Strecht Simulados · Guia de preparação



Guia de preparação DVA-C02

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.

4 capítulos por domínio	**28** questões comentadas	**72%** 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 DVA-C02: 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 72%).

Mapa do exame

Cada prova do curso tem 65 questões e 130 minutos, no formato do exame. Referência de aprovação: 72%.

Domínio (capítulo)	Peso	Neste guia
1. Desenvolvimento com os serviços da AWS	—	18 conceitos · 14 pegadinhas
2. Implantação	—	17 conceitos · 14 pegadinhas
3. Segurança	—	18 conceitos · 14 pegadinhas
4. Troubleshooting e Otimização	—	18 conceitos · 13 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 · Desenvolvimento com os serviços da AWS

Uma empresa está desenvolvendo um backend serverless em AWS Lambda para processar requisições HTTP recebidas via Amazon API Gateway. Em produção, o tempo de resposta da API está variando bastante, principalmente nas primeiras requisições após períodos de inatividade. O desenvolvedor quer reduzir essa variabilidade e melhorar a previsibilidade da latência, mantendo o modelo serverless. Qual solução é MAIS adequada para esse cenário?

A. Usar Provisioned concurrency na função crítica e ajuste de memória para reduzir o tempo de execução
B. Migrar o backend para uma aplicação em instâncias Amazon EC2 por trás de um Application Load Balancer para eliminar cold starts
C. Aumentar apenas o timeout da função Lambda e configurar o API Gateway para usar cache em todas as rotas
D. Habilitar DLQ (Dead Letter Queue) para a função Lambda e diminuir o timeout

D2 · Desenvolvimento com os serviços da AWS

Um desenvolvedor está criando uma aplicação que armazena e processa objetos no Amazon S3. A aplicação deverá executar diferentes ações dependendo do tipo de evento gerado no bucket (por exemplo, criação de objeto com um prefixo específico, exclusão lógica via versão, ou restauração de objeto arquivado). O desenvolvedor quer minimizar o código de roteamento dentro da lógica da aplicação e deseja que o próprio serviço de mensageria faça o fan-out adequado para diferentes consumidores. Qual abordagem é a mais adequada para esse cenário?

A. Configurar o bucket S3 para enviar eventos diretamente para uma única fila Amazon SQS e implementar toda a lógica de roteamento dentro da aplicação consumidora.
B. Habilitar o Amazon EventBridge para ler diretamente os objetos do S3 e encaminhá-los automaticamente para diferentes APIs HTTP, sem necessidade de configuração adicional.
C. Usar eventos do S3 publicados num tópico SNS que distribui às filas SQS de cada componente com filtros de assinatura
D. Configurar o bucket S3 para invocar diretamente uma função AWS Lambda única, que então envia mensagens.

D3 · Desenvolvimento com os serviços da AWS

Um time de desenvolvimento está construindo uma aplicação serverless que receberá eventos de diversos produtores (IoT, serviços internos e aplicações de terceiros) por meio do Amazon EventBridge. Os desenvolvedores precisam testar em ambiente de desenvolvimento novas regras de roteamento e transformações de eventos sem impactar a produção, mas também desejam reutilizar o máximo possível da infraestrutura já criada, incluindo esquemas, barramentos e regras. Qual é a abordagem MAIS apropriada para atender a esses requisitos?

A. Criar um novo event bus personalizado em produção, apontar temporariamente os produtores para esse novo bus para testes e, após validar, mover as regras para o event bus original
B. Utilizar AWS SAM ou AWS CDK para definir a infraestrutura como código e criar stacks separados para desenvolvimento e produção, cada um com seu próprio event bus, regras, permissões e targets
C. Habilitar o recurso de archive e replay do EventBridge no event bus de produção e usar o replay para testar novas regras diretamente na conta de produção
D. Criar regras condicionais no event bus de produção que enviem eventos de teste para targets de desenvolvimento e eventos reais para targets de produção, diferenciando-os só por um campo de payload

D4 · Implantação

Um desenvolvedor está configurando um pipeline de implantação no AWS CodePipeline para uma aplicação web em contêineres executada no Amazon ECS com Fargate. O código-fonte está no AWS CodeCommit, os testes automatizados rodam no AWS CodeBuild e a imagem é armazenada no Amazon ECR. O desenvolvedor quer garantir que, em caso de falha na nova versão, o tráfego volte automaticamente para a versão anterior, minimizando downtime e risco de erro humano. Qual é a abordagem MAIS adequada para atender a esse requisito usando serviços gerenciados da AWS?

A. Configurar o AWS CodePipeline com estágios de Source (CodeCommit), Build (CodeBuild) e Deploy usando o AWS CodeDeploy com tipo de implantação Blue/Green para ECS, habilitando validações pós-implantação e rollback automático baseado em falhas em alarmes do Amazon CloudWatch.
B. Configurar o AWS CodePipeline com estágios de Source (CodeCommit), Build (CodeBuild) e Deploy (ECS) usando implantação do tipo Rolling Update, e fazer rollback manual caso a nova versão apresente falhas.
C. Configurar o AWS CodePipeline com estágios de Source (CodeCommit), Build (CodeBuild) e Deploy usando AWS CloudFormation para atualizar o serviço ECS com política de implantação All-at-once, e usar Change Sets para reverter à mão em caso de falhas.
D. Configurar o AWS CodePipeline com estágios de Source (CodeCommit) e Build (CodeBuild) só, e agendar um script em uma instância Amazon EC2 via cron para fazer o deploy no ECS com a AWS CLI e reverter à mão em caso de falhas.

D5 · Implantação

Um time de desenvolvimento deseja automatizar a implantação de uma aplicação em contêiner no Amazon ECS (usando Fargate) sempre que um novo commit é feito no branch 'main' de um repositório GitHub. A equipe quer evitar gerenciar servidores de build e prefere usar serviços totalmente gerenciados da AWS. Qual abordagem atende a esses requisitos com o menor esforço operacional?

A. Criar instâncias EC2 dedicadas para build, configurar webhooks do GitHub para disparar scripts de shell que executam docker build, docker push para o Amazon ECR e chamar à mão a atualização do serviço ECS
B. Configurar um pipeline no AWS CodePipeline integrado ao GitHub como fonte, usar AWS CodeBuild para build da imagem Docker e enviar para o Amazon ECR, seguido de uma ação de implantação no Amazon ECS
C. Usar o AWS CodeCommit como repositório de código-fonte, migrando o código do GitHub, e configurar um pipeline com CodePipeline e CodeDeploy para implantação em ECS
D. Criar um job do AWS Batch que é disparado por eventos do Amazon EventBridge ao detectar novos commits no GitHub, para construir a imagem e atualizar o serviço ECS

D6 · Implantação

Você está trabalhando em um microserviço crítico implantado em AWS Elastic Beanstalk usando a plataforma Docker. Em ambiente de produção, a aplicação precisa: (1) aplicar migrações de banco de dados automaticamente em cada novo deploy; (2) evitar tempo de inatividade durante a implantação; (3) permitir rollback rápido para a versão anterior caso uma migração falhe. Qual é a abordagem MAIS adequada para atender a esses requisitos, seguindo boas práticas de implantação na AWS?

A. Configurar um script de migração diretamente no Dockerfile para ser executado no ENTRYPOINT, de forma que cada container execute as migrações ao iniciar, utilizando o tipo de implantação padrão do Elastic Beanstalk.
B. Criar um script de migração no diretório '.platform/hooks/postdeploy' e usar uma implantação 'Rolling with additional batch' no Elastic Beanstalk, combinada com versionamento de aplicações para permitir rollback.
C. Habilitar 'Immutable deployment' no Elastic Beanstalk e configurar a aplicação para rodar as migrações apenas no primeiro container que iniciar na nova versão, utilizando um lock em banco para evitar concorrência.
D. Armazenar o script de migração em um bucket S3 e usar um Lambda acionado por upload de nova versão de aplicação no S3 para rodar as migrações, enquanto o Elastic Beanstalk utiliza implantação 'All at once' para reduzir o tempo total de deploy.

D7 · Segurança

Uma aplicação de processamento de pedidos em execução no Amazon ECS (Fargate) precisa ler e gravar dados em uma tabela do Amazon DynamoDB. Os requisitos de segurança da empresa determinam que nenhuma chave de acesso da AWS seja armazenada no código, em variáveis de ambiente ou em arquivos de configuração. Além disso, a solução deve seguir o princípio do menor privilégio e ser simples de gerenciar. Como o desenvolvedor deve configurar o acesso da aplicação ao DynamoDB?

A. Anexar uma política gerenciada 'AmazonDynamoDBFullAccess' diretamente ao usuário raiz da conta e configurar o ECS para assumir automaticamente essas permissões.
B. Criar um usuário do IAM com política de acesso ao DynamoDB, gerar chaves de acesso e armazená-las criptografadas no AWS Systems Manager Parameter Store para a aplicação ler em tempo de execução.
C. Criar uma função do IAM (IAM Role) para a tarefa do ECS com uma política que permita somente as ações necessárias no DynamoDB e associar essa função à definição da tarefa do Fargate.
D. Criar uma função do IAM para o serviço Amazon ECS, anexar uma política com permissões amplas ao DynamoDB e compartilhar essa função entre todas as aplicações em ECS para simplificar o gerenciamento.

D8 · Segurança

Uma empresa precisa armazenar segredos de aplicação (como chaves de API e strings de conexão) usados por funções AWS Lambda. Os segredos devem ser criptografados em repouso com chaves gerenciadas pela AWS e a aplicação deve conseguir recuperá-los em tempo de execução com baixa latência e controle de acesso detalhado via IAM. Qual serviço da AWS atende melhor a esses requisitos?

A. AWS Secrets Manager
B. AWS KMS
C. AWS Systems Manager Parameter Store (parâmetros seguros)
D. Amazon S3 com criptografia do lado do servidor (SSE-S3)

D9 · Segurança

Um time está criando um pipeline de CI/CD no AWS CodePipeline para implantar funções AWS Lambda que acessam dados sensíveis em uma tabela Amazon DynamoDB. A equipe de segurança exige que: (1) o código nunca seja armazenado em texto puro dentro do pipeline, (2) segredos de conexão sejam rotacionados automaticamente e (3) as funções Lambda utilizem credenciais temporárias ao acessar a tabela. Qual combinação de ações atende melhor a esses requisitos seguindo boas práticas de segurança na AWS?

A. Armazenar o código-fonte em um repositório GitHub público, usar parâmetros SecureString do AWS Systems Manager Parameter Store sem KMS e conceder permissões ao DynamoDB por meio de chaves de acesso permanentes armazenadas no Parameter Store.
B. Armazenar o código-fonte em um repositório privado do AWS CodeCommit, usar AWS Secrets Manager para armazenar e rotacionar segredos, e configurar um role do IAM para a função Lambda com permissões mínimas necessárias para acessar o DynamoDB por meio de credenciais temporárias.
C. Armazenar variáveis de ambiente criptografadas diretamente no CodePipeline, usar AWS KMS para criptografar o código-fonte e configurar a função Lambda com chaves de acesso estáticas de um usuário IAM.
D. Compactar o código-fonte em um arquivo ZIP protegido por senha e armazená-lo em um bucket S3 público, armazenar segredos em variáveis de ambiente da função Lambda sem criptografia e anexar a política AmazonDynamoDBFullAccess diretamente à função.

D10 · Troubleshooting e Otimização

Uma função AWS Lambda escrita em Python, acionada por eventos do Amazon SQS, está apresentando aumento no tempo de processamento e consumo de memória após algumas semanas em produção. O desenvolvedor já confirmou que não há aumento significativo no volume de mensagens nem mudanças no código de negócio. Logs do CloudWatch mostram que o tempo de inicialização (init) da função está estável, mas o tempo de execução (invoke) cresce gradualmente ao longo da vida útil do container da função. Qual é a ação MAIS apropriada para identificar e mitigar a causa raiz desse problema?

A. Reduzir o tamanho do lote (batch size) do evento do SQS para apenas uma mensagem por invocação, garantindo que cada execução processe menos dados e, assim, consuma menos memória.
B. Alterar a função Lambda para ser executada em uma VPC privada e habilitar o Elastic Network Interface (ENI) dedicado, garantindo mais estabilidade na rede e reduzindo o tempo de execução.
C. Ativar o AWS X-Ray para a função Lambda e revisar o código em busca de estruturas de dados que crescem indefinidamente entre invocações, garantindo que variáveis globais sejam limpas ou recriadas a cada execução.
D. Aumentar a memória alocada para a função Lambda para o dobro do valor atual e configurar o tempo máximo de execução (timeout) para o valor máximo permitido, reduzindo a chance de falhas por falta de recursos.

D11 · Troubleshooting e Otimização

Uma aplicação em contêiner no Amazon ECS (com Launch Type Fargate) está apresentando picos de latência em uma API crítica durante horários de pico. As métricas do Amazon CloudWatch mostram que o CPUUtilization das tasks está consistentemente em 80-90% e o Application Load Balancer possui TargetResponseTime elevado apenas nesses períodos. O time já aumentou o número mínimo de tasks no serviço, mas o problema persiste e o custo aumentou significativamente. Como o desenvolvedor pode otimizar a performance e o custo de forma mais eficiente?

A. Habilitar o AWS X-Ray na aplicação para identificar gargalos de código, otimizar trechos mais lentos e reduzir o tempo de processamento por requisição antes de alterar a infraestrutura.
B. Configurar o Application Load Balancer para usar só o algoritmo de roteamento 'least outstanding requests' para distribuir melhor as requisições entre as tasks existentes, sem mudar a aplicação.
C. Criar uma fila Amazon SQS entre o Application Load Balancer e o serviço ECS para desacoplar as requisições síncronas, reduzindo a latência percebida pelo cliente.
D. Aumentar a capacidade vCPU e memória de cada task Fargate e desligar o Auto Scaling do serviço ECS para manter uma quantidade fixa de tasks mais potentes.

D12 · Troubleshooting e Otimização

Um desenvolvedor está investigando um aumento súbito de erros HTTP 5xx em um serviço backend exposto por meio de um Application Load Balancer (ALB). O serviço é executado em instâncias Amazon EC2 em um Auto Scaling group. Os logs de acesso do ALB mostram vários erros 502 e 504, mas os logs da aplicação nas instâncias não registram exceções correspondentes. O desenvolvedor já verificou que não há aumento de latência na aplicação nem uso excessivo de CPU/memória nas instâncias. Qual é a maneira MAIS eficaz de identificar a causa raiz do problema para otimizar a estabilidade do serviço?

A. Aumentar o número mínimo de instâncias no Auto Scaling group e configurar health checks mais agressivos no ALB para reduzir a probabilidade de erros 5xx
B. Configurar métricas personalizadas no Amazon CloudWatch para monitorar o uso de CPU, memória e disco e criar alarmes para cada pico de utilização nas instâncias
C. Habilitar logs de fluxo do VPC (VPC Flow Logs) para analisar conexões rejeitadas e pacotes descartados, correlacionando-os à mão com os timestamps dos erros 5xx no ALB
D. Ativar o AWS X-Ray para a aplicação, instrumentar o código para rastreamento de ponta a ponta e analisar os traces detalhados das requisições passando pelo ALB até o backend

01

CAPÍTULO 1

Desenvolvimento com os serviços da AWS

É o domínio de maior peso, 32% do conteúdo pontuado no guia oficial do DVA-C02, e cobra o código do dia a dia: Lambda (concorrência, cold start, inicialização fora do handler, Layers, imagem de contêiner), mensageria (SQS, SNS, EventBridge, Kinesis e Step Functions), API Gateway, DynamoDB e S3. Preciso assumir que eventos assíncronos chegam pelo menos uma vez e por isso o consumidor tem de ser idempotente, com DLQ para mensagens problemáticas e visibility timeout maior que o tempo de processamento. A prova descreve uma arquitetura serverless com um sintoma (duplicidade, perda de mensagens, cold start, erro 429, timeout) e pede o ajuste certo com menor esforço. Onde mais erro é confundir SQS (fila), SNS (fan-out), EventBridge (roteamento por regras) e Step Functions (orquestração com estado), e achar que aumentar timeout ou memória resolve cold start.

Neste capítulo

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

Tópicos-chave

Entrega pelo menos uma vez e idempotência

S3, SNS, EventBridge, SQS e invocações assíncronas de Lambda podem entregar o mesmo evento mais de uma vez, atrasado ou fora de ordem. O consumidor deve ser idempotente: registra o ID único da operação (por exemplo, no DynamoDB com escrita condicional) antes do efeito e ignora repetições. Ajustar timeout ou visibility timeout não elimina duplicatas.

SQS padrão x FIFO

A fila padrão tem alto throughput, entrega pelo menos uma vez e ordenação best-effort. A FIFO garante ordem por MessageGroupId e deduplicação (inclusive content-based deduplication, com corpo determinístico), mas tem limites de throughput mais restritos. FIFO não é a escolha para ganhar vazão.

Visibility timeout, maxReceiveCount e DLQ

O visibility timeout deve ser maior que o tempo máximo de processamento (e que o timeout da Lambda); se for menor, a mensagem reaparece e é processada em duplicidade. Uma DLQ associada com maxReceiveCount desvia mensagens que falham repetidamente para análise. Como referência rápida, o visibility timeout deve ser pelo menos 6x o timeout da função quando a Lambda consome a fila pelo mapeamento de origem. Em fluxo simples evento para função, como SNS invocando Lambda, a DLQ nativa da própria Lambda guarda os eventos que esgotaram as tentativas.

Long polling e lotes no SQS

Long polling (WaitTimeSeconds maior que zero, até 20 segundos) reduz respostas vazias e chamadas ReceiveMessage desnecessárias. Combinado com MaxNumberOfMessages, cada chamada retorna até 10 mensagens, aumentando throughput e reduzindo custo.

Event source mapping da Lambda

Controla o consumo de SQS, Kinesis e DynamoDB Streams: batchSize (mensagens por invocação), maximumBatchingWindowInSeconds (janela de agregação) e maximumConcurrency (invocações simultâneas para aquela fonte). Com fila SQS, o partial batch response devolve só as mensagens que falharam, evitando reprocessar o lote inteiro; em streams (Kinesis ou DynamoDB Streams), registros ruins são isolados com bisectBatchOnFunctionError.

bisectBatchOnFunctionError

Em streams (Kinesis ou DynamoDB Streams), divide o lote ao meio quando a função falha, isolando registros ruins sem descartar o restante. Combina com destino em SQS ou SNS para os registros que continuam falhando.

SNS e o padrão SNS para SQS para Lambda

SNS é pub/sub de push: faz fan-out para vários assinantes, com filtros de assinatura por mensagem. Colocar uma fila SQS com DLQ entre o SNS e a Lambda adiciona buffer, retentativas e proteção contra picos e throttling no downstream. SNS chamando endpoint HTTP ou Function URL diretamente sofre com picos e erros 429/5xx.

