mostra quem alterou cada linha. Baixar ZIP traz só arquivos de um branch, sem histórico. Release publica uma versão a partir de uma tag com notas e binários; as notas podem ser geradas automaticamente a partir dos PRs e editadas antes de publicar.

Macetes para memorizar

💡 Arquivos de comunidade

CONTRIBUTING = como; CODE_OF_CONDUCT = comportamento; SECURITY = segredo de falha; SUPPORT = socorro.

💡 CODEOWNERS

Owners = donos do código chamados para revisar; três casas possíveis: raiz, .github/ ou docs/.

Insights

Pulse = pulso da semana; Traffic = trânsito de visitas e clones; Contributors = quem commitou; Community = checklist.

Template x fork

Template nasce limpo (sem histórico); fork nasce com memória (histórico e vínculo).

Mudanças de manutenção

Arquivar congela, transferir muda de casa, renomear redireciona.

Sem licença

Sem LICENSE, sem permissão: direitos autorais padrão.

Tabela de memorização: não confunda

Quando é...	...e quando é
CONTRIBUTING.md O cenário fala de padrões de código e processo de envio de mudanças.	**CODE_OF_CONDUCT.md** O cenário fala de convivência e comportamento esperado dos participantes.
SECURITY.md Como relatar vulnerabilidades de forma responsável.	**SUPPORT.md** Onde os usuários pedem ajuda (fórum, canal de suporte).
Template Padronizar novos projetos começando sem o histórico e sem vínculo com o original.	**Fork** Contribuir ou divergir mantendo histórico e vínculo com o original.
Clone Contribuir: traz arquivos, histórico e branches.	**Download ZIP** Só obter os arquivos de um branch, sem histórico.
Pulse Atividade recente: PRs mesclados, issues abertas na última semana.	**Traffic** Visitas e clones nos últimos dias.
Arquivar Repositório sem manutenção que deve continuar visível como referência, só leitura.	**Tornar privado** Ocultar o conteúdo de quem não tem acesso.
Tópicos Facilitar a descoberta por assunto na busca e na exploração.	**Tags do Git** Marcar versões do código.

Números que caem

Item	Valor
Peso do domínio na prova	**10–15% (referência de 6 itens em uma prova de 50)**

Locais aceitos para o CODEOWNERS	**Três: raiz, .github/ ou docs/**
Arquivos que podem entrar no primeiro commit ao criar pela interface	**Três: README, .gitignore e licença**

Alertas de prova (pegadinhas)

⚠ **Alerta de prova:** Escolher CODE_OF_CONDUCT.md para orientar padrões de código e envio de mudanças.
Como não cair: Código de conduta é convivência; processo técnico de contribuição é CONTRIBUTING.md.

⚠ **Alerta de prova:** Achar que repositório público sem LICENSE é de domínio público ou recebe MIT automaticamente.
Como não cair: Sem licença valem os direitos autorais padrão; nada é permitido explicitamente e o GitHub não escolhe licença.

⚠ **Alerta de prova:** Escolher Insights, Pulse para ver visitas e clones.
Como não cair: Pulse é atividade de PRs e issues; visitas e clones ficam em Traffic.

⚠ **Alerta de prova:** Achar que o template copia todo o histórico ou que template e fork mantêm vínculo com o original.
Como não cair: Template começa sem histórico; só o fork mantém histórico e vínculo.

⚠ **Alerta de prova:** Achar que o ZIP traz todos os branches ou que é igual ao clone.
Como não cair: O ZIP traz um branch, sem histórico. Para contribuir, clone.

⚠ **Alerta de prova:** Fazer fork e apagar o original para mudar o dono do repositório.
Como não cair: Perde issues e estrelas; template ou upload perdem o histórico. Para mudar de dono com histórico e issues: transferir.

⚠ **Alerta de prova:** Tornar privado ou apagar um repositório antigo que deve continuar como referência.
Como não cair: Privado oculta e apagar remove. Visível e só leitura = arquivar.

⚠ **Alerta de prova:** Achar que links antigos param de funcionar ao renomear o repositório.
Como não cair: O GitHub redireciona; mesmo assim atualize os remotos locais.

