

Questão 1 — Gabarito e Rubrica (Peso 2,0)

Enunciado (resumo):

Definir o que é **conflito de merge** e apresentar **duas boas práticas** (Git Flow) que poderiam evitar/minimizar o problema no **serviço de *match*** do Glink, com **justificativas**.

Critérios de Correção (com pontuação parcial)

Critério	Pontos	O que vale nota máxima	Parcial	Zero
C1. Conceito de “conflito de merge”	0,5	Explica com precisão que é a incompatibilidade automática entre mudanças concorrentes na mesma região do código (ou em arquivos de dependência acoplados), exigindo resolução manual .	Explica conflito mas sem citar a necessidade de resolução manual OU sem apontar que ocorre em regiões coincidentes .	Confunde com erro de compilação/execução ou define como “bug do Git”.
C2. Contextualização ao Git Flow	0,3	Relaciona explicitamente ao Git Flow: feature branches partindo de <code>develop</code> , <code>pull requests</code> para <code>develop</code> , <code>release/hotfix</code> e o impacto de sincronização tardia (branches long-lived).	Menciona branches e PRs, mas sem citar fluxo Git Flow ou sem conectar a sincronização como causa.	Sem ligação com Git Flow/fluxo de branches.
C3. Prática 1 – Relevância e profundidade	0,5	Sugere uma prática altamente pertinente ao caso (ex.: sincronizar/atualizar frequentemente a branch de feature a partir de develop via <code>merge</code> ou <code>rebase</code> ; PRs pequenos e frequentes ; limitar escopo da feature ; pair programming na área sensível), explica como reduz conflitos e conecta ao serviço de <i>match</i> do Glink.	Prática pertinente, com explicação genérica (sem amarrar ao serviço de <i>match</i> /contexto).	Prática irrelevante ou vaga (ex.: “usar testes” sem relação com conflitos).
C4. Prática 2 – Relevância e profundidade	**0,5	Segunda prática complementar e relevante (ex.: Code Owners/review obrigatório no módulo de <i>match</i> ; fatiar mudanças por arquivo/módulo ; acordos de código/contratos de módulo ; feature flags para isolar caminhos), bem justificada e amarrada ao contexto .	Prática pertinente, mas com justificativa superficial ou sem conexão ao caso.	Irrelevante/repetida/sem justificativa.
C5. Justificativa bem embasada e conexão com o Glink	0,2	A resposta cita elementos do caso (ex.: “mesmo método do serviço de <i>match</i> ”, “3 devs backend”, “alto fluxo na mesma classe”) e argumenta causalmente por que as práticas mitigam o conflito neste cenário .	Alude ao contexto, mas de forma genérica, sem ser específico ao serviço de <i>match</i> .	Sem contextualização.

Total: $0,5 + 0,3 + 0,5 + 0,5 + 0,2 = 2,0$ pontos

Exemplos de Conteúdo Esperado (resumo orientador)

Definição correta (C1)

“Conflito de *merge* ocorre quando duas alterações concorrentes atingem a **mesma região de código** (por exemplo, o mesmo método da classe `MatchScoring`), impedindo a fusão automática; exige **resolução manual** para decidir qual versão prevalece ou como combiná-las.”

Boas práticas com justificativa (C3, C4, C5)

1. Sincronização frequente com `develop` (*merge/rebase*)

- **Por quê relevante?** No Git Flow, *feature branches* ficam desatualizadas rapidamente; sincronizar reduz o “salto” entre bases de código.
- **Conexão Glink:** Como **dois devs editaram o mesmo método** do serviço de *match*, integrar cedo e frequentemente diminui a chance de ambos acumularem mudanças conflitantes.

2. PRs pequenos e escopo limitado

- **Por quê relevante?** Menos linhas alteradas = menor superfície de conflito e revisão mais rápida.
- **Conexão Glink:** Em vez de “refatorar o método inteiro” + “ajustar pesos do índice” no mesmo PR, separar **refatoração e ajuste do algoritmo** em PRs distintos no módulo de *match*.

