IT資格道場

DVA-C03 学習用テキスト

予備知識ゼロから合格ラインへ —— 読んで理解する1冊目

無料サンプル(第1章まで)

AWS Certified Developer - Associate (DVA-C03) 試験ガイド(docs.aws.amazon.com・2026-11-12取得)準拠
2026年11月版

SAMPLE

ご利用にあたって

- このテキストは「IT資格道場」(www.shikaku-doujou.com)の購入者限定テキストのうち、学習用テキストの第1章までを収録した**無料サンプル**です。どなたでもご覧いただけます。
- 無料サンプルも著作物です。転載・改変しての再配布・二次販売はできません。
- AI モデルの学習データとしてのご利用(学習目的の入力・データセット化を含む)はできません。
- 製品版(学習用テキスト・頻出度別ノート・暗記用ノートの3点)は、ご購入者を識別できる透かし入りのPDFとして提供されます。
- 本PDFはスマートフォン・タブレットでの閲覧に合わせたA5版面・文字サイズで組版しています。

SAMPLE

目次

この教材の使い方

試験の基本情報

試験の設計図：4 分野と配点

第1章 AWS のサービスによる開発
[1.1.1/1.1.2/1.1.3/1.1.4/1.1.5/1.1.6/1.1.7/1.1.8/1.1.9/1.1.10/1.2.1/1.2.2/1.2.3/1.2.4/1.3.1/1.3.2/1.3.3/1.3.4]

疎結合の見分けと、キュー・トピック・イベントバスによる分離 [1.1.1]

同期呼び出しと非同期呼び出しの選び分け [1.1.2]

SDK の再試行と指数バックオフ、スロットリング、グレースフルデグラデーション [1.1.3]

API Gateway の API の種類と作り込み [1.1.4]

SDK と CLI の認証情報、ページネーション、バッチ API、エラー処理 [1.1.5]

Kinesis のストリームと Amazon Data Firehose [1.1.6]

JSON のシリアル化と DynamoDB の型付き属性値、Base64 とバイナリ [1.1.7]

イベント駆動の実装: EventBridge、S3、SNS、SQS [1.1.8]

AI 支援開発ツールによるコードの生成・レビュー・最適化 [1.1.9]

Amazon Bedrock などのマネージド AI サービスをアプリに組み込む [1.1.10]

Lambda 関数から VPC 内のプライベートリソースへ届く [1.2.1]

Lambda 関数の設定: メモリ、タイムアウト、環境変数、レイヤー、同時実行 [1.2.2]

非同期呼び出しとイベントソースマッピングのエラー処理 [1.2.3]

関数を API Gateway、DynamoDB、SQS、Kinesis とつなぐ [1.2.4]

データのライフサイクル: S3 ライフサイクル、DynamoDB の TTL、保持期間 [1.3.1]

アクセスパターンに合わせたデータストアの選び方 [1.3.2]

キャッシュの戦略と、ElastiCache・DAX の選び分け [1.3.3]

DynamoDB の設計と操作: キー、整合性、Query と Scan、インデックス、トランザクション [1.3.4]

サンプルはここまでです

SAMPLE

3 冊の使い方は、ダウンロード欄の案内を参照してください。

SAMPLE

この教材の使い方

このテキストの目的は、**AWS Certified Developer - Associate（DVA-C03）**（実施は Amazon Web Services（AWS））の合格です。

Developer - Associate が問うのは、サービスの名前ではなく、**コードと設定でサービスをどう動かすか**です。SDK の呼び方、Lambda や API Gateway の設定値、認証情報と鍵の扱い、パイプラインとデプロイ、ログとトレースによる原因の切り分けが、場面つきで出題の中心になるでしょう。公式の試験ガイドは、次の 4 つの分野から成ります。

- AWS のサービスによる開発
- セキュリティ
- テストとデプロイ
- トラブルシューティングと最適化

第 1 章〜第 4 章は、その順番どおりに並べ、各節の見出しの角かっこに試験ガイドのスキル番号を付けています。

試験の基本情報

項目	内容
試験名	AWS Certified Developer - Associate（**DVA-C03**）
運営	Amazon Web Services
問題数	**65 問**（採点対象 50 問＋採点されない 15 問）
出題形式	**択一（4 択から 1 つ）と複数選択（5 つ以上から 2 つ以上）**
試験時間	**130 分**
合格ライン	**100〜1,000 のスケールスコアで 720 点**（分野ごとの足切りはなく、全体で 720 点に届けば合格です）
受験料	20,000 円（150 USD）

試験の設計図：4 分野と配点

分野	配点	対応する章
① AWS のサービスによる開発	**30%**	第 1 章
② セキュリティ	**26%**	第 2 章
③ テストとデプロイ	**22%**	第 3 章
④ トラブルシューティングと最適化	**22%**	第 4 章

第 1 章がいちばん厚く、全体の 3 割です。ただし残りの 3 分野を合わせると 7 割になるので、どの分野も外せないでしょう。

SAMPLE

第1章 AWS のサービスによる開発 [1.1.1/1.1.2/1.1.3/1.1.4/1.1.5/1.1.6/1.1.7/1.1.8/1.1.9/1.1.10/1.2.1/1.2.2/1

疎結合の見分けと、キュー・トピック・イベントバスによる分離 [1.1.1]

密結合は、呼び出す側が相手の場所と速度と稼働を前提にしている形です。注文の受付が在庫の処理を直接呼び、応答が返るまで待つ構成がその例で、在庫側が止まれば、受付も止まるしかありません。疎結合は二者の間に仲介を置き、送る側と受ける側が互いの状態を知らずに動ける形です。仲介の部品は3つあります。SQS のキュー、SNS のトピック、EventBridge のイベントバスです。

SQS は受け手が取りに行く pull 型のキューです。標準キューは少なくとも1回の配信（at-least-once）で、同じメッセージが複数回届くことがあり、順序は努力目標（best-effort）にとどまります[^c1-1]。受け取ったメッセージは**可視性タイムアウト**の間だけ他の受け手から隠れ、既定は30秒、最大は受信から12時間です[^c1-2]。保持期間の既定は4日（60秒〜14日）、メッセージの最大サイズは1,048,576バイトです[^c1-3]。空振りの受信を減らすには、ReceiveMessage に WaitTimeSeconds を付けてロングポーリングにします。待ち時間は最大20秒です。[^c1-4]

SNS は発行者が1通を送ると、複数の購読者へ即時に押し出す push 型のトピックです。トピックの先に SQS のキューを購読させる形を**ファンアウト**と呼びます。SNS だけでは受け手が落ちていた間のメッセージは残りませんが、キューを挟むとメッセージが残り、受け手は後から処理できます。[^c1-5]

EventBridge はイベントで部品をつなぐサーバーレスのサービスです。イベントバスがイベントを受け取り、ルールに合うものを0個以上のターゲットへ届けます。バスには Custom Event Bus と従来型の Custom Event Bus - Classic があり、新しいアプリは Custom Event Bus から始めます[^c1-6]。Custom Event Bus は、期間を決めてイベントを保持し、順序どおりの配信、リプレイ、JSON 以外のペイロードに対応します。[^c1-6]Classic のルールは、1つにつきターゲットを5つまで持てます。[^c1-7]

部品	届け方	受け手の数	向く場面
SQS	受け手が取りに行く	1つのキューを複数の受け手が分け合う	負荷をためて順に処理する
SNS	発行者が押し出す	購読者すべてに同じ内容	1通を複数の系統へ配る
EventBridge	ルールで振り分けて押し出す	ルールごとに0個以上のターゲット	内容に応じて宛先を変える、SaaS や他アカウントのイベントを受ける

順序と重複の制御が要るときは **FIFO キュー**を使います。同じ重複排除 ID のメッセージは、5分の重複排除間隔の中なら新たに配信されません。コンテンツベースの重複排除を有効にすると、本文の SHA-256 ハッシュから重複排除 ID が作られます。[^c1-8]メッセージ属性はハッシュに含まれません。[^c1-8]

キューの価値は、入る速さと出る速さを切り離すことにあります。夜間のバッチで注文が普段の10倍入っても、キューがその差を吸収し、消費者は自分の速度で処理を続けます。消費者の数は生産者と無関係に増減でき、下流の DB が耐えられる並列度に絞ることもできます（消費者を3本に固定するなど）。

処理の流れは「受信、処理、削除」の3段です。削除を呼ぶ前に可視性タイムアウトが切れると、メッセージは再び見えて、別の受け手が同じものを処理します。処理時間より短い値にしてはいけません。重複はこの仕組みの上で必ず起

SAMPLE

こり得るので、受け手は**冪等**に作ります。冪等とは、同じ入力を2回処理しても結果が変わらない性質のことです（二度目は何も起こりません）。

たとえば、注文メッセージを受けて在庫を1つ減らす処理を、そのまま2回実行すると在庫が2つ減ります。注文 ID を DynamoDB に条件付きで書き込み、書けたときだけ在庫を減らす形にすると、再配信されても減るのは1回です。標準キューの再配信そのものは止められません。何度処理しても同じ結果になるこの形が、重複への備えです。繰り返し処理に失敗するメッセージは、受信回数の上限（maxReceiveCount）を決めてデッドレターキューへ移します。標準キューでは、デッドレターキューへ移ってもエンキュー時刻は元のままで、期限は元の送信時刻から数えます（デッドレターキューの保持期間は元のキューより長くする）。FIFO キューでは、移った時点でエンキュー時刻が改まります。[^c1-17]

同期呼び出しと非同期呼び出しの選び分け [1.1.2]

同期は、呼び出し側が結果を待つ形です。非同期は、受け付けの確認だけを受け取り、処理は後ろで進むので、結果は待ちません。結果がすぐ要る画面操作は同期、時間のかかる処理や取りこぼしを避けたい連携は非同期が向きます。