⚠ **Alerta de prova:** Usar tags do Git ou mais branches para que o repositório apareça em buscas por assunto.
Como não cair: Tags marcam versões. Descoberta por assunto = tópicos.

⚠ **Alerta de prova:** Achar que quem tem o link ou segue o dono vê um repositório privado.
Como não cair: Só pessoas com acesso concedido veem; 'qualquer usuário logado' descreveria repositório público.

⚠ **Alerta de prova:** Escolher Proteção do branch padrão ou Arquivamento para apagar branches após o merge.
Como não cair: Proteção controla a main e arquivamento deixa tudo só leitura. Limpar branches após merge = exclusão automática.

⚠ **Alerta de prova:** Usar FUNDING.yml ou SUPPORT.md para mostrar como citar o software.
Como não cair: FUNDING.yml é patrocínio, SUPPORT.md é ajuda. Citação = CITATION.cff.

⚠ **Alerta de prova:** Confundir .gitignore com .gitkeep, .gitconfig ou .gitattributes.
Como não cair: Ignorar arquivos = .gitignore. .gitkeep mantém pastas, .gitconfig configura o usuário, .gitattributes define atributos por caminho.

⚠ **Alerta de prova:** Criar uma tag ou um fork chamado main para renomear o branch padrão.
Como não cair: Tags não são branches e fork não renomeia. Renomeie o branch padrão nas configurações.

Como cai na prova

📝 **Como cai:** 'Qual arquivo...' com quatro arquivos especiais como alternativas.
Como resolver: Identifique o verbo do cenário (orientar contribuição, convivência, reportar falha, pedir ajuda, revisar por caminho, citar) e associe ao arquivo; os outros são arquivos reais de outra finalidade.

📝 **Como cai:** 'Onde o mantenedor vê...' métricas do repositório.
Como resolver: É sempre Insights; escolha a subpágina: visitas/clones = Traffic, atividade = Pulse, commits por pessoa = Contributors, padrões de comunidade = Community.

📝 **Como cai:** Manutenção do repositório com uma restrição (manter histórico, continuar visível, só leitura).
Como resolver: Leia a restrição: manter issues e histórico mudando o dono = transferir; visível e só leitura = arquivar; base sem histórico = template. Descarte apagar ou copiar por upload.

📝 **Como cai:** 'O que acontece quando...' (renomear, editar na web, sem licença).
Como resolver: Renomear redireciona; editar na web gera commit; sem licença valem direitos autorais padrão. Alternativas extremas (para de funcionar, apaga histórico) estão erradas.

📝 **Como cai:** Diferença entre duas formas de obter o código (clone x ZIP, template x fork).
Como resolver: Pergunte: traz histórico? mantém vínculo? Clone e fork trazem histórico; ZIP e template, não.

Gatilhos do enunciado

Se o enunciado trouxer...	Pense em
chamar automaticamente revisores para mudanças em uma pasta	**CODEOWNERS**
regras de convivência	**CODE_OF_CONDUCT.md**
padrões de código e processo de envio de mudanças	**CONTRIBUTING.md**
relatar vulnerabilidades de forma responsável	**SECURITY.md**
onde pedir ajuda sobre o projeto	**SUPPORT.md**
como citar o software em artigos	**CITATION.cff**
mesmos arquivos de saúde em todos os repositórios da organização	**Repositório .github da organização**
arquivos de vídeo muito grandes	**Git Large File Storage**
visível como referência, apenas leitura	**Arquivar o repositório**
repositório pessoal passa para a organização mantendo issues e histórico	**Transferir**
listar PRs mesclados desde a versão anterior sem escrever à mão	**Notas de release geradas automaticamente**
lista do que falta para seguir padrões de comunidade	**Insights, Community**
imagem exibida ao compartilhar o link em redes sociais	**Prévia social**
novos repositórios a partir da estrutura de um existente	**Marcar como template**

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

- CONTRIBUTING é processo técnico; CODE_OF_CONDUCT é convivência.
- Sem LICENSE, valem os direitos autorais padrão.
- Transferir mantém issues e histórico; template não traz histórico.
- Visitas e clones ficam em Insights, Traffic, não em Pulse.