Amazon EventBridge

Barramento de eventos com regras por padrão que roteiam para vários destinos (por exemplo, uma fila SQS por microsserviço), com arquivo e replay para reprocessamento. Testa-se com PutEvents e acompanha-se pelas métricas das regras no CloudWatch. Não elimina duplicatas por si só.

AWS Step Functions

Orquestra múltiplos serviços com estado e fluxo visual, definido em Amazon States Language com estados Task, Choice, Retry e Catch. O tipo Standard atende fluxos de longa duração com histórico de execução e aprovação manual por callback com task token. Para um fluxo simples de receber e gravar, adiciona complexidade desnecessária.

Kinesis Data Streams

Streaming de alta taxa com reprocessamento por shard e múltiplos consumidores lendo o mesmo fluxo de forma independente. Integra-se nativamente à Lambda como fonte de eventos. Não é a escolha para uma fila simples de tarefas.

Reserved concurrency x Provisioned concurrency

Reserved concurrency reserva e também limita a concorrência de uma função, isolando-a e protegendo outras cargas ou o downstream. Provisioned concurrency mantém ambientes pré-inicializados para eliminar cold start e dar latência previsível; pode ser aplicada ao alias de produção e ajustada em horários de pico com EventBridge chamando PutProvisionedConcurrencyConfig ou com Application Auto Scaling.

Inicialização fora do handler

Clientes do SDK, conexões e configurações criados no escopo global são reaproveitados entre invocações no mesmo ambiente de execução, reduzindo latência. Importe apenas o necessário (SDK modular) para encolher o pacote. Variáveis globais também servem de cache local com TTL para parâmetros e segredos.

Lambda Layers e imagens de contêiner

Layers guardam dependências compartilhadas por várias funções e deixam o pacote de cada função enxuto. Funções empacotadas como imagem de contêiner não usam Layers: tudo vai na imagem, preferencialmente a imagem base oficial da Lambda com multi-stage build. O timeout máximo é de até 15 minutos nos dois formatos.

API Gateway: throttling, timeout e CORS

Throttling por estágio ou método com rate e burst, e usage plans com API keys definindo cotas por grupo de clientes; o excesso retorna 429, e o cliente deve usar retry exponencial. O timeout de integração tem limite máximo de 29 segundos e não pode ser desligado; trocar Lambda Proxy por não-proxy não muda isso. CORS se habilita no próprio API Gateway, sem código.

URLs pré-assinadas do S3

O backend gera a URL e o cliente faz upload ou download direto no S3, sem trafegar o arquivo pela aplicação. Os cabeçalhos assinados, como Content-Type, precisam ser exatamente os enviados pelo cliente, senão a requisição falha. Para arquivos grandes, upload multipart com URLs pré-assinadas e, se necessário, S3 Transfer Acceleration.

DynamoDB, DAX e TTL

DynamoDB é a primeira opção de banco para serverless com acesso por chave, baixa latência e escala automática (sessões, carrinho, metadados). DAX acelera leituras frequentes com latência muito baixa via endpoint do cluster no SDK. TTL expira itens automaticamente, de forma assíncrona, útil para sessões.

Testes locais com SAM CLI

sam local invoke e sam local start-api executam funções e APIs localmente, podendo apontar para DynamoDB Local, Step Functions Local ou serviços reais. Testes unitários usam mocks sem dependências externas; testes de integração interagem com serviços reais ou emulados. Isolar o SDK atrás de interfaces (repositório, adaptadores) facilita os mocks.

Retry com backoff exponencial e jitter

Para throttling e erros transitórios (DynamoDB, APIs externas), o SDK deve repetir com espera crescente e aleatoriedade, distinguindo erros transitórios de definitivos. É o primeiro ajuste barato antes de escalar recursos.

Macetes para memorizar

Mensageria

SQS enfileira, SNS espalha, EventBridge roteia, Kinesis transmite, Step Functions orquestra.

Visibility timeout

A mensagem tem de ficar escondida mais tempo do que a função leva para trabalhar.

Concorrência Lambda

Provisioned aquece, Reserved reserva e limita.

Eventos assíncronos

Pelo menos uma vez = idempotente sempre.

Handler

Fora do handler o que é caro e reutilizável; dentro, só a lógica do evento.

API Gateway

29 segundos é o teto: passou disso, vira assíncrono.

Resiliência

Retry com backoff e jitter, DLQ e idempotência: o tripé de toda integração assíncrona.

Tabela de memorização: não confunda

Quando é...	...e quando é

Amazon SQS Fila de trabalho: buffer, processamento assíncrono ponto a ponto, retentativas, DLQ e controle do ritmo pelo consumidor.	**Amazon SNS** Broadcast pub/sub: um evento distribuído (push) para vários assinantes, com filtros de assinatura.
Amazon EventBridge Receber, filtrar e rotear eventos de várias fontes para vários destinos por regras de padrão.	**AWS Step Functions** Orquestrar passos com estado, condições, retries, Catch e aprovação manual.
Kinesis Data Streams Vários consumidores independentes lendo o mesmo fluxo de dados em tempo real, com reprocessamento.	**SQS** Fila simples de tarefas consumida e apagada após o processamento.
Provisioned concurrency Eliminar cold start e garantir latência previsível em funções críticas ou picos conhecidos.	**Reserved concurrency** Isolar a função, garantir capacidade a ela e limitar a taxa contra o downstream.
SQS FIFO Ordem estrita por grupo e deduplicação são requisitos.	**SQS padrão** Throughput alto com tolerância a duplicatas tratadas por idempotência.
CloudFront + AWS WAF Cache na borda por caminho e proteção HTTP sem mexer no código.	**AWS Global Accelerator** Acelerar tráfego TCP/UDP e melhorar disponibilidade global, sem cache.
Amazon ElastiCache for Redis Cache de sessão em aplicação distribuída.	**DynamoDB Accelerator (DAX)** Acelerar leituras frequentes de uma tabela DynamoDB existente.

Números que caem

Item	Valor
Peso de Desenvolvimento com os serviços da AWS no guia oficial DVA-C02	**32% do conteúdo pontuado**
Timeout máximo de integração do API Gateway	**29 segundos**
Timeout máximo da Lambda (ZIP ou imagem de contêiner)	**até 15 minutos**
Long polling do SQS (WaitTimeSeconds)	**até 20 segundos**
Mensagens por chamada ReceiveMessage (MaxNumberOfMessages)	**até 10**
Referência de visibility timeout com Lambda consumindo SQS	**pelo menos 6x o timeout da função**
Durabilidade do Amazon S3	**11 noves**
Código de erro de throttling no API Gateway	**429**

Alertas de prova (pegadinhas)

⚠ **Alerta de prova:** Aumentar timeout e memória da Lambda para acabar com cold start.
Como não cair: Timeout só muda o tempo máximo de execução. Cold start se ataca com provisioned concurrency e inicialização enxuta fora do handler.

⚠ **Alerta de prova:** Migrar para fila FIFO para aumentar throughput.
Como não cair: FIFO tem limites de throughput mais restritos; serve para ordem e deduplicação, não para vazão.

⚠ **Alerta de prova:** Reduzir o visibility timeout para as mensagens voltarem mais rápido.
Como não cair: Mensagens reaparecem antes do fim do processamento e geram duplicidade. O visibility timeout deve superar o tempo de processamento.

⚠ **Alerta de prova:** Achar que ajustar visibility timeout ou long polling elimina duplicatas no SQS.
Como não cair: A entrega continua pelo menos uma vez. A solução é consumidor idempotente.

⚠ **Alerta de prova:** Manter o SNS chamando diretamente o Function URL e só aumentar a reserved concurrency.
Como não cair: Sem buffer, picos geram 429/5xx e perda. Coloque SQS com DLQ entre SNS e Lambda.

⚠ **Alerta de prova:** Trocar a integração Lambda Proxy por não-proxy para escapar do timeout do API Gateway.
Como não cair: O limite de 29 segundos continua. Para processos longos, use padrão assíncrono (SQS, SNS, Step Functions).

⚠ **Alerta de prova:** Desligar o timeout do API Gateway e controlar o tempo só na Lambda.
Como não cair: Não é possível desabilitar; o API Gateway sempre tem limite máximo.

⚠ **Alerta de prova:** Usar Global Accelerator para ganhar cache e WAF na borda.
Como não cair: Global Accelerator não faz cache nem integra WAF para inspeção HTTP. Use CloudFront com WAF.

⚠ **Alerta de prova:** Habilitar 'Active tracing' no SNS esperando o mesmo trace ID em todos os consumidores.
Como não cair: No fluxo assíncrono, propague o X-Amzn-Trace-Id na mensagem e use o SDK do X-Ray no assinante.

⚠ **Alerta de prova:** Fazer consumidores lerem por pull de um tópico SNS FIFO.
Como não cair: SNS é push para assinaturas. Para consumo controlado, assine filas SQS FIFO ao tópico.

⚠ **Alerta de prova:** Habilitar DLQ para reduzir latência.
Como não cair: DLQ trata falhas, não desempenho.

⚠ **Alerta de prova:** Usar Step Functions para orquestrar o simples 'receber do SNS e gravar no DynamoDB'.
Como não cair: É complexidade e custo sem ganho. Lambda assinante com DLQ ou SQS no meio resolve.

⚠ **Alerta de prova:** Adicionar Layers a uma função empacotada como imagem de contêiner.
Como não cair: Imagem de contêiner não usa Layers; as dependências vão na imagem.

⚠ **Alerta de prova:** Configurar a fonte SQS para descartar a mensagem após uma única falha.
Como não cair: Falhas intermitentes viram perda. Use retentativas com maxReceiveCount e DLQ.

Como cai na prova

📝 **Como cai:** Arquitetura serverless com sintoma de duplicidade ou perda de mensagens.
Como resolver: Identifique a fonte de entrega pelo menos uma vez; escolha idempotência com ID único, DLQ e visibility timeout adequado. Descarte FIFO só para vazão e reduções de timeout.

📝 **Como cai:** Latência alta só nas primeiras invocações ou após períodos sem tráfego.
Como resolver: É cold start: provisioned concurrency (no alias, ajustável em picos) e inicialização fora do handler. Descarte aumentar timeout, DLQ e memória como solução principal.

📝 **Como cai:** Escolher um único serviço entre SQS, SNS, Kinesis, EventBridge e Step Functions.
Como resolver: Pergunte: é fila (SQS), broadcast (SNS), stream com múltiplos leitores (Kinesis), roteamento por regras (EventBridge) ou orquestração com estado (Step Functions)?

📝 **Como cai:** Integração SNS ou API diretamente na Lambda sofrendo picos, 429 e throttling no DynamoDB.
Como resolver: Insira SQS como buffer, ajuste batchSize e concorrência, use DLQ e retry com backoff e jitter.

📝 **Como cai:** Requisição síncrona via API Gateway que pode demorar mais que o limite.
Como resolver: Alinhe timeouts de ponta a ponta respeitando o máximo de 29 segundos ou mude para padrão assíncrono (SQS, SNS, Step Functions).

Gatilhos do enunciado

Se o enunciado trouxer...	Pense em
upload direto para o S3 sem passar pelo backend	**URL pré-assinada (e multipart para arquivos grandes)**
upload com URL pré-assinada falha com assinatura inválida	**Content-Type assinado diferente do enviado pelo cliente**
mensagens processadas em duplicidade	**Idempotência no consumidor + visibility timeout maior que o processamento**
mensagens venenosas que falham sempre	**DLQ com maxReceiveCount**
muitas respostas vazias do ReceiveMessage	**Long polling até 20 segundos e até 10 mensagens por chamada**
falha de poucas mensagens força reprocessar o lote inteiro	**Partial batch response (SQS) ou bisectBatchOnFunctionError (streams)**
um evento para vários consumidores com filtros diferentes	**SNS fan-out para filas SQS com filtros de assinatura**
barramento de eventos, roteamento por padrão, replay	**Amazon EventBridge**
fluxo com condições, retries, Catch, aprovação manual	**Step Functions (Standard com task token para callback)**
init duration alta após inatividade, latência imprevisível	**Provisioned concurrency no alias de produção**
erro 429 no API Gateway	**Throttling rate/burst por estágio, usage plans e retry exponencial no cliente**
várias funções com as mesmas bibliotecas, pacote grande	**Lambda Layers**
leituras frequentes no DynamoDB com latência muito baixa	**DAX**
testar Lambda e API localmente	**SAM CLI (sam local invoke, sam local start-api) e DynamoDB Local**

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

- Todo consumidor assíncrono precisa ser idempotente.
- Visibility timeout maior que o tempo de processamento; DLQ com maxReceiveCount para falhas repetidas.
- Cold start = provisioned concurrency + init fora do handler, não mais timeout.
- API Gateway tem teto de 29 segundos de integração e não pode ser desligado.

Essenciais para a prova (checklist)

- ☐ Sei explicar entrega pelo menos uma vez e implementar idempotência com ID único.
- ☐ Sei configurar visibility timeout, maxReceiveCount e DLQ corretamente.
- ☐ Sei quando usar SQS padrão ou FIFO e as limitações da FIFO.
- ☐ Sei ajustar batchSize, maximumBatchingWindowInSeconds, maximumConcurrency e partial batch response.
- ☐ Sei diferenciar SQS, SNS, EventBridge, Kinesis e Step Functions por cenário.
- ☐ Sei diferenciar reserved de provisioned concurrency.
- ☐ Sei por que inicializar clientes do SDK fora do handler e quando usar Layers.
- ☐ Sei os limites de 29 segundos no API Gateway e de 15 minutos na Lambda.
- ☐ Sei gerar URL pré-assinada e evitar erro de assinatura por Content-Type.
- ☐ Sei testar localmente com SAM CLI e separar teste unitário de teste de integração.

Questões de revisão do capítulo

R1.1 · Desenvolvimento com os serviços da AWS

Um desenvolvedor está criando um backend sem servidor que processa eventos do Amazon S3 usando AWS Lambda. Em ambiente de testes, a função funciona corretamente, mas em produção, sob alta carga, alguns eventos do S3 não são processados e outros são processados mais de uma vez. O desenvolvedor precisa adaptar o código para lidar corretamente com esse comportamento e garantir consistência no processamento. O que ele deve fazer?

A. Tornar a função idempotente, tolerando eventos repetidos, atrasados ou fora de ordem
B. Ativar VPC na função Lambda, garantindo maior estabilidade de rede e, assim, eliminando a possibilidade de perda ou duplicação de eventos.
C. Configurar a função Lambda para o modo síncrono, garantindo que cada evento seja processado exatamente uma vez antes de retornar sucesso.
D. Aumentar o tempo máximo de execução (timeout) da função Lambda para evitar que eventos sejam descartados durante picos de carga.

R1.2 · Desenvolvimento com os serviços da AWS

Um desenvolvedor precisa processar arquivos de imagem enviados por usuários para um bucket Amazon S3. Sempre que um novo objeto JPG for criado em um prefixo específico, o código deve ser executado para gerar miniaturas e salvar o resultado em outro bucket. O desenvolvedor quer minimizar a administração de servidores, garantir escalabilidade automática e manter o custo baixo. Qual é a abordagem MAIS adequada?

A. Configurar uma função AWS Lambda acionada por eventos de criação de objeto no bucket S3, processar a imagem e salvar a miniatura em outro bucket S3.
B. Utilizar um AWS Step Functions State Machine que é executado manualmente pelo desenvolvedor sempre que novas imagens forem enviadas ao bucket S3.
C. Implantar um contêiner no Amazon ECS com uma tarefa agendada para listar objetos no bucket S3 e processar as imagens encontradas.
D. Criar uma instância Amazon EC2 com um script em loop verificando o bucket S3 periodicamente usando cron para detectar novos arquivos e processá-los.

R1.3 · Desenvolvimento com os serviços da AWS

Um desenvolvedor está construindo uma aplicação serverless que consome mensagens de uma fila do Amazon SQS e grava dados em uma tabela do Amazon DynamoDB. A aplicação usa uma função AWS Lambda escrita em Python. O desenvolvedor deseja reduzir o tempo de inicialização da função e melhorar a performance sob alta carga, mantendo os custos sob controle. Qual abordagem atende melhor a esses requisitos?

A. Configurar a função Lambda com memória mínima e sem variáveis de ambiente, para forçar a AWS a reutilizar o mesmo ambiente de execução por mais tempo.
B. Aumentar o timeout da função Lambda e desligar a concorrência reservada, permitindo que a AWS escale automaticamente sem limites.
C. Ativar Lambda provisioned concurrency e usar AWS SDK v3 modular (boto3 com importações específicas) junto com conexões reutilizáveis para o DynamoDB fora do handler.
D. Configurar a função Lambda para ser invocada só de dentro de uma VPC com NAT Gateway, garantindo que o cold start seja sempre reduzido.

R1.4 · Desenvolvimento com os serviços da AWS

Uma aplicação serverless recebe eventos do Amazon S3 e os processa em uma função AWS Lambda escrita em Python. Esses eventos disparam gravações em uma tabela Amazon DynamoDB. Em horários de pico, o volume de gravações aumenta e ocasionalmente ocorrem falhas transitórias de escrita (por exemplo, ProvisionedThroughputExceededException). O desenvolvedor quer implementar uma solução no código da função Lambda que trate automaticamente essas falhas transitórias, sem alterar a configuração de capacidade da tabela. Qual abordagem é a MAIS adequada?

A. Criar uma segunda tabela DynamoDB de backup e gravar simultaneamente nas duas tabelas
B. Usar Retry exponencial com jitter nas escritas ao DynamoDB via boto3, tratando os erros transitórios
C. Ativar o modo de billing on-demand da tabela DynamoDB para escalar automaticamente e confiar só nos retries nativos do serviço Lambda
D. Adicionar uma fila Amazon SQS entre o S3 e o Lambda para desacoplar a carga e desligar os retries no código da função

Respostas comentadas

R1.1. A) Tornar a função idempotente, tolerando eventos repetidos, atrasados ou fora de ordem
Por quê: A integração entre Amazon S3 e AWS Lambda segue um modelo de entrega de eventos pelo menos uma vez (at-least-once). Isso significa que o mesmo evento pode ser entregue mais de uma vez e, em casos raros, algum evento pode ser atrasado ou reentregue. Portanto, o código da função Lambda precisa ser idempotente: ao receber o mesmo evento novamente, o processamento não deve causar efeitos colaterais adicionais (por exemplo, não gravar registros duplicados, não faturar duas vezes, etc.).
✗ B) Incorreta. Colocar a função Lambda em uma VPC trata de conectividade com recursos privados (como RDS, ElastiCache, etc.) e não altera o comportamento de entrega de eventos do S3. Não elimina duplicações nem perdas ocasionais.
✗ C) Incorreta. Não é possível configurar a invocação de triggers do S3 para modo síncrono com garantia de exatamente uma vez. O padrão da integração continua sendo assíncrono e com entrega pelo menos uma vez, então a idempotência ainda é...
✗ D) Incorreta. Aumentar o timeout pode evitar falhas por tempo excedido, mas não resolve o modelo de entrega pelo menos uma vez. Mesmo com mais tempo, o S3 ainda pode reenviar eventos, e o problema de duplicidade permanecerá.