Outras práticas que também pontuam bem (se bem justificadas e conectadas ao caso):

- **Code Owners / revisão obrigatória** para arquivos do *match scoring*;
- **Fatiar por módulo/arquivo** (ex.: extrair funções auxiliares para reduzir edição simultânea do mesmo método);
- **Acordos de design / contratos de interface** (definir claramente responsabilidades do serviço de *match* vs. *perfil*);
- **Pair/Mob programming** em trechos críticos;
- **Feature flags** para evoluir o algoritmo sem editar o mesmo bloco central.

Resposta-modelo (exemplo conciso, não único)

Conflito de merge é quando mudanças concorrentes atingem a **mesma região do código** (ex.: o método de cálculo do Índice de Compatibilidade no serviço de *match*), impedindo a fusão automática e exigindo **resolução manual**.

Prática 1: Sincronizar frequentemente a feature branch com `develop` (via *merge* ou *rebase*). No Git Flow, *features* longas aumentam divergência. No Glink, como **dois devs editaram o mesmo método**, integrar cedo reduz a chance de conflitos grandes e difíceis.

Prática 2: PRs pequenos e focados. Separar refatoração estrutural do ajuste de pesos do algoritmo diminui a superfície de conflito, acelera revisão e evita que dois devs alterem o **mesmo bloco** simultaneamente no serviço de *match*.

Erros comuns (para desconto rápido)

- Tratar conflito como “falha do Git” ou “erro de build” (–C1).
- Sugerir “escrever testes” sem relacionar à prevenção de **conflitos de merge** (prática irrelevante → –C3/C4).
- Falar de *branching* genérico sem mencionar o **Git Flow** (–C2).
- Não **justificar causalmente** como a prática reduz o conflito **no serviço de match** (–C5).
- Mencionar “usar *microservices*” como solução ao conflito de merge (confunde arquitetura com SCM).

Guia de atribuição rápida (checklist)

- Definiu corretamente conflito de merge (0–0,5)
- Citou Git Flow/contexto de branches/PRs (0–0,3)

- Prática 1 pertinente + explicada + contexto (0–0,6)
- Prática 2 pertinente + explicada + contexto (0–0,6)
- Amarra com o caso Glink/serviço de match (0–0,2)

Questão 2 — Gabarito e Rubrica (Peso 3,0)

Enunciado (resumo):

A user story “Como usuário, quero ver meu índice de compatibilidade com outros perfis para entender melhor quem combina comigo” tem gerado interpretações diferentes.

O aluno deve:

1. Identificar **duas lacunas** (ambiguidade ou incompletude);
2. **Reformular** a história com **critérios de aceitação claros**;
3. **Justificar** como essas boas práticas contribuem para a **qualidade e redução de retrabalho**.

Critérios de Correção (com pontuação parcial)

Critério	Pontos	O que vale nota máxima	Parcial	Zero
C1. Identificação das lacunas da user story	**0,6	Identifica duas lacunas relevantes e específicas, como: • Ausência de critério de cálculo ou unidade de medida do índice (percentual, faixa etc.); • Falta de contexto de exibição (onde e quando aparece); • Ambiguidade sobre com quem comparar (todos os perfis? apenas compatíveis?); • Falta de critérios de aceitação/testabilidade .	Identifica apenas uma lacuna relevante ou cita duas mas genéricas (“está incompleta”).	Nenhuma lacuna relevante identificada.
C2. Reformulação da user story	0,6	Apresenta nova história completa, clara e testável , seguindo o formato <i>Como [persona], quero [funcionalidade], para [benefício]</i> , adaptada ao contexto do Glink (ex.: “Como usuário do Glink interessado em RPGs, quero visualizar meu índice de compatibilidade com outros perfis diretamente na tela de descoberta, para decidir com quem iniciar uma conversa.”).	Reformula parcialmente ou usa formato incorreto, mas melhora a clareza.	Não reformula ou apenas repete a história original.
C3. Definição de critérios de aceitação claros e mensuráveis	0,6	Lista pelo menos 3 critérios objetivos , como: • O índice deve ser exibido como percentual (0–100%) ao lado da foto do perfil; • O cálculo deve considerar pelo menos 3 tags em comum; • O carregamento não deve exceder 300 ms; • Deve ser atualizado automaticamente a cada nova interação.	Lista 1–2 critérios, parcialmente mensuráveis ou genéricos (“mostrar corretamente o índice”).	Não apresenta critérios de aceitação.
C4. Justificativa da importância da boa escrita de user stories	0,6	Explica de forma clara e fundamentada que boas user stories melhoram a comunicação entre PO, dev e QA , garantem clareza e testabilidade , reduzem retrabalho e ambiguidade , e contribuem para qualidade e rastreabilidade .	Menciona um ou dois aspectos sem aprofundar (ex.: “melhora a comunicação”).	Sem justificativa ou genérica (“porque fica melhor”).
C5. Conexão com o contexto do Glink	0,3	Faz menção explícita a elementos do caso, como Índice de Compatibilidade Geek , tela de descoberta , PO , time Scrum , feedbacks dos usuários , ou microserviço de match .	Contexto genérico (“no app” / “na equipe”).	Sem conexão com o caso.

