IT資格道場

AWS DEA-C01 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 公式 Exam Guide(DEA-C01 v1.1・2026年9月6日取得)準拠
2026年7月版

本書のご利用にあたって

- 本書は「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。
- 無料サンプルであっても著作物です。本書の転載・改変しての再配布・二次販売を禁じます。
- 本書を AI モデルの学習データとして利用すること(学習目的の入力・データセット化を含む)を禁じます。
- 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。

不正な配布を見つけた場合は info@shikaku-doujou.com までご連絡ください。

目次

この教材群の使い方(3点セット)

種類	役割
① 学習用テキスト(本書)	これだけで問題が解けるようになる本編
② 頻出度別ノート	テキストを読んだら次はこれ。頻出順に絞った問題と解説で「解ける」に橋渡し。試験直前の最終チェックにも
③ 暗記用ノート	覚えるべき要点だけを一問一答に凝縮。空き時間の反復と用語の再確認用

使い方: ① 学習用を読む → ② 頻出度別で解き方を掴む → ③ 問題演習で腕試し → ④ 頻出度別に戻って再確認(試験直前もここ)。暗記用は空き時間や用語の再確認に。

このテキストの位置づけ: 本書は①の学習用テキストです。まずはここを通読し、これだけで問題が解けるようになることを目標にしてください。

はじめに ── この教材の使い方

このテキストは、Amazon Web Services（AWS）が実施する **AWS Certified Data Engineer – Associate（試験コード DEA-C01）** の合格を目的に作られています。

この試験が問うのは、個々のサービスの暗記ではありません。問われるのは、**データを取り込んで変換し、要件に合うデータストアに貯め、パイプラインとして自動化・監視し、認証・認可・暗号化・監査で守れること**です。公式の試験ガイドは、想定される受験者を「AWS を使ったデータエンジニアリングの経験が2〜3年、一般的なデータエンジニアリングの経験が1〜3年」としています。

本書は AWS の公式教材ではなく、IT資格道場が独自に書き下ろしたオリジナルの教材です。実際に出題された問題を再現したものではありません。

試験の基本情報

項目	内容
試験名	AWS Certified Data Engineer – Associate（DEA-C01）
実施	Amazon Web Services（Pearson VUE のテストセンター、またはオンライン監督試験）
問題数	**65問**（うち採点対象 50問・非採点 15問。どれが非採点かは受験者には分かりません）
試験時間	**130分**
出題形式	**択一（4肢1正解）／複数選択（5肢以上から2つ以上）** の2形式
合格ライン	**スケールスコア 100〜1,000 のうち 720**
採点方式	全体で合否を決める補償型。分野ごとの足切りはありません

受験料・試験時間・出題数・試験ガイドは改定されることがあります。 お申し込みの前に、AWS 公式サイトで最新の情報をご確認ください。

想定される受験者

公式ガイドが想定するのは、**AWS でのデータエンジニアリング経験が2〜3年、一般的なデータエンジニアリング経験が1〜3年**ある人です。とはいえ本書は、**データ基盤の用語と AWS の基礎から順に積み上げる構成**にしてあり

ます。Glue や Redshift を触ったことがなくても、章の順に読めば必要な知識がそろいます。

出題範囲の全体像 —— 4つのドメインと本書の章

ドメイン	分野	採点対象に占める割合（公式）	本書
1	データの取り込みと変換	**34%**	第1章
2	データストアの管理	**26%**	第2章
3	データの運用とサポート	**22%**	第3章
4	データのセキュリティとガバナンス	**18%**	第4章

本書の章の並びは、試験ガイドの並びとまったく同じです。節見出しの末尾にある `[1.1]` のような番号は、試験ガイドの Task statement の番号です。公式ガイドを手元に置いて対照しながら読めます。AWS はドメインごとの割合だけを公表しており、Task 単位の出題数は公表していません。本書もそれ以上の数字は書きません。

サービス名・機能名は英語のまま覚える

本文では、**サービス名・機能名・設定項目名を英語表記のまま**書いています（Glue Data Catalog、Kinesis Data Firehose、Redshift Spectrum、Lake Formation、Step Functions、Glue Data Quality …）。本番の設問文と選択肢に出るのはこの表記であり、日本語訳だけを覚えても画面では出会いません。英語の名前を見た瞬間に「どのドメインの、どの場面の機能か」が反射で出る状態を目指してください。