> **Dica de prova:** Sempre que usar integrações de eventos assíncronos na AWS (como S3, SNS, EventBridge ou invocações assíncronas de Lambda), assuma o modelo de entrega at-least-once e projete suas funções para serem idempotentes.

R1.2. A) Configurar uma função AWS Lambda acionada por eventos de criação de objeto no bucket S3, processar a imagem e salvar a miniatura em outro bucket S3.

Por quê: Esta alternativa é a mais adequada porque combina processamento orientado a eventos do Amazon S3 com AWS Lambda, eliminando a necessidade de gerenciar servidores. O Lambda escala automaticamente com o volume de uploads, é altamente integrado ao S3 e cobra apenas pelo tempo de execução, atendendo aos requisitos de escalabilidade automática, baixa administração e custo reduzido.

✗ B) Incorreta. Depende de execução manual, não é orientado a eventos de forma automática e não escala de maneira transparente com o volume de uploads, além de introduzir risco operacional por depender de ações humanas.

✗ C) Incorreta. ECS com tarefa agendada também usa polling, não reage imediatamente a novos uploads e adiciona complexidade operacional na configuração de cluster/serviço, fugindo do objetivo de minimizar administração.

✗ D) Incorreta. Exige gerenciamento de instância EC2 (patching, dimensionamento, alta disponibilidade) e usa polling, o que é menos eficiente e mais custoso que uma arquitetura baseada em eventos.

> **Dica de prova:** Quando o requisito envolver 'acionar código sempre que um objeto for criado no S3', pense primeiro em eventos do S3 integrados ao AWS Lambda. Essa combinação é padrão de arquitetura serverless para processamento de arquivos, reduzindo custo e esforço operacional.

R1.3. C) Ativar Lambda provisioned concurrency e usar AWS SDK v3 modular (boto3 com importações específicas) junto com conexões reutilizáveis para o DynamoDB fora do handler.

Por quê: Esta alternativa é a mais adequada porque combina técnicas específicas para reduzir cold starts e melhorar a performance sob carga. O uso de provisioned concurrency garante que um certo número de ambientes de execução já esteja inicializado, diminuindo o impacto de cold starts em picos de tráfego.

✗ A) Incorreta. Reduzir memória geralmente aumenta o tempo de execução e pode piorar a performance. A quantidade de memória influencia a CPU disponível;

✗ B) Incorreta. Aumentar o timeout não reduz cold start nem melhora performance; apenas permite que execuções mais lentas não falhem por timeout.

✗ D) Incorreta. Colocar a função em VPC com NAT Gateway normalmente aumenta o tempo de cold start (devido à configuração de ENIs), em vez de reduzi-lo, além de adicionar custo de NAT.

> **Dica de prova:** Para reduzir cold starts em Lambda: use provisioned concurrency para endpoints de baixa latência, inicialize clientes de SDK fora do handler, importe apenas o que for necessário no código, e evite colocar a função em VPC se não precisar acessar recursos privados.

R1.4. B) Usar Retry exponencial com jitter nas escritas ao DynamoDB via boto3, tratando os erros transitórios

Por quê: Esta alternativa é a mais adequada porque segue as boas práticas de desenvolvimento com DynamoDB e AWS SDK. Implementar retry exponencial com jitter no código da Lambda, tratando explicitamente erros transitórios (como ProvisionedThroughputExceededException ou throttling), permite que a aplicação seja resiliente sem depender exclusivamente de mudanças de capacidade ou de infraestrutura adicional. O padrão de backoff exponencial com jitter reduz contenção e colisões de novas tentativas em cenários de pico de carga.

✗ A) Incorreta. Gravar em uma segunda tabela não evita throttling na tabela principal e aumenta a complexidade e o custo. Também não aborda diretamente o tratamento de falhas transitórias na escrita.

✗ C) Incorreta. Migrar para on-demand pode ajudar a escalar, mas não substitui a necessidade de lidar com falhas transitórias no código. Além disso, a pergunta pede uma solução no código da função, não mudança de configuração da tabela.

✗ D) Incorreta. Inserir SQS melhora o desacoplamento, mas não resolve, por si só, os erros de throughput do DynamoDB. Ainda é necessário tratar retries em nível de aplicação; desativar todos os retries no código agrava o problema.

Dica de prova: Sempre que trabalhar com serviços suscetíveis a throttling (DynamoDB, APIs externas, etc.), implemente retries com backoff exponencial e, idealmente, jitter. Use os mecanismos de tratamento de exceção da SDK (como boto3) para distinguir erros transitórios de erros permanentes e só repetir o que for realmente...

02

CAPÍTULO 2

Implantação

Aqui eu preciso dominar o fluxo de CI/CD nativo da AWS: CodeCommit ou GitHub como origem, CodeBuild para build e testes, Amazon ECR para as imagens, CodePipeline orquestrando e CodeDeploy executando a estratégia de implantação. A prova quase sempre descreve um serviço em contêiner no Amazon ECS com Fargate, ou funções Lambda com AWS SAM, e pede a solução com menor esforço operacional, zero downtime e rollback automático. Onde mais erro é cair em alternativas com rolling update padrão do ECS, scripts no CodeBuild, cron em EC2 ou passos manuais no console quando o enunciado exige blue/green, canary e reversão automática. Também cai muito infraestrutura como código: um único template parametrizado (CloudFormation, SAM ou CDK) reaproveitado em vários ambientes e contas, em vez de templates duplicados ou recursos criados à mão. No guia oficial do DVA-C02 este domínio vale 24% do conteúdo pontuado.

Neste capítulo

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

Tópicos-chave

AWS CodePipeline

É o orquestrador do fluxo de CI/CD: organiza estágios de origem, build, teste, aprovação manual e deploy, com integrações nativas com ECR, CodeBuild, CodeDeploy, ECS, Lambda, Elastic Beanstalk e CloudFormation. Ele não compila nem implanta sozinho, delega ao CodeBuild e às ações de deploy. Sempre que o enunciado pede pipeline gerenciado com aprovações manuais e facilidade de inserir testes, a resposta gira em torno dele.

AWS CodeBuild

Executa build e testes (unitários, integração, E2E) definidos no buildspec.yml. Em pipelines de contêiner, faz o docker build e o docker push para o Amazon ECR, precisando de privilégios para rodar Docker. É executor dentro do pipeline, não orquestrador e nem alvo de execução da aplicação.

AWS CodeDeploy

É quem cuida da estratégia de implantação (in-place ou blue/green, canary ou linear) para EC2, Lambda e ECS, com rollback automático por falha de implantação ou por alarme do Amazon CloudWatch. No ECS, a implantação usa AppSpec e taskdef.json. Quando a prova fala em controle de tráfego entre versões e reversão automática, o CodeDeploy é a peça central.

Blue/green no ECS com CodeDeploy

Serviço ECS com deployment controller do CodeDeploy, Application Load Balancer com dois target groups (blue e green), health checks e troca de tráfego gradual com rollback automático. Hooks de ciclo de vida como BeforeAllowTraffic e AfterAllowTraffic permitem validar a nova versão antes de expor usuários, e o termination wait time mantém a versão antiga disponível para reversão. Fórmula da prova: ECS + ALB (dois target groups) + CodeDeploy, orquestrados pelo CodePipeline.

Rolling update padrão do ECS

É a atualização nativa do serviço ECS, também usada pela ação de deploy 'Amazon ECS' do CodePipeline. É mais simples e reduz indisponibilidade, mas não oferece controle fino de tráfego entre versões, testes antes de rotear tráfego nem rollback automático no nível do blue/green do CodeDeploy. Serve quando o requisito é só atualizar o serviço a cada nova imagem.

Canary, linear e AllAtOnce

Canary envia primeiro uma pequena parcela do tráfego e depois o restante (exemplo no SAM: Canary10Percent5Minutes). Linear aumenta o tráfego em incrementos iguais a intervalos fixos (exemplo: Linear10PercentEvery1Minute). AllAtOnce muda todo o tráfego de uma vez, mesmo dentro de um blue/green, por isso não atende a 'começar com pequena parcela'.

Versões e aliases do Lambda

Uma versão publicada é imutável; o alias (por exemplo, prod) é um ponteiro para uma versão e pode dividir tráfego entre duas versões. Implantação gradual move o alias aos poucos e o rollback é apontar o alias de volta para a versão anterior. Evita alterar o $LATEST diretamente em produção.

AutoPublishAlias e DeploymentPreference no SAM

Atributos do template SAM que publicam uma nova versão a cada deploy, movem o alias e integram com o CodeDeploy para canary ou linear com rollback por alarmes do CloudWatch. Dispensam scripts de implantação. É a resposta padrão para Lambda + SAM + deploy gradual com reversão rápida.

sam build, sam package, sam deploy e samconfig.toml

Comandos padrão para compilar, empacotar e implantar aplicações serverless com mínimo de scripts. Para vários ambientes, usa-se um único template com parâmetros, variando valores pelo samconfig.toml ou por overrides de parâmetros no pipeline, além do stack-name e de tags.

Build once, deploy many

O artefato é gerado uma única vez no estágio de build (pacote em bucket S3 versionado ou imagem no ECR) e o mesmo artefato é promovido de homologação para produção. Rodar sam build ou docker build de novo em cada ambiente quebra a garantia de que produção recebe exatamente o que foi testado.

Tags imutáveis de imagem

A imagem recebe uma tag imutável, como o SHA do commit, o número do build ou o digest, e o pipeline atualiza explicitamente a task definition do ECS com essa referência. Isso facilita rollback, auditoria e rastreabilidade, ao contrário de reaproveitar uma tag genérica.

CloudFormation parametrizado e StackSets

Para dev, homologação e produção, prefira um único template parametrizado e várias stacks, usando Parameters, Mappings e Conditions, com valores sensíveis no Parameter Store ou Secrets Manager. Para padronizar recursos (inclusive o próprio pipeline) em várias contas e regiões, use StackSets, com assunção de role entre contas (sts:AssumeRole) em vez de chaves estáticas.

Blue/green no Elastic Beanstalk

No Elastic Beanstalk, blue/green significa dois ambientes distintos e a troca do tráfego por swap de CNAME depois de validar o ambiente green. All at once, Rolling, Rolling with additional batch e Immutable são políticas de atualização dentro de um mesmo ambiente e não substituem um blue/green verdadeiro.

.platform/hooks e .ebextensions

No Elastic Beanstalk, scripts em .platform/hooks (predeploy, postdeploy etc.) automatizam tarefas ligadas ao ciclo de implantação, como migração de banco, limpeza de cache e ajuste de permissões. O .ebextensions serve para ajustes leves de configuração; instalação pesada repetida a cada deploy deve virar uma AMI pré-preparada.

CodeDeploy com Auto Scaling

Associar o Auto Scaling Group ao deployment group faz com que toda instância nova receba automaticamente a revisão atual por meio do lifecycle hook. Isso evita instâncias novas subindo com versão antiga, sem depender de AMI atualizada à mão, user data com git clone ou implantações agendadas.

Configuração por ambiente

A mesma imagem ou pacote roda em todos os ambientes; o que muda é a configuração, guardada no Systems Manager Parameter Store (parâmetros gerais e SecureString) ou no Secrets Manager (credenciais), lida pela aplicação ou pelo pipeline. Segredos não ficam em repositório, arquivos JSON criptografados ou variáveis editadas à mão no console.

Gates automáticos no CodePipeline

Ações de teste com CodeBuild executam testes de integração, smoke tests ou chamadas a APIs externas e falham o pipeline antes de promover o artefato. Ações de invocação Lambda implementam validações customizadas, como chamar o endpoint /health após o staging e devolver sucesso ou falha, bloqueando a promoção.

Macetes para memorizar

Papéis dos serviços Code*

CodeCommit guarda, CodeBuild compila, CodePipeline rege, CodeDeploy entrega.

Blue/green em ECS

ECS + ALB com dois target groups + CodeDeploy = troca gradual com volta automática.

Lambda em produção

Alias é a ponte para produção: versão não muda, o alias é que anda.

Canary

Canário na mina: primeiro um pouco, se sobreviver vai o resto.

Elastic Beanstalk

Blue/green no Beanstalk = dois ambientes e troca de CNAME; o resto (All at once, Rolling, Immutable) é dentro de um ambiente só.

Artefatos

Build once, deploy many: quem testou é quem vai para produção.

Distratores

Console, laptop, cron em EC2, SSH e script manual = quase sempre alternativa errada.

Tabela de memorização: não confunda

Quando é...	...e quando é
CodeDeploy blue/green para ECS O enunciado exige zero downtime, troca gradual de tráfego, testes antes de expor usuários e rollback automático por saúde ou alarme.	**Rolling update padrão do ECS (ação nativa ECS no CodePipeline)** Basta atualizar o serviço automaticamente a cada nova imagem, sem exigência de controle de tráfego ou reversão automática.
Canary Requisito de começar com pequena parcela de tráfego ou instâncias e ampliar aos poucos conforme a confiança.	**AllAtOnce** Não há exigência de gradualidade; todo o tráfego pode mudar de uma vez (com maior risco).

Blue/green do Elastic Beanstalk (dois ambientes + swap de CNAME) Pede-se blue/green verdadeiro com validação isolada da nova versão e rollback instantâneo trocando o CNAME de volta.	**Rolling with additional batch (mesmo ambiente)** Pede-se manter a capacidade durante a atualização dentro de um único ambiente, por exemplo junto com hooks postdeploy para migração.
CodeDeploy in-place com rollback automático Instâncias EC2 existentes podem ser atualizadas no lugar e o requisito é reverter sozinho para a última revisão bem-sucedida em caso de falha ou alarme.	**CodeDeploy blue/green com ALB e Auto Scaling** O requisito é minimizar downtime em EC2 atrás de load balancer, testando a nova frota antes de redirecionar o tráfego.
CodePipeline com estágios e aprovação manual Há estágios de build, teste e aprovação humana antes de produção, com histórico de execução.	**Regra do EventBridge no push do ECR acionando Lambda** O requisito é automação total, sem intervenção manual: ao chegar imagem nova, criar revisão da task definition e atualizar o serviço ECS.
Template único parametrizado com várias stacks Vários ambientes (dev, homologação, produção, efêmeros por branch) a partir do mesmo código de infraestrutura.	**StackSets** Padronizar o mesmo template em várias contas e regiões a partir de um ponto central.

Números que caem

Item	Valor
Peso de Implantação no guia oficial DVA-C02	**24% do conteúdo pontuado**
Exemplo de configuração canary no SAM	**Canary10Percent5Minutes**
Exemplo de configuração linear no SAM	**Linear10PercentEvery1Minute**

Alertas de prova (pegadinhas)

⚠ **Alerta de prova:** Usar o AWS Batch, disparado por nova imagem no ECR, para rodar testes e atualizar o ECS.
Como não cair: AWS Batch é para computação em lote, não orquestra pipeline nem oferece aprovação manual. Pipeline de implantação é CodePipeline.

⚠ **Alerta de prova:** Agendar um Step Functions que faz polling no ECR e atualiza o ECS.
Como não cair: Funciona, mas exige lógica customizada e vai contra 'menor esforço operacional'. O CodePipeline aceita o ECR como origem nativamente.

⚠ **Alerta de prova:** Achar que a ação nativa ECS do CodePipeline (rolling update) já entrega rollback automático por saúde.
Como não cair: Rolling update não reverte sozinho como o blue/green do CodeDeploy; quando aparecer 'rollback automático' + 'zero downtime' em ECS, exija CodeDeploy.

⚠ **Alerta de prova:** Gerenciar o blue/green com scripts no CodeBuild trocando target groups do ALB.
Como não cair: Scripts aumentam complexidade e risco; o CodeDeploy tem suporte nativo a blue/green e rollback no ECS.

⚠ **Alerta de prova:** Escolher Blue/Green com AllAtOnce quando se pede começar com pequena parcela.
Como não cair: AllAtOnce muda todo o tráfego de uma vez. Pequena parcela e expansão gradual é Canary (ou Linear).

⚠ **Alerta de prova:** Tratar 'Rolling with additional batch' como blue/green no Elastic Beanstalk.
Como não cair: Rolling atua dentro de um único ambiente. Blue/green no Beanstalk exige dois ambientes e swap de CNAME.

⚠ **Alerta de prova:** Usar Provisioned Concurrency ou Auto Scaling para controlar a porcentagem de tráfego entre versões Lambda.
Como não cair: Provisioned Concurrency trata capacidade e cold start, não roteamento. Divisão de tráfego entre versões é feita pelo alias.

⚠ **Alerta de prova:** Criar duas funções Lambda (v1 e v2) com dois endpoints no API Gateway e trocar a URL no cliente.
Como não cair: Isso obriga mudança no frontend e não dá controle de porcentagem. Use versões + alias + DeploymentPreference.

⚠ **Alerta de prova:** Colocar a migração de banco no ENTRYPOINT do contêiner.
Como não cair: Cada instância executaria a migração a cada inicialização, gerando condições de corrida. Use .platform/hooks/postdeploy ligado ao ciclo de implantação.

⚠ **Alerta de prova:** Rodar sam build e sam deploy separadamente em cada pipeline de ambiente.
Como não cair: Cada ambiente reconstrói o próprio pacote e quebra o 'build once, deploy many'. Empacote uma vez, guarde em S3 versionado e promova o mesmo artefato.

⚠ **Alerta de prova:** Considerar atualização in-place do CloudFormation ou política All-at-once como blue/green para ECS Fargate.
Como não cair: In-place não isola as versões nem faz traffic shifting gradual; o blue/green com rollback é do CodeDeploy.

⚠ **Alerta de prova:** Garantir a versão atual em instâncias novas do Auto Scaling com AMI atualizada à mão, user data com git clone ou deploy agendado.
Como não cair: Associe o Auto Scaling Group ao deployment group do CodeDeploy; o lifecycle hook implanta a revisão atual em cada instância nova.

⚠ **Alerta de prova:** Usar Elastic Beanstalk com .ebextensions para gerenciar uma aplicação serverless de API Gateway, Lambda e DynamoDB.
Como não cair: Beanstalk é voltado a aplicações em instâncias e contêineres. Serverless se descreve em SAM ou CloudFormation com CodePipeline/CodeBuild.

⚠ **Alerta de prova:** Chamar a API externa de validação dentro do Dockerfile durante o build da imagem.
Como não cair: O build fica dependente de serviço externo e perde reprodutibilidade. Faça a validação numa ação de teste do CodeBuild no pipeline, antes do deploy.

Como cai na prova

📝 **Como cai:** Cenário longo de microsserviço em ECS Fargate com requisitos numerados (build e testes, imagem versionada, blue/green, rollback automático, sem downtime).
Como resolver: Marque cada requisito; elimine opções com passos manuais, scripts customizados, GitHub Actions fora da AWS ou rolling update; fique com CodePipeline + CodeBuild + ECR + CodeDeploy blue/green para ECS com ALB.