Critério	Pontos	O que vale nota máxima	Parcial	Zero
C6. Clareza e coerência da resposta	**0,3	Texto organizado, coeso e bem redigido, facilitando a correção.	Ideia compreensível mas confusa ou com erros menores.	Texto fragmentado ou incoerente.

Total: $0,6 + 0,6 + 0,6 + 0,6 + 0,3 = 3,0$ pontos

Exemplos de Conteúdo Esperado (resumo orientador)

Lacunas (C1)

- A história **não especifica o contexto** (“onde” o índice aparece – na tela de descoberta, no perfil, no chat?).
- Falta um **critério de cálculo** ou **unidade de medida** (é um percentual? usa quais tags?).
- **Critérios de aceitação** inexistentes — o QA não sabe o que validar.
- Ambiguidade sobre o **usuário-alvo** (todos os usuários ou apenas compatíveis?).

Reformulação adequada (C2)

Como usuário do Glink interessado em cultura geek,
quero visualizar meu índice de compatibilidade (em %) com outros perfis na tela de descoberta,
para decidir de forma rápida com quem iniciar uma conversa baseada em interesses em comum.

Critérios de aceitação (C3)

1. O índice de compatibilidade deve aparecer como percentual de 0–100% logo abaixo do nome do perfil sugerido.
2. O cálculo deve considerar apenas tags de interesse compartilhadas em pelo menos 3 categorias.
3. O tempo de exibição não pode exceder **300 ms** após o carregamento do perfil.
4. Ao atualizar tags do usuário, o índice deve ser recalculado automaticamente.
5. Testes automatizados devem validar o valor exibido conforme o mock de referência.

Justificativa e relação com qualidade (C4, C5)

- Histórias claras e mensuráveis permitem **entendimento comum** entre PO, desenvolvedores e QA.
- Critérios de aceitação tornam o requisito **testável** e **verificável**, garantindo **rastreabilidade**.
- No contexto do Glink, evitam divergências sobre **como calcular o índice de compatibilidade, onde exibir e como medir desempenho**, reduzindo retrabalho e aumentando a **qualidade percebida**.

Exemplo de resposta-modelo (concisa e correta)

A história apresenta duas lacunas: (1) não define **como o índice é calculado ou exibido** (percentual? em qual tela?), e (2) **falta de critérios de aceitação** que tornem o requisito testável.

Reformulando:

Como usuário do Glink interessado em RPGs, **quero** visualizar meu índice de compatibilidade em percentual com outros perfis diretamente na tela de descoberta, **para** escolher com quem iniciar uma conversa.

Critérios de aceitação:

- (1) O índice deve aparecer como percentual (0–100%);
- (2) O cálculo deve considerar pelo menos 3 tags em comum;
- (3) O carregamento deve ocorrer em até 300 ms.

Essa reformulação garante **clareza e testabilidade**, reduz **retrabalho** e melhora a **qualidade do software**, pois evita que diferentes membros do time interpretem o requisito de forma divergente.

Erros comuns (para desconto rápido)

- Reformular sem seguir o formato *Como/Quero/Para* (-C2).

- Citar critérios genéricos (“deve funcionar corretamente”).
- Dizer apenas “está incompleta” sem apontar lacunas específicas (–C1).
- Ignorar o contexto Glink (–C5).
- Explicar o que é user story sem responder ao pedido da questão.