全設問に効く判断軸 ——「3つの問い」

問い	何を切り分けるか	分かれ方
問い1: どの工程の話か	取り込み・変換／保存・カタログ／自動化・監視／セキュリティ	同じサービス名でも工程が違えば答えが変わります（例: Glue は「ETL ジョブ」「Data Catalog」「Data Quality」の3つの顔で出ます）
問い2: 要件は何を最優先しているか	レイテンシ・スループット・コスト・運用負荷	ストリーミングかバッチか、サーバーレスかクラスタか、はこの軸だけで一意に決まります
問い3: 最小権限・最小露出で満たせているか	IAM のスコープ・ネットワーク経路・ログの残し方	要件を満たす案が複数あるとき、正解はより狭い権限・より閉じた経路のものです

4ステップ学習法

購入者限定テキストは3冊で1セットです。

- **STEP 1（理解する）**: 本書「学習用テキスト」を通読し、4つのドメインの地図と3つの問いを頭に入れる
- **STEP 2（解き方を掴む）**: 「頻出度別ノート」で頻出度の高い論点を解いて、解き方を掴む
- **STEP 3（腕試し）**: 本サイトの問題演習で、本番と同じ2形式に慣れる
- **STEP 4（再確認）**: 「頻出度別ノート」に戻って総ざらい。試験直前もここへ戻る

「暗記用テキスト」は本線には入れません。通勤時間や休憩時間など、**空き時間の反復と用語の再確認**に並走させてください。

それでは、第1章から始めます。

第1章 データの取り込みと変換（ドメイン1・採点対象の34%）

このドメインは採点対象の34%を占め、四つのドメインの中で最大です。問われるのは「サービスを知っているか」ではなく、要件（遅延・件数・順序・再実行・コスト）から取り込み方式と変換方式を選び、壊れたときに直せるかどうかです。本章では、Task statement 1.1〜1.4 の順に、判断の分かれ目を数字と対応表で押さえていきます。

1-1 データを取り込む [1.1]

取り込み（ingestion）は、外にあるデータを自分の管理下に運び込む工程です。ここでの設計ミスは下流すべてに伝播し、あとから直すのがいちばん高くつきます。判断の軸は次の五つです。

軸	問い	主に効くサービス
遅延	秒単位か、分単位か、日次でよいか	Kinesis Data Streams / Amazon Data Firehose / AWS Glue
件数と帯域	1秒あたり何レコード・何MBか	Kinesis のシャード数、MSK のパーティション数
順序	同じキーの順序を守る必要があるか	Kinesis のパーティションキー、DynamoDB Streams
再実行	失敗した日をやり直せるか	保持期間、job bookmarks、S3 の生データ保存

軸	問い	主に効くサービス
コスト	常時起動か、来たときだけか	provisioned と serverless の別

1-1-1 ストリーミングソースから読む（Read data from streaming sources）[1.1]

ストリーミングソースは、レコードが到着し続ける前提のデータ源です。試験で名前が挙がるのは Amazon Kinesis、Amazon Managed Streaming for Apache Kafka（Amazon MSK）、Amazon DynamoDB Streams、AWS DMS、AWS Glue、Amazon Redshift の六つです。それぞれ「何を流すためのものか」が違うので、まず役割を切り分けます。

ソース	何が流れるか	典型的な受け手
Amazon Kinesis Data Streams	アプリが put した任意のレコード	Lambda、Managed Service for Apache Flink、Amazon Data Firehose
Amazon MSK	Apache Kafka のトピックのメッセージ	Lambda、Flink、Amazon Data Firehose、Glue streaming
Amazon DynamoDB Streams	テーブル項目の変更（item-level change）	Lambda、DynamoDB Streams Kinesis adapter
AWS DMS	ソースDBのトランザクションログ由来の変更（CDC）	Amazon S3、Kinesis Data Streams、Amazon Redshift ほか
AWS Glue	streaming ETL job としてストリームを読み、変換して書く	Amazon S3、Amazon Redshift

ソース	何が流れるか	典型的な受け手
Amazon Redshift	取り込み先。ストリーミング取り込みで Kinesis / MSK から直接読む	—

Amazon Kinesis Data Streams はシャード（shard）という単位でスループットを持ちます。provisioned mode では、1シャードあたり書き込みが最大 1 MB/秒 または 1,000 レコード/秒、読み取りが最大 2 MB/秒 または 2,000 レコード/秒です。読み取りは GetRecords で行い、1回の呼び出しで最大 10,000 レコード・最大 10 MB を取得でき、1シャードあたり毎秒5回の読み取りトランザクションが上限です。10 MB を返した直後の5秒間は例外が返ります。

on-demand mode では、新規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、トラフィックに応じて自動でスケールします。既定で作成できる on-demand のデータストリームは50本までです。provisioned と on-demand の切り替えは、1本のストリームにつき24時間で2回までという制限があります。

レコードのデータペイロードは base64 エンコード前で最大 10 MiB です。PutRecords は1リクエストで最大500レコードを扱えます。保持期間（retention period）は最小24時間、最大 8,760時間（365日）です。IncreaseStreamRetentionPeriod / DecreaseStreamRetentionPeriod で変更します。