📝 **Como cai:** Lambda versionada com SAM precisa ir para produção aos poucos com rollback rápido.
Como resolver: Procure versão + alias + DeploymentPreference (Canary ou Linear). Descarte $LATEST direto, duas funções separadas e Provisioned Concurrency como mecanismo de roteamento.

📝 **Como cai:** Padronizar ambientes em várias contas ou criar homologação igual à produção.
Como resolver: Escolha IaC: template único parametrizado e nova stack, ou StackSets para multi-conta. Descarte console, template exportado, template duplicado e troca de plataforma (ECS para Beanstalk).

📝 **Como cai:** Quatro alternativas quase iguais, diferindo só na estratégia (rolling x blue/green, AllAtOnce x Canary, com ou sem CodeDeploy).
Como resolver: Leia o final de cada alternativa; a palavra que decide é a estratégia. Associe 'rollback automático' e 'controle de tráfego' a CodeDeploy blue/green ou canary.

📝 **Como cai:** Pergunta curta pedindo um único serviço para automatizar implantação de contêiner.
Como resolver: Orquestrar o fluxo inteiro = CodePipeline; IaC e change sets = CloudFormation; repositório = CodeCommit; eventos sozinhos não orquestram estágios (EventBridge).

Gatilhos do enunciado

Se o enunciado trouxer...	Pense em
pipeline gerenciado, integrações nativas, aprovação manual, adicionar testes no meio	**AWS CodePipeline + CodeBuild (build/testes) + ação de deploy (CodeDeploy, ECS, Lambda)**
ECS/Fargate, zero downtime, rollback automático, health check	**CodeDeploy blue/green para ECS com ALB e dois target groups**
implantar para pequena parcela e expandir gradualmente	**Canary (ou Linear) no CodeDeploy**
Lambda com SAM, deploy gradual e rollback rápido	**Versão imutável + alias + AutoPublishAlias e DeploymentPreference**
mesmo padrão em várias contas ou regiões	**CloudFormation/CDK versionado no Git + StackSets e roles entre contas**
dev, homologação e produção com mesma infraestrutura	**Um template parametrizado e uma stack por ambiente (samconfig.toml ou parâmetros)**
garantir que produção recebe o mesmo pacote testado	**Build once, deploy many: artefato em S3 versionado ou imagem com tag imutável no ECR**
blue/green no Elastic Beanstalk	**Dois ambientes + swap de CNAME**
migração de banco durante deploy no Elastic Beanstalk	**.platform/hooks/postdeploy**
instâncias novas do Auto Scaling sobem com versão antiga	**Auto Scaling Group associado ao deployment group do CodeDeploy**
nova imagem no ECR, automatizar sem intervenção manual	**Regra do EventBridge acionando Lambda que registra nova task definition e atualiza o serviço**
validar /health ou smoke test antes de promover	**Ação de invocação Lambda ou ação de teste CodeBuild no CodePipeline**
EKS com atualização gradual pelo pipeline	**Deployment Kubernetes com RollingUpdate e readiness probes; CodeBuild atualiza a tag e roda kubectl apply**
ambientes efêmeros por branch ou pull request	**IaC com stack nomeada por branch e parâmetro Environment, criada e destruída pelo pipeline**

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

- Rollback automático e troca gradual de tráfego em ECS, EC2 ou Lambda vêm do CodeDeploy, não do rolling update nem de scripts.
- Um template parametrizado para todos os ambientes; StackSets para várias contas e regiões.
- Promova o mesmo artefato (S3 versionado ou imagem com tag imutável) entre ambientes.

Essenciais para a prova (checklist)

☐ Sei montar o pipeline CodePipeline + CodeBuild + ECR + CodeDeploy para ECS Fargate com blue/green.

☐ Sei diferenciar rolling update do ECS de blue/green com CodeDeploy e dizer qual dá rollback automático.

☐ Sei explicar Canary, Linear e AllAtOnce e reconhecer Canary10Percent5Minutes e Linear10PercentEvery1Minute.

☐ Sei usar versão, alias, AutoPublishAlias e DeploymentPreference para deploy gradual de Lambda com SAM.

☐ Sei por que um template único parametrizado é melhor que um template por ambiente e quando usar StackSets.

☐ Sei aplicar build once, deploy many com S3 versionado e tags imutáveis de imagem.

☐ Sei fazer blue/green no Elastic Beanstalk com swap de CNAME e usar .platform/hooks para migrações.

☐ Sei garantir que instâncias novas do Auto Scaling recebam a revisão atual pelo CodeDeploy.

☐ Sei inserir gates automáticos (ação CodeBuild de teste, ação Lambda) e aprovação manual no CodePipeline.

Questões de revisão do capítulo

R2.1 · Implantação

Uma empresa mantém um aplicativo containerizado em Amazon ECS usando Fargate. Durante o processo de implantação de uma nova versão, a equipe deseja garantir que novas tarefas recebam apenas tráfego HTTP se passarem em verificações de integridade, e que seja possível rollback automático caso a nova versão apresente falhas logo após o deploy. Qual a maneira MAIS apropriada de configurar a implantação para atender a esses requisitos com o mínimo de código customizado?

A. Configurar um serviço ECS com tipo de implantação Rolling update padrão e usar scripts customizados em cada container para registrar e remover as tarefas de um Application Load Balancer.
B. Ativar Auto Scaling no ECS Service e reduzir à mão o número de tarefas antigas, aumentando o número de tarefas novas após verificar à mão os logs de erro.
C. Utilizar o AWS CloudFormation para atualizar a definição de tarefa ECS e executar um stack update toda vez que houver nova versão, confiando nos health checks do container sem load balancer.
D. Definir um serviço ECS com suporte a deployments gerenciados pelo AWS CodeDeploy, utilizar um Application Load Balancer com health checks e configurar uma implantação blue/green no CodeDeploy.

R2.2 · Implantação

Uma equipe de desenvolvimento deseja implementar blue/green deployment para uma aplicação containerizada em produção usando AWS Elastic Beanstalk. O objetivo é reduzir ao máximo o tempo de indisponibilidade e possibilitar rollback rápido em caso de falhas na nova versão. Qual abordagem atende melhor a esse requisito?

A. Atualizar o mesmo ambiente Elastic Beanstalk com a nova versão utilizando o método de implantação 'All at once' e configurar health checks mais agressivos
B. Criar dois ambientes Elastic Beanstalk separados (blue e green) e usar a funcionalidade de swap de CNAME para alternar o tráfego entre eles após validar o ambiente green
C. Configurar o método de implantação 'Rolling with additional batch' em um único ambiente Elastic Beanstalk e aumentar o número mínimo de instâncias durante a implantação
D. Criar um Application Load Balancer separado e registrar manualmente as instâncias de uma Auto Scaling group criado fora do Elastic Beanstalk para simular blue/green

R2.3 · Implantação

Você é responsável por implantar uma nova versão de um microserviço containerizado em produção na AWS. O serviço roda em Amazon ECS com capacidade Fargate, exposto por meio de um Application Load Balancer (ALB). O requisito é: (1) ter *blue/green deployment* com possibilidade de rollback rápido; (2) validar a nova versão com tráfego real de forma progressiva, com métricas de erro e latência; (3) não alterar manualmente configurações de DNS externas nem interromper conexões ativas. Qual abordagem atende melhor a esses requisitos?

A. Configurar um serviço ECS com duas revisões distintas em execução simultânea, alterar à mão o ALB target group principal para apontar para a nova tarefa, e em caso de falha voltar à mão para o target group anterior
B. Utilizar AWS CloudFormation com atualizações *in-place* e política de rollback automática, sem alterar a configuração padrão de tráfego do ECS, confiando só em health checks de tarefa para validar a nova versão
C. Configurar um segundo serviço ECS em outro cluster Fargate com um novo ALB, atualizar à mão o registro DNS externo para apontar para o novo ALB após testes, e reverter o DNS se a implantação falhar
D. Usar CodeDeploy blue/green para ECS com dois target groups no ALB, troca gradual monitorada por alarmes e CodePipeline

R2.4 · Implantação

Uma startup está migrando um monólito para microsserviços em AWS. Eles querem padronizar a criação e implantação de funções AWS Lambda, incluindo variáveis de ambiente, permissões IAM e integrações com Amazon API Gateway, garantindo que todas as funções sigam o mesmo padrão em diferentes contas e regiões. Qual é a MELHOR abordagem para atender a esse requisito com o mínimo de esforço operacional contínuo?

A. Configurar um pipeline no AWS CodePipeline que só execute testes unitários e, em seguida, faça upload do pacote de código Lambda diretamente para o console do Lambda de forma manual
B. Definir a infraestrutura como código usando AWS CloudFormation ou AWS SAM, descrevendo funções Lambda, permissões IAM, variáveis de ambiente e integrações com API Gateway em templates reutilizáveis
C. Utilizar AWS Elastic Beanstalk com um arquivo .ebextensions para gerenciar o ciclo de vida das funções Lambda e das integrações com API Gateway
D. Criar um script Bash que use AWS CLI para empacotar e implantar à mão cada função Lambda, executando o script toda vez que uma mudança for necessária

Respostas comentadas

R2.1. D) Definir um serviço ECS com suporte a deployments gerenciados pelo AWS CodeDeploy, utilizar um Application Load Balancer com health checks e configurar uma implantação blue/green no CodeDeploy.

Por quê: Esta alternativa é a mais adequada porque combina Amazon ECS com AWS CodeDeploy em um modelo de implantação blue/green. Nesse modelo, o CodeDeploy integra-se diretamente com um Application Load Balancer (ALB), usando health checks para direcionar o tráfego apenas para as novas tarefas saudáveis. Além disso, o CodeDeploy oferece rollback automático baseado em falhas de health check ou alarmes do CloudWatch logo após o deploy, reduzindo a necessidade de scripts customizados e lógica adicional no código da aplicação.

✗ A) Incorreta. Depende de scripts customizados dentro do container para manipular o ALB, o que aumenta a complexidade operacional e foge das práticas recomendadas. Também não fornece rollback automático de forma nativa.

✗ B) Incorreta. Exige intervenção manual para ajustar o número de tarefas e analisar logs, tornando o processo sujeito a erros humanos e sem rollback automático. O Auto Scaling não gerencia versões de tarefa nem lógica de blue/green por si só.

✗ C) Incorreta. O CloudFormation atualiza a definição de tarefa, mas sem ALB não há roteamento fino de tráfego nem health checks no nível do load balancer para controlar a exposição da nova versão.

> **Dica de prova:** Para cenários de ECS + Fargate em que você precisa de controle refinado de tráfego, health checks e rollback automático, pense em: ECS Service + Application Load Balancer + CodeDeploy (deployment type blue/green). Esse trio costuma aparecer em questões envolvendo zero-downtime e rollback automático.

R2.2. B) Criar dois ambientes Elastic Beanstalk separados (blue e green) e usar a funcionalidade de swap de CNAME para alternar o tráfego entre eles após validar o ambiente green

Por quê: Esta alternativa é a melhor abordagem para blue/green deployment no Elastic Beanstalk. Ao manter dois ambientes separados (blue = produção atual, green = nova versão), a equipe pode implantar, testar e validar a nova versão no ambiente green sem impactar usuários. Quando tudo estiver validado, basta usar o recurso nativo de 'swap CNAME' do Elastic Beanstalk, que alterna o DNS de forma rápida e reversível.

✗ A) Incorreta. 'All at once' atualiza todas as instâncias simultaneamente no mesmo ambiente, aumentando risco de indisponibilidade e tornando o rollback mais lento, pois exige nova implantação ou recriação do ambiente.

✗ C) Incorreta. 'Rolling with additional batch' melhora a disponibilidade, mas ainda atua em um único ambiente. Não é um verdadeiro blue/green e dificulta rollback instantâneo, pois não há ambiente anterior completo preservado.

✗ D) Incorreta. Criar ALB e Auto Scaling fora do Elastic Beanstalk adiciona complexidade desnecessária e não aproveita os recursos nativos de blue/green do serviço.

> **Dica de prova:** Para lembrar: em Elastic Beanstalk, blue/green = dois ambientes distintos + swap de CNAME. Métodos 'rolling' e 'all at once' são estratégias de atualização dentro de um mesmo ambiente, não substituem um verdadeiro blue/green deployment.

R2.3. D) Usar CodeDeploy blue/green para ECS com dois target groups no ALB, troca gradual monitorada por alarmes e CodePipeline

Por quê: Esta alternativa é a melhor porque usa o mecanismo nativo de blue/green deployment para ECS oferecido pelo AWS CodeDeploy. Com o tipo de implantação 'Blue/Green' para ECS, você configura dois target groups do ALB (um para o ambiente blue e outro para o green). O CodeDeploy faz o *traffic shifting* gradual entre eles, permitindo validar a nova versão com tráfego real enquanto monitora métricas via Amazon CloudWatch (por exemplo, taxa de erros 5xx, latência p95).

✗ A) Incorreta. Depende de troca manual de target group no ALB, o que é propenso a erro e não oferece *traffic shifting* gradual, além de dificultar rollback automatizado.

✗ B) Incorreta. Atualizações *in-place* do CloudFormation não representam um verdadeiro blue/green para ECS Fargate com ALB. Além disso, confiar apenas em health checks de tarefa não permite estratégia de *canary* ou *linear traffic shifting*...

✗ C) Incorreta. Cria um segundo ALB e exige alterações manuais de DNS, o que acrescenta latência de propagação DNS e risco de inconsistência. Também não há *traffic shifting* progressivo automático nem rollback imediato;

> **Dica de prova:** Para blue/green em ECS com Fargate, decore que o trio mais comum é: ECS (serviço), ALB (dois target groups) e CodeDeploy (tipo Blue/Green), frequentemente orquestrado por CodePipeline.

R2.4. B) Definir a infraestrutura como código usando AWS CloudFormation ou AWS SAM, descrevendo funções Lambda, permissões IAM, variáveis de ambiente e integrações com API Gateway em templates reutilizáveis

Por quê: Esta alternativa é a melhor abordagem porque usar AWS CloudFormation ou AWS SAM permite descrever toda a infraestrutura (funções Lambda, permissões IAM, variáveis de ambiente, integrações com API Gateway) como código em templates reutilizáveis. Isso facilita padronização entre contas e regiões, versionamento em repositórios Git e automação de implantações com esforço operacional reduzido ao longo do tempo.

✗ A) Incorreta. O pipeline descrito não automatiza a criação/atualização da infraestrutura; ainda exige upload manual do código e não garante padronização de permissões, variáveis de ambiente e integrações em múltiplas contas e regiões.

✗ C) Incorreta. AWS Elastic Beanstalk é voltado para deploy de aplicações (por exemplo, web) e não é a ferramenta adequada para gerenciar diretamente funções Lambda e integrações com API Gateway em padrão de microsserviços serverless.

✗ D) Incorreta. Depende de scripts manuais com AWS CLI, o que aumenta risco de erro humano, dificulta padronização entre contas/regiões e não oferece os benefícios completos de infraestrutura como código declarativa.

> **Dica de prova:** Para aplicações serverless, prefira AWS SAM ou CloudFormation em vez de scripts manuais. Templates permitem replicar o mesmo padrão em várias contas/regiões com um único comando (por exemplo, 'sam deploy' ou 'cloudformation deploy'), além de facilitar rollback e controle de versão.

03

CAPÍTULO 3

Segurança

Este domínio gira em torno de três ideias que se repetem o tempo todo: código rodando na AWS usa role com credenciais temporárias, nunca usuário IAM com access key; segredo fica no Secrets Manager ou no Parameter Store como SecureString, nunca em código, imagem, user data ou variável de ambiente; e usuário final é autenticado pelo Amazon Cognito com JWT validado no API Gateway. A prova testa se eu sei a role certa para cada computação (execution role da Lambda, task role do ECS, instance profile do EC2, IRSA no EKS), quando o Secrets Manager é obrigatório (rotação automática, integração com RDS) e quando o Parameter Store basta, e como restringir o KMS com key policy. Onde mais erro é em alternativas que guardam a access key em lugar aparentemente seguro, criptografada ou no próprio Secrets Manager: continua sendo credencial estática. No guia oficial do DVA-C02 este domínio vale 26% do conteúdo pontuado.

Neste capítulo

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

Tópicos-chave

Role IAM para a computação

Lambda usa execution role, ECS usa task role, EC2 usa instance profile e EKS usa IRSA. A role entrega credenciais temporárias via STS, renovadas automaticamente, e o SDK as obtém sem nenhuma chave no código. Se a alternativa cria usuário IAM para a aplicação, quase certamente está errada.

Task role do ECS

Role associada à task definition, dando a cada tarefa as próprias credenciais de menor privilégio. Em ECS/Fargate é preferida ao instance profile, que seria compartilhado por todas as tarefas do host. É ela que recebe permissões como ssm:GetParameter ou secretsmanager:GetSecretValue nos ARNs necessários.

IRSA (IAM Roles for Service Accounts)

No EKS, a role IAM é ligada a uma service account do Kubernetes usada pelos pods que precisam acessar a AWS. Assim cada microsserviço tem permissões restritas, por exemplo a um bucket específico, sem chaves estáticas.

Menor privilégio

Conceder somente as ações necessárias nos ARNs exatos, como sns:Publish no ARN do tópico ou ssm:GetParameter em /prod/db/password. Trocar 'Resource': '*' pelo ARN da tabela e evitar políticas FullAccess. Cada função deve ter a própria role, sem reaproveitar uma role ampla.

Trust policy e políticas baseadas em recurso

A trust policy define quem pode assumir a role, por exemplo codepipeline.amazonaws.com para a role do pipeline. Políticas baseadas em recurso complementam a role, como a política da fila SQS, a bucket policy ou a política do repositório ECR com aws:sourceVpce. Role com trust para EC2 não serve para aplicação rodando no navegador.

AWS Secrets Manager

Serviço específico para segredos: rotação automática, integração nativa com RDS, auditoria e controle por IAM. A aplicação lê o segredo em tempo de execução com secretsmanager:GetSecretValue, de preferência com cache em memória, e os logs registram identificadores, nunca valores.

Parameter Store SecureString

Parâmetro criptografado com KMS, lido em tempo de execução com ssm:GetParameter (mais kms:Decrypt quando a chave é do cliente). Ótimo para configurações e segredos sem rotação complexa. Parâmetro do tipo String ou padrão sem criptografia deixa o valor em texto puro.

KMS: key policy + IAM

O controle de uma chave KMS tem duas camadas: a key policy diz quem pode usar a chave e a política IAM da role diz quem pode chamar as APIs do KMS. Para restringir quem descriptografa, use chave gerenciada pelo cliente (CMK) com key policy liberando só a role específica. A chave gerenciada pela AWS (como aws/ssm) não permite esse ajuste de policy.