Lambda の Invoke API では、InvocationType の値で決まります。RequestResponse なら同期で、関数の応答が返るまで待ちます。Event なら非同期で、Lambda がイベントを内部のキューに入れ、すぐに成功を返します（HTTP 202）。実際に関数へ渡すのは別のプロセスです[^c1-9]。ペイロードの上限は、同期が要求・応答とも6 MB、ストリーミング応答が200 MB、非同期が1 MBです。[^c1-10]

API Gateway の REST API は統合タイムアウトが50ミリ秒〜29秒で、Lambda、HTTP、AWS の統合すべてに当てはまります。ペイロードは10 MBまでです。Regional と private の API は29秒を超える引き上げを申請でき、その代わりにリージョンのスロットル上限が下がることがあります。エッジ最適化の API は引き上げられません[^c1-11]。29秒を超えうる処理を同期で待たせるのは設計の誤りで、受け付けだけ返して非同期に回します。

SQS による非同期処理は、受け手が自分のペースで取り出すので、送る側は待たずに済み、受け手の停止も吸収できます。**EventBridge** は、イベントの内容を見て宛先を変えるときに向きます。1つの出来事を複数の部品が別々の処理で受ける、イベント駆動のワークフローに向きます（受け手は互いを知りません）。点と点の統合なら **EventBridge Pipes** が合い、変換とエンリッチメントを備えます。多数のソースから多数のターゲットへ振り分けるなら、イベントバスです。[^c1-6]

Step Functions は、ワークフローの種類が実行の性質を決めます。Standard は最長1年まで続き、各ステップが1回だけ実行される前提（exactly-once）で、履歴を残します。Express は最長5分で、非同期なら少なくとも1回の実行（at-least-once）なので冪等な処理に向き、同期なら最大1回の実行（at-most-once）です。Express の用途は、大量のイベント処理です（IoT データの取り込み、ストリーミングデータの処理と変換、モバイルアプリのバックエンドなど）。[^c1-13]**StartSyncExecution** は Express だけで使える同期の実行で、終わるまで待って結果を返します。Standard には使えません[^c1-13]。応答は200 OK で返り、実行が失敗しても HTTP のコードには現れません。[^c1-115]

場面で見ます。Web 画面から押された注文を確定し、その結果を画面に返す必要があり、処理は3秒で終わります。この場合は API Gateway と Lambda の同期呼び出しが合います。これに対し、確定後の請求書 PDF の生成と配信メールは待たせる必要がありません。注文確定の後に SQS か EventBridge へイベントを出し、別の部品が非同期に受けます。

SAMPLE

SDK の再試行と指数バックオフ、スロットリング、グレースフルデグラデーション [1.1.3]

一時的なエラーやスロットリングは、時間を置いてやり直すと通ることがあります。やり直しの間隔を倍々に延ばす方式を**指数バックオフ**と呼びます（1秒、2秒、4秒の順）。全員が同じ間隔で再試行すると、また同時に殺到するので、間隔にランダムな揺らぎ（**ジッター**）を加えます。

AWS SDK の既定は standard モードで、指数バックオフ＋ジッターと再試行の枠（retry quota）を使います。以下の式と回数は、環境変数 AWS_NEW_RETRIES_2026=true で有効にする新しい挙動の値です（設定しない SDK は従来の挙動で、バックオフの時間、retry quota の消費、サービスごとの既定値が異なります）。待ち時間の式は、random(0,1) × min(20,000 ms, base_delay × 2^retry) です。base_delay は一時エラーが50 ms（DynamoDB と DynamoDB Streams のクライアントは25 ms）、スロットリングが1,000 ms で、1回の待ちは最大20秒になります[^c1-14]。一時エラーには RequestTimeout のようなエラー、スロットリングには ThrottlingException のようなエラーが入ります。standard モードは、一時エラーに短い待ち、スロットリングに長い待ちを使います。[^c1-14]最大試行回数は初回を含めて3回（初回1回＋再試行2回）で、DynamoDB と DynamoDB Streams のクライアントだけは4回です。環境変数 AWS_MAX_ATTEMPTS か設定ファイルの max_attempts で変え、1にすると再試行しません。再試行のモードは、環境変数 AWS_RETRY_MODE か設定ファイルの retry_mode で選びます。値の優先は、コードの設定、環境変数、共有の設定ファイル、SDK の既定の順です。[^c1-14]boto3 の Config オブジェクトでは max_attempts が初回を含まない再試行の回数になるので、初回を含む回数は total_max_attempts で指定します。retry quota はトークンバケットで、再試行のたびにトークンを消費し、成功すると補充されます。枯渇すると再試行せずにエラーを返し、最初のリクエストも遅らせません。adaptive モードは、これにクライアント側のレート制限を加え、スロットリングを検知すると最初のリクエストまで遅らせます。

```
import boto3
from botocore.config import Config

cfg = Config(retries={"mode": "standard", "total_max_attempts": 5})
ddb = boto3.client("dynamodb", config=cfg)
```

スロットリングは、サービスが「多すぎる」と返すことで、HTTP では 429 や ThrottlingException の形で現れます。API Gateway の既定の上限はアカウントとリージョンで1秒あたり10,000リクエストで、バーストの最大は5,000リクエストです。超えると 429 Too Many Requests を返します[^c1-15]。再試行が増えるほど重複した実行も増えるので、書き込みには**冪等性**が要ります。EC2 の RunInstances のように、呼び出しに ClientToken を付けると、同じトークンの再送は新しい実行にならず、最初の結果が返ります[^c1-16]。SQS のデッドレターキューは、maxReceiveCount の回数を受信しても処理されないメッセージを移す先です。標準キューでは期限の基準が元の送信時刻なので、デッドレターキューの保持期間は元のキューより長くします。[^c1-17]

EventBridge のターゲットへの配信は、既定で最大24時間、最大185回まで再試行します。使い切るとイベントは捨てられますが、ターゲットにデッドレターキュー（SQS）を設定すると失敗したイベントを残せます。[^c1-18]

グレースフルデグラデーションは、依存先が壊れたときに、全体を止めずに機能を縮めて動き続ける考え方です。実装の型は3つあります。呼び出しにタイムアウトを必ず付けること。失敗したら既定値やキャッシュした古い値を返すフォールバックを用意すること。失敗が続く相手への呼び出しを一時的に止めるサーキットブレーカーを置くことです。推薦の部品が応答しないとき、推薦欄だけ空にして商品ページは表示し続ける動きが典型です。ブレーカーが働いて

SAMPLE

いる間は、その依存先を呼びません。一部の処理だけが失敗したときは、成功と失敗の内訳を呼び出し元に返すか、失敗した要求をデッドレターキューに入れて後から再試行し、失敗の内容は記録に残します。[^c1-118]再試行を重ねても、止まった相手は戻りません。ブレーカーで一度休ませるほうが先です。

場面で考えます。書き込み API を呼ぶ関数が、スロットリングのたびに再試行し、二重に課金される事故が出ました。再試行そのものは正しい動きで、SDK は初回を含めて3回までと決まっています。直すのは書き込みの側で、SDK の設定ではありません。ClientToken のような冪等性トークンを付けるか、条件付き書き込みで二重の登録を拒みます。

API Gateway の API の種類と作り込み [1.1.4]

API Gateway の API は3種類あります。**REST API** は機能がそろった完全版で、**HTTP API** は機能を絞って低価格にした版、**WebSocket API** は双方向の持続接続を扱う版です。API キー、クライアント別のスロットリング、リクエスト検証、AWS WAF 連携、private エンドポイントが必要なら REST API を選びます。そうした機能が要らなければ HTTP API が安く済みます[^c1-19]。WebSocket API は $connect、$disconnect、$default の3つの定義済みルートに加え、独自のルートを作れます。[^c1-20]

統合には種類があり、API 定義の type 値で表します。Lambda プロキシ統合は AWS_PROXY、それ以外の Lambda カスタム統合と AWS サービス統合は AWS です。HTTP プロキシ統合は HTTP_PROXY、HTTP カスタム統合は HTTP になります。API Gateway 自身が応答を返す mock 統合（MOCK）もあります[^c1-21]。プロキシ統合は要求をそのまま渡すので設定が少なく、バックエンドの変更に追従しやすい反面、変換はバックエンドの責任になります。カスタム統合では、要求と応答の変換を API Gateway が担います（バックエンドは変換を知りません）。

変換には**マッピングテンプレート**を使います。Velocity Template Language（VTL）で書き、Content-Type ごとに定義します。その Content-Type に対応するテンプレートがなければ、本文はそのまま統合先に渡ります。[^c1-22]

```
#set($body = $input.path('$'))
{
  "orderId": "$input.params('id')",
  "quantity": $body.quantity
}
```

テンプレートの中では、$input で要求の本文やパラメータを読み、$context で要求の文脈を、$util で補助関数を使います。1つのテンプレートの上限は300 KB です。[^c1-11]

リクエストの検証は、REST API の機能です。必須のパラメータ（URI、クエリ文字列、ヘッダー）の有無か、モデル（JSON スキーマ）で本文を検証し、不正なら 400 を呼び出し元へ返してバックエンドを呼びません。検証の結果は CloudWatch Logs に出ます[^c1-24]。**使用量プラン**は、API キーを持つ顧客ごとにスロットリングの上限と一定期間のクォータを定める仕組みで、API やメソッドの単位で設定します。API キーは認証の手段ではなく、利用量を区別する印です[^c1-25]。認可には Cognito や Lambda オーソライザー、IAM を別に使います。

キャッシュはステージ単位です。TTL の既定は300秒、最大3,600秒で、TTL を0にすると無効になります。キャッシュできる応答の最大サイズは1,048,576バイトです[^c1-26]。クライアントが Cache-Control: max-age=0 を送るとキャッシュを素通りして統合先から取り直しますが、そのクライアントに許可が要ります。

SAMPLE

CORS は、ブラウザが別オリジンの API を呼ぶときの許可です。REST API で非シンプルな要求（プリフライト）に応えるには、OPTIONS メソッドを mock 統合で作り、200 の応答に Access-Control-Allow-Headers、Access-Control-Allow-Methods、Access-Control-Allow-Origin を付けます[^c1-27]。Lambda プロキシ統合では、実際の GET や POST の応答にも Access-Control-Allow-Origin を関数が返さないと、ブラウザが拒否します。HTTP API は CORS の設定を持ち、OPTIONS のルートが無くてもプリフライトに自動で応えます。[^c1-28]