Amazon MSK は Apache Kafka をマネージドで動かすサービスです。Kinesis Data Streams と迷ったら、次の対比で切ります。

観点	Kinesis Data Streams	Amazon MSK
API	AWS 独自（PutRecord / GetRecords / SubscribeToShard）	Apache Kafka の API がそのまま使える
スケール単位	shard	partition（トピック内）
既存資産	Kafka のコード資産は書き換えが要る	Kafka のプロデューサ・コンシューマをそのまま移せる
運用	シャード数を増減（UpdateShardCount）または on-demand	ブローカー数・パーティション数・MSK Serverless
選ぶ理由	AWS サービスとの結線が最短	Kafka 互換が要件、または既存 Kafka からの移行

誤答肢の切り方として、「オンプレミスの Kafka からできるだけ書き換えずに移したい」という設定で Kinesis Data Streams を選ばせる肢は、Kafka API 互換という要件を無視しているので切れます。逆に「AWS のサービスだけで最小構成」で MSK を選ぶと、ブローカー運用という不要な重さを持ち込むことになります。

Amazon DynamoDB Streams は、テーブルの項目が変更されたときにその変更レコードを流します。DynamoDB の変更データキャプチャ（change data capture）には二つのモデルがあり、DynamoDB Streams と Kinesis Data Streams for DynamoDB です。違いは次のとおりです。

観点	DynamoDB Streams	Kinesis Data Streams for DynamoDB
保持期間	24時間	最大1年

観点	DynamoDB Streams	Kinesis Data Streams for DynamoDB
コンシューマ数	1シャードあたり最大2	1シャードあたり最大5、enhanced fan-out なら最大20
重複レコード	出ない	まれに出ることがある
順序	同一項目の変更はテーブルでの変更順どおりに並ぶ	各レコードのタイムスタンプ属性で実際の順序を判定する
スループット制限	テーブルとリージョンのクォータに従う	無制限
処理の受け手	AWS Lambda、DynamoDB Streams Kinesis adapter	Lambda、Amazon Managed Service for Apache Flink、Amazon Data Firehose、AWS Glue streaming ETL

両方を同じテーブルで有効にすることもできます。「変更履歴を24時間より長く保持して後追い分析したい」「5つ以上のコンシューマが同じ変更を読む」なら Kinesis Data Streams for DynamoDB、「重複を出したくない・項目単位の厳密な順序が要る」なら DynamoDB Streams、という切り分けになります。

AWS DMS はデータベースの移行と継続レプリケーションのサービスで、ストリーミング源としては change data capture（CDC）を担当します。タスクの型は三つです。

タスク型	動き	使う場面
full load	既存データを一度だけコピー	停止できる移行、初回ロードだけ

タスク型	動き	使う場面
full load plus CDC	既存データを移したあと、変更を流し続ける	ダウンタイムを詰めた移行
CDC only	ターゲットにデータがある前提で、変更だけ流す	別手段で初期ロード済みの継続同期

CDC はソースDBのネイティブAPIでトランザクションログを読みます。Oracle は LogMiner API または binary reader API で SCN（system change number）を基準に、Microsoft SQL Server は MS-Replication または MS-CDC を使い LSN（log sequence number）を基準に、MySQL は行ベースの binlog を、PostgreSQL は logical replication slot と test_decoding プラグインを使います。ソース側の準備（Oracle の supplemental logging、MySQL の row-level binary logging）が要る点は覚えておいてください。

CDC の開始点は、カスタムのタイムスタンプ（custom CDC start time）か、ログ上のネイティブな位置（CDC native start point）で指定します。PostgreSQL はタイムスタンプから LSN や SCN へ写像する仕組みが無いため、custom CDC start time に対応しません。またタスク作成時に決めた CDC 開始点は後から変更できず、変えたければ新しいタスクを作ります。

重要な注意として、AWS DMS の CDC はリアルタイムのレプリケーションを保証するものではありません。レイテンシはソースの負荷・ネットワーク・レプリケーションインスタンスのリソース・ターゲットの取り込み能力で変わり、数分以上に伸びることもあります。CDC レイテンシに関する SLA はありません。「ミリ秒単位の即時反映が必須」という要件に DMS CDC を当てる肢は、この一点で切れます。

AWS Glue はストリーミング ETL job（job command は gluestreaming）として Kinesis Data Streams や Amazon MSK を読み、変換して Amazon S3

や Amazon Redshift に書けます。低ボリュームのストリーミングジョブ向けに G.025X というワーカータイプが用意されており、これは AWS Glue version 3.0 以降のストリーミングジョブでのみ選べます。