Criptografia em repouso gerenciada

Quando o requisito é criptografar sem mudar a aplicação e com menor esforço, use a criptografia nativa do serviço, como a do DynamoDB com chave gerenciada pela AWS (AWS owned key). Se o requisito pede controle de política e auditoria de uso da chave, use CMK do cliente integrada ao serviço (DynamoDB, S3, RDS).

Amazon Cognito User Pool

Diretório de usuários que faz login, federa com provedores externos via OIDC, SAML ou login social e emite tokens JWT. Integrado ao API Gateway como Cognito authorizer, que valida assinatura, expiração, issuer e audience do token enviado no cabeçalho Authorization. Claims e grupos do token permitem autorização por papel.

Amazon Cognito Identity Pool

Troca uma identidade autenticada por credenciais AWS temporárias, mapeando-a para roles IAM. É a resposta quando um app web ou mobile precisa acessar serviços AWS diretamente sem chaves fixas no cliente.

Authorizers do API Gateway

Cognito authorizer valida JWT do User Pool; JWT authorizer nativo (HTTP APIs) aceita tokens de um provedor OIDC configurando issuer e audience; Lambda authorizer é código customizado para regras específicas, como validar header secreto ou consultar introspecção; autorização IAM (execute-api:Invoke) é para chamadas entre serviços com credenciais da role.

API keys e usage plans

Servem para identificar clientes, aplicar cotas e controlar consumo. Não são mecanismo de autenticação forte de usuário final e não substituem Cognito, JWT ou IAM.

ALB com HTTPS e autenticação

O listener HTTPS (porta 443) usa certificado do AWS Certificate Manager e pode exigir autenticação OIDC (Cognito ou outro IdP) antes de encaminhar ao target group. Quando o enunciado pede criptografar todo o tráfego, de ponta a ponta, o target group também deve usar HTTPS até as instâncias.

Autenticação IAM no RDS

A aplicação conecta ao RDS MySQL ou PostgreSQL com token temporário gerado pelo SDK, e a role (por exemplo, task role do ECS) recebe rds-db:connect. Elimina senha de banco armazenada na aplicação.

VPC endpoints

Endpoint de gateway do S3 mantém o tráfego do EC2 ou ECS dentro da VPC e aceita política restringindo a role. Endpoint de interface (AWS PrivateLink) atende serviços como Secrets Manager sem sair para a internet. Políticas de recurso podem checar aws:sourceVpce.

Segurança de JWT

Para revogar tokens antes de expirarem, inclua o claim jti e consulte uma denylist no DynamoDB com leitura fortemente consistente, ou use introspecção no IdP com tokens curtos. Para assinar JWT entre serviços, use chave KMS assimétrica: quem emite chama a API Sign e os demais validam com a chave pública.

Bucket central de logs do CloudTrail

Bucket dedicado com bucket policy liberando s3:PutObject para cloudtrail.amazonaws.com condicionado por aws:SourceArn, SSE-KMS com CMK e Object Lock em modo Compliance para imutabilidade.

Macetes para memorizar

Credenciais
Quem roda na AWS usa role; quem guarda chave perde a questão.

Segredos
Rotação? Secrets Manager. Só configuração? Parameter Store SecureString.

Cognito
User Pool diz quem você é (JWT); Identity Pool diz o que você pode na AWS (credenciais).

KMS
A chave tem duas portas: key policy e política IAM; as duas precisam abrir.

API Gateway
API key mede consumo, não autentica.

EKS
Pod não usa chave, usa role por service account (IRSA).

Onde nunca guardar segredo
Segredo não mora em código, imagem, user data, repositório, bucket S3 nem como valor fixo em variável de ambiente: a aplicação lê do Secrets Manager ou do Parameter Store SecureString em tempo de execução.

Tabela de memorização: não confunda

Quando é...	...e quando é
AWS Secrets Manager O enunciado fala em rotação automática, credenciais de banco com integração nativa ao RDS, chaves de API com ciclo de vida e auditoria.	**Parameter Store SecureString** O foco é configuração e segredos simples, sem rotação complexa, com criptografia KMS e custo menor.
Cognito User Pool Autenticar usuários finais (login, federação OIDC/SAML/social) e emitir JWT para o API Gateway validar.	**Cognito Identity Pool** Dar ao app cliente credenciais AWS temporárias, via role IAM, para acessar serviços diretamente.
Cognito authorizer ou JWT authorizer nativo Os tokens vêm do User Pool ou de um provedor OIDC e o requisito é reduzir código de segurança.	**Lambda authorizer** Há regra customizada que o nativo não cobre, como validar header secreto enviado pelo CloudFront ou checar revogação em tempo real.
Autorização IAM no API Gateway (execute-api:Invoke) Chamadas internas entre serviços da mesma conta (EC2, Lambda, ECS) usando a role.	**Cognito User Pool authorizer** APIs consumidas por usuários finais em apps web ou mobile.

CMK gerenciada pelo cliente É preciso key policy restrita a roles específicas, auditoria de uso e controle sobre a chave.	**Chave gerenciada pela AWS** Basta criptografia em repouso sem mudança na aplicação e com o menor esforço.
SSE-KMS com CMK É preciso controlar quem pode descriptografar os objetos (por role) e auditar.	**SSE-S3** Basta criptografia em repouso, sem controle refinado de quem descriptografa.
Task role do ECS Cada tarefa ECS/Fargate precisa das próprias permissões mínimas.	**Instance profile do EC2** A aplicação roda diretamente em instâncias EC2.

Números que caem

Item	Valor
Peso de Segurança no guia oficial DVA-C02	**26% do conteúdo pontuado**
Porta do listener HTTPS do ALB para tráfego criptografado	**443**

Alertas de prova (pegadinhas)

⚠ **Alerta de prova:** Guardar a access key de um usuário IAM no Secrets Manager, no Parameter Store ou criptografada com KMS e considerar resolvido.
Como não cair: Continua sendo credencial estática de longa duração. Se o requisito é não usar chaves, a resposta é role (execution role, task role, instance profile, IRSA).

⚠ **Alerta de prova:** Variáveis de ambiente criptografadas com KMS bastam para segredos.
Como não cair: O segredo continua exposto como variável e sujeito a logs e dumps. Leia em tempo de execução do Secrets Manager ou do Parameter Store SecureString.

⚠ **Alerta de prova:** Proteger a API com API keys distribuídas aos clientes.
Como não cair: API key é controle de uso e cota, não autenticação. Use Cognito, JWT authorizer ou IAM.

⚠ **Alerta de prova:** Habilitar CORS com Access-Control-Allow-Origin restrito para impedir acesso direto à API.
Como não cair: CORS só orienta o navegador; não impede chamadas diretas. Valide token ou header secreto no API Gateway (Lambda authorizer, JWT, IAM).

⚠ **Alerta de prova:** Usar SecureString com a chave gerenciada pela AWS (aws/ssm) quando o enunciado exige CMK do cliente e acesso restrito por role.
Como não cair: Crie CMK própria e ajuste a key policy para permitir uso apenas pela role de execução, além da política IAM.

⚠ **Alerta de prova:** Guardar segredo como parâmetro String (sem criptografia) no Parameter Store, restringindo por IAM.
Como não cair: Sem SecureString o valor fica em texto puro. Segredo exige criptografia KMS.

⚠ **Alerta de prova:** Usar bucket S3 com SSE-S3 como cofre de segredos.
Como não cair: SSE-S3 não controla quem descriptografa e S3 não é serviço de segredos. Use Secrets Manager ou Parameter Store.

⚠ **Alerta de prova:** Key policy concedendo acesso irrestrito à conta e controle só por políticas IAM de usuários.
Como não cair: Aumenta o risco de uso indevido. Restrinja a key policy às roles que realmente usam a chave.

⚠ **Alerta de prova:** Escrever um Lambda authorizer próprio para validar JWT do Cognito.
Como não cair: É código a manter e sujeito a bugs; o Cognito authorizer ou o JWT authorizer nativo validam assinatura, expiração, issuer e audience sem código.

⚠ **Alerta de prova:** Usar role com trust policy para EC2 em uma aplicação frontend no navegador.
Como não cair: Role de EC2 é para instância. Cliente web ou mobile obtém credenciais temporárias pelo Cognito Identity Pool.

⚠ **Alerta de prova:** Parameter Store com SecureString para senha de RDS que precisa de rotação automática.
Como não cair: A rotação manual continua com você. Rotação automática integrada ao RDS é Secrets Manager.

⚠ **Alerta de prova:** Criar usuários IAM para cada usuário final e federar via STS AssumeRoleWithWebIdentity.
Como não cair: IAM não é diretório de usuários finais. Federação de usuários é Cognito User Pool.

⚠ **Alerta de prova:** Ligar log DEBUG com variáveis de ambiente e payloads para investigar.
Como não cair: Isso expõe segredos nos logs. Registre identificadores, não valores.

⚠ **Alerta de prova:** Colocar credenciais no user data da instância ou embutidas na imagem do contêiner.
Como não cair: Ficam em históricos, artefatos e logs. Role no runtime + segredo lido do serviço gerenciado em tempo de execução.

Como cai na prova

📝 **Como cai:** Aplicação em Lambda/ECS/EC2 hoje usa access key em variável ou arquivo; pede-se a solução mais segura.
Como resolver: Elimine toda alternativa que mantém chave de usuário IAM, mesmo criptografada ou guardada em serviço de segredos. Escolha a role do runtime com permissões mínimas nos ARNs necessários.

📝 **Como cai:** Segredos de banco ou API precisam de rotação, auditoria e leitura em runtime.
Como resolver: Rotação explícita aponta para Secrets Manager + role com secretsmanager:GetSecretValue. Sem rotação, Parameter Store SecureString + ssm:GetParameter. Descarte S3, variáveis de ambiente e criptografia manual.

📝 **Como cai:** API pública para apps web e mobile com login federado e menor código de segurança no backend.
Como resolver: Cognito User Pool federando o IdP + authorizer nativo no API Gateway validando o JWT. Descarte API keys, Lambda authorizer desnecessário e usuários IAM.

📝 **Como cai:** Requisito de controlar quem pode descriptografar dados ou parâmetros com KMS.
Como resolver: Procure CMK gerenciada pelo cliente com key policy restrita à role e permissão kms:Decrypt na política IAM. Descarte SSE-S3 e chave gerenciada pela AWS.

📝 **Como cai:** Política com 'Resource': '*' ou FullAccess precisa ser corrigida.
Como resolver: Mantenha só as ações necessárias e troque o recurso pelo ARN exato (tabela, tópico, parâmetro, segredo).

Gatilhos do enunciado

Se o enunciado trouxer...	Pense em
rotação automática de segredos, credenciais de RDS	**AWS Secrets Manager**
parâmetros de configuração sensíveis, sem rotação, custo baixo	**Parameter Store SecureString com KMS**

Lambda, ECS ou EC2 acessando S3/DynamoDB sem chaves	**Execution role, task role ou instance profile com menor privilégio**
pods do EKS precisam acessar S3	**IRSA**
login social ou corporativo, OAuth 2.0, OIDC, JWT, API Gateway	**Cognito User Pool + Cognito authorizer (ou JWT authorizer em HTTP APIs)**
app mobile ou web acessando AWS com credenciais temporárias	**Cognito Identity Pool mapeando para roles IAM**
instâncias EC2 chamando API interna do API Gateway	**Autorização IAM no método + execute-api:Invoke na role**
apenas determinadas funções podem descriptografar	**CMK do cliente + key policy restrita + permissão kms:Decrypt na role**
criptografar em repouso sem mudar a aplicação, menor esforço	**Criptografia nativa do serviço com chave gerenciada pela AWS**
tráfego ao S3 ou ao Secrets Manager sem sair da VPC	**Endpoint de gateway (S3) ou endpoint de interface/PrivateLink**
criptografar todo o tráfego, de ponta a ponta, com ALB	**Listener HTTPS 443 com certificado ACM e target group em HTTPS**
conectar ao RDS sem senha armazenada	**Autenticação IAM do RDS com rds-db:connect e token do SDK**
API só pode ser acessada via CloudFront	**Lambda authorizer validando header secreto enviado pelo CloudFront**
revogar JWT antes da expiração	**Claim jti + denylist no DynamoDB (leitura fortemente consistente) ou introspecção**

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

- Access key de usuário IAM guardada em qualquer lugar continua errada quando existe role.
- Rotação automática e integração com RDS = Secrets Manager.
- Restringir descriptografia = CMK do cliente + key policy + IAM.
- API keys e CORS não são autenticação.

Essenciais para a prova (checklist)

- ☐ Sei qual role usar em Lambda, ECS, EC2 e EKS (execution role, task role, instance profile, IRSA).
- ☐ Sei escrever permissão de menor privilégio com ação exata e ARN específico.
- ☐ Sei quando escolher Secrets Manager e quando Parameter Store SecureString.

- ☐ Sei que parâmetro String no Parameter Store fica em texto puro.
- ☐ Sei diferenciar Cognito User Pool de Identity Pool.
- ☐ Sei escolher entre Cognito authorizer, JWT authorizer, Lambda authorizer e autorização IAM no API Gateway.
- ☐ Sei explicar as duas camadas de controle do KMS e quando usar CMK do cliente.
- ☐ Sei configurar HTTPS ponta a ponta no ALB com certificado ACM.
- ☐ Sei usar autenticação IAM no RDS e VPC endpoints para manter tráfego privado.

Questões de revisão do capítulo

R3.1 · Segurança

Uma aplicação serverless utiliza AWS Lambda com integração ao Amazon API Gateway. O time de desenvolvimento precisa garantir que apenas clientes autorizados consigam invocar as funções Lambda e, ao mesmo tempo, quer evitar o gerenciamento manual de chaves de acesso de longo prazo. A solução deve permitir revogação rápida, rotação simples e integração nativa com o API Gateway. Qual é a abordagem MAIS adequada do ponto de vista de segurança para esse cenário?

A. Criar um Custom Lambda Authorizer que valida uma lista estática de tokens compartilhados, armazenados em texto puro no código do authorizer, para liberar ou negar o acesso às rotas.
B. Expor diretamente a URL da função Lambda e usar chaves de acesso IAM de longo prazo gravadas em variáveis de ambiente do cliente para assinar as requisições.
C. Configurar o API Gateway com um Amazon Cognito User Pool Authorizer, emitindo JSON Web Tokens (JWT) para os clientes, e usar políticas de autorização baseadas em claims para controlar o acesso às rotas.
D. Ativar API Keys no API Gateway e usá-las como único mecanismo de controle de acesso, sem qualquer validação adicional de identidade ou autorização.

R3.2 · Segurança

Um time de desenvolvimento mantém variáveis de ambiente sensíveis (como chaves de API e segredos de banco de dados) diretamente no código-fonte de um aplicativo serverless em execução no AWS Lambda. A equipe deseja aumentar a segurança desses segredos sem alterar a forma como a função é invocada e minimizando mudanças de código. Qual é a abordagem MAIS adequada seguindo boas práticas de segurança da AWS?

A. Manter as variáveis sensíveis no código-fonte, mas proteger o repositório Git
B. Armazenar as variáveis sensíveis em arquivos JSON dentro do bucket Amazon S3 e configurar a função Lambda para ler e descriptografar esses arquivos à mão usando a AWS SDK
C. Desligar variáveis de ambiente e passar os segredos no payload da requisição para o Lambda por meio do Amazon API Gateway, usando só HTTPS para proteger os dados em trânsito
D. Mover os valores sensíveis para SecureString no Parameter Store com KMS e lê-los na função

R3.3 · Segurança

Você está projetando a autenticação de uma API em Amazon API Gateway que será consumida por aplicações web e mobile. Os requisitos são: (1) permitir autenticação federada com vários provedores de identidade (IdPs) externos via OpenID Connect e SAML 2.0; (2) manter o mínimo de código customizado possível para emissão e validação de tokens; (3) armazenar atributos de autorização em grupos e atributos de usuário centralizados; (4) permitir rotação automática de chaves de assinatura dos tokens. Qual abordagem atende melhor a esses requisitos com o menor esforço operacional?

A. Implementar um serviço customizado em AWS Lambda para autenticação, integrando à mão com cada IdP via SDK, emitindo tokens JWT assinados com uma chave gerenciada no AWS KMS, e usar um Lambda Authorizer no API Gateway.
B. Criar usuários IAM para cada usuário final, usar federation via STS AssumeRoleWithWebIdentity com cada IdP e configurar o API Gateway para aceitar só chamadas assinadas com Signature Version 4.
C. Configurar um Amazon Cognito User Pool como provedor de identidade, federando com IdPs externos via OIDC/SAML, e usar um Authorizer do tipo JWT no API Gateway validando os tokens do User Pool.
D. Configurar o API Gateway com um Custom Domain protegido por AWS WAF, usar autenticação básica (Basic Auth) em um Lambda Authorizer e armazenar credenciais em AWS Secrets Manager, aplicando criptografia com KMS.

R3.4 · Segurança

Uma empresa precisa armazenar segredos de aplicação (como credenciais para acessar um banco de dados externo) de forma segura na AWS. Os desenvolvedores querem que a aplicação obtenha esses segredos de forma programática, com rotação automática e trilha de auditoria de acessos, sem armazenar segredos em variáveis de ambiente ou arquivos de configuração. Qual é a solução MAIS apropriada para esse requisito?

A. Salvar os segredos em um banco de dados Amazon DynamoDB com criptografia em repouso e restringir o acesso ao DynamoDB apenas à aplicação.
B. Criar um bucket S3 privado para armazenar um arquivo JSON com os segredos criptografados e usar uma chave KMS para descriptografar em tempo de execução.
C. Utilizar o AWS Secrets Manager para armazenar os segredos, habilitar rotação automática e conceder acesso via políticas IAM anexadas ao papel de execução da aplicação.
D. Armazenar os segredos em parâmetros SecureString no AWS Systems Manager Parameter Store, usando criptografia com AWS KMS e controlar o acesso via IAM.

Respostas comentadas

R3.1. C) Configurar o API Gateway com um Amazon Cognito User Pool Authorizer, emitindo JSON Web Tokens (JWT) para os clientes, e usar políticas de autorização baseadas em claims para controlar o acesso às rotas.

Por quê: Esta alternativa é a mais adequada porque o Amazon Cognito User Pool Authorizer integra-se nativamente ao API Gateway, emitindo tokens JWT de curta duração. Isso elimina a necessidade de gerenciar chaves de acesso de longo prazo no cliente, facilita rotação e revogação (revogando tokens, desabilitando usuários ou alterando o User Pool) e permite autorização granular baseada em claims contidas no token.

✗ A) Incorreta. Armazenar tokens estáticos em texto puro no código e validá-los em um Custom Authorizer não resolve o problema de rotação e revogação de forma eficiente.

✗ B) Incorreta. Utilizar chaves de acesso IAM de longo prazo no cliente é uma grave má prática de segurança. Essas credenciais podem vazar facilmente (por exemplo, código-fonte, logs, app mobile descompilado) e são difíceis de revogar de forma...