ゲートウェイレスポンスは、API Gateway 自身が返すエラーの形を変える設定です。型ごとに既定の状態コードがあり、たとえば ACCESS_DENIED は403、THROTTLED は429です。型を指定しなければ DEFAULT_4XX が使われます[^c1-29]。CORS のヘッダーをここに足すと、認可エラーや 429 でもブラウザが本文を読めます。429 のうち、使用量プランのクォータの超過は QUOTA_EXCEEDED、スロットリングの上限の超過は THROTTLED の型です。Cognito や Lambda のオーソライザーが拒否したときの 403 は ACCESS_DENIED の型です。[^c1-29]

場面で考えます。取引先ごとに1秒あたりの呼び出し回数を決め、超えた分を制限したい公開 API が、HTTP API で作られていました。HTTP API には API キーとクライアント別のスロットリングがないので、REST API に移し、取引先ごとの API キーを使用量プランに結び付けます。ブラウザからも呼ばれるなら、OPTIONS を足し、ゲートウェイレスポンスにも Access-Control-Allow-Origin を付けておきます。付けておかないと、429 を受けたブラウザは本文を読めません。リクエストの検証を効かせれば、不正な本文はバックエンドに届く前に 400 で止まります。

SDK と CLI の認証情報、ページネーション、バッチ API、エラー処理 [1.1.5]

SDK は決まった順で複数の場所に認証情報を探し、有効なものが見つかった時点で止めます。順序は SDK ごとに少し違い、期限切れのものは自動で更新されます[^c1-30]。IAM Identity Center の一時認証情報も SDK が更新しますが、サインインのセッション自体が切れたら、aws sso login でサインインをやり直します。[^c2-21]Python の Boto3 では次の順です。[^c1-31]

1. client() の引数で渡した認証情報
2. Session の引数で渡した認証情報
3. 環境変数（AWS_ACCESS_KEY_ID、AWS_SECRET_ACCESS_KEY、AWS_SESSION_TOKEN）
4. assume role プロバイダー
5. web identity 付きの assume role プロバイダー
6. IAM Identity Center
7. 共有の認証情報ファイル（~/.aws/credentials）
8. コンソールのログイン認証情報
9. 設定ファイル（~/.aws/config）
10. Boto2 の設定ファイル
11. コンテナの認証情報プロバイダー
12. EC2 のインスタンスメタデータ（IAM ロール）

覚え方は「コードで直接渡したものが最優先、環境変数が次、ロールの引き受けと Identity Center とファイル類を挟み、コンテナと EC2 の実行環境の認証情報が最後」です。EC2 や ECS、Lambda 上のコードに、長期のアクセスキーを埋めてはいけません。ロールの一時認証情報を使わせます。同じマシンに環境変数とプロファイルの両方があれば、環境変数が勝ちます。意図したプロファイルが使われないときは、まず環境変数を疑います。

SAMPLE

署名付き URL の有効期限は、IAM ユーザーの認証情報で SigV4 署名したもので最長7日です。一時認証情報で署名した URL は、元の認証情報の期限より長くできません。EC2 のロールの認証情報は通常6時間です。[^c1-32]

ページネーションは、結果が1回で返りきらないときに続きを取る仕組みです。応答に含まれる NextToken（サービスによっては Marker、DynamoDB の Query と Scan は LastEvaluatedKey）を、次の要求に渡して続けます。SDK のページネーターは、この繰り返しを肩代わりします。**ウェイター**は、リソースが目的の状態になるまで一定間隔でポーリングする補助で、テーブルの作成完了などを待つときに使います。

バッチ API は、1回の呼び出しで複数の項目を扱い、往復の回数を減らします（容量は項目ごとに計算されます）。DynamoDB の BatchGetItem は1回に最大100件、BatchWriteItem は最大25件を扱います。BatchWriteItem は PutItem と DeleteItem だけで、更新はできません[^c1-33]。一部だけ処理できなかった項目は応答の UnprocessedItems（読み込みは UnprocessedKeys）に入って返るので、自分で再送します。

SDK のエラーは、サービスが返す例外（エラーコード付き）と、通信の問題に分かれます。Boto3 では ClientError の応答に、Error（エラーコードの Code とエラーメッセージの Message）と、リクエスト ID（ResponseMetadata の RequestId）が入っています。問い合わせるときは、このリクエスト ID が手がかりになります（調査の依頼にも使います）。再試行できる例外（スロットリングや一時的なサーバーエラー）は SDK が自動で再試行し、入力の誤りや権限の不足は再試行しても直らないので、コードで直します。DynamoDB の ConditionalCheckFailedException は条件が偽だったことを示し、再送しても結果は変わりません。[^c1-34]

```
from botocore.exceptions import ClientError


try:
    table.put_item(Item=item, ConditionExpression="attribute_not_exists(pk)")
except ClientError as e:
    if e.response["Error"]["Code"] == "ConditionalCheckFailedException":
        pass  # すでにあるので何もしない
    else:
        raise
```

場面で考えます。オンプレミスのバッチが、プロファイルの assume role で動くはずなのに、別の権限で動いています。調べると、環境変数 AWS_ACCESS_KEY_ID が残っていました。Boto3 は環境変数を assume role プロバイダーより先に読むので、プロファイルの設定は使われません。環境変数を外すしかありません。大きな表の取り出しが途中で終わるときは、応答の LastEvaluatedKey が残っているのに、続きを呼んでいない場合が大半です（件数の上限に達したとは限りません）。

Kinesis のストリームと Amazon Data Firehose [1.1.6]

ストリーミングデータは、止まらずに流れ込む記録の列です。AWS の主な受け皿は2つで、Kinesis Data Streams は自分のコードで読み出し、Amazon Data Firehose は読み出さずに保存先へ配ります。

Kinesis Data Streams のレコードは、シーケンス番号、パーティションキー、データ本体で構成されます。パーティションキーは MD5 で128ビットの値になり、その値が属するハッシュ範囲のシャードへ入ります。同じキーのレコードは

SAMPLE

同じシャードに入り、順序もそのシャードの中でしか保たれません。キーに偏りがあると特定のシャードに負荷が集まるので、値の種類が多く、均等に散るキーを選びます。

1シャードの処理能力は、書き込みが1 MB/秒または1,000レコード/秒、読み取りが2 MB/秒または2,000レコード/秒までです[^c1-35]。シャード数は UpdateShardCount で変えます。レコードのデータ本体は、base64 の前で最大10 MiB です。PutRecords は1回に最大500レコードで、合計も10 MiB までです[^c1-36]。一部のレコードだけ書けないことがあり、応答の FailedRecordCount と、各レコードの ErrorCode（ProvisionedThroughputExceededException など）を見て、書けなかったものだけを再送します。

たとえば、書き込みが3 MB/秒で2,500レコード/秒なら、MB で見ると3シャード、レコード数で見ると3シャードが要ります。必要数は大きいほうで決まるので3です。読み取りは3シャードで合計6 MB/秒ですが、共有の読み取りでは消費者全員でこの枠を分け合います。3 MB/秒ずつ読む消費者が3本あると9 MB/秒が要り、足りません。

レコードの保持は既定で24時間で、最大8760時間（365日）まで延ばせます。IncreaseStreamRetentionPeriod で延ばし、DecreaseStreamRetentionPeriod で最小24時間まで縮められます[^c1-37]。1回の GetRecords は最大10,000レコード・10 MB までです。1シャードあたり1秒に5回の読み取りと、合計2 MB/秒までの上限があり、10 MB を返した直後の5秒間の呼び出しは例外になります。[^c1-35]

拡張ファンアウト（enhanced fan-out）は、消費者ごとにシャードあたり2 MB/秒の専用の帯域を与える仕組みです。共有では消費者が増えるほど1本あたりの帯域が減りますが、拡張ファンアウトは増えても減りません。レコードは SubscribeToShard を使って HTTP/2 で押し出されます。登録できる消費者は1ストリームあたり20本（On-demand Advantage モードは50本）です。[^c1-38]

容量モードは2つあります。オンデマンドは容量の計画が要らず自動でスケールし、プロビジョンドはシャード数を自分で指定します。新しいオンデマンドのストリームは、既定で書き込み4 MB/秒・読み取り8 MB/秒から始まります。モードは24時間に2回まで切り替えられます。[^c1-35]

消費者の側には3つの選択肢があります。**KCL**（Kinesis Client Library）は、シャードの割り当てや再分割後の追従を引き受けるライブラリです。Lambda のイベントソースマッピングは、シャードを読んでバッチで関数を呼びます（消費者のコードは関数だけです）。**ParallelizationFactor** を1〜10で指定すると、1シャードあたり同時に処理するバッチ数が増え、順序はパーティションキー単位で保たれます。読み取りが追いついていないことは、**IteratorAge**（最後に読んだレコードの古さ）の指標が伸びることで分かります。

Amazon Data Firehose は、ストリームを S3、Redshift、OpenSearch、Splunk、Iceberg、HTTP エンドポイントなどへ配るフルマネージドのサービスです。受け手のコードも、シャードの管理も要りません。受け取ったデータをバッファに貯め、設定したサイズか間隔のどちらかが先に満ちたときに配ります。そのため配信はリアルタイムではなく、ニアリアルタイムです。[^c1-40]

変換の Lambda を挟むこともできます。Firehose がバッファ済みのバッチを関数に同期で渡し、返ってきたデータを配信先へ送ります。変換用のバッファサイズは0.2〜3 MB（既定1 MB）、間隔は0〜900秒（既定60秒）です。同期呼び出しなので要求と応答はともに6 MB が上限です[^c1-41]。変換を有効にすると、変換前のデータを別の S3 バケットへ控えることができます。