Essenciais para a prova (checklist)

- ☐ Sei associar README, LICENSE, CONTRIBUTING, CODE_OF_CONDUCT, SECURITY, SUPPORT, CODEOWNERS e CITATION.cff ao seu propósito.
- ☐ Sei para que serve o repositório .github de uma organização.
- ☐ Sei a diferença entre template e fork e entre clone e download ZIP.
- ☐ Sei quando arquivar, transferir ou renomear um repositório e o que acontece com os links.
- ☐ Sei o papel do branch padrão e como renomeá-lo.
- ☐ Sei qual subpágina de Insights mostra cada métrica.
- ☐ Sei usar About, tópicos e prévia social para apresentar o repositório.
- ☐ Sei quando usar Git LFS e o que acontece ao editar um arquivo na web.

Questões de revisão do capítulo

R7.1 · Repositórios do GitHub

Qual arquivo define as regras de convivência e comportamento esperado dos participantes de um projeto?

A. SECURITY.md
B. CODE_OF_CONDUCT.md
C. CHANGELOG.md
D. CONTRIBUTING.md

R7.2 · Repositórios do GitHub

Qual arquivo orienta novos colaboradores sobre padrões de código e o processo de envio de mudanças?

A. .gitignore
B. CONTRIBUTING.md
C. CODE_OF_CONDUCT.md
D. CODEOWNERS

R7.3 · Repositórios do GitHub

Um repositório público não tem arquivo LICENSE. O que isso implica para quem quer reutilizar o código?

A. Valem os direitos autorais padrão
B. O código é automaticamente de domínio público
C. O GitHub escolhe uma licença MIT
D. Qualquer uso comercial é liberado

R7.4 · Repositórios do GitHub

Qual página de Insights resume a atividade recente do repositório, como PRs mesclados e issues abertas na última semana?

A. Network
B. Forks
C. Pulse
D. Traffic

Respostas comentadas

R7.1. B) CODE_OF_CONDUCT.md

Por quê: O GitHub oferece modelos, como o Contributor Covenant.

✗ A) Incorreta. Política de segurança.
✗ C) Incorreta. Registro de mudanças.
✗ D) Incorreta. Guia técnico de contribuição.

> **Dica de prova:** Regras de convivência: CODE_OF_CONDUCT.

R7.2. B) CONTRIBUTING.md

Por quê: O GitHub mostra um link para o CONTRIBUTING ao abrir issues e PRs.

✗ A) Incorreta. Lista arquivos ignorados.
✗ C) Incorreta. Define regras de convivência, não o processo de contribuição.
✗ D) Incorreta. Define responsáveis por revisão.

> **Dica de prova:** Como contribuir: CONTRIBUTING.md.

R7.3. A) Valem os direitos autorais padrão

Por quê: Licenças como MIT, Apache 2.0 e GPL definem direitos de uso.

✗ B) Incorreta. Não é o caso.
✗ C) Incorreta. O GitHub não escolhe licenças.
✗ D) Incorreta. Não há permissão explícita.

> **Dica de prova:** Sem LICENSE: direitos autorais padrão.

R7.4. C) Pulse

Por quê: O período do Pulse pode ser ajustado.

✗ A) Incorreta. Mostra o gráfico de branches.

✗ B) Incorreta. Mostra a rede de forks.

✗ D) Incorreta. Mostra visitas e clones.

> **Dica de prova:** Atividade recente: Pulse.

DIA DA PROVA

Estratégia de prova

A GH-900 é uma prova de fundamentos: cada enunciado descreve uma necessidade concreta e pede o comando, o recurso, o arquivo ou a configuração exata que a resolve. As alternativas erradas quase sempre são recursos reais do GitHub ou comandos reais do Git com outra função, então ganhar a prova é saber o propósito preciso de cada um, não apenas reconhecer o nome. Aparecem também itens de múltipla resposta (escolha duas) e pares que se invertem na alternativa errada, como fetch e pull ou fork e clone. Detalhes de restrição no enunciado (sem reescrever o histórico, mantendo as alterações, sem ser membro, sem escrever código) definem a resposta. Fundamentos de Git e GitHub responde por 25–30% e é onde a prova mais pontua.