✗ D) Incorreta. API Keys no API Gateway servem principalmente para controle de uso (throttling, quotas e billing), não para autenticação/autorização forte de usuários.

> **Dica de prova:** Para APIs públicas ou mobile/web, priorize Amazon Cognito (User Pools + Federated Identities) com JWT integrado ao API Gateway. Use API Keys apenas como complemento para controle de consumo, e nunca armazene chaves IAM de longo prazo em clientes.

R3.2. D) Mover os valores sensíveis para SecureString no Parameter Store com KMS e lê-los na função

Por quê: Esta alternativa é a mais adequada porque utiliza o AWS Systems Manager Parameter Store com parâmetros do tipo SecureString, que suportam criptografia gerenciada pelo AWS KMS. Isso permite armazenar segredos de forma centralizada, com controle de acesso via IAM, auditoria e rotação facilitada. A função Lambda pode ser configurada para obter esses valores em tempo de execução ou via integração com variáveis de ambiente, reduzindo alterações de código e mantendo um fluxo de invocação simples e seguro.

✗ A) Incorreta. Proteger o repositório com MFA é boa prática, mas não resolve o problema central de ter segredos em texto claro no código-fonte. Se o código vazar (logs, backups, forks, screenshots, etc.), os segredos continuam expostos.

✗ B) Incorreta. Armazenar segredos em S3 exige implementação manual de leitura e criptografia/descriptografia, aumentando a complexidade e a superfície de erro.

✗ C) Incorreta. Enviar segredos no payload a cada requisição aumenta a exposição (logs de API, ferramentas de observabilidade, clientes comprometidos) e complica o consumo. HTTPS protege apenas em trânsito;

> **Dica de prova:** Para o exame DVA-C02, memorize que segredos e credenciais NUNCA devem estar em código ou em texto claro em variáveis de ambiente. Use serviços gerenciados de segredos: AWS Systems Manager Parameter Store (SecureString) e AWS Secrets Manager.

R3.3. C) Configurar um Amazon Cognito User Pool como provedor de identidade, federando com IdPs externos via OIDC/SAML, e usar um Authorizer do tipo JWT no API Gateway validando os tokens do User Pool.

Por quê: Esta alternativa usa Amazon Cognito User Pools como provedor de identidade gerenciado, o que atende a todos os requisitos com mínimo esforço de código e operação. O Cognito oferece federação nativa com diversos IdPs (OpenID Connect e SAML 2.0), gerencia o ciclo de vida dos usuários, grupos e atributos de autorização, emite tokens JWT (ID/Access tokens) e faz rotação automática das chaves de assinatura (publicadas via JWKs).

✗ A) Incorreta. Exige implementação e manutenção de um serviço customizado em Lambda para lidar com vários IdPs, mapeamento de claims, emissão e rotação de chaves JWT.

✗ B) Incorreta. IAM não é adequado para gerenciar usuários finais (de aplicações web/mobile) em grande escala; é pensado para identidades de nível conta/serviço.

✗ D) Incorreta. Autenticação básica com usuário/senha em Authorizer não explora federação com IdPs via OIDC/SAML nem fornece tokens JWT padronizados.

> **Dica de prova:** Para APIs serverless no API Gateway que precisam de login de usuários finais (web/mobile) com federação social ou corporativa, pense em Amazon Cognito User Pools + JWT Authorizer.

R3.4. C) Utilizar o AWS Secrets Manager para armazenar os segredos, habilitar rotação automática e conceder acesso via políticas IAM anexadas ao papel de execução da aplicação.

Por quê: Esta alternativa é a mais adequada porque o AWS Secrets Manager foi projetado especificamente para gerenciamento de segredos, incluindo rotação automática integrada para diversos tipos de credenciais (como bancos de dados e serviços externos), integração nativa com IAM para controle de acesso granular e registro de chamadas via AWS CloudTrail, fornecendo trilha de auditoria completa.

✗ A) Incorreta. Usar DynamoDB apenas para guardar segredos não é uma boa prática; embora ofereça criptografia em repouso, não foi projetado para gerenciamento de segredos, nem provê mecanismos nativos de rotação automática, o que fere o...

✗ B) Incorreta. Armazenar segredos em um arquivo no S3 criptografado com KMS é possível, mas exige que a equipe implemente manualmente a lógica de rotação, versionamento e auditoria de acessos, aumentando a complexidade operacional e o risco de...

✗ D) Incorreta. O AWS Systems Manager Parameter Store com SecureString oferece criptografia e controle de acesso, porém não fornece, por padrão, rotação automática de segredos com a mesma facilidade e integração que o AWS Secrets Manager.

> **Dica de prova:** Quando o enunciado enfatizar rotação automática de segredos, integração simples com aplicações e auditoria de acessos, pense primeiro em AWS Secrets Manager. Já o Parameter Store é excelente para configurações e segredos menos dinâmicos, ou quando custo menor for um fator decisivo.

04

CAPÍTULO 4

Troubleshooting e Otimização

Neste domínio (no guia oficial chamado Solução de problemas e otimização, 18% do conteúdo pontuado) a prova descreve um sintoma em produção, como latência intermitente, erros 5xx, throttling, filas crescendo ou cold starts, e pede o primeiro passo certo. O raciocínio vencedor é observar antes de escalar: AWS X-Ray para traces e service map, métricas do CloudWatch (Init Duration, Throttles, ConcurrentExecutions, IteratorAge, ThrottledRequests, idade da mensagem mais antiga) e logs estruturados. Depois vêm as otimizações típicas: memória da Lambda que também aumenta CPU, clientes e conexões fora do handler, RDS Proxy para explosão de conexões, operações em lote, backoff exponencial com jitter e redesenho de chave de partição para hot partitions. Onde mais erro é escolher alternativas que só mascaram o problema, como aumentar timeout, dobrar memória sem evidência, adicionar tarefas ou mover para VPC.

Neste capítulo

18 tópicos-chave · 13 pegadinhas · 7 macetes · 14 gatilhos · 4 questões comentadas

Tópicos-chave

AWS X-Ray

Rastreamento distribuído que mostra o caminho completo da requisição (API Gateway, Lambda, ALB, banco, serviços externos) e o tempo em cada trecho. O service map dá a visão global e os traces mostram o detalhe; anotações e subsegmentos instrumentam chamadas ao RDS, DynamoDB, S3 ou APIs externas. É a primeira escolha quando CPU e memória estão baixas, mas latência e erros sobem.

CloudTrail não é tracing

CloudTrail registra chamadas de API para auditoria; não mostra o tempo gasto em cada parte de uma requisição de aplicação. Para gargalo de latência, use X-Ray junto com CloudWatch Logs e métricas.

Init Duration x Duration

Init Duration alta com Duration estável indica cold start. A resposta é provisioned concurrency no alias estável de produção e inicialização mais enxuta (clientes e bibliotecas fora do handler, pacote menor).

Memória da Lambda = CPU e rede

Aumentar a memória aumenta proporcionalmente CPU e throughput de rede, o que costuma reduzir a duração de funções CPU-bound, mas não garante custo total menor. Não resolve gargalo de I/O externo, muitas chamadas individuais ou cold start.

Reuso de clientes e conexões

Criar o cliente do DynamoDB, boto3, conexões de banco e clientes HTTP fora do handler aproveita o ambiente de execução reutilizado. Ajuste pools de conexão, keep-alive, timeouts curtos e retries. Cuidado: estruturas globais que crescem a cada invocação causam vazamento de memória.

Amazon RDS Proxy

Faz pooling e reuso de conexões entre muitas instâncias concorrentes da Lambda (ou aplicações) e o RDS/Aurora. É a solução quando há muitas conexões curtas, esgotamento de conexões ou latência de conexão, em vez de só aumentar max_connections ou escalar o banco.

Throttling da Lambda

Monitore Throttles, ConcurrentExecutions e UnreservedConcurrentExecutions. Use reserved concurrency para isolar a função crítica e, se necessário, solicite aumento de limite da conta. Reduzir a concorrência também é alavanca para não sobrecarregar o DynamoDB ou o banco.

ProvisionedThroughputExceededException no DynamoDB

Indica capacidade de leitura ou escrita excedida. Duas frentes: ajustar capacidade (auto scaling da capacidade provisionada ou modo on-demand) e melhorar o cliente com retry exponencial com jitter, batch e menos escritas por requisição.

Hot partition

Quando o tráfego se concentra em poucas chaves de partição, aumentar a capacidade da tabela não resolve, pois ela é distribuída entre partições. Redesenhe a chave: chaves compostas ou sharding lógico (como userId#1, userId#2) para espalhar a carga.

Operações em lote

Trocar chamadas por registro por BatchGetItem e BatchWriteItem, ler até 10 mensagens por ReceiveMessage e enviar várias métricas por PutMetricData reduz chamadas remotas, custo e latência. Chamadas independentes podem rodar em paralelo na mesma invocação (asyncio ou threads).

IteratorAge em Kinesis

IteratorAge alto mostra que a função não acompanha o stream. Ajuste BatchSize (menor para processar mais rápido), ParallelizationFactor (mais consumidores por shard), número de shards e concorrência. Lotes muito grandes demoram mais, aumentam a chance de timeout e, em caso de falha, fazem muitos registros serem reprocessados.

Métricas de fila

ApproximateNumberOfMessagesVisible e ApproximateAgeOfOldestMessage mostram backlog no SQS. Em consumidores ECS, o Service Auto Scaling deve usar métrica customizada baseada na fila, não CPU, quando a CPU não é o gargalo.

CloudWatch EMF e Logs Insights

O Embedded Metric Format publica métricas customizadas a partir de logs estruturados. CloudWatch Logs Insights consulta logs para encontrar dados relevantes. Métricas customizadas não substituem tracing quando o problema é de latência entre serviços.

Nível de log

Controle o nível por variável de ambiente (por exemplo, LOG_LEVEL), com INFO ou ERROR por padrão e DEBUG só quando necessário. Reduzir o volume de log e a retenção corta latência e custo do CloudWatch Logs; desligar logs inteiramente tira visibilidade.

Lambda SnapStart

Reaproveita um snapshot do estado já inicializado para reduzir a inicialização a frio, em runtimes compatíveis como Java. Na prova, propor SnapStart para função Node.js ou Python é alternativa errada; para esses casos, a resposta é provisioned concurrency e inicialização fora do handler.

Lambda em VPC

Colocar a função em VPC sem necessidade adiciona ENIs, NAT e rotas e pode aumentar o cold start. Só use VPC quando a função precisa acessar recursos privados; tirar a função da VPC e expor o banco publicamente é inseguro.

Diagnóstico com ALB

Access logs do ALB e métricas como TargetProcessingTime ajudam a separar latência do balanceador e da aplicação; erros 502 e 504 sem exceções no app apontam para conexões, keep-alive e timeouts. Least outstanding requests e keep-alive HTTP entre ALB e tarefas ajudam quando a latência sobe com CPU baixa; health check no caminho da aplicação com grace period adequado evita derrubar instâncias que ainda estão iniciando.

Read replicas x Multi-AZ

Gargalo de leitura no RDS se resolve com read replicas e direcionando as leituras para elas. Multi-AZ é alta disponibilidade, não ganho de desempenho de leitura.

Macetes para memorizar

💡 Ordem de ataque

Primeiro enxergar (X-Ray, métricas, logs), depois otimizar, por último escalar.

💡 Memória Lambda

Na Lambda, memória é o botão da CPU.

Cold start

Init alto = frio; Provisioned esquenta, fora do handler economiza.

Banco relacional + Lambda

Muita Lambda, pouca conexão: chame o RDS Proxy.

DynamoDB

Throttling geral = capacidade; throttling em uma chave = partição quente.

RDS

Réplica lê, Multi-AZ salva.

Timeout

Aumentar timeout é varrer o problema para debaixo do tapete.

Tabela de memorização: não confunda

Quando é...	...e quando é
AWS X-Ray Encontrar onde está a latência ou o erro dentro do fluxo de uma requisição entre serviços.	**AWS CloudTrail** Auditar quem chamou qual API da AWS e quando.
Provisioned concurrency O problema é cold start (Init Duration alta, latência após inatividade).	**Aumentar memória da Lambda** A função é CPU-bound e a duração cai com mais CPU.
RDS Proxy Explosão de conexões curtas vinda de alta concorrência da Lambda, com o banco sem saturar.	**Aumentar max_connections ou escalar o banco** O próprio banco está saturado de CPU ou I/O (e, para leitura, prefira read replicas).
Ajustar capacidade (auto scaling ou on-demand) O throttling ocorre na tabela como um todo, por picos de tráfego.	**Redesenhar a chave de partição** O tráfego se concentra em poucas chaves (hot partition).
Read replicas Escalar leituras de um RDS sobrecarregado.	**Multi-AZ** Alta disponibilidade e failover.
Auto Scaling do ECS por métrica de fila Consumidores de SQS com backlog crescendo e CPU baixa.	**Auto Scaling do ECS por CPU** A carga do serviço realmente se reflete em CPU e memória.

Números que caem

Item	Valor
Peso de Solução de problemas e otimização no guia oficial DVA-C02	**18% do conteúdo pontuado**

Alertas de prova (pegadinhas)

⚠ **Alerta de prova:** Aumentar o timeout da Lambda para resolver lentidão ou throttling.
Como não cair: Timeout maior só deixa a função falhar mais tarde e ocupar concorrência por mais tempo. Diagnostique com X-Ray e métricas e ataque a causa.

⚠ **Alerta de prova:** Dobrar a memória quando o gargalo é I/O de rede ou muitas chamadas individuais ao DynamoDB.
Como não cair: Mais CPU não resolve I/O. Use operações em lote, paralelismo e reuso de clientes.

⚠ **Alerta de prova:** Ativar SnapStart numa função Node.js ou Python.
Como não cair: SnapStart não se aplica a esses runtimes. Use provisioned concurrency e init fora do handler.

⚠ **Alerta de prova:** Colocar a Lambda em VPC (com NAT ou ENI dedicada) para ganhar desempenho.
Como não cair: VPC adiciona complexidade e pode aumentar o cold start. Só use VPC para acessar recursos privados.

⚠ **Alerta de prova:** Aumentar a capacidade provisionada da tabela para resolver throttling concentrado em uma chave.
Como não cair: A capacidade é distribuída entre partições; hot partition exige redesenho da chave (sharding lógico).

⚠ **Alerta de prova:** Aumentar max_connections e ligar provisioned concurrency para resolver estouro de conexões no RDS.
Como não cair: A raiz é o padrão explosivo de conexões por instância de Lambda. Use RDS Proxy.

⚠ **Alerta de prova:** Auto scaling por CPU quando CPU e memória estão abaixo do limite e a fila cresce.
Como não cair: Escale pela métrica da fila ou investigue I/O externo; CPU não é o gargalo.

⚠ **Alerta de prova:** Aumentar timeout e maximumBatchingWindowInSeconds para reduzir latência.
Como não cair: A janela de lote adiciona espera; tende a aumentar a latência.

⚠ **Alerta de prova:** Aumentar timeout do ALB de 60 para 120 segundos para acabar com os 504.
Como não cair: Só mascara o problema. Encontre o trecho lento com X-Ray e métricas do ALB.

⚠ **Alerta de prova:** Usar CloudTrail para investigar latência intermitente entre serviços.
Como não cair: CloudTrail é auditoria de API. Tracing de requisição é X-Ray.

⚠ **Alerta de prova:** Adicionar tarefas ECS ou migrar para EC2 sem entender a causa.
Como não cair: Mascara o problema e eleva custo. Observe antes de escalar.

⚠ **Alerta de prova:** Migrar o DynamoDB para RDS para fugir do throttling.
Como não cair: Muda toda a arquitetura sem necessidade. Ajuste capacidade, chave e retries.

⚠ **Alerta de prova:** Escolher Multi-AZ para melhorar leitura no RDS.
Como não cair: Multi-AZ é disponibilidade. Leitura escala com read replicas.

Como cai na prova

📝 **Como cai:** Latência intermitente ou erros 5xx, CPU e memória baixas, várias opções de escalar recursos.
Como resolver: Descarte escalar, aumentar timeout e mexer na rede; escolha observabilidade ponta a ponta (X-Ray com subsegmentos, métricas e logs) para achar o componente lento.

📝 **Como cai:** Lambda com métricas citadas (Init Duration, Duration, Throttles, ConcurrentExecutions).
Como resolver: Leia qual métrica mudou: Init alto = cold start; Throttles = concorrência; Duration alto com CPU-bound = memória; Duration alto por I/O = batch, paralelismo, reuso de clientes.

📝 **Como cai:** DynamoDB devolvendo ProvisionedThroughputExceededException em picos.
Como resolver: Verifique se é pico geral (auto scaling, on-demand, retry com jitter) ou chave concentrada (redesenho da chave). Descarte trocar de banco e aumentar timeout.

📝 **Como cai:** Fila SQS ou stream Kinesis acumulando.
Como resolver: Compare produção e consumo; ajuste batch, concorrência, ParallelizationFactor ou paralelismo no consumidor, alinhe visibility timeout e use DLQ; para falhas parciais, partial batch response na fila SQS e bisectBatchOnFunctionError no stream.

📝 **Como cai:** Lambda + RDS com estouro de conexões e latência sob alta concorrência.
Como resolver: RDS Proxy para pooling, conexão criada fora do handler e, para cold start, provisioned concurrency.

Gatilhos do enunciado

Se o enunciado trouxer...	Pense em

latência alta com CPU e memória baixas	**X-Ray para achar dependência lenta (banco, API externa, conexões)**
Init Duration aumentando, Duration estável	**Cold start: provisioned concurrency e init fora do handler**
cliente criado dentro do handler a cada invocação	**Mover clientes e conexões para o escopo global**
muitas conexões curtas ao RDS vindas de Lambda	**RDS Proxy**
ProvisionedThroughputExceededException ou ThrottledRequests	**Auto scaling ou on-demand + retry exponencial com jitter**
throttling concentrado em poucas chaves	**Hot partition: sharding lógico ou chave composta**
ThrottlingException na Lambda, função crítica afetada	**Reserved concurrency e pedido de aumento de limite**
IteratorAge crescendo no Kinesis	**BatchSize menor, ParallelizationFactor maior, mais shards**
função CPU-bound lenta	**Aumentar memória (mais CPU proporcional)**
loop de chamadas por item ao DynamoDB	**BatchGetItem e BatchWriteItem**
custo e latência altos de CloudWatch Logs	**Nível de log por variável de ambiente e retenção menor**
erros 502 no API Gateway com função de 25 segundos e timeout de 30	**Alinhar timeouts ao limite de 29 segundos da integração**
gargalo de leitura no RDS	**Read replicas, não Multi-AZ**
métricas customizadas a partir de logs	**CloudWatch Embedded Metric Format (EMF)**

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

- Observar antes de escalar: X-Ray, métricas e logs vêm primeiro.
- Mais memória na Lambda = mais CPU; não cura cold start nem I/O.
- RDS Proxy para conexões da Lambda; read replicas para leitura; Multi-AZ só disponibilidade.
- Hot partition se resolve na chave, não na capacidade.