Amazon Redshift は、この文脈では受け手側です。ストリーミング取り込みで Kinesis Data Streams や Amazon MSK からマテリアライズドビュー（materialized view）として直接読み込めるため、Amazon S3 を経由せずに秒単位で分析対象に載せられます。「Firehose で S3 に落としてから COPY」より遅延が短くなる、という対比で押さえてください。

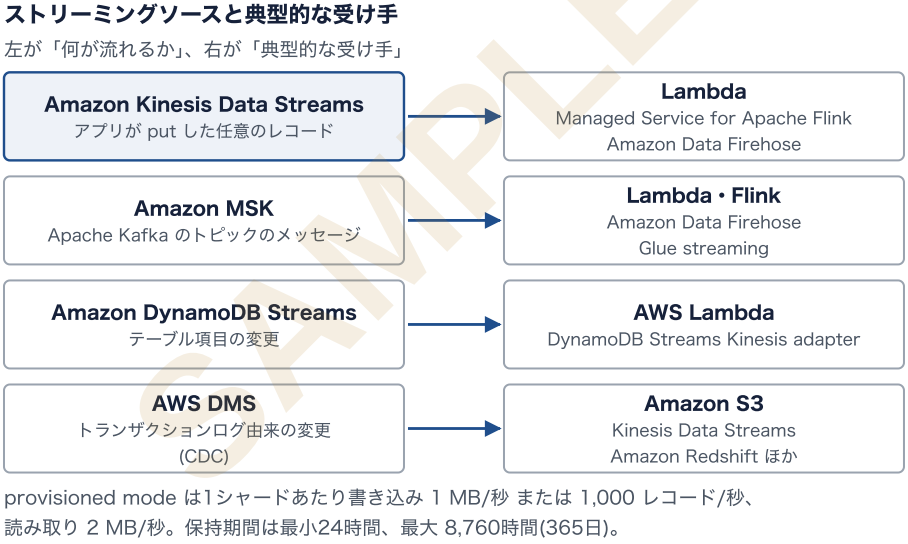

図 1-1　ストリーミングソースと典型的な受け手

1-1-2 バッチソースから読む（Read data from batch sources）[1.1]

バッチソースは、まとまった塊として置いてあるデータです。試験で挙がるのは Amazon S3、AWS Glue、Amazon EMR、AWS DMS、Amazon Redshift、AWS

Lambda、Amazon AppFlow です。

ソース	バッチでの役割	効く場面
Amazon S3	データレイクの一次置き場。オブジェクト単位で読む	生データの保管と再処理の起点
AWS Glue	crawlers でスキーマを起こし、ETL jobs で読み書き	サーバーレスな定期 ETL
Amazon EMR	Spark / Hive で大量データを分散処理	既存の Spark・Hive 資産、細かいチューニングが要る処理
AWS DMS	full load でDBを一括吸い出す	オンプレDBやRDSからの初回移行
Amazon Redshift	UNLOAD で外へ出し、COPY で取り込む	DWH と S3 の往復
AWS Lambda	小さなファイルをイベント駆動で処理	1ファイルあたり短時間で終わる軽い加工
Amazon AppFlow	SaaS からのデータ転送	Salesforce・Zendesk・Slack などからの取り込み

Amazon AppFlow はフルマネージドの統合サービスで、SaaS アプリケーションと AWS サービスの間でデータを交換します。ソースは Salesforce、Zendesk、Slack、Marketo、ServiceNow など、宛先は Amazon S3、Amazon Redshift、Snowflake などです。コードを書かずにフローを作れること、AWS PrivateLink と統合してインターネットを経由しない private なデータ転送ができること、Amazon S3 に転送したデータを AWS Glue Data Catalog にカタログ登録できること、パーティションと集約の設定で下流のクエリ性能を上げられることが特徴です。Amazon AppFlow Custom Connector SDK（Python と Java）で独自コネクタも作れます。

「Salesforce のレコードを Amazon Redshift に日次で入れたい」という設定で、Lambda に API を叩かせるコードを書く肢と Amazon AppFlow を使う肢が並んだら、後者が正解です。AppFlow はコネクタ・スケジュール・エラー処理・監視を持っているため、自作コードは運用負債になるからです。

AWS Lambda をバッチ取り込みに使うときの制約は明確です。関数のタイムアウトは最大 900秒（15分）、メモリは 128 MB から 10,240 MB まで 1 MB 刻み、/tmp のストレージは 512 MB から 10,240 MB まで 1 MB 刻みです。「数十GBのファイルを一度に変換したい」に Lambda を当てる肢は、この15分とメモリで切れます。そこは AWS Glue か Amazon EMR の領分です。

1-1-3 バッチ取り込みの適切な設定オプション（configuration options for batch ingestion） [1.1]