要件	選ぶもの
レコードを自分のコードで読み、順序や再読み込みが要る	Kinesis Data Streams

要件	選ぶもの
受けたデータをそのまま S3 や OpenSearch に置きたい	Data Firehose
受信側が複数あり、それぞれに専用の帯域が要る	Data Streams の拡張ファンアウト

JSON のシリアル化と DynamoDB の型付き属性値、Base64 とバイナリ [1.1.7]

シリアル化は、メモリ上のデータを、送れる形（文字列やバイト列）に変えることです。逆シリアル化はその逆の変換を指します（受け取った文字列を、コードで扱える値に戻す）。AWS の API は JSON を使うものが多く、SDK がこの変換と署名を肩代わりします。

DynamoDB の低レベル API は、すべての値に**型記述子**を付けます。S（文字列）、N（数値）、B（バイナリ）、BOOL、NULL、M（マップ）、L（リスト）、そして SS、NS、BS（それぞれの集合）です[^c1-42]。数値は精度を保つために、JSON の中でも文字列で表します（`{"age": {"N": "25"}}`）。バイナリは JSON に型がないので、base64（RFC 4648）で符号化して送ります。JSON は転送の形式にすぎず、保存の形式ではありません。[^c1-42]

数値の精度は38桁までで、それ以上が要るときは文字列で持ちます。アイテムの最大サイズは400 KB です[^c1-43]。この変換を毎回手で書くのは大変なので、高水準のインターフェースが用意されています。Python の Boto3 のテーブルリソース、JavaScript の DocumentClient（v3 では lib-dynamodb の DynamoDBDocumentClient）は、言語の型と型記述子を自動で変換します。リソースでは、DynamoDB の Number が Python の Decimal 型に対応します（数値を比べるときは型に注意）。言語の型から型記述子への変換はマーシャリング（marshall）と呼ばれ、高水準のインターフェースが内部で行います。低レベルのクライアントを使うときだけ、型記述子を自分で付けます。

```
# 低レベルのクライアント：型記述子を自分で付ける
client.put_item(
    TableName="orders",
    Item={"pk": {"S": "o-1"}, "qty": {"N": "3"}},
)
# テーブルリソース：型記述子は不要
table.put_item(Item={"pk": "o-1", "qty": 3})
```

シリアル化の落とし穴は、受け渡しの途中で形が変わる所にあります。S3 のイベント通知に入るオブジェクトのキー名は URL エンコードされています。`red flower.jpg` は `red+flower.jpg` で届くので、関数で受けるときにデコードしてから S3 を引きます[^c1-44]。SNS から SQS や HTTP/S へ送るときは、既定では SNS のメタデータで包んだ JSON が届きます。raw message delivery を有効にすると、メタデータが取り除かれ、メッセージがそのまま届きます。[^c1-45]

API Gateway では、テキストのペイロードは UTF-8 の JSON 文字列として扱われ、JPEG、GZip、XML の一部などはバイナリです。バイナリを扱うには、API に **binary media types** を設定します。クライアントの Accept ヘッダーの先頭の値で、バイナリで返すかどうかを判断します。ブラウザのように Accept の順序を制御できない相手には、`*/*` を設定します。統合の側では contentHandling を CONVERT_TO_BINARY または CONVERT_TO_TEXT にして、base64 のテキストとバイナリを変換します[^c1-46]。CONVERT_TO_BINARY は base64 の文字列をバイナリに戻し、CONVERT_TO_TEXT はバイナリを base64 の文字列にします。binaryMediaTypes も contentHandling も設

SAMPLE

定しないと、ペイロードはすべて UTF-8 の JSON 文字列として扱われます。[^c1-46]Lambda プロキシ統合の関数がバイナリを返すときは、本文を base64 にして isBase64Encoded を true にします。

場面で考えます。S3 のイベントで起動した関数が、名前に空白を含むオブジェクトを見つけられません。通知のキー名が URL エンコードされているためで、デコードしてから GetObject を呼びます。DynamoDB に数値を入れるときも同じ発想で、低レベルのクライアントなら `{"N": "3"}` のように、型記述子と文字列で渡します。

イベント駆動の実装: EventBridge、S3、SNS、SQS [1.1.8]

イベント駆動は、「何かが起きた」という通知を起点に処理が動く形です。通知を出す部品と受ける部品は、互いを知らなくて済みます（間にバスやキューが入ります）。

EventBridge のルールは、イベントパターンに合うイベントだけをターゲットへ送ります。パターンは JSON で書き、値の一致のほかに次の指定ができます。

- prefix（前方一致）や suffix（後方一致）
- numeric（数値の比較。`[">", 10, "<=", 20]` のように範囲も書ける）
- exists（項目の有無）
- anything-but（それ以外）

```
{
  "source": ["app.orders"],
  "detail-type": ["OrderPlaced"],
  "detail": {"amount": [{"numeric": [">=", 10000]}]}
}
```

複数の項目を並べると AND、1つの項目の値の配列は OR です[^c1-47]。配信に失敗したときの既定は、最大24時間・最大185回の再試行で、デッドレターキューを付ければ捨てられたイベントを残せます。[^c1-18]

EventBridge Scheduler は、予定を作るサーバーレスのスケジューラーです。スケジュールは3種類です。rate（一定間隔）、cron（暦に沿った繰り返し）、one-time（1回限りで `at(yyyy-mm-ddThh:mm:ss)` と書く）で、タイムゾーンを指定して評価されます。配信の柔軟な時間枠、再試行の回数、失敗した呼び出しを持ち続ける最大の時間も設定できます。[^c1-48]

S3 のイベント通知の宛先は、SNS のトピック、SQS のキュー、Lambda 関数、EventBridge です。SQS の FIFO キューは宛先に使えないので、FIFO へ送りたいときは EventBridge を経由します[^c1-49]。イベントの種類は s3:ObjectCreated:*（Put、Post、Copy、CompleteMultipartUpload を含む）や s3:ObjectRemoved:*（Delete、DeleteMarkerCreated）のように指定します。通知は少なくとも1回の配信を想定した設計で、通常は数秒で届きますが、1分以上かかることもあるため、届く時刻に頼る処理は組めません。同じ通知が重複して届くことがあるので、受け手は冪等に作ります。

SNS のフィルタポリシーは、購読ごとに付ける JSON の条件です。購読者は既定ではトピックの全メッセージを受け取りますが、ポリシーを付けると条件に合う分だけを受け取ります。ポリシーの適用範囲は FilterPolicyScope で選び、MessageAttributes（既定）ならメッセージ属性に、MessageBody なら本文に対して評価します[^c1-50]。受け手側で捨てる処理が要らなくなり、ファンアウトの先ごとに必要な分だけ届きます。

SAMPLE

SQS の標準キューと FIFO キューの違いは次のとおりです。

項目	標準	FIFO
順序	努力目標	メッセージグループの中で厳密
配信	少なくとも1回（重複あり）	重複排除の間隔内は重複なし
並列	受け手を増やすだけ	グループ単位で直列、グループ間は並列
Lambda のバッチサイズの最大	10,000	10

FIFO では、送信時に MessageGroupId を必ず付けます（最大128文字）。付けない送信は、エラーで返されます。[^c1-119]同じグループのメッセージは順に1つずつ処理されます[^c1-3]。重複排除は、MessageDeduplicationId を明示するか、コンテンツベースの重複排除を有効にして本文から作ります。メッセージの遅延（DelaySeconds）の既定は0秒で、最大は15分です。[^c1-3]

たとえば、店舗ごとに注文を順番どおりに処理し、全店舗は並列に進めたいとします。店舗 ID を MessageGroupId にした FIFO キューにします。1つのグループに処理の遅い注文があると、そのグループだけが詰まり、他の店舗は止まりません。

AI 支援開発ツールによるコードの生成・レビュー・最適化 [1.1.9]

AI 支援開発ツールは、コードの下書き、補完、見直し、脆弱性の検出を AI に手伝わせる道具です。試験では、道具の名前と役割、そして今の状態を結び付けられるかが問われます（名前の入れ替わりも出題の対象です）。

Kiro は仕様駆動の開発を行うツールです。Kiro の spec は、要件、設計、タスクを構造化した成果物で、3つのファイルで構成されます。requirements.md（バグ修正なら bugfix.md）、design.md、tasks.md です[^c1-51]。要件を決め、設計し、タスクに分けてから、タスクごとにコードを生成する流れです。曖昧な指示で一気にコードを出すのとは違い、後から検証できる記録が残ります。

Kiro には、さらに2つの仕組みがあります。**steering** は、パターンやライブラリ、標準といった決まりごとを Kiro に伝えるファイルです。プロジェクト直下の `.kiro/steering/` はそのワークスペースだけ、ホーム直下の `~/.kiro/steering/` はすべてのワークスペースに効きます[^c1-52]。**hooks** は、ファイルの変更、ツールの呼び出し、タスクの完了といったイベントを契機に、シェルコマンドまたはエージェントへの指示を自動で実行します。プロジェクトの設定は `.kiro/hooks/` に置きます。[^c1-53]

コードレビューの自動化では、名前が変わった道具に注意します。Amazon Q Developer の IDE プラグインは2027年4月30日にサポートが終わり、同等の機能が必要なら Kiro を使うよう案内されています（Kiro はエージェント型のコーディング、チャット、MCP に対応します）。コンソールとコンソールのモバイルアプリの会話の体験は AWS assistant という名前に変わり、IDE とコマンドラインの Q Developer は名前が続きます[^c1-54]。Amazon CodeGuru Reviewer は、2025年11月7日以降、新しいリポジトリの関連付けを作れず、既存の関連付けだけ使えます。代わりに、Amazon Q Developer のコードレビュー（SAST、シークレットの露出の検出、SCA＝依存関係の脆弱性の検出、品質問題の検出）と、Amazon Inspector のコードセキュリティが案内されています。[^c1-55]

使い分けは、作る前の段階か、書いた後の段階かで考えます。要件から設計、実装計画を作らせたいなら Kiro の spec です。書いたコードの補完やリファクタリングの提案、セキュリティのスキャンは、IDE に組み込む AI の機能の役割です。どの道具の出力も、確認なしにはマージしません。

