



Desafio Técnico — Pessoa Desenvolvedora Android Sênior

GOK | Inovação Digital

Visão Geral do Desafio

Você está sendo entrevistado para uma posição **Android Sênior** em uma consultoria que atende projetos de alta criticidade transacional — incluindo SDKs financeiros, apps de pagamento e sistemas bancários mobile.

O Tech Lead vai simular um **cenário real de um dos projetos ativos**.

Atenção: Este não é um exercício acadêmico. O problema apresentado representa um bug com impacto financeiro direto em produção. A profundidade da sua resposta será proporcional à senioridade demonstrada.

O Problema

"Temos um app Android de pagamentos que processa operações Pix. O usuário inicia uma transferência, o app chama nossa API via BFF, e a operação é confirmada ou rejeitada. O problema: estamos tendo um aumento de reclamações de usuários dizendo que transferências aparecem como 'em processamento' por tempo indeterminado — e em alguns casos o usuário tenta novamente e a operação é processada em duplicata."

Sua Tarefa

Projete e implemente — ou descreva com profundidade técnica suficiente para que um dev júnior consiga implementar — uma solução Android que:

- 1. Realize a chamada de transferência Pix com **resiliência** adequada
- 2. Gerencie corretamente o **estado da operação** do ponto de vista do cliente
- 3. **Previna duplicatas no lado cliente**, sem depender exclusivamente do backend
- 4. Garanta que a **UI reflita sempre um estado consistente**, mesmo em falha de rede, timeout ou resposta ambígua do servidor

Restrições Técnicas

Requisito	Detalhe
Linguagem	Kotlin com Coroutines e Flow
Arquitetura	Clean Architecture com separação clara de camadas
Estado da UI	Exposto via <code>StateFlow</code> no ViewModel
Modelagem de estados	Máquina de estados explícita — sem booleanos soltos ou flags implícitas
Escopo de coroutines	<code>GlobalScope</code> é proibido
Testabilidade	Contratos de interfaces definidos; localização dos testes indicada
Camada de rede	Retrofit + OkHttp

Entrada e Saída Esperadas

Entrada — dados da transferência

```
data class PixTransferRequest(  
    val destinationKey: String,  
    val amount: BigDecimal,  
    val description: String?,  
    val idempotencyKey: String // gerado pelo cliente antes do envio  
)
```

Saída — estado exposto para a UI

```
sealed class PixTransferUiState {  
    object Idle : PixTransferUiState()  
    object Loading : PixTransferUiState()  
    data class Success(val receipt: TransferReceipt) : PixTransferUiState()  
    data class Error(val type: PixTransferError) : PixTransferUiState()  
    object AwaitingConfirmation : PixTransferUiState() // estado ambíguo – resp  
}  
  
sealed class PixTransferError {  
    object NetworkTimeout : PixTransferError()  
    object InsufficientFunds : PixTransferError()  
    object DuplicateOperation : PixTransferError()  
    data class Unknown(val message: String) : PixTransferError()  
}
```

Atenção: O estado `AwaitingConfirmation` é **intencional e obrigatório**. Candidatos que o omitirem ou o tratarem como erro genérico serão questionados diretamente sobre as implicações para o negócio.

O Que Deve Ser Entregue

Código 100% compilável não é obrigatório. O esperado é:

- [] **Diagrama mental ou textual** da máquina de estados
- [] **Estrutura das camadas** — quais classes existem, em qual camada e qual responsabilidade
- [] **Implementação do ViewModel** com `StateFlow` e tratamento dos estados
- [] **Implementação ou contrato do Use Case** de transferência
- [] **Estratégia de idempotência** no lado cliente
- [] **Estratégia de retry** — com justificativa clara de quando *não* fazer retry
- [] **Comportamento da UI** em cada estado, especialmente em `AwaitingConfirmation`

Edge Cases — O Que Você Deve Endereçar

Cenário	Comportamento esperado
Timeout sem resposta do servidor	App entra em <code>AwaitingConfirmation</code> . Nenhuma nova tentativa automática. Usuário é informado e tem opção de consultar o status
Usuário fecha o app durante a operação	Estado é persistido via Room. Ao reabrir, a operação em curso é recuperada e exibida corretamente
Duplo toque no botão "Transferir"	Segunda chamada ignorada no ViewModel — não chega ao Use Case
Resposta HTTP 5xx	Sem retry automático. Em operações financeiras, 5xx não é idempotente sem confirmação explícita
Resposta HTTP 409 (Conflict)	Mapeado para <code>PixTransferError.DuplicateOperation</code> com mensagem clara ao usuário
Perda de conexão durante polling	Coroutines canceladas corretamente, sem vazamentos
Quando regenerar a <code>idempotencyKey</code> ?	Apenas quando o usuário inicia explicitamente uma nova operação — nunca em retry automático da mesma tentativa

Critérios de Avaliação

❑ Eliminatórios — ausência ou erro grave descarta o candidato

Critério	O que é avaliado
Máquina de estados explícita	Estados modelados de forma clara, sem booleanos implícitos ou flags acopladas
Idempotência no cliente	<code>idempotencyKey</code> gerada antes da chamada, mantida na mesma tentativa, nunca regenerada em retry automático
Ausência de retry automático em timeout financeiro	Compreensão de que retry cego em operações não idempotentes é a causa raiz do bug
<code>AwaitingConfirmation</code> tratado com seriedade	Não descartado como erro genérico — implicações para o usuário e para o negócio claramente articuladas