バッチ取り込みの設定は、次の四つに整理できます。

第一に、増分か全量かです。AWS Glue の job bookmarks は、前回までに処理したデータの状態を保存し、再実行時に新しい分だけを処理させる仕組みです。JDBC データソース、Relationalize トランスフォーム、そして一部の Amazon S3 ソースに実装されています。AWS Glue version 1.0 以降で job bookmarks が使える S3 のソース形式は JSON、CSV、Apache Avro、XML、Parquet、ORC です。

job bookmarks のオプションは三つです。

設定値	動き
Enable	実行後に状態を更新し、次回は前回チェックポイント以降の新しいデータだけを処理する

設定値	動き
Disable	ブックマークを使わず、毎回データセット全体を処理する。これが既定
Pause	前回の成功実行以降の増分（または job-bookmark-from と job-bookmark-to で指定した範囲）を処理するが、状態は更新しない

Amazon S3 の入力ソースでは、オブジェクトの最終更新時刻（last modified time）を見て再処理対象を判定します。前回実行以降にファイルが更新されていれば、再度処理されます。JDBC ソースでは、既定でプライマリキーがブックマークキーとして使われますが、これは連続的に増加または減少している（欠番が無い）ことが前提です。大文字小文字を区別する列名はブックマークキーに使えません。

運用上の落とし穴として、ブックマークを有効にしたままデータソースのパスだけを変えて transformation_ctx を変えないと、古いブックマーク状態が使われて新しいパスのファイルが「処理済み」と誤判定され、取りこぼしが起きます。全部を processing し直したいときは ResetJobBookmark（AWS CLI なら aws glue reset-job-bookmark --job-name）でリセットします。ただしリセットしてもターゲットのファイルは掃除されないので、出力先を別にしないと重複データが残ります。

図 1-3　AWS Glue job bookmarks —— 増分で読むための状態

第二に、並列度とワーカーです。AWS Glue のワーカーは DPU（Data Processing Unit）で測られ、1 DPU は 4 vCPU と 16 GB のメモリに相当します。

ワーカータイプ	DPU	vCPU / メモリ	ディスク	推奨用途
G.025X	0.25	2 vCPU / 4 GB	84 GB	低ボリュームのストリーミングジョブ（Glue 3.0以降のストリーミング専用）
G.1X	1	4 vCPU / 16 GB	94 GB	変換・結合・クエリなど大半のジョブ

ワーカータイプ	DPU	vCPU / メモリ	ディスク	推奨用途
G.2X	2	8 vCPU / 32 GB	138 GB	同上、より重い処理
G.4X	4	16 vCPU / 64 GB	256 GB	最も要求の重い変換・集約・結合
G.8X	8	32 vCPU / 128 GB	512 GB	同上
G.12X	12	48 vCPU / 192 GB	768 GB	非常に大きくリソース集約的な処理
G.16X	16	64 vCPU / 256 GB	1024 GB	最大級の処理
R.1X〜R.8X	1〜8	メモリ最適化構成	—	out-of-memory が頻発する、メモリとCPUの比率が高い処理

G.12X、G.16X、R.1X〜R.8X は起動レイテンシが大きい点が明記されています。また AWS Glue version 2.0 以降では Maximum capacity を指定できず、Worker type と Number of workers を指定します。

第三に、実行クラスです。Flexible execution class（Flex）は、事前検証・テスト・単発のデータロードなど急がないジョブ向けで、AWS Glue version 3.0 以降かつ G.1X または G.2X のワーカータイプで使えます。G.12X、G.16X、R.1X〜R.8X は Flex に対応しません。Flex の課金は「その時点で動いていたワーカー数 × 動いた時間」の合計なので、Max Capacity × 実行時間 という

単純計算とは異なります。時間に厳しいワークロードには standard execution class を使います。

第四に、再試行・タイムアウト・同時実行です。Number of retries は 0 から 10 で、タイムアウトに達したジョブは再起動されません。Job timeout の最大は 7日（10,080分）で、未指定時の既定は AWS Glue 4.0 以前が 2,880分、AWS Glue 5.0 以降が 480分です。Maximum concurrency の既定は1で、前の実行が走っている間に新しい実行が始まるとエラーになります。Job run queuing を有効にすると、サービスクォータやリソース不足で即座に実行できないジョブ実行が即失敗せずキューで待ち、リソースが空き次第自動で開始します。

「日次ジョブが前日分の実行と重なって二重書き込みが起きた」という障害設定では、Maximum concurrency が1のまま失敗させる設計か、Job run queuing で待たせる設計かを選ばせる問題になります。二重処理を許さないなら concurrency 1、遅れても必ず処理したいなら queuing です。

1-1-4 データAPIを消費する（Consume data APIs）[1.1]