Guia de atribuição rápida (checklist)

- Identificou duas lacunas concretas (0–0,8)
- Reformulou de forma clara e no formato correto (0–0,8)
- Indicou critérios de aceitação mensuráveis (0–0,8)
- Justificou relação com qualidade/testabilidade (0–0,6)
- Conectou ao caso Glink (0–0,3)
- Redação coesa e compreensível (0–0,2)

Questão 3 — Gabarito e Rubrica (Peso 5,0)

Enunciado (resumo):

O tempo de resposta do cálculo do “Índice de Compatibilidade Geek” subiu de **250 ms** para **600 ms**, ultrapassando o requisito não funcional definido (≤ 300 ms).

O Scrum Master propôs uma retrospectiva técnica, e o PO quer entender se o problema decorre de falha de processo, má especificação ou decisões arquiteturais.

O aluno deve responder:

1. Fatores do modelo de processo (Scrum + Git Flow) que podem ter contribuído;
2. Ações de melhoria de processo e qualidade (técnicas e gerenciais);
3. Estratégias para garantir verificação contínua de requisitos não funcionais (desempenho).

Critérios de Correção (com pontuação parcial)

Critério	Pontos	O que vale nota máxima	Parcial	Zero
C1. Identificação de fatores de processo relacionados ao problema (Scrum + Git Flow)	1,0	Cita e explica 2 ou mais fatores específicos que podem ter levado ao aumento da latência, como: <ul style="list-style-type: none"> • Falhas no refinamento da <i>user story</i> (não detalhou requisito de desempenho); • Falta de <i>Definition of Done</i> com testes de performance; • Integração tardia no Git Flow (<i>feature branches</i> long-lived); • Falta de inspeção técnica nas <i>reviews</i>; • Planejamento de sprint sem incluir critérios não funcionais. 	Identifica apenas um fator relevante, ou mais de um mas com explicações superficiais.	Resposta genérica ou incorreta (“foi erro do desenvolvedor”).
C2. Proposição de ações de melhoria técnica e de processo	1,5	Apresenta ações complementares e específicas para prevenir recorrência, tais como: <ul style="list-style-type: none"> • Adotar testes automatizados de desempenho na pipeline CI/CD; • Revisar <i>Definition of Done</i> para incluir requisitos não funcionais; • Revisões de arquitetura (<i>Architecture Review Board</i> ou <i>tech debt grooming</i>); • Monitoramento contínuo de latência; • Revisão de estratégias de branching e integração contínua; • Sprint de estabilização técnica. 	Apresenta 1–2 ações genéricas (ex.: “testar melhor”), sem relação clara com o problema de desempenho.	Ações irrelevantes ou desconectadas do caso.
C3. Garantia contínua de verificação de	1,0	Explica como operacionalizar a verificação contínua de desempenho, sugerindo práticas como: <ul style="list-style-type: none"> • Métricas 	Menciona monitoramento ou testes, mas sem	Não apresenta mecanismos concretos de

Critério	Pontos	O que vale nota máxima	Parcial	Zero
requisitos não funcionais		automáticas integradas ao pipeline (latência, throughput, consumo de CPU); • Ambientes de staging com testes de carga; • Uso de ferramentas (JMeter, Gatling, Lighthouse); • Alertas automáticos e limites (ex.: Prometheus, Grafana, Sentry); • Revisão sistemática de NFRs a cada sprint.	detalhar integração contínua ou sem explicitar periodicidade.	verificação contínua.
C4. Justificativa bem embasada e causal	1,0	Argumenta logicamente como as causas e ações propostas se relacionam (ex.: “falta de <i>DoD</i> levou à ausência de testes automatizados, que permitiu a regressão de desempenho não detectada antes do deploy”). Mostra domínio da causalidade entre processo e qualidade .	Justifica parcialmente ou apenas enumera itens.	Sem justificativa causal.
C5. Conexão com o contexto do Glink	0,3	Faz referência explícita ao caso — menciona serviço de match, cálculo do índice, latência de 600 ms, Scrum, Git Flow, PO/Scrum Master, etc.	Alusão genérica ao app.	Sem relação contextual.
C6. Clareza, estrutura e profundidade da análise	0,2	Texto bem estruturado (introdução, análise, conclusão); argumentação técnica coerente.	Compreensível, mas superficial ou com falhas de organização.	Incoerente ou confuso.