SAMPLE

Amazon Bedrock などのマネージド AI サービスをアプリに組み込む [1.1.10]

マネージド AI サービスは、モデルを自分で動かさずに、API として呼び出して使えるサービスです。この試験で問われるのは、モデルの選定やプロンプトの設計ではなく、アプリのコードから呼び出す側です。

Amazon Bedrock は、100以上の基盤モデルに安全にアクセスして、生成 AI のアプリを作る完全マネージドのサービスです[^c1-56]。推論の API は bedrock-runtime のエンドポイントにあり、主に次の3つです。

API	本文の形	使い所
InvokeModel	モデルごとに異なる	特定のモデルの形式で直接呼ぶ
InvokeModelWithResponseStream	モデルごとに異なる	生成中の出力を順に受ける
Converse、ConverseStream	messages 形式で全対応モデルに共通	モデルを切り替える会話型のアプリ

ConverseStream は、Converse と同じ messages 形式のまま、生成中の出力を順に返します。[^c1-57]

Converse は、messages に役割と内容を並べ、system で指示を、inferenceConfig で全モデル共通の推論パラメータを渡します。モデル固有のパラメータは additionalModelRequestFields に入れます[^c1-57]。モデルを入れ替えてもコードをほぼ直さなくて済むのが利点です。modelId には、モデルまたは推論プロファイルの ID か ARN を指定します。動画生成のように時間のかかる処理には StartAsyncInvoke があり、後から結果を取り出します。[^c1-58]

```
import boto3

rt = boto3.client("bedrock-runtime")
res = rt.converse(
    modelId="MODEL_ID",
    system=[{"text": "簡潔に答えてください。"}],
    messages=[{"role": "user", "content": [{"text": "SQS とは何ですか。"}]}],
    inferenceConfig={"maxTokens": 300, "temperature": 0.2},
)
text = res["output"]["message"]["content"][0]["text"]
```

処理能力を固定の費用で確保したいときは **Provisioned Throughput** を購入します。購入した分が時間単位で課金され、カスタマイズしたモデルを呼び出す手段にもなります。一部のモデルは、カスタムモデルのデプロイでオンデマンドにも呼び出せます。[^c1-59]

社内の文書を根拠に答えさせるなら **Knowledge Bases** を使います。アプリからは、関連する断片だけを取り出す Retrieve と、取り出して回答まで生成する RetrieveAndGenerate の2つの操作があります[^c1-60]。**Agents** は、モデルが手順を考えて複数の呼び出しを行う仕組みで、アプリは InvokeAgent に agentId、agentAliasId、sessionId、inputText を渡します。sessionId を同じにすると会話が続き、応答はチャンクのストリームで返ります。[^c1-61]

Amazon Bedrock AgentCore は、任意のフレームワークとモデルで作ったエージェントを、安全に大規模に構築・展開・運用するプラットフォームです。Harness、Runtime、Memory などのモジュールを、単独でも組み合わせても使えます[^c1-62]。Runtime では、利用者のセッションごとに専用のマイクロ VM が割り当てられ、CPU、メモリ、ファイルシステムが分離されます。セッション間でデータが混ざらないのがこの構成の利点です。[^c1-116]

SAMPLE

Amazon Quick は、試験ガイドの表記で、公式のページでは AWS Quick です。自然言語のチャットと AI エージェントで、作業の自動化、データの分析、Web アプリの作成、調査を行うサービスで、Amazon QuickSight から発展しました。QuickSight は AWS Quick Sight として続いています[^c1-63]。試験の対象外にあるのは Amazon Quick Sight で、Amazon Quick とは別です（名前が近い点が引っかけになります）。

選び分けは、自分のアプリに生成機能を入れるなら Bedrock、アプリが使う判断の流れ自体をエージェントに任せて運用するなら AgentCore、業務の利用者が自分で使うチャットや分析なら Quick です。

Lambda 関数から VPC 内のプライベートリソースへ届く [1.2.1]

既定では、Lambda 関数は Lambda が管理する VPC の中で動き、インターネットへ出られます。自分の VPC にある RDS や ElastiCache、EC2 のようなプライベートリソースへ届かせるには、関数にその VPC の設定を加えます。指定するのは、サブネットとセキュリティグループです。VPC に接続した関数は、その VPC の中のリソースだけに届き、インターネットへは出られません。出口は別に用意します。[^c1-64]

接続の要点は3つあります。1つめは、そのリソースがあるプライベートサブネットにつなぐことです。パブリックサブネットにつないでも、インターネットへの経路やパブリック IP は得られません。2つめは、実行ロールに VPC 接続の権限を持たせることで、AWS 管理ポリシーの AWSLambdaVPCAccessExecutionRole を付けると、ec2:CreateNetworkInterface などの必要な権限が入ります。3つめは、セキュリティグループで、関数からリソースの通信を許す規則を書くことです。リソース側のセキュリティグループが関数からの通信を許可していないと、接続は成立しません。

接続のために Lambda が作るネットワークインターフェースは、**Hyperplane ENI** と呼ばれます。同じサブネットとセキュリティグループの組み合わせを使う関数の間で共有され、Lambda は既存の ENI を再利用します。

VPC 内の関数がインターネットや AWS のサービスへ出る道は、2通りです。

出口	向く行き先	要るもの
NAT ゲートウェイ	インターネット上の API、AWS のサービスの公開エンドポイント	パブリックサブネットに置いた NAT ゲートウェイと、プライベートサブネットの経路
VPC エンドポイント	AWS のサービス	S3 と DynamoDB はゲートウェイエンドポイント（追加料金なし）、他はインターフェースエンドポイント

S3 と DynamoDB にはゲートウェイエンドポイントとインターフェースエンドポイントの両方があり、ゲートウェイエンドポイントは、インターネットゲートウェイや NAT デバイスなしで届きます[^c1-65]。AWS のサービスだけに用があるなら、NAT より VPC エンドポイントのほうが閉じた経路になります。

RDS Proxy は、DB への接続をプールして共有する仕組みです。Lambda は呼び出しが増えると実行環境が増え、それぞれが DB に接続を開くので、接続数が急増して DB が過負荷になりかねません。プロキシを挟むと、多数の関数の接続をまとめて少数の接続で DB に渡します。クライアントに IAM 認証を強制でき、プロキシから DB へは IAM 認証か Secrets Manager の資格情報で接続します。DB のフェイルオーバーの間も接続を保ちます[^c1-66]。Lambda から RDS へつなぐ場面で「接続数が足りない」「フェイルオーバー時に切れる」と出たら、RDS Proxy が答えです。

場面で考えます。VPC のプライベートサブネットにつないだ関数が、RDS には届くのに、外部の決済 API の呼び出しだけタイムアウトします。プライベートサブネットに NAT ゲートウェイへの経路がないのが原因で、パブリックサブネッ

SAMPLE

トにつなぎ直しても直りません。NAT ゲートウェイを置いて経路を足します。同じ関数が S3 にだけ用があるなら、ゲートウェイエンドポイントで足ります。DB の接続数が上限に達するなら、RDS Proxy を挟むのが正解です（NAT では解決しません）。

Lambda 関数の設定: メモリ、タイムアウト、環境変数、レイヤー、同時実行 [1.2.2]

メモリは128〜10,240 MB（1 MB 刻み）です。CPU はメモリに比例して割り当てられ、1,769 MB で1 vCPU 相当になります。メモリを増やすと CPU も増えるので、CPU を使う処理は、メモリを上げると速くなります[^c1-10]。**タイムアウト**の既定は3秒で、1秒刻みで最大900秒（15分）です[^c1-68]。**エフェメラルストレージ**（/tmp）は512〜10,240 MB で、保存時に AWS 管理のキーで暗号化されます。[^c1-69]

設定	既定	範囲
メモリ	128 MB	128〜10,240 MB
タイムアウト	3秒	1〜900秒
/tmp	512 MB	512〜10,240 MB
環境変数	なし	合計4 KB
レイヤー	なし	1関数に5つまで
デプロイパッケージ	なし	zip を API や SDK で直接上げるなら50 MB、解凍後はレイヤー込みで250 MB、コンテナイメージは10 GB

環境変数は、コードを変えずに値を変える手段で、合計4 KB までです。Lambda は保存時に環境変数を暗号化します。別の鍵を使うなら KMSKeyARN を設定し、値をクライアント側で暗号化したい場合は暗号化ヘルパーを使います[^c1-71]。秘密の値は環境変数に平文で持ってはいけません。Secrets Manager などから取り出します。**ハンドラー**は、呼び出しの入口になる関数の指定で、Python では `ファイル名.関数名`（例 `app.lambda_handler`）の形です。**ランタイム**は、コードを動かす言語の実行環境で、Lambda が用意するものか、コンテナイメージ内に自分で入れたものを使います。

レイヤーは、ライブラリや共通のコードを関数から切り出して共有する zip です。実行時に /opt へ展開され、関数のコードから読めます（ランタイムごとのパスが通っています）。1関数に付けられるのは5つまでで、zip で配置した関数にだけ使えます。コンテナイメージの関数は、必要なものをイメージに入れます[^c1-72]。**拡張機能**（Extensions）は、監視やセキュリティのツールを関数に組み込む仕組みです。external extension は関数とは別プロセスで動き、関数の呼び出しが終わった後も動き続けます。internal extension は、ランタイムのプロセスの内側で動きます。[^c1-73]

同時実行は、同時に処理中のリクエストの数です。アカウントの既定の上限は、1リージョンで1,000です。これは引き上げを申請できます[^c1-10]。関数ごとの設定は2つあり、混同しやすい所です。

項目	予約済み同時実行（Reserved）	プロビジョニング済み同時実行（Provisioned）
目的	関数の同時実行の上限と下限を兼ねる	初期化済みの実行環境を前もって用意する
効果	他の関数に枠を取られない。上限を超える分はスロットリング	コールドスタートを減らす

SAMPLE

項目	予約済み同時実行（Reserved）	プロビジョニング済み同時実行（Provisioned）
追加料金	かからない	かかる
設定先	関数	エイリアスかバージョン（$LATEST には使えない）