外部システムからデータを取るとき、ファイルではなく API を叩く形になることがあります。API を消費するときの設計論点は五つです。

論点	対処
認証情報の保管	AWS Secrets Manager にAPIキー・トークンを置き、コードに埋め込まない
ページング	next token / cursor を保存し、途中失敗から再開できるようにする
レート制限	429 応答に対して exponential backoff with jitter で再試行する

論点	対処
ページ数の増加	1回の Lambda で終わらないなら Step Functions で分割する
差分取得	updated_since 形式のパラメータと、最後に取った時刻の保存先（DynamoDB など）を持つ

AWS 側のデータAPIとしては、Amazon Redshift Data API が代表です。これは HTTP ベースで Amazon Redshift に SQL を投げられる仕組みで、持続的なJDBC接続を張らずに Lambda や Step Functions から非同期にクエリを実行できます。VPC 内に関数を置かなくても呼べること、認証情報を AWS Secrets Manager または一時的な認証情報で扱えることが利点です。

Lambda で API を消費するときのペイロード上限も押さえます。同期呼び出しのリクエストとレスポンスはそれぞれ 6 MB、レスポンスストリーミングを使う場合は 200 MB、非同期呼び出しは 1 MB です。「API から返る100 MBのJSONを Lambda が同期で返す」設計は 6 MB の壁で成立しません。Amazon S3 に書いてキーだけ返す形に変えます。

1-1-5 スケジューラを設定する — Amazon EventBridge、Apache Airflow、time-based schedules [1.1]

jobs と crawlers を定期的に走らせる方法は三系統あります。

手段	何ができるか	向く場面
Amazon EventBridge	cron 式と rate 式のスケジュール、イベントパターンでの起動	AWS サービス横断の起動、イベント駆動と時刻駆動の両方

手段	何ができるか	向く場面
EventBridge Scheduler	cron と rate の定期実行に加え、一度きりの実行、柔軟な時間枠、再試行上限、失敗時の最大保持時間	大量のスケジュールを一元管理したいとき
Apache Airflow（Amazon MWAA）	DAG で依存関係つきのスケジュール	タスク間に順序と分岐がある日次パイプライン
AWS Glue の time-based schedule	ジョブとクローラを cron 式で起動	Glue だけで完結する定期処理

Amazon EventBridge はイベント駆動アーキテクチャのためのサーバーレスサービスで、イベントを取り込み・フィルタし・変換して配送します。処理と配送の方式は二つあり、event buses は多数のソースから多数のターゲットへルーティングするルーター、EventBridge Pipes は1つのソースから1つのターゲットへの point-to-point 統合です。Pipes は配送前の高度な変換とエンリッチメントに対応します。Pipes のターゲットに event bus を置いて、そこから複数ターゲットへ配る構成もよく使われます（たとえば DynamoDB stream をソースにした pipe から event bus へ、という形です）。

EventBridge Scheduler は、cron 式と rate 式による繰り返しパターン、または一度きりの呼び出しを、ひとつのマネージドサービスで作成・実行・管理できるサーバーレスのスケジューラです。配送の柔軟な時間枠（flexible time windows）、再試行の上限、失敗した API 呼び出しの最大保持時間を設定できます。

「毎日午前2時に Glue job を起動する」だけなら Glue のスケジュールや EventBridge のスケジュールルールで足ります。「Glue job が終わったら crawler を回し、その成功を待って Redshift に COPY し、どこかで失敗したら通知する」なら、依存関係の表現ができる AWS Step Functions か Amazon

MWAA が要ります。ここが誤答肢の分かれ目で、依存関係のある多段処理に対して「EventBridge の cron を三つ並べる」という肢は、前段の完了を待たない点で誤りです。

1-1-6 イベントトリガーを設定する — Amazon S3 Event Notifications、EventBridge [1.1]

Amazon S3 Event Notifications は、バケットで特定のイベントが起きたときに通知を出す機能です。設定はバケットの notification サブリソースに保存します。

通知できるイベントには、新規オブジェクト作成イベント、オブジェクト削除イベント、restore object イベント、Reduced Redundancy Storage（RRS）のオブジェクト消失イベント、レプリケーションイベント、S3 Lifecycle の有効期限イベントと移行イベント、S3 Intelligent-Tiering の自動アーカイブイベント、オブジェクトのタグ付けイベント、オブジェクト ACL PUT イベントがあります。

送信先は次の四つです。

送信先	注意点
Amazon SNS トピック	ファンアウトして複数の購読者へ配る
Amazon SQS キュー	FIFO キューは S3 の通知先として直接はサポートされない
AWS Lambda 関数	durable functions は通知先としてサポートされない
Amazon EventBridge	ルールで細かくフィルタでき、SQS FIFO へも中継できる