Como responder

✓ Administre o tempo: são 100 minutos para cerca de 40 a 60 itens, e a aprovação exige 700 de 1000; mantenha um ritmo constante, marque os itens em dúvida e reserve o fim da prova para revisá-los.

✓ Sublinhe mentalmente a restrição do enunciado (já enviado, sem histórico, só leitura, sem ser proprietário, sem instalar nada) antes de olhar as alternativas.

✓ Para cada alternativa, diga em uma frase o que aquele recurso realmente faz; se a frase não responde ao cenário, elimine.

✓ Em múltipla resposta, marque exatamente o número pedido e confira se as duas escolhidas atendem ao mesmo critério do enunciado.

✓ Desconfie de pares invertidos (fetch/pull, fork/clone, base/compare, Cloud/Server): leia as duas metades da frase antes de marcar.

✓ Elimine alternativas desproporcionais (apagar o repositório, recriar o fork, criar outro repositório) e recursos inexistentes (fork da issue, Copilot em Pages, estrela em issue).

✓ Em sintaxe (comandos, YAML, Markdown, filtros), confira se a opção pertence ao comando ou chave citada: --staged é do diff, --soft é do reset, needs é entre jobs, @ é menção.

✓ Distribua a revisão final priorizando Fundamentos de Git e GitHub, que tem o maior peso.

Erros que derrubam candidatos

✗ Marcar um recurso real de outra finalidade só porque o nome parece familiar (Pulse em vez de Traffic, CODE_OF_CONDUCT em vez de CONTRIBUTING).

✗ Ignorar se o commit já foi compartilhado e escolher reset --hard com push forçado em vez de revert.

✗ Confundir detectar com prevenir em segurança: secret scanning em vez de push protection.

✗ Inverter pares de conceitos (fetch e pull, fork e clone, base e compare, Enterprise Cloud e Server).

✗ Escolher squash quando o cenário pede preservar commits, ou rebase and merge quando pede um único commit.

✗ Achar que github.dev executa builds ou que variáveis do Actions são secretas.

✗ Marcar mais ou menos alternativas do que o pedido nos itens de múltipla resposta.

Na semana da prova

- ☐ Pratique focando nos domínios de maior peso e anote cada erro como par 'confundi X com Y'.
- ☐ Revise os comandos de desfazer (restore, reset --soft/--hard, revert, amend, reflog) e de remoto (fetch, pull, push -u, tags).
- ☐ Revise os arquivos especiais do repositório e as subpáginas de Insights em uma única lista de propósito.
- ☐ Revise as tabelas de papéis (Read, Triage, Write, Maintain, Admin) e os recursos de segurança (secret scanning, push protection, code scanning, Dependabot).
- ☐ Revise as chaves do YAML do Actions, os eventos de disparo e os planos e recursos do Copilot.
- ☐ Na véspera, leia só as seções de lembrar e gatilhos de cada domínio e descanse.

REVISÃO FINAL

Folha de cola

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

1. Colaboração no GitHub

‣ um commit só no branch de destino → **Squash and merge**

‣ cada commit no destino, sem commit de merge → **Rebase and merge**

‣ preservar commits e registrar a integração com commit extra → **Create a merge commit**

‣ propor a correção exata de uma linha para aplicar com um clique → **Sugestão de alteração**

‣ vários comentários enviados juntos com parecer final → **Iniciar uma revisão**

⚠ Rebase and merge aplica cada commit; quem junta é Squash and merge.

⚠ Squash junta tudo. Preservar e registrar a integração com commit extra = Create a merge commit.

⚠ Rebase and merge não cria commit de merge; quem cria é Create a merge commit.

2. Comunidade do GitHub

‣ contribuir sem permissão de escrita → **Fork, alterar e abrir um PR**

‣ fork desatualizado em relação ao original → **Sync fork**

‣ apoiar financeiramente mantenedores → **GitHub Sponsors**

‣ botão Sponsor no repositório → **.github/FUNDING.yml**

‣ práticas de código aberto dentro da empresa → **InnerSource**

⚠ Estrela é favorito e apreço; notificações vêm do Watch e cópia é o fork.