予約済み同時実行は、アカウントの上限から100を残した分までしか割り当てられません。1,000の上限なら、1つの関数に1,000すべてを予約することはできません[^c1-75]。予約済み同時実行を0にすると関数は実行されなくなり、非同期のイベントは再試行なしで、設定済みのデッドレターキューか失敗時の送信先（on-failure destination）へ送られます。事故の拡大を止める非常ボタンとして使えます[^c1-76]。関数ごとのスケールの速さは、1つの関数につき10秒あたり1,000実行環境が上限です。[^c1-10]

プロビジョニング済み同時実行はエイリアスかバージョンに設定するので、イベントソースがそのエイリアスを指していないと、効果が出ません。[^c1-77]

場面で考えます。画像を縮小する関数が、タイムアウト10秒の設定で、ときどき切れます。CPU を使う処理なので、メモリを上げて CPU を増やします（タイムアウトの値だけ延ばしても根本は直りません）。別の関数が同時実行を使い切り、重要な関数がスロットリングされているなら、重要な関数に予約済み同時実行を割り当てます。初回の呼び出しだけ遅いなら、エイリアスにプロビジョニング済み同時実行を設定します。$LATEST には設定できません。

非同期呼び出しとイベントソースマッピングのエラー処理 [1.2.3]

失敗したイベントの行き先は、呼び出しの種類で扱いが変わります。同期呼び出しは、エラーを呼び出し元にそのまま返し、Lambda は再試行しません。非同期呼び出しは、Lambda が再試行します。イベントソースマッピングは、読み取りの種類（キューかストリームか）で挙動が違います。

非同期呼び出しで関数がエラーを返すと、Lambda は既定でさらに2回、合計3回まで実行します。1回目と2回目の間は1分、2回目と3回目の間は2分待ちます。MaximumRetryAttempts は0〜2で、MaximumEventAgeInSeconds は60〜21,600秒（最大6時間）です[^c1-78]。0 にすると再試行しません。再試行しない設定でも、失敗したイベントは失敗時の送信先に残せます。[^c1-120]再試行を使い切るか、イベントがこの年齢を超えると、イベントは捨てられるか、設定した行き先に送られます。

行き先は**送信先**（destinations）と**デッドレターキュー**（DLQ）の2種類で、違いは次の表のとおりです。

項目	送信先（destinations）	デッドレターキュー
条件	成功と失敗を別々に設定できる	失敗した（捨てられる）イベントだけ
宛先	SQS 標準、SNS 標準、S3（失敗のみ）、Lambda、EventBridge	SQS 標準か SNS 標準
送る内容	呼び出しの記録（要求と応答の詳細）	イベントそのもの（エラーコードと先頭 1 KB のエラーメッセージは属性に付く）

FIFO のキューと FIFO のトピックは、どちらにも使えません。送れなかったときは、CloudWatch に DestinationDeliveryFailures の指標が出ます[^c1-76]。成功のあとに別の処理を続けたいときや、失敗した理由まで残したいときは、送信先を使います。

SAMPLE

イベントソースマッピング（ESM）のうち、**SQS を読む場合**は、バッチのメッセージを受け取り、関数が成功すれば Lambda が削除します。エラーのときは、バッチ全体が可視性タイムアウトのあとに再び見えます。部分的なバッチ応答を有効にしていても、関数が例外で終わると、バッチ全体が失敗として扱われます。[^c1-80]成功していた分も再処理されるので、関数は冪等に作るしかありません。再処理を最小にするのが**部分的なバッチ応答**です。ESM の FunctionResponseTypes に ReportBatchItemFailures を指定し、関数が失敗したメッセージの ID だけを、`batchItemFailures` の `itemIdentifier` に入れて返します。[^c1-80]

```
def handler(event, context):
    failures = []
    for r in event["Records"]:
        try:
            process(r["body"])
        except Exception:
            failures.append({"itemIdentifier": r["messageId"]})
    return {"batchItemFailures": failures}
```

失敗を繰り返すメッセージは、キュー側の redrive policy（maxReceiveCount）でデッドレターキューへ移します。デッドレターキューは ESM ではなく、元のキューに付けます。

ストリーム（Kinesis、DynamoDB Streams）を読む場合は、失敗したバッチ全体を再試行し、エラーが続くと、そのシャードの処理は解消するか、レコードが期限切れになるまで止まります。止まった状態は Iterator Age の指標で見つけます[^c1-81]。止まり方を和らげる設定が4つあります。[^c1-82]

- MaximumRetryAttempts: 再試行の最大回数。既定は-1で無限
- MaximumRecordAgeInSeconds: 古すぎるレコードを捨てる年齢。既定は-1で無限
- BisectBatchOnFunctionError: エラー時にバッチを2つに割って再試行し、壊れたレコードを絞り込む
- OnFailure の送信先: 捨てたバッチの情報を送信先に残す

ストリームでも ReportBatchItemFailures が使えます。Kinesis では、itemIdentifier にレコードのシーケンス番号を入れ、複数の失敗があれば最小のシーケンス番号が再開の位置になります。[^c1-83]

場面で考えます。Kinesis を読む関数が、不正な形式のレコードを1件受けるたびに例外を投げ、そのシャードが数時間止まりました。MaximumRecordAgeInSeconds と MaximumRetryAttempts に有限の値を入れ、BisectBatchOnFunctionError を有効にして、OnFailure に送信先を付けます。壊れた1件は切り分けられて残り、他のレコードは止まりません。

関数を API Gateway、DynamoDB、SQS、Kinesis とつなぐ [1.2.4]

統合は3つの形があります。コードから AWS の API を呼ぶ形、イベントソースマッピングで Lambda がサービスを読みに行く形、そして送信先で結果を渡す形です。

API Gateway との統合は、Lambda プロキシ統合が標準です。API Gateway が要求の全体を1つのイベントにして関数に渡し、関数は決まった形のオブジェクトを返さなければなりません。イベントには、httpMethod、headers と multiValueHeaders 、queryStringParameters 、pathParameters 、stageVariables 、requestContext 、body 、

SAMPLE

isBase64Encoded が入ります。`?status=shipped` のようなクエリ文字列は queryStringParameters に、`/orders/{id}` のようなパスの変数は pathParameters に入ります。requestContext には、ステージ名、リクエスト ID、認可された呼び出し元の情報が入ります。[^c1-84] 関数の応答は、statusCode、headers、body、isBase64Encoded が基本で、body は文字列です。[^c1-84]

```
import json


def handler(event, context):
    return {
        "statusCode": 200,
        "headers": {"Content-Type": "application/json"},
        "body": json.dumps({"ok": True}),
        "isBase64Encoded": False,
    }
```

応答の形を満たさないと、API Gateway は 502 を返します。エラーを返すときも、statusCode に 400 や 500 を入れた同じ形のオブジェクトを返します。画像などのバイナリを返すときは、body を Base64 の文字列にして isBase64Encoded を true にし、API のバイナリメディアタイプに対象の型を足します（ブラウザから呼ぶなら */*）。[^c1-121]HTTP API は、Lambda 統合のペイロード形式を payloadFormatVersion の1.0か2.0で選びます。[^c1-85]

SQS、Kinesis、DynamoDB Streams との統合は、ESM の設定です。[^c1-82]

項目	Kinesis、DynamoDB Streams	SQS
BatchSize の既定	100（最大10,000）	10（標準の最大10,000、FIFOの最大10）
MaximumBatchingWindowInSeconds	0〜300秒、既定0	0〜300秒、既定0
ParallelizationFactor	1〜10	なし
最大同時実行（ScalingConfig）	なし	2〜1,000

ParallelizationFactor は、1 つのシャードから同時に処理するバッチの数です。シャードを増やさずに処理を速めたいときに上げます。[^c1-122]

既定のウィンドウ0は「レコードがあればすぐ起動」です。SQS の標準キューでバッチサイズを10より大きくするときは、ウィンドウを1秒以上にします。[^c1-87]

SQS の ESM は、5つのバッチを5つの同時呼び出しで処理し始め、メッセージが残っていれば1分あたり最大300ずつ呼び出しを増やします。1つの ESM が同時に処理できる上限は1,250です[^c1-88]。下流を守るために同時実行を絞るなら、ESM の Maximum concurrency を使います。これは関数の予約済み同時実行とは別の設定で、特定のキューに限った上限です。SQS を読む関数のタイムアウトは、キューの可視性タイムアウト以下にします。キューの可視性タイムアウトは、関数のタイムアウトの6倍に、バッチウィンドウの値を足した長さを勧められています。[^c1-87]

イベントフィルタリングは、ESM の FilterCriteria に書くパターンで、条件に合うイベントだけが関数を起動します。1つの ESM に既定で最大5つのフィルターを定義でき、それらは OR で結ばれます[^c1-89]。合わないイベントでは関

SAMPLE

数が起動しないので、起動回数と費用が減ります。

DynamoDB Streams は、表の変更を順序つきで最大24時間保持します。StreamViewType は KEYS_ONLY、NEW_IMAGE 、OLD_IMAGE 、NEW_AND_OLD_IMAGES から選び、更新の前後の値が要るなら NEW_AND_OLD_IMAGES にします。[^c1-90]

リソースベースのポリシーと実行ロールは、向きが逆です。実行ロールは「関数が他のサービスに対して何をしてよいか」を決め、信頼先は lambda.amazonaws.com です。リソースベースのポリシーは「誰が関数を呼んでよいか」を関数に付けます。API Gateway、S3、SNS などのサービスが関数を呼ぶには、そのサービスをプリンシパルにして lambda:InvokeFunction を許可します。条件に aws:SourceAccount を付ければ、呼び出し元のリソースを持つアカウントを絞れるので、ほかのアカウントの同じ名前のバケットからは呼べません。[^c1-123]ポリシーの上限は20 KB で、これを超える数の許可は書けません。[^c1-91]