配送の性質として、Amazon S3 のイベント通知は「少なくとも1回（at least once）」の配送で設計されています。通常は数秒で届きますが、1分以上かかることもあります。ここは試験で狙われる箇所です。「必ず1回だけ届く」と書いてある肢は誤りで、受け手側を冪等（idempotent）に作る必要があります。

もうひとつの定番の落とし穴が、通知の起点になっているバケットに、その通知で起動した処理が書き戻す構成です。Lambda がアップロードのたびに起動し、その Lambda が同じバケットへオブジェクトを置くと、自分自身を間接的に起動し続ける実行ループになります。回避策は、バケットを入力用と出力用に分けるか、トリガーを入力用のプレフィックスに限定することです。

SQS FIFO キューへ S3 のイベントを送りたい場合は、Amazon EventBridge を経由させます。この一手が選択肢に出てきたら、それが正解になりやすい形です。

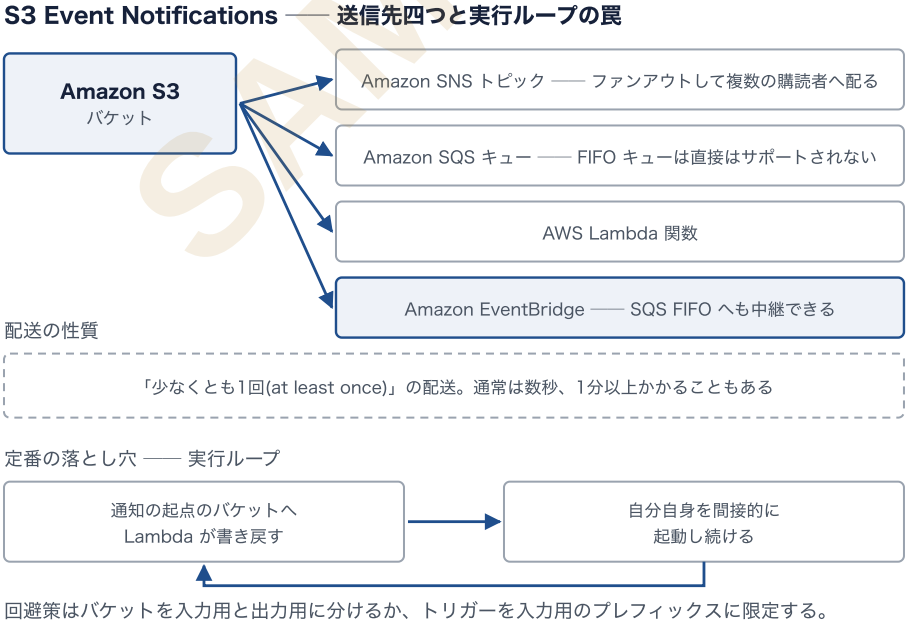

図 1-4　S3 Event Notifications ―― 送信先四つと実行ループの罠

1-1-7 Kinesis から Lambda function を呼ぶ（Call a Lambda function from Kinesis）[1.1]

Kinesis Data Streams と AWS Lambda の結線は、event source mapping というマッピングで作ります。Lambda はストリームからレコードをバッチで読み、関数を同期的に呼び出します。1つのバッチには単一のシャード（単一のデータストリーム）のレコードだけが入ります。

コンシューマの形は二つです。

形	動き	特徴
standard iterator（共有スループット）	Lambda が各シャードを毎秒1回の基本レートでポーリングする	読み取りスループットを他のコンシューマと共有する
enhanced fan-out（専有スループット）	Kinesis が HTTP/2 の SubscribeToShard でレコードを push する	シャードごとに専用の接続、レイテンシが小さい

バッチングの挙動として、既定ではレコードが到着し次第すぐ呼び出されます。小さすぎるバッチでの呼び出しを避けたいときは batching window を設定し、最大5分までレコードをためられます。Lambda は、バッチが満たされるか、batching window が切れるか、バッチが 6 MB のペイロード上限に達するまで読み続けます。

並列度は ParallelizationFactor で調整します。1つのシャードを同時に複数の Lambda 呼び出しで処理する設定で、既定は1、最大は10です。ParallelizationFactor を2にすると、100シャードのストリームに対して最大200の同時呼び出しが可能になります。データ量の変動が大きく IteratorAge

が高いときの処方箋がこれです。並列度を上げても、パーティションキー単位の順序は保たれます。

処理速度を上げるもうひとつの手が、シャードを増やすことです。Lambda は各シャード内のレコードを順序どおりに処理し、関数がエラーを返すとそのシャードの後続レコードの処理を止めます。シャードが多いほど同時に処理されるバッチが増え、ひとつのエラーが全体に及ぼす影響が小さくなります。

重要な前提として、Lambda の event source mapping は各イベントを「少なくとも1回」処理し、レコードの重複処理が起こり得ます。関数コードは冪等に書く必要があります。