Essenciais para a prova (checklist)

- ☐ Sei usar X-Ray (service map, traces, anotações, subsegmentos) para localizar gargalos.
- ☐ Sei ler Init Duration, Duration, Throttles, ConcurrentExecutions e IteratorAge.
- ☐ Sei quando aumentar memória da Lambda ajuda e quando não ajuda.

- ☐ Sei por que inicializar clientes e conexões fora do handler e evitar estado global crescente.
- ☐ Sei quando usar RDS Proxy, read replicas e Multi-AZ.
- ☐ Sei tratar ProvisionedThroughputExceededException e hot partitions no DynamoDB.
- ☐ Sei otimizar consumidores SQS e Kinesis com batch, concorrência e ParallelizationFactor.
- ☐ Sei reduzir custo e latência de logs com nível ajustável por variável de ambiente.
- ☐ Sei que SnapStart é para runtimes compatíveis como Java e que, na prova, não é resposta para Node.js ou Python.

Questões de revisão do capítulo

R4.1 · Troubleshooting e Otimização

Um desenvolvedor está otimizando uma função AWS Lambda escrita em Python que consome mensagens de uma fila Amazon SQS. Em ambiente de produção, o CloudWatch Logs mostra aumento significativo no tempo médio de execução e no uso de memória após a adição de novas bibliotecas ao pacote de implantação. O desenvolvedor também observou que a função está realizando o download de um mesmo arquivo de configuração do Amazon S3 a cada invocação. Qual abordagem oferece a MELHOR otimização de desempenho e custo para essa função, mantendo a mesma lógica de negócio?

A. Aumentar a memória configurada da função Lambda para o dobro do valor atual e reduzir a concorrência reservada para diminuir o número de execuções simultâneas.
B. Configurar a função Lambda para rodar dentro de uma VPC e utilizar o Amazon EFS para armazenar o arquivo de configuração de forma persistente entre as invocações.
C. Mover o arquivo de configuração para variáveis de ambiente e reduzir o timeout da função Lambda para minimizar o tempo de execução.
D. Utilizar camadas (Lambda Layers) para armazenar as bibliotecas adicionadas e carregar o arquivo de configuração em variáveis globais fora do handler para reaproveitar entre invocações.

R4.2 · Troubleshooting e Otimização

Uma função AWS Lambda escrita em Python processa mensagens de um tópico Amazon SNS e grava resultados em uma tabela do Amazon DynamoDB. Em produção, usuários relatam alta latência e, ocasionalmente, timeouts na API que aciona o SNS. As métricas do CloudWatch mostram aumento de 'Duration' e 'Throttles' na Lambda, além de 'ProvisionedThroughputExceededException' no DynamoDB. O que o desenvolvedor deve fazer para otimizar o desempenho e reduzir erros de limitação?

A. Habilitar reservas de concorrência (reserved concurrency) para a função Lambda e ajustar a capacidade provisionada ou habilitar modo on-demand na tabela DynamoDB
B. Reduzir a memória configurada da função Lambda para forçar menos concorrência e diminuir o número de gravações simultâneas no DynamoDB
C. Aumentar o tempo limite (timeout) da função Lambda e ativar o AWS X-Ray apenas em ambiente de desenvolvimento para identificar gargalos de código
D. Configurar o SNS para reter mensagens por mais tempo e processá-las em lotes maiores na Lambda, reduzindo o número de invocações

R4.3 · Troubleshooting e Otimização

Um desenvolvedor implantou uma função AWS Lambda escrita em Python que realiza consultas frequentes a uma tabela do Amazon DynamoDB. Após algumas semanas em produção, métricas do Amazon CloudWatch indicam aumento significativo no tempo de execução da função e custos de computação mais altos que o esperado. A análise de logs mostra que a função está passando muito tempo estabelecendo conexões com o DynamoDB e inicializando bibliotecas em cada invocação. Qual é a maneira MAIS eficiente de otimizar o desempenho e reduzir custos sem alterar a lógica de negócio principal da função?

A. Aumentar o tempo limite (timeout) e a memória alocada para a função Lambda para reduzir o risco de timeouts e permitir mais recursos de CPU
B. Inicializar o cliente do DynamoDB e as bibliotecas pesadas fora do handler da função Lambda, aproveitando o ambiente de execução reutilizado entre invocações
C. Configurar a função Lambda para ser invocada em uma VPC privada com sub-redes múltiplas para distribuir a carga de conexão com o DynamoDB
D. Mover toda a lógica de consulta para um script cron em uma instância Amazon EC2 e chamar a instância via HTTP sempre que a função Lambda precisar de dados

R4.4 · Troubleshooting e Otimização

Um Lambda que processa mensagens de uma fila Amazon SQS Standard começou a registrar tempos de processamento muito acima do esperado. A função realiza apenas operações de CPU leve e consultas rápidas em um banco de dados Amazon DynamoDB, mas o CloudWatch Logs mostra um aumento súbito no 'Init Duration' das invocações, enquanto a 'Duration' da execução permanece praticamente constante. O desenvolvedor já confirmou que não houve alteração recente de código nem de dependências. O que deve ser feito para reduzir a latência de inicialização e estabilizar o tempo total de resposta da função?

A. Configurar provisioned concurrency na função Lambda e garantir que o deployment use alias estável para atender o tráfego de produção
B. Aumentar a memória configurada da função Lambda para o dobro do valor atual, pois isso reduz automaticamente o 'Init Duration' em qualquer cenário
C. Ativar o dimensionamento provisionado (provisioned throughput) na tabela DynamoDB para reduzir o tempo de consulta e, assim, diminuir o 'Init Duration'
D. Migrar a função para um contêiner em execução no AWS Fargate, pois o Lambda não é adequado para workloads que acessam DynamoDB com baixa latência

Respostas comentadas

R4.1. D) Utilizar camadas (Lambda Layers) para armazenar as bibliotecas adicionadas e carregar o arquivo de configuração em variáveis globais fora do handler para reaproveitar entre invocações.

Por quê: Esta alternativa é a melhor abordagem porque ataca diretamente as duas causas principais de degradação: tamanho do pacote de implantação e trabalho repetitivo a cada invocação. Ao mover as bibliotecas adicionais para uma camada (Lambda Layer), o pacote de implantação da função fica menor, o que reduz o tempo de cold start e facilita o versionamento e o reuso entre funções.

✗ A) Incorreta. Aumentar memória pode até reduzir a duração em alguns cenários, mas aumenta o custo por invocação. Não resolve a raiz do problema (pacote inchado e download repetido do S3).

✗ B) Incorreta. Colocar a função em uma VPC e usar Amazon EFS adiciona latência de rede e complexidade de configuração (ENIs, sub-redes, segurança).

✗ C) Incorreta. Variáveis de ambiente são úteis para pequenos valores de configuração, mas não substituem bem um arquivo de configuração maior em S3.

> **Dica de prova:** Para otimizar Lambda: (1) Use variáveis globais para conexões e dados de configuração que podem ser reutilizados entre invocações; (2) Utilize Lambda Layers para bibliotecas grandes e compartilhadas; (3) Evite I/O repetitivo (S3, banco, APIs externas) dentro do handler sempre que possível;

R4.2. A) Habilitar reservas de concorrência (reserved concurrency) para a função Lambda e ajustar a capacidade provisionada ou habilitar modo on-demand na tabela DynamoDB

Por quê: Esta alternativa é a melhor solução porque atua nos dois pontos principais de limitação identificados pelas métricas: concorrência da Lambda e capacidade do DynamoDB. Ao configurar 'reserved concurrency', o desenvolvedor controla o número máximo de execuções simultâneas, evitando estouros repentinos de concorrência que pressionam o DynamoDB.

✗ B) Incorreta. Reduzir a memória não é uma estratégia adequada de controle de concorrência. Menos memória tende a deixar a função mais lenta, o que pode aumentar a duração média das execuções e, indiretamente, manter a concorrência alta por...

✗ C) Incorreta. Apenas aumentar o timeout não resolve o problema de limitação nem de throughput do DynamoDB. Aumentar o tempo de execução pode inclusive mascarar o problema e aumentar custos.

✗ D) Incorreta. Aumentar retenção no SNS e usar lotes maiores não endereça diretamente a causa raiz: limitação de throughput do DynamoDB e concorrência descontrolada da Lambda.

> **Dica de prova:** Quando vir métricas de 'Throttles' na Lambda e 'ProvisionedThroughputExceededException' no DynamoDB, pense em dois ajustes complementares: controle de concorrência na camada de consumo (Lambda) e ajuste do modelo de capacidade do DynamoDB (provisionado com auto scaling bem configurado ou on-demand).

R4.3. B) Inicializar o cliente do DynamoDB e as bibliotecas pesadas fora do handler da função Lambda, aproveitando o ambiente de execução reutilizado entre invocações

Por quê: Esta alternativa é a correta porque mover a criação do cliente do DynamoDB e a inicialização de bibliotecas pesadas para fora do handler permite que esses objetos sejam reutilizados entre invocações na mesma instância de execução da Lambda. Isso reduz o overhead de inicialização (cold start e setup em cada chamada), diminuindo o tempo total de execução e, consequentemente, o custo. Essa é uma prática recomendada da AWS para otimizar o desempenho de funções Lambda que acessam serviços como DynamoDB.

✗ A) Incorreta. Aumentar timeout e memória pode mitigar timeouts, mas não otimiza a causa raiz (inicialização repetida de recursos).

✗ C) Incorreta. Colocar a função Lambda em uma VPC privada aumenta a complexidade de rede (NAT, ENIs, etc.) e costuma introduzir overhead adicional de inicialização de rede.

✗ D) Incorreta. Migrar a lógica para uma instância EC2 adiciona complexidade operacional, remove a vantagem de escalabilidade automática da Lambda e não resolve diretamente o problema de overhead de inicialização dentro da função.

> **Dica de prova:** Sempre que possível, inicialize clientes de SDK (como boto3), conexões e bibliotecas pesadas fora do handler da Lambda. O código fora do handler é executado apenas na inicialização do ambiente, e o runtime reutiliza a mesma instância entre invocações, reduzindo o custo por chamada.

R4.4. A) Configurar provisioned concurrency na função Lambda e garantir que o deployment use alias estável para atender o tráfego de produção

Por quê: A métrica 'Init Duration' alta indica que o problema está em cold starts da função Lambda (fase de inicialização do ambiente), não na execução do código em si. A forma recomendada de reduzir e estabilizar esse tempo é usar provisioned concurrency, que mantém um número configurado de instâncias da função já inicializadas e prontas para atender requisições. Ao associar o tráfego de produção a um alias com provisioned concurrency, você diminui a variação de latência causada por cold starts.

✗ B) Incorreta. Aumentar memória pode, em alguns casos, reduzir um pouco o tempo de inicialização, mas não garante a eliminação de cold starts nem a estabilização da latência.

✗ C) Incorreta. Ajustar throughput provisionado do DynamoDB pode ajudar em gargalos de leitura/escrita, mas não afeta o 'Init Duration' da função Lambda, que é o tempo de inicialização do runtime, não de consulta ao banco.

✗ D) Incorreta. Migrar para Fargate é uma mudança arquitetural grande e desnecessária para este cenário. Lambda é totalmente adequado; o problema é resolvido com provisioned concurrency, sem precisar migrar para contêiner gerenciado.

Dica de prova: Sempre que o CloudWatch indicar aumento em 'Init Duration' com 'Duration' estável, pense em cold starts. Em ambientes de produção sensíveis à latência, use provisioned concurrency em conjunto com aliases estáveis e, se necessário, ajuste memória apenas após analisar métricas detalhadas de performance.

DIA DA PROVA

Estratégia de prova

O DVA-C02 pensa como um desenvolvedor que quer a solução gerenciada, segura e com menor esforço operacional: a alternativa certa quase sempre usa a integração nativa entre serviços (CodePipeline com CodeDeploy, Cognito com API Gateway, SQS com Lambda) em vez de scripts, servidores dedicados ou passos manuais. Os enunciados são cenários longos com requisitos numerados ou qualificadores decisivos, como zero downtime, rollback automático, rotação automática, sem chaves estáticas e sem alterar o código. As alternativas erradas são plausíveis e muitas vezes diferem da certa em um único detalhe: rolling em vez de blue/green, AllAtOnce em vez de Canary, chave de acesso guardada em lugar seguro em vez de role, aumentar timeout em vez de diagnosticar. Em troubleshooting, a prova premia quem observa antes de escalar.

Como responder

✓ Conheça o formato: 65 perguntas, das quais 50 são pontuadas e 15 não pontuadas e não identificadas; trate todas como se valessem.

✓ Há múltipla escolha (uma correta e três distratores) e múltipla resposta (duas ou mais corretas entre cinco ou mais opções); confira no enunciado quantas marcar.

✓ Não deixe nada em branco: pergunta não respondida conta como errada e não há penalidade por chutar.

✓ Sublinhe mentalmente os qualificadores (menor esforço operacional, sem downtime, rollback automático, rotação, sem chaves estáticas, sem mudar o código) e use-os para eliminar opções.

✓ Elimine de cara alternativas com console manual, scripts em laptop, cron em EC2, SSH, chaves de usuário IAM ou segredos em variáveis de ambiente.

✓ Quando duas opções parecem iguais, compare o final de cada uma: a diferença costuma estar na estratégia (blue/green, canary) ou no serviço de segredos.

✓ Em perguntas de diagnóstico, prefira a opção que identifica a causa (X-Ray, métricas, logs) antes das que escalam recursos ou aumentam timeouts.

✓ A nota é em escala de 100 a 1.000, com mínimo de 720, e o modelo é compensatório: domínio fraco pode ser compensado pelos outros, então priorize Desenvolvimento (32%) e Segurança (26%).

Erros que derrubam candidatos

✗ Escolher rolling update ou scripts quando o enunciado pede blue/green com rollback automático.

✗ Aceitar access key de usuário IAM só porque está criptografada ou guardada no Secrets Manager.

✗ Confundir Parameter Store com Secrets Manager quando há rotação automática no requisito.

✗ Achar que aumentar timeout ou memória resolve cold start, throttling ou gargalo de I/O.

✗ Trocar SQS, SNS, EventBridge e Step Functions entre si, ou usar FIFO para ganhar throughput.

✗ Tratar API keys ou CORS como mecanismo de autenticação.

✗ Esquecer que eventos assíncronos chegam pelo menos uma vez e que o consumidor deve ser idempotente.

Na semana da prova

- ☐ Refazer as provas completas cronometradas e revisar cada erro pelo motivo da alternativa errada, não só pela certa.
- ☐ Revisar as comparações de cada capítulo: SQS x SNS x EventBridge x Step Functions, Secrets Manager x Parameter Store, User Pool x Identity Pool, rolling x blue/green.
- ☐ Decorar os números que caem: 29 segundos do API Gateway, 15 minutos da Lambda, long polling até 20 segundos, até 10 mensagens por ReceiveMessage.
- ☐ Repassar as listas de pegadinhas e gatilhos de cada domínio na véspera, sem estudar conteúdo novo.
- ☐ Na prova, marcar para revisão as perguntas longas e voltar a elas no fim, sem deixar nenhuma em branco.

REVISÃO FINAL

Folha de cola

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

1. Desenvolvimento com os serviços da AWS

‣ upload direto para o S3 sem passar pelo backend → **URL pré-assinada (e multipart para arquivos grandes)**

‣ upload com URL pré-assinada falha com assinatura inválida → **Content-Type assinado diferente do enviado pelo cliente**

‣ mensagens processadas em duplicidade → **Idempotência no consumidor + visibility timeout maior que o processamento**

‣ mensagens venenosas que falham sempre → **DLQ com maxReceiveCount**

‣ muitas respostas vazias do ReceiveMessage → **Long polling até 20 segundos e até 10 mensagens por chamada**

⚠ Timeout só muda o tempo máximo de execução. Cold start se ataca com provisioned concurrency e inicialização enxuta fora do handler.

⚠ FIFO tem limites de throughput mais restritos; serve para ordem e deduplicação, não para vazão.

⚠ Mensagens reaparecem antes do fim do processamento e geram duplicidade. O visibility timeout deve superar o tempo de processamento.

2. Implantação

‣ pipeline gerenciado, integrações nativas, aprovação manual, adicionar testes no meio → **AWS CodePipeline + CodeBuild (build/testes) + ação de deploy (CodeDeploy, ECS, Lambda)**

‣ ECS/Fargate, zero downtime, rollback automático, health check → **CodeDeploy blue/green para ECS com ALB e dois target groups**

‣ implantar para pequena parcela e expandir gradualmente → **Canary (ou Linear) no CodeDeploy**

‣ Lambda com SAM, deploy gradual e rollback rápido → **Versão imutável + alias + AutoPublishAlias e DeploymentPreference**

‣ mesmo padrão em várias contas ou regiões → **CloudFormation/CDK versionado no Git + StackSets e roles entre contas**

⚠ AWS Batch é para computação em lote, não orquestra pipeline nem oferece aprovação manual. Pipeline de implantação é CodePipeline.

⚠ Funciona, mas exige lógica customizada e vai contra 'menor esforço operacional'. O CodePipeline aceita o ECR como origem nativamente.

⚠ Rolling update não reverte sozinho como o blue/green do CodeDeploy; quando aparecer 'rollback automático' + 'zero downtime' em ECS, exija CodeDeploy.

3. Segurança

‣ rotação automática de segredos, credenciais de RDS → **AWS Secrets Manager**

‣ parâmetros de configuração sensíveis, sem rotação, custo baixo → **Parameter Store SecureString com KMS**

‣ Lambda, ECS ou EC2 acessando S3/DynamoDB sem chaves → **Execution role, task role ou instance profile com menor privilégio**

‣ pods do EKS precisam acessar S3 → **IRSA**

‣ login social ou corporativo, OAuth 2.0, OIDC, JWT, API Gateway → **Cognito User Pool + Cognito authorizer (ou JWT authorizer em HTTP APIs)**

⚠ Continua sendo credencial estática de longa duração. Se o requisito é não usar chaves, a resposta é role (execution role, task role, instance profile, IRSA).

⚠ O segredo continua exposto como variável e sujeito a logs e dumps. Leia em tempo de execução do Secrets Manager ou do Parameter Store SecureString.

⚠ API key é controle de uso e cota, não autenticação. Use Cognito, JWT authorizer ou IAM.

4. Troubleshooting e Otimização

‣ latência alta com CPU e memória baixas → **X-Ray para achar dependência lenta (banco, API externa, conexões)**

‣ Init Duration aumentando, Duration estável → **Cold start: provisioned concurrency e init fora do handler**

‣ cliente criado dentro do handler a cada invocação → **Mover clientes e conexões para o escopo global**

‣ muitas conexões curtas ao RDS vindas de Lambda → **RDS Proxy**

‣ ProvisionedThroughputExceededException ou ThrottledRequests → **Auto scaling ou on-demand + retry exponencial com jitter**

⚠ Timeout maior só deixa a função falhar mais tarde e ocupar concorrência por mais tempo. Diagnostique com X-Ray e métricas e ataque a causa.