⚠ Watch só gera notificações; nem estrela nem Watch dão acesso.

⚠ Sem licença nada é permitido. Código aberto = código disponível sob licença aberta.

3. Fundamentos de Git e GitHub

‣ trocar de branch com urgência sem comitar o trabalho pela metade → **git stash (depois git stash pop)**

‣ commit perdido depois de git reset --hard → **git reflog para achar o hash**

‣ commit com erro já enviado à main compartilhada → **git revert com o hash**

‣ aplicar só um commit de outro branch → **git cherry-pick**

‣ quem alterou cada linha e em qual commit → **git blame (ou a visão Blame no GitHub)**

⚠ fetch só baixa e atualiza origin/main; a main local só muda após o merge. pull = fetch + integração.

⚠ Isso reescreve o histórico compartilhado. Em branch compartilhado, a resposta é git revert, que cria um commit novo.

⚠ --soft é opção do git reset. revert e push não têm essa opção.

4. Gerenciamento de Projetos no GitHub

‣ sprints → **Campo de iteração**

‣ prioridade Alta, Média ou Baixa → **Campo de seleção única**

‣ estimativa de esforço somada na visão → **Campo numérico**

‣ data prevista de entrega por item → **Campo de data**

‣ linha do tempo → **Layout Roadmap**

⚠ Rótulo só classifica. Prazo e barra de progresso = milestone.

⚠ Roadmap é linha do tempo e Board é colunas. Planilha = Table.

⚠ Kanban = Board, com colunas definidas por um campo como Status.

5. Privacidade, Segurança e Administração

‣ bloquear token antes de chegar ao repositório → **Proteção de push**

‣ token de API enviado por engano em commit → **Varredura de segredos (e revogar o token)**

‣ CodeQL, vulnerabilidades e erros no código → **Code scanning**

‣ IA sugere correção para alerta do code scanning → **Copilot Autofix**

‣ dependência com vulnerabilidade conhecida e PR de atualização → **Dependabot**

⚠ Secret scanning detecta; quem bloqueia antes é a proteção de push.

⚠ Dependabot trata de dependências e secret scanning de credenciais. Código-fonte = code scanning.

⚠ Write grava código e Admin tem controle total. Organizar sem escrever código = Triage.

6. Práticas Modernas de Desenvolvimento

‣ job deploy só depois do job test → **needs: test**

‣ rodar todo dia em um horário, sem depender de commits → **schedule com cron (UTC)**

‣ iniciar manualmente pela interface → **workflow_dispatch**

‣ testes quando um PR é aberto ou recebe commits → **pull_request**

‣ várias versões do runtime no mesmo workflow → **strategy.matrix**

⚠ uses chama actions, env define variáveis e runs-on define o executor. Ordem entre jobs = needs.

⚠ repository_dispatch é por API. Botão manual na interface = workflow_dispatch.

⚠ github.dev não tem computação nem terminal; quem tem é o Codespaces.

7. Repositórios do GitHub

‣ chamar automaticamente revisores para mudanças em uma pasta → **CODEOWNERS**

‣ regras de convivência → **CODE_OF_CONDUCT.md**

‣ padrões de código e processo de envio de mudanças → **CONTRIBUTING.md**

‣ relatar vulnerabilidades de forma responsável → **SECURITY.md**

‣ onde pedir ajuda sobre o projeto → **SUPPORT.md**

⚠ Código de conduta é convivência; processo técnico de contribuição é CONTRIBUTING.md.

⚠ Sem licença valem os direitos autorais padrão; nada é permitido explicitamente e o GitHub não escolhe licença.

⚠ Pulse é atividade de PRs e issues; visitas e clones ficam em Traffic.

GABARITO

Respostas do diagnóstico

D1. C) Em .github/pull_request_template.md → revise o capítulo 1

Por quê: Também é possível ter vários modelos em .github/PULL_REQUEST_TEMPLATE/.

✗ A) Incorreta. Define workflows.
✗ B) Incorreta. Documenta o projeto.
✗ D) Incorreta. Lista arquivos ignorados.

> **Dica de prova:** Modelo de PR: pull_request_template.md.