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Principal": {"Service": "s3.amazonaws.com"},
    "Action": "lambda:InvokeFunction",
    "Resource": "arn:aws:lambda:us-east-2:111122223333:function:my-function",
    "Condition": {"StringEquals": {"aws:SourceAccount": "111122223333"}}
  }]
}
```

S3 や SNS から呼ばれる関数で「呼び出しが拒否される」なら、実行ロールを直すのではなく、関数のリソースベースのポリシーに許可を足します。ESM（SQS、Kinesis、DynamoDB Streams）は逆で、Lambda がそのサービスを読みに行くので、実行ロールにキューやストリームを読む権限が要ります。

場面で考えます。SQS を読む関数のタイムアウトが60秒で、バッチウィンドウが10秒のとき、キューの可視性タイムアウトは 6×60＋10 で370秒以上にします。30秒のままだと、処理中のメッセージが再び見えて、別の呼び出しに二重に処理されかねません。下流の DB が耐えられるのが同時20本までなら、ESM の Maximum concurrency を20にします。

データのライフサイクル: S3 ライフサイクル、DynamoDB の TTL、保持期間 [1.3.1]

データは、置きっぱなしにすると費用と管理の負担が増える一方で、自然には減りません。AWS には、期限や経過日数で自動的に移し、消す仕組みが各サービスにあります。

S3 ライフサイクルのルールは、2種類のアクションを持ちます。**transition** は、低コストのストレージクラスへ移すことで、**expiration** は、期限が来たオブジェクトを削除することです。ルールの対象は、キーのプレフィックス、タグ、オブジェクトのサイズで絞ります[^c1-92]。プレフィックスごとに期間を変えるなら、プレフィックスごとにルールを分けます。[^c1-92]移せる向きは、Standard から Standard-IA、Intelligent-Tiering、One Zone-IA、Glacier Instant Retrieval、Glacier Flexible Retrieval、Glacier Deep Archive へと、コストの安いほうへ下る一方向です。ストレージクラスを指定しないで置いたオブジェクトは、S3 Standard になります。[^c1-95]

SAMPLE

場面	ルールのアクション
30日たったログを安い層へ移す	Transition（Standard-IA など）
1年たったら削除する	Expiration
バージョニング有効で、古いバージョンを消す	NoncurrentVersionExpiration（NoncurrentDays、残す世代数は NewerNoncurrentVersions）
完了しなかったマルチパートアップロードを片付ける	AbortIncompleteMultipartUpload（DaysAfterInitiation）
最新が削除マーカーだけになったものを消す	ExpiredObjectDeleteMarker

AbortIncompleteMultipartUpload には、タグのフィルターを指定できません。ExpiredObjectDeleteMarker は、同じルールに Days と並べて指定できません[^c1-93]。未完了のマルチパートは、見えない領域に溜まって課金され続けるので、バケットごとに7日などで掃除するルールを入れます。

```
{
  "Rules": [{
    "ID": "tier-then-expire",
    "Status": "Enabled",
    "Filter": {"Prefix": "logs/"},
    "Transitions": [{"Days": 30, "StorageClass": "STANDARD_IA"}],
    "Expiration": {"Days": 365},
    "AbortIncompleteMultipartUpload": {"DaysAfterInitiation": 7}
  }]
}
```

128 KB 未満のオブジェクトは、既定ではどのストレージクラスにも transition しません。サイズの条件（ObjectSizeGreaterThan、ObjectSizeLessThan）を指定すると変えられます[^c1-94]。最小保存期間より前に移したり消したりしても、その分の課金が残ります。Standard-IA と One Zone-IA は30日、Glacier Instant Retrieval と Glacier Flexible Retrieval は90日、Glacier Deep Archive は180日です[^c1-95]。**S3 Versioning** は、同じバケット内に複数のバージョンを保持する機能で、ライフサイクルと組み合わせて古いバージョンを管理します。

DynamoDB の TTL は、アイテムの期限を属性に持たせて自動で消す機能です。使うには、まず表で TTL を有効にし、期限を入れる属性の名前を指定します。アイテムを作るときに限らず、更新するときにも、期限を計算して TTL の属性に入れ直してかまいません。[^c1-124]TTL の属性は Number 型の Unix エポック秒で保存します。期限が切れたアイテムは、期限から数日以内に自動で削除され、書き込みのスループットを消費しません。消える時刻は保証されないので、期限切れで未削除のアイテムが読み取りに出ることがあります。そのため、読み取り側で TTL 属性を使ったフィルター式をかけて除きます。削除はサービスによるものとして DynamoDB Streams に記録され、ユーザーによる削除とは区別できます。インデックスからも、通常の削除と同じように消えます。[^c1-96]

保持期間の既定は、サービスごとに違います。

サービス	既定	設定できる範囲
CloudWatch Logs	無期限	ロググループごとに設定

SAMPLE

サービス	既定	設定できる範囲
SQS のメッセージ	4日	60秒〜14日(1,209,600秒)
Kinesis Data Streams	24時間	最大365日
DynamoDB Streams	24時間	固定

CloudWatch Logs は、既定で無期限に残り続けて費用が増えるので、ロググループごとに保持期間を設定します。アカウントレベルの保持ポリシーを設定すると、ロググループ側の設定を上書きします[^c1-97]。SQS の保持期間を過ぎたメッセージは、処理されていなくても消えます。

場面で考えます。バージョニングを有効にしたログ用バケットで、30日で Standard-IA へ移し、1年で削除し、古いバージョンは90日で消し、未完了のマルチパートは7日で片付けたいとします。ルールは、Transition、Expiration、NoncurrentVersionExpiration、AbortIncompleteMultipartUpload の4つのアクションで書けます。5 KB の小さなログは、既定では Standard-IA に移りません。サイズの条件を指定しないかぎり、Standard に残り続けます。

アクセスパターンに合わせたデータストアの選び方 [1.3.2]

データストアの選択は、どう読み、どう書くかで決まります。同じデータでも、キーで1件を引くのか、条件で絞るのか、結合や集計をするのかで、向く場所が変わります。

要件	選ぶもの	理由
キーで1件を高速に読み書きする。規模が大きい	DynamoDB	キーと値、ドキュメントの形で、ミリ秒単位
複雑なクエリ、JOIN、トランザクション、スキーマの制約	RDS、Aurora	リレーショナル
画像、動画、ログ、バックアップのような大きなオブジェクト	S3	容量の制約が実質なく、安価
繰り返し読まれる値を高速に返したい	ElastiCache、DAX	メモリ上のキャッシュ

DynamoDB のアイテムの最大サイズは400 KB です。それより大きなデータは S3 に置き、アイテムには S3 のキーを持たせる形にします[^c1-43]。RDS は、接続数が急増しても RDS Proxy が接続をまとめ、フェイルオーバーのときも接続を保ちます。[^c1-66]

S3 の整合性は、すべてのリージョンで、オブジェクトの PUT と DELETE の直後の読み取りが最新を返す強い整合性(read-after-write)です。新規書き込みも上書きも削除も同じです。[^c1-99]

ストレージクラスは、アクセスの頻度と取り出しの速さで選びます。

クラス	向く用途	取り出し	最小保存期間
Standard	頻繁にアクセスする	ミリ秒	なし
Intelligent-Tiering	アクセスの頻度が分からない、変わる	ミリ秒	なし
Standard-IA、One Zone-IA	月に1回程度。One Zone-IA は1つの AZ のみ	ミリ秒	30日

SAMPLE

クラス	向く用途	取り出し	最小保存期間
Glacier Instant Retrieval	四半期に1回程度	ミリ秒	90日
Glacier Flexible Retrieval	年に1回程度	分〜時間（復元が必要）	90日
Glacier Deep Archive	年に1回未満	時間（復元が必要）	180日

Intelligent-Tiering は、アクセスの履歴を見て自動で層を移します[^c1-95]。アクセスの頻度が読めないデータは、ライフサイクルで下ろすより、このクラスに置きます。Flexible Retrieval と Deep Archive のオブジェクトは、そのままでは読めません。先に復元を要求します。

場面で考えます。利用者のプロフィール（1 KB）をキーで毎秒数万回読み書きし、添付の画像（5 MB）を保存し、月次の売上を表の結合で集計したいとします。プロフィールは DynamoDB、画像は S3 に置いてアイテムにキーを持たせ、集計は Aurora か RDS に任せます。画像を DynamoDB に入れようとしても、アイテムは400 KB までなので入りません。

キャッシュの戦略と、ElastiCache・DAX の選び分け [1.3.3]

キャッシュは、読み取りの多いデータを、速い場所へ置いて、元のデータストアへの負荷を減らします。戦略は、キャッシュに入れる時期で2つに分かれます（読む時と書く時）。

遅延読み込み（lazy loading）は、アプリがまずキャッシュを読み、ミス（データがない、または期限切れ）なら DB から読み、そのデータをキャッシュへ書く方式です。必要なデータだけがキャッシュに載り、キャッシュのノードが壊れても、ミスで DB を読むだけなので動き続けます。反面、ミスのたびに「キャッシュ読み、DB 読み、キャッシュ書き」の3往復がかかり、キャッシュのデータが古くなることがあります。[^c1-101]

書き込みスルー（write-through）は、DB に書くたびにキャッシュも更新する方式です。キャッシュのデータは古くなりません。書き込みごとにキャッシュと DB の2回の書き込みが要るので遅延が増え、読まれないデータもキャッシュに載ります。[^c1-101]

2つは組み合わせて使います。書き込みスルーで新鮮さを保ち、遅延読み込みで空のキャッシュを埋め、**TTL**（有効期限）を付けて古いデータと使われないデータを自然に消します。TTL が短いほど新鮮ですが、ミスが増えます。

ElastiCache は、Valkey、Memcached、Redis OSS から選べ、サーバーレスのキャッシュとノードベースのクラスターの2つの形で使えます[^c1-102]。Memcached は、単純なモデルで、複数コア・スレッドの大きなノードを使え、ノードの増減でスケールし、オブジェクトのキャッシュに向きます。一方 Valkey と Redis OSS は、複雑なデータ型を扱え、クラスターモードを有効にするとオンラインのリシャーディングもできます（Memcached はデータ型が単純で、オンラインのリシャーディングはできません）。[^c1-103]

DAX（DynamoDB Accelerator）は、DynamoDB 専用のインメモリキャッシュです。アプリは DAX のクライアントに切り替えるだけで、DynamoDB と同じ API で使えます。キャッシュは2つあり、item cache は GetItem と BatchGetItem の結果を主キーで保持し、query cache は Query と Scan の結果を、条件の値で保持します。item cache の TTL は既定で5分です。[^c1-104]DAX を通さずに DynamoDB を直接更新すると、DAX はその変更を知らないので、TTL が切れるまで古い値が消えません。[^c1-125]

SAMPLE

項目	DAX	ElastiCache
対象のデータストア	DynamoDB のみ	任意（RDS、計算結果、セッションなど）
アプリの変更	クライアントの差し替えだけ	キャッシュの読み書きのコードを書く
得意な読み取り	結果整合性の読み取りを、マイクロ秒に縮める	自由な戦略

BatchWriteItem、UpdateItem、DeleteItem、PutItem の書き込みは write-through で、DynamoDB の表へ先に、次に DAX へ書き、両方に書けたときだけ成功になります。強い整合性の読み取りは DAX を素通りして DynamoDB に届き、結果は DAX にキャッシュされません。そのため、強い整合性が必須の読み取りや、書き込みが中心のアプリには向きません。[^c1-104]

API Gateway のキャッシュは、ステージ単位で応答を保持します。TTL の既定は300秒、最大3,600秒です[^c1-26]。選び分けは、DynamoDB の読み取りを速くしたいなら DAX、DynamoDB 以外の DB の読み取りを軽くしたいなら ElastiCache、API の応答そのものを使い回したいなら API Gateway のキャッシュです。

場面で考えます。RDS の商品マスタを読む画面が重く、更新は1日に数回、読み取りは毎秒数千回です。読み取りが圧倒的に多いので、ElastiCache に遅延読み込みで載せ、TTL を数分にします。更新直後だけ古い値が出る点は、許容するしかありません。一方、DynamoDB のランキングを強い整合性で読む処理には、DAX は効きません。強い整合性の読み取りは DAX を素通りし、結果もキャッシュされないためです。

DynamoDB の設計と操作: キー、整合性、Query と Scan、インデックス、トランザクション [1.3.4]

パーティションキーとソートキーが、DynamoDB の主キーです。パーティションキーだけの単純なキーと、ソートキーを加えた複合キーがあります。パーティションキーの値でデータの置き場所（パーティション）が決まるので、アクセスが全キーに均等に散るように、**カーディナリティ**（値の種類）の高い属性を選びます。1つのパーティションは、読み取り3,000ユニット/秒、書き込み1,000ユニット/秒までを提供します[^c1-106]。注文日のような値の少ない属性や、人気商品のような特定のキーへの集中は、**ホットパーティション**になります。負荷が偏るときは、キーに乱数や番号を足して書き込みを分ける write sharding を使います。

RCU と WCU は、容量の単位です。強い整合性の読み取りは、4 KB まで1 RCU です。結果整合性の読み取りは0.5 RCU で、トランザクションの読み取りは2 RCU です（トランザクションは2倍かかります）。書き込みは、1 KB まで1 WCU で、トランザクションは2 WCU です。アイテムのサイズは、読み取りは4 KB 単位、書き込みは1 KB 単位に切り上げます。[^c1-33]

計算の例を示します。

- 6 KB のアイテムを強い整合性で読む: 6÷4を切り上げて2 RCU
- 同じアイテムを結果整合性で読む: 2×0.5で1 RCU
- 2.5 KB のアイテムを毎秒100件書く: 2.5を切り上げて3 WCU、3×100で300 WCU
- 同じ書き込みをトランザクションにする: 600 WCU

読み取りの整合性の既定は結果整合性です。表と LSI は強い整合性の読み取りも選べますが、GSI と DynamoDB Streams の読み取りは常に結果整合性です[^c1-108]。強い整合性にするには、読み取りの要求に ConsistentRead を指定します。

SAMPLE

Query と Scan は、読み方が違います。Query は、パーティションキーを必ず指定し、ソートキーの条件（KeyConditionExpression）で、そのパーティションの中から読みます。ソートキーが時刻なら BETWEEN で期間を切り出せ、ソートキーの逆順で読むには ScanIndexForward を false に、件数は Limit で絞ります。[^c1-126]Scan は、表（またはインデックス）の全体を読むので、大きな表では向きません。どちらも1回の呼び出しで最大1 MB までで、続きがあれば LastEvaluatedKey が返るので、ExclusiveStartKey に渡して続けます。**FilterExpression** は、読んだあとで結果を絞る条件で、1 MB の上限は絞る前に適用され、消費する読み取り容量も絞る前の分です。フィルターは費用を減らしません[^c1-109]。返す属性は **ProjectionExpression** で絞ります。