❑ Diferenciadores — separa sênior de pleno

Critério	O que é avaliado
Separação de camadas	ViewModel não chama Retrofit; Use Case não conhece ViewModel; Repository é abstrato
Persistência de estado	Considera interrupção do processo e reabertura do app — Room ou equivalente utilizado
Testabilidade explicitada	Localização dos testes por camada e dependências mockadas com MockK
Cancelamento de coroutines	Comportamento correto quando o ViewModel é destruído durante a operação

Critério	O que é avaliado
UX no estado ambíguo	Proposta clara: mensagem informativa, opção de consultar status, ausência do botão “Tentar novamente”

□ Excelência — indica candidato acima do esperado

Critério	O que é avaliado
Polling ou webhook para <code>AwaitingConfirmation</code>	Consulta de status com backoff exponencial, cancelamento correto e timeout máximo definido
Segurança	<code>idempotencyKey</code> não previsível (UUID v4); estado ambíguo não reexibido como erro para evitar nova tentativa do usuário
Observabilidade	Log estruturado dos estados da operação para auditoria — especialmente relevante em contexto fintech
Trade-offs explicitados	Candidato menciona o que <i>não</i> está fazendo e por quê, demonstrando consciência das decisões de arquitetura

Raciocínio Esperado — Padrão de Profundidade

Dica estratégica: Este é o nível de pensamento que o Tech Lead espera ouvir. Não é a única resposta correta — é o padrão mínimo de profundidade para uma posição sênior.

Passo 1 — Identificar o problema real antes de codar

"O bug de duplicata não é um bug de rede — é um bug de estado. O app não distingue 'a operação falhou' de 'a operação foi enviada, mas não recebi confirmação'. Quando o usuário tenta novamente, o app trata os dois casos da mesma forma. Preciso primeiro modelar os estados possíveis da operação."

Passo 2 — Definir a máquina de estados

"Uma transferência Pix, do ponto de vista do cliente, tem os seguintes estados: Idle → Loading → Success, Error ou AwaitingConfirmation. O estado crítico é o terceiro — é quando enviei a requisição, mas não sei o que aconteceu com ela. Esse estado deve persistir, não ser apagado."

Passo 3 — Decidir sobre retry

"Não farei retry automático em timeout. Se o servidor recebeu a requisição mas o ACK se perdeu, um retry com a mesma idempotencyKey pode ser seguro — mas somente se o backend implementar idempotência corretamente. Como não tenho essa garantia, deixo o usuário decidir. Retry automático só é seguro quando tenho certeza de que a operação não foi processada."

Passo 4 — Estratégia de idempotência

"A idempotencyKey é gerada uma única vez, antes de qualquer tentativa, e fica associada a essa intenção de transferência. Se o usuário cancelar e iniciar uma nova transferência, gero uma nova chave. Se tentar novamente após um timeout, uso a mesma chave — isso permite que o backend detecte a duplicata e retorne o resultado original."

Passo 5 — Estrutura das camadas

"O ViewModel expõe `StateFlow<PixTransferUiState>` , coordena o Use Case e protege contra double-tap com um flag de operação em andamento ou via `stateIn` . O Use Case contém a lógica de negócio — chama o Repository, trata os retornos e persiste o estado no banco local antes de retornar ao ViewModel. O Repository abstrai a fonte de dados: API ou banco local, dependendo do estado atual. A camada de dados mantém `RemoteDataSource` (Retrofit) e `LocalDataSource` (Room) separados."

Passo 6 — UX para `AwaitingConfirmation`

"Quando entro em `AwaitingConfirmation` , exibio uma tela específica — não uma tela de erro. O texto seria: 'Sua transferência está sendo processada. Isso pode levar alguns instantes. Consulte seu extrato para confirmar.' Ofereço um botão 'Verificar status', que dispara uma consulta manual ao backend. Não ofereço 'Tentar novamente' — isso aumentaria o risco de duplicata do ponto de vista do usuário."

Passo 7 — Ciclo de vida e persistência

"Se o usuário fechar o app enquanto está em `Loading` ou `AwaitingConfirmation` , persisto esse estado no Room. Ao reabrir, faço uma consulta de status automática usando a `idempotencyKey` salva. Se o servidor confirmar que a operação foi processada, exibio `Success` . Se negar, exibio `Error` . Se ainda estiver processando, mantenho `AwaitingConfirmation` com atualização de status."

Dica estratégica: O candidato que chega ao Passo 7 de forma natural — sem ser guiado — demonstra maturidade real em ambientes transacionais críticos. É esse nível de pensamento que a GOK precisa em produção.

Career Agent **PRO**

A preparação perfeita para cada vaga de emprego. Inteligência aplicada ao seu contexto e à empresa que você quer entrar.

 www.career-agent-pro.com

 support@main.career-agent-pro.com

© 2026 Career Agent PRO. Todos os direitos reservados.

PREPARAÇÃO · INTELIGÊNCIA · RESULTADO