Total: $1,0 + 1,5 + 1,0 + 1,0 + 0,3 + 0,2 = 5,0$ pontos

Exemplos de Conteúdo Esperado (resumo orientador)

(C1) Fatores de processo que contribuíram

- **Falha no refinamento:** o requisito de desempenho (≤ 300 ms) não foi detalhado nem estimado na *planning*.
- **Ausência de “Definition of Done” técnico:** equipe considerou concluído sem validar performance.
- **Branches long-lived:** atrasaram integração de otimizações no algoritmo de compatibilidade.
- **Falta de revisões de arquitetura:** não houve revisão da complexidade algorítmica antes do deploy.
- **Ausência de testes automatizados de carga:** a regressão passou despercebida.

(C2) Ações de melhoria

- **Incluir NFRs na Definition of Done** e no backlog, para que cada *story* relevante possua métricas de performance.
- **Automatizar testes de desempenho** (ex.: JMeter) e integrá-los à pipeline CI/CD.
- **Implantar monitoramento contínuo (Prometheus + Grafana)** para medir latência em ambiente de staging e produção.
- **Planejar revisões de arquitetura e de código focadas em desempenho.**
- **Usar integração contínua com builds menores** (reduz tempo entre merges e feedback técnico).
- **Sprint técnica para refatorar o serviço de match**, eliminando gargalos.

(C3) Garantia contínua de verificação de NFRs

- Configurar *pipelines* com **gatilhos automáticos de performance regression** (ex.: falhar build se >300 ms).
- Adotar **telemetria contínua** em produção (tracing distribuído) para detectar variações de tempo de resposta.

- Revisar NFRs a cada sprint review/retrospectiva, atualizando metas conforme feedback real.
- Criar um dashboard de qualidade visível à equipe com KPIs de latência, disponibilidade e throughput.

(C4) Justificativa (exemplo de causalidade)

A ausência de testes automatizados de desempenho e de um *Definition of Done* técnico permitiu que alterações no cálculo de compatibilidade fossem entregues sem validação do tempo de resposta. Somado ao uso de *feature branches* long-lived, que atrasou a integração e dificultou a detecção precoce da regressão, isso resultou na elevação da latência de 250 ms para 600 ms. Incorporar práticas de verificação contínua e automação de métricas na pipeline evita que esse tipo de falha chegue à produção, reforçando a **qualidade sistêmica** do Glink.

Exemplo de resposta-modelo (concisa e correta)

O aumento da latência do serviço de *match* pode ter sido causado por **fallhas de processo**: ausência de critérios de desempenho na *Definition of Done*, *feature branches* long-lived no Git Flow e falta de testes automatizados de carga durante as *reviews*.

Ações de melhoria: incluir requisitos não funcionais nos *sprint plannings*, automatizar testes de performance na pipeline de CI/CD, revisar periodicamente a arquitetura e adotar monitoramento contínuo com alertas de latência. Para garantir a **verificação contínua dos NFRs**, o time deve criar uma pipeline com limiares automáticos de rejeição (>300 ms), usar ferramentas como JMeter e Prometheus para medir desempenho e incluir esses resultados nas *retrospectives*.

Isso assegura que cada incremento de software mantenha o desempenho previsto, fortalecendo a qualidade e a confiabilidade do produto Glink.

Erros comuns (para desconto rápido)

- Culpar “infraestrutura” sem citar fatores de processo (–C1).
- Sugerir ações genéricas (“testar mais”, “otimizar código”) sem explicar como implementá-las (–C2).
- Não relacionar o caso ao Scrum ou Git Flow (–C1, –C5).
- Ignorar o requisito não funcional ou não citar métricas (–C3).
- Explicação linear sem relação causa–efeito (–C4).
- Falta de contextualização com o Glink (–C5).

Guia de atribuição rápida (checklist)

- Identificou fatores de processo Scrum/Git Flow (0–1,2)
- Propôs melhorias técnicas e gerenciais específicas (0–1,5)
- Demonstrou mecanismos de verificação contínua de NFRs (0–1,2)
- Justificou causalmente as relações (0–0,8)
- Conectou explicitamente ao caso Glink (0–0,2)
- Estrutura e clareza textual (0–0,1)