```
from boto3.dynamodb.conditions import Key, Attr

res = table.query(
    KeyConditionExpression=Key("pk").eq("user-1") & Key("sk").begins_with("2026-"),
    FilterExpression=Attr("status").eq("OPEN"),
    ProjectionExpression="sk, amount",
    ConsistentRead=True,
)
```

大きな表の Scan は、**並列スキャン**で速くします。TotalSegments に並列度を指定し、各ワーカーが別の Segment の値で自分の区画を Scan します。[^c1-109]

LSI と GSI は、主キー以外の検索路を足す二次インデックスです。[^c1-111][^c1-117]

項目	LSI（ローカル）	GSI（グローバル）
キー	表と同じパーティションキー、別のソートキー	表と別のパーティションキーとソートキー
作れる時期	表の作成時だけ	あとからでも作れる
整合性	強い整合性も選べる	結果整合性のみ
1表あたりの上限	5（増やせない）	20（AWS Support への連絡で引き上げ可）
制約	同じパーティションキーの値で、表とインデックスを合わせて10 GB まで	なし

運用中の表に LSI が要るときは、LSI を定義した新しい表を作ってデータを移します。1 つのパーティションキーの値が 10 GB に近づくなら、キーの値を分けてアイテムコレクションを小さくします。[^c1-111]

「あとから別のキーで検索したい」なら GSI で、「同じパーティションの中で別の並びが欲しく、強い整合性が要る」なら表の作成時に LSI を入れます。インデックスへ写す属性（projection）は、KEYS_ONLY（キー属性のみで最小）、INCLUDE（指定した非キー属性を足す）、ALL（全属性で最大）から選びます。表のキー属性は常にインデックスに入ります。[^c1-112]

条件付き書き込みは、ConditionExpression が真のときだけ書き込む仕組みです。attribute_not_exists(pk) と書くと、同じキーのアイテムがまだなければ新規作成でき、あれば ConditionalCheckFailedException になります。これで二重登録を防げます（既存のアイテムは上書きされません）。**楽観的ロック**は、アイテムに version 属性を持たせ、

SAMPLE

読んだ値と同じときだけ更新して +1 する形です。別の誰かが先に更新していると条件が偽になって失敗するので、読み直してやり直します。

```
table.update_item(
    Key={"pk": "item-1"},
    UpdateExpression="SET qty = :q, version = version + :one",
    ConditionExpression="version = :v",
    ExpressionAttributeValues={":q": 5, ":v": 3, ":one": 1},
)
```

トランザクションは、TransactWriteItems で、最大100個の書き込みアクションを、全部成功か全部失敗かの1回にまとめます。対象は同じアカウントと同じリージョンの1つ以上の表に限られ、アクションの合計サイズは4 MB を超えられません。操作は冪等で、ClientRequestToken を付けて再送できます。1 つのトランザクションの中で、同じアイテムに 2 つ以上の操作は書けません。条件は、Put や Update 自身の ConditionExpression に付けます。[^c1-113]BatchWriteItem との違いは、バッチは一部が失敗し得るのに対し、トランザクションは全体が1つの単位になることです[^c1-113]。容量は読み取りも書き込みも2倍かかります。

容量モードは、オンデマンドとプロビジョンドです。オンデマンドは、読み書きの回数に応じて課金され、容量の計画が要らず、負荷が予測できないときに向きます。プロビジョンドは、RCU と WCU を指定し、負荷が予測できて安定しているときに向きます[^c1-114]。負荷の波が大きい新サービスはオンデマンド、毎日ほぼ同じ量が流れる処理はプロビジョンドが合います。

場面で考えます。注文表のパーティションキーが customerId、ソートキーが orderDate のとき、全顧客の当日分の注文を一覧にしたくなりました。Scan は表の全体を読み、1 MB ずつしか返さず、フィルターをかけても容量は減らないので向きません。orderDate を新しいパーティションキーにした GSI なら Query で引けますが、当日の書き込みが1つのキーに集中します。日付に 0〜9 の番号を足して10個に分け（write sharding）、10本の Query の結果をまとめて読みます。強い整合性が要る読み取りは GSI では選べないので、表そのもので引く設計にするのが筋でしょう。

SAMPLE

サンプルはここまでです

続き(第2章〜巻末)は、DVA-C03 問題集のご購入で、頻出度別ノート・暗記用ノートと合わせて3点すべてダウンロードできます。

SAMPLE

DVA-C03 学習用テキスト(無料サンプル)

版

2026年11月版(AWS Certified Developer - Associate (DVA-C03) 試験ガイド(docs.aws.amazon.com・2026-11-12取得)準拠)

発行

IT資格道場(https://www.shikaku-doujou.com)

お問い合わせ

info@shikaku-doujou.com

AWS(Amazon Web Services)は Amazon.com, Inc. またはその関連会社の商標です。記載されているその他の会社名・製品名・サービス名は、各社の商標または登録商標です。

本教材は公式シラバスの内容を基に作成したもので、AWS との提携・承認関係はありません。

本文・図表・設問はすべてオリジナルの著作物です。無断での転載・複製・再配布・二次販売はできません。お気づきの点は info@shikaku-doujou.com までご連絡ください。

SAMPLE