D2. C) Squash and merge → revise o capítulo 1

Por quê: Squash deixa o histórico mais limpo; merge commit preserva todos os detalhes.

✗ A) Incorreta. Mantém os commits e cria um commit de merge.
✗ B) Incorreta. Desfaz um merge.
✗ D) Incorreta. Reaplica os commits sem commit de merge.

> **Dica de prova:** Um commit só: squash and merge.

D3. A) Sugestão de alteração → revise o capítulo 1

Por quê: Sugestões podem ser aplicadas em lote como um commit.

✗ B) Incorreta. Não aplica a mudança.
✗ C) Incorreta. Não aplica a mudança.
✗ D) Incorreta. É excessivo para uma linha.

> **Dica de prova:** Correção de uma linha: sugestão de alteração.

D4. B) Código disponível sob licença aberta → revise o capítulo 2

Por quê: A licença define direitos e obrigações de quem reutiliza.

✗ A) Incorreta. Sem licença, nada é permitido.
✗ C) Incorreta. Não é aberto.
✗ D) Incorreta. Não é o critério.

> **Dica de prova:** Código aberto: licença aberta.

D5. C) Mais pessoas podem revisar o código → revise o capítulo 2

Por quê: Transparência não substitui boas práticas de segurança.

✗ A) Incorreta. O código é público.
✗ B) Incorreta. Não é verdade.
✗ D) Incorreta. Não é verdade.

> **Dica de prova:** Código aberto: mais revisores.

D6. C) Cria um histórico público de trabalho → revise o capítulo 2

Por quê: Contribuições também desenvolvem colaboração e revisão de código.

✗ A) Incorreta. Revisões fazem parte.

✗ B) Incorreta. Não há garantia.

✗ D) Incorreta. Não é o caso.

> **Dica de prova:** Carreira: portfólio público.

D7. C) Branch, commits, PR, revisão e merge → revise o capítulo 3

Por quê: No GitHub Flow, a main fica sempre pronta para implantar.

✗ A) Incorreta. A ordem está incorreta.

✗ B) Incorreta. O trabalho é feito em um branch.

✗ D) Incorreta. A ordem está invertida.

> **Dica de prova:** GitHub Flow: branch, commit, PR, revisão, merge.

D8. A) git switch main → revise o capítulo 3

Por quê: git checkout main também funciona.

✗ B) Incorreta. Gerencia remotos.

✗ C) Incorreta. Cria uma tag.

✗ D) Incorreta. Apaga o branch.

> **Dica de prova:** Mudar de branch: git switch.

D9. B) GitHub Enterprise Server → revise o capítulo 3

Por quê: O GitHub Enterprise Cloud é hospedado pelo GitHub; o Server, pelo cliente.

✗ A) Incorreta. É um aplicativo cliente.

✗ C) Incorreta. Hospedado pelo GitHub.

✗ D) Incorreta. É um aplicativo cliente.

> **Dica de prova:** Auto-hospedado: GitHub Enterprise Server.

D10. A) Table → revise o capítulo 4

Por quê: Visões salvas guardam filtros, agrupamentos e ordenações.

✗ B) Incorreta. Visão em linha do tempo.

✗ C) Incorreta. Visão em colunas.

✗ D) Incorreta. Página de atividade do repositório.

> **Dica de prova:** Planilha no Projects: Table.

D11. B) Workflows internos do Projects → revise o capítulo 4

Por quê: Workflows internos também adicionam itens novos automaticamente.

✗ A) Incorreta. Não automatiza projetos.
✗ C) Incorreta. Define revisores.
✗ D) Incorreta. Não automatiza projetos.

> **Dica de prova:** Status automático: workflows do Projects.

D12. A) Workflow de adição automática → revise o capítulo 4

Por quê: Filtros podem usar rótulos, tipos e outros critérios.

✗ B) Incorreta. Não afeta projetos.
✗ C) Incorreta. Não afeta projetos.
✗ D) Incorreta. Não afeta projetos.

> **Dica de prova:** Adicionar itens sozinho: auto-add.

Agora, treine no formato real

Faça as 6 provas completas do curso GH-900: 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.