⚠ Mais CPU não resolve I/O. Use operações em lote, paralelismo e reuso de clientes.

⚠ SnapStart não se aplica a esses runtimes. Use provisioned concurrency e init fora do handler.

GABARITO

Respostas do diagnóstico

D1. A) Usar Provisioned concurrency na função crítica e ajuste de memória para reduzir o tempo de execução → revise o capítulo 1

Por quê: Esta alternativa é a mais adequada porque a provisão de concorrência (provisioned concurrency) mantém instâncias da função Lambda pré-inicializadas, reduzindo ou eliminando o impacto de cold starts e tornando a latência mais previsível. Ajustar o tamanho de memória também pode reduzir o tempo de execução (mais CPU alocada), melhorando ainda mais o desempenho sem deixar de usar o modelo serverless.

✗ B) Incorreta. Migrar para EC2 pode reduzir problemas de cold start, mas foge do objetivo de manter o modelo serverless. É uma mudança arquitetural maior, com mais gerenciamento de infraestrutura, e não é necessária quando a provisão de...

✗ C) Incorreta. Aumentar apenas o timeout não resolve o problema de latência variável; apenas permite que execuções lentas durem mais tempo.

✗ D) Incorreta. DLQ é útil para tratar falhas assíncronas, não para reduzir latência. Diminuir o timeout pode até aumentar erros por timeout, sem qualquer impacto positivo na variabilidade de resposta ou nos cold starts.

> **Dica de prova:** Em provas da AWS, quando o problema é latência imprevisível em Lambda (especialmente após períodos de inatividade), pense em "provisioned concurrency". Se a exigência é previsibilidade de tempo de resposta (picos conhecidos, low-latency, endpoints críticos), a provisão de concorrência é o mecanismo padrão.

D2. C) Usar eventos do S3 publicados num tópico SNS que distribui às filas SQS de cada componente com filtros de assinatura → revise o capítulo 1

Por quê: Esta alternativa é a mais adequada porque utiliza um padrão de fan-out desacoplado: o Amazon S3 publica eventos em um tópico Amazon SNS, e cada assinatura do SNS pode aplicar filtros (message filtering) para entregar somente determinados tipos de eventos a filas Amazon SQS distintas. Assim, cada componente da aplicação consome apenas os eventos relevantes, reduzindo a necessidade de lógica complexa de roteamento no código e aproveitando recursos nativos de filtragem e distribuição de mensagens da AWS.

✗ A) Incorreta. Centraliza todo o roteamento em uma única aplicação consumidora da fila SQS, aumentando a complexidade do código, o acoplamento e a responsabilidade desse componente, em vez de usar o fan-out nativo via SNS e filtragem por...

✗ B) Incorreta. O EventBridge não 'lê diretamente' objetos do S3 sem configuração; é necessário definir regras específicas para eventos suportados.

✗ D) Incorreta. Exige uma função Lambda única responsável por orquestrar o roteamento para múltiplas filas, o que também cria um ponto único de acoplamento e maior complexidade, além de adicionar uma camada de execução desnecessária quando o...

> **Dica de prova:** Quando precisar distribuir eventos para múltiplos consumidores com diferentes necessidades de filtragem, pense no padrão S3 -> SNS -> SQS/Lambda. O SNS faz o fan-out e o filtro por mensagem, enquanto SQS ou Lambda ficam responsáveis pelo processamento desacoplado em cada fluxo.

D3. B) Utilizar AWS SAM ou AWS CDK para definir a infraestrutura como código e criar stacks separados para desenvolvimento e produção, cada um com seu próprio event bus, regras, permissões e targets → revise o capítulo 1

Por quê: Esta alternativa é a mais adequada porque segue boas práticas de isolamento entre ambientes usando infraestrutura como código (IaC). Com AWS SAM ou AWS CDK, o time pode modelar event buses, regras, permissões e targets em templates versionados, e então implantar stacks distintos para desenvolvimento, homologação e produção.

✗ A) Incorreta. Move temporariamente os produtores para um novo event bus em produção, o que é arriscado e complexo de coordenar. Isso não fornece um ambiente de desenvolvimento isolado;

✗ C) Incorreta. Archive e replay no event bus de produção são úteis para depuração e reprocessamento de eventos, mas testar novas regras diretamente na produção viola boas práticas de separação de ambientes.

✗ D) Incorreta. Misturar eventos de teste e produção no mesmo event bus, diferenciando apenas por campos no payload, aumenta o risco de configuração incorreta e vazamento de eventos de teste para targets produtivos.

> **Dica de prova:** Sempre que possível, modele sua arquitetura serverless com infraestrutura como código e crie ambientes separados (dev, test, prod) em contas distintas ou, no mínimo, em stacks separados. Reutilize o código dos templates, não os recursos em si, entre ambientes. Isso reduz riscos e facilita automação com CI/CD.

D4. A) Configurar o AWS CodePipeline com estágios de Source (CodeCommit), Build (CodeBuild) e Deploy usando o AWS CodeDeploy com tipo de implantação Blue/Green para ECS, habilitando validações pós-implantação e rollback automático baseado em falhas em alarmes do Amazon CloudWatch. → revise o capítulo 2

Por quê: Esta alternativa é a mais adequada porque utiliza o AWS CodeDeploy com estratégia Blue/Green para serviços ECS, integrado ao CodePipeline. O Blue/Green permite manter a versão atual (blue) em produção enquanto a nova versão (green) é provisionada e testada.

✗ B) Incorreta. Utiliza Rolling Update no ECS, o que não oferece rollback automático nativo baseado em métricas ou testes. Em caso de falhas, o desenvolvedor precisa reverter manualmente, o que não atende ao requisito de rollback automático e...

✗ C) Incorreta. O uso de AWS CloudFormation com política All-at-once aumenta o risco de downtime e não provê rollback automático fino baseado em falhas de runtime ou alarmes;

✗ D) Incorreta. Depende de scripts manuais em uma instância EC2 e de agendamentos via cron, afastando-se da abordagem totalmente gerenciada e automatizada.

> **Dica de prova:** Para ECS e EKS, quando o requisito inclui rollback automático e controle refinado de tráfego entre versões, pense em integrações do AWS CodeDeploy com estratégias Blue/Green ou Canary.

D5. B) Configurar um pipeline no AWS CodePipeline integrado ao GitHub como fonte, usar AWS CodeBuild para build da imagem Docker e enviar para o Amazon ECR, seguido de uma ação de implantação no Amazon ECS → revise o capítulo 2

Por quê: Esta alternativa utiliza uma cadeia totalmente gerenciada e nativa para CI/CD na AWS: o AWS CodePipeline integra-se diretamente ao GitHub como fonte, o AWS CodeBuild realiza o build da imagem Docker sem necessidade de gerenciar servidores de build e a imagem é enviada para o Amazon ECR. Em seguida, o CodePipeline executa uma ação de implantação no Amazon ECS (Fargate), atualizando a task definition e o serviço.

✗ A) Incorreta. Exige gerenciar instâncias EC2, configurar scripts manuais e webhooks, aumentando a complexidade operacional e fugindo do requisito de usar apenas serviços totalmente gerenciados.

✗ C) Incorreta. Obriga a migração do código de GitHub para CodeCommit, o que não é um requisito do cenário. Além disso, o CodeDeploy não é necessário para implantar em ECS via pipeline, já que o CodePipeline integra-se diretamente ao ECS.

✗ D) Incorreta. AWS Batch não é o serviço indicado para pipelines de CI/CD padrão. Usar Batch e EventBridge para essa finalidade é mais complexo e não segue o caminho nativo recomendado para automação de builds e deploys em ECS.

> **Dica de prova:** Para cenários de CI/CD em que o código está no GitHub e o alvo é ECS ou Lambda, pense em: CodePipeline (orquestração) + CodeBuild (build/teste) + ECR (imagens) + ECS/Lambda (deploy). Isso tende a ser a combinação mais cobrada no exame para fluxos totalmente gerenciados.

D6. B) Criar um script de migração no diretório '.platform/hooks/postdeploy' e usar uma implantação 'Rolling with additional batch' no Elastic Beanstalk, combinada com versionamento de aplicações para permitir rollback. → revise o capítulo 2

Por quê: Esta alternativa é a mais adequada porque segue o padrão recomendado de hooks de plataforma do Elastic Beanstalk para operações de pós-implantação. O diretório '.platform/hooks/postdeploy' garante que as migrações sejam executadas após o novo código estar implantado no ambiente, mas antes de o tráfego ser totalmente direcionado para a nova versão em todos os instâncias.

✗ A) Incorreta. Colocar migrações no ENTRYPOINT do container faz com que cada instância execute migrações em cada inicialização, o que pode causar condições de corrida, bloqueios e falhas de concorrência no banco de dados.

✗ C) Incorreta. Embora 'Immutable deployment' seja uma boa prática para segurança e previsibilidade, a opção não descreve um mecanismo claro e confiável de orquestração das migrações com o ciclo de vida do Elastic Beanstalk.

✗ D) Incorreta. Usar um Lambda disparado por upload em S3 desacopla as migrações do ciclo de implantação real do Elastic Beanstalk.

> **Dica de prova:** Para Elastic Beanstalk, memorize o uso de '.platform/hooks' (predeploy, postdeploy, etc.) para automatizar tarefas de infraestrutura ligadas ao ciclo de implantação, como migrações de banco, limpeza de cache e ajustes de permissões.

D7. C) Criar uma função do IAM (IAM Role) para a tarefa do ECS com uma política que permita somente as ações necessárias no DynamoDB e associar essa função à definição da tarefa do Fargate. → revise o capítulo 3

Por quê: Esta alternativa é a correta porque utiliza uma função do IAM específica para a tarefa do ECS (task role), que é a forma recomendada de conceder credenciais temporárias para workloads em contêineres sem armazenar chaves de acesso. A política anexada à função pode ser limitada apenas às operações necessárias no DynamoDB, atendendo ao princípio do menor privilégio e evitando o uso de credenciais estáticas.

✗ A) Incorreta. Conceder permissões ao usuário raiz é uma grave violação de boas práticas de segurança e não há mecanismo para o ECS 'assumir' permissões do usuário raiz.

✗ B) Incorreta. Ainda depende de chaves de acesso de um usuário do IAM, mesmo que criptografadas no Parameter Store. Isso vai contra a recomendação de não usar credenciais estáticas em código ou configuração quando funções do IAM podem ser...

✗ D) Incorreta. Usa uma função do IAM, mas com permissões amplas e compartilhadas entre várias aplicações. Isso dificulta o controle de acesso granular e fere o princípio do menor privilégio, além de aumentar o impacto de um possível...

> **Dica de prova:** Sempre que um serviço da AWS (como ECS, Lambda ou EC2) precisar acessar outros serviços, pense primeiro em funções do IAM (roles) com políticas mínimas necessárias, em vez de criar usuários do IAM com chaves de acesso estáticas.

D8. A) AWS Secrets Manager → revise o capítulo 3

Por quê: O AWS Secrets Manager foi projetado especificamente para gerenciar segredos (senhas, chaves de API, tokens, etc.), oferecendo: - Criptografia em repouso usando chaves gerenciadas pela AWS (KMS). - Integração nativa com AWS Lambda para recuperação em tempo de execução. - Controle de acesso detalhado via políticas IAM. - Rotação automática de segredos (quando configurada), o que melhora a postura de segurança.

✗ B) Incorreta. KMS é um serviço de gerenciamento de chaves, não de armazenamento de segredos. Ele criptografa/decriptografa dados, mas não oferece por si só o armazenamento, versionamento e rotação de segredos usados pela aplicação.

✗ C) Incorreta. Suporta parâmetros seguros e criptografia com KMS, mas o cenário pede explicitamente um serviço focado em segredos, com recursos mais avançados de gerenciamento e rotação, o que é melhor atendido pelo Secrets Manager.

✗ D) Incorreta. Apesar de criptografar dados em repouso, não é o serviço recomendado para armazenamento de segredos de aplicação. Falta controle fino e funcionalidades específicas para segredos (rotação, integração direta com Lambda, etc.).

> **Dica de prova:** Quando a questão mencionar explicitamente 'segredos de aplicação', 'rotação de segredos', 'integração com Lambda/RDS' e controle de acesso via IAM, pense primeiro em AWS Secrets Manager. Se a ênfase estiver em parâmetros de configuração (não necessariamente segredos) e custo menor, considere SSM Parameter Store.

D9. B) Armazenar o código-fonte em um repositório privado do AWS CodeCommit, usar AWS Secrets Manager para armazenar e rotacionar segredos, e configurar um role do IAM para a função Lambda com permissões mínimas necessárias para acessar o DynamoDB por meio de credenciais temporárias. → revise o capítulo 3

Por quê: Esta alternativa segue as melhores práticas de segurança na AWS para CI/CD e acesso a dados sensíveis. O código-fonte é armazenado em um repositório privado do CodeCommit, evitando exposição em texto puro em locais inseguros. Os segredos são gerenciados pelo AWS Secrets Manager, que suporta rotação automática de credenciais (por exemplo, para bancos de dados ou outros segredos).

✗ A) Incorreta. Repositório GitHub público expõe o código a qualquer pessoa, o que é inaceitável para workloads sensíveis. Parâmetros SecureString sem KMS limitam a proteção criptográfica e o controle granular.

✗ C) Incorreta. Armazenar variáveis de ambiente criptografadas apenas no CodePipeline não resolve o problema de rotação automática de segredos.

✗ D) Incorreta. Armazenar o código em um bucket S3 público, mesmo com ZIP protegido por senha, é uma falha crítica de segurança.

> **Dica de prova:** Sempre que possível, use roles do IAM associadas a serviços (Lambda, ECS, EC2, etc.) em vez de chaves de acesso estáticas. Para segredos, prefira AWS Secrets Manager (com rotação automática) ou Parameter Store com criptografia KMS.

D10. C) Ativar o AWS X-Ray para a função Lambda e revisar o código em busca de estruturas de dados que crescem indefinidamente entre invocações, garantindo que variáveis globais sejam limpas ou recriadas a cada execução. → revise o capítulo 4

Por quê: Esta alternativa é a mais adequada porque o crescimento gradual do tempo de execução e do consumo de memória ao longo da vida útil do container indica fortemente um possível memory leak lógico ou acúmulo de estado em variáveis globais/estáticas entre invocações. Ao ativar o AWS X-Ray, o desenvolvedor consegue rastrear a execução, identificar trechos de código que acumulam dados ou mantêm conexões abertas e, em seguida, ajustar o código para limpar ou recriar estruturas de dados a cada execução.

✗ A) Incorreta. Reduzir o batch size pode diminuir a carga por invocação, mas não explica o crescimento progressivo ao longo da vida do container. Se o problema é acúmulo em variáveis globais, ele continuará ocorrendo, apenas em ritmo diferente.

✗ B) Incorreta. Colocar a função em VPC privada e usar ENI dedicado não tem relação com o crescimento gradual de uso de memória ao longo das invocações. Isso poderia até aumentar a latência de cold starts, sem resolver o problema observado.

✗ D) Incorreta. Aumentar memória e timeout apenas mascara o problema, não o identifica nem o corrige. Se há vazamento lógico de memória ou acúmulo de estado, mais recursos apenas atrasam o sintoma, sem tratar a causa raiz.

> **Dica de prova:** Em funções Lambda, qualquer dado armazenado em variáveis globais é potencialmente reutilizado entre invocações no mesmo ambiente. Use isso para otimizar (por exemplo, reaproveitando conexões), mas tome cuidado para não acumular listas, dicionários ou caches sem limpeza.

D11. A) Habilitar o AWS X-Ray na aplicação para identificar gargalos de código, otimizar trechos mais lentos e reduzir o tempo de processamento por requisição antes de alterar a infraestrutura.

→ revise o capítulo 4

Por quê: Esta alternativa é a melhor abordagem porque foca na otimização da aplicação antes de escalar ainda mais a infraestrutura. Ao habilitar o AWS X-Ray, o desenvolvedor consegue identificar chamadas de downstream lentas, trechos de código ineficientes, consultas a banco de dados mal otimizadas ou dependências externas que causam aumento no tempo de resposta.

✗ B) Incorreta. O algoritmo de roteamento do ALB (por padrão, round-robin com certas otimizações) raramente é a causa de picos de CPU consistente em todas as tasks.

✗ C) Incorreta. Não é possível colocar Amazon SQS diretamente entre o ALB e o ECS em um fluxo HTTP síncrono tradicional. Além disso, o problema é de alta utilização de CPU e tempo de processamento;

✗ D) Incorreta. Aumentar vCPU e memória e desabilitar Auto Scaling apenas mascara o problema e aumenta os custos. O uso de CPU já está alto e manter tasks mais potentes fixas não resolve gargalos de código ou de dependências internas.

> **Dica de prova:** Sempre que identificar alta utilização de recursos (CPU, memória, I/O) em conjunto com aumento de latência, investigue primeiro a causa na camada de aplicação (profiling, tracing, otimização de queries, caching) antes de simplesmente aumentar recursos.

D12. D) Ativar o AWS X-Ray para a aplicação, instrumentar o código para rastreamento de ponta a ponta e analisar os traces detalhados das requisições passando pelo ALB até o backend → revise o capítulo 4

Por quê: Esta alternativa é a melhor abordagem porque o AWS X-Ray fornece rastreamento distribuído de ponta a ponta, permitindo visualizar o caminho completo de cada requisição, tempos de resposta por segmento (ALB, rede, aplicação, dependências externas) e erros em cada etapa.

✗ A) Incorreta. Aumentar instâncias e endurecer health checks pode mascarar o problema, não identificá-lo. Isso pode até reduzir temporariamente alguns erros, mas não revela a causa raiz (como falhas intermitentes em dependências externas ou...

✗ B) Incorreta. Métricas personalizadas de CPU/memória/disco ajudam em capacidade e monitoração de recursos, mas o cenário já indica que não há gargalo nesses recursos.

✗ C) Incorreta. VPC Flow Logs ajudam a inspecionar tráfego de rede em nível de fluxo (aceito/rejeitado), mas não fornecem visibilidade em nível de aplicação ou correlação detalhada das requisições com erros 5xx.

> **Dica de prova:** Para problemas intermitentes de erros 5xx, combine logs de ALB, CloudWatch Logs e rastreamento distribuído (AWS X-Ray). X-Ray é especialmente útil para mapear dependências, identificar operações lentas e timeouts e visualizar o impacto de serviços externos (como bancos de dados, APIs de terceiros ou filas) sem...

Agora, treine no formato real

Faça as 6 provas completas do curso DVA-C02: 390 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 Amazon Web Services (AWS). As questões são inéditas e não reproduzem a prova real.