拡張機能（extensions）を使っている場合、Lambda は次のバッチを送る前に拡張機能の完了を待ちません。そのため拡張機能が次のバッチ処理と並走し、アカウントの同時実行設定や上限に触れてスロットリングを起こすことがあります。シャード数より高い同時実行メトリクスが出ていたら、これを疑います。

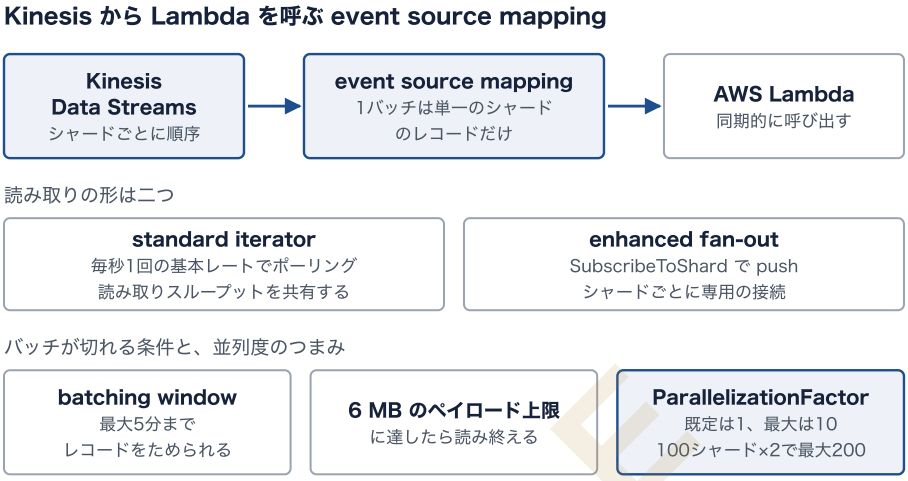

図 1-2　Kinesis から Lambda を呼ぶ event source mapping

Kinesis の集約（aggregation）を使う場合の挙動は、enhanced fan-out の有無で変わります。enhanced fan-out を使わない場合、集約イベントの中の全イベントが同じパーティションキーを持ち、それが集約イベントのパーティションキーと一致している必要があります。異なるパーティションキーが混ざると、パーティションキー単位の順序保証ができません。enhanced fan-out を使う場合、Lambda はまず集約イベントを個々のイベントへデコードします。集約イベントは中身と異なるパーティションキーを持てますが、パーティションキーに対応しないイベントは破棄され失われます。破棄されたイベントは処理されず、設定した失敗先（failure destination）にも送られません。

1-1-8 データソースへの接続を許すIPアドレスのallowlistを作る [1.1]

外部のデータソース（SaaS のAPI、取引先のデータベース、オンプレミスのシステム）に AWS 側から接続するとき、相手側が接続元IPアドレスの allowlist

を要求することがあります。ここで問題になるのは、Lambda や AWS Glue の実体は AWS 側で動的に配置され、送信元IPが固定されないという点です。

固定するための組み立ては次のとおりです。

手段	どう固定されるか	使う場面
NAT Gateway と Elastic IP	プライベートサブネットからの外向き通信が Elastic IP に集約される	Lambda や Glue を VPC 接続にし、外部SaaSへ出る
VPC endpoints（AWS PrivateLink）	そもそもインターネットへ出ずに AWS サービスへ到達する	S3・DynamoDB など AWS サービスが相手
Security groups	受信側の許可をIP範囲やセキュリティグループ単位で絞る	RDS・Redshift・MSK など VPC 内リソースへの接続
Amazon Redshift のセキュリティグループとサブネットグループ	クラスタへ入れる接続元を限定する	BIツールや ETL からの接続

Lambda を VPC に接続すると、その VPC のルーティングに従って外部へ出るため、NAT Gateway に付けた Elastic IP が送信元IPになります。相手先にはこの Elastic IP を伝えます。逆に、VPC に接続していない Lambda は AWS が管理するIP範囲から出るため、固定できません。「取引先が接続元IPの登録を求めている」という設定で、VPC 接続なしの Lambda をそのまま使う肢は切れます。

Elastic network interface（ENI）のクォータにも注意します。VPC あたりの ENI は既定で500で、これは Amazon EFS など他サービスと共有です。VPC 接続の Lambda を大量に並べると、この枠と VPC のIPアドレス枯渇が効いてきます。AWS Glue の VPC ベースのジョブでも、IP不足がジョブ起動時に検知

されればキューイングが働きますが、ドライバ起動後のエグゼキュータ確保中に枯渇するとジョブは失敗します。

1-1-9 スロットリングとレート制限への対処（throttling and overcoming rate limits） [1.1]

throttling は、サービスが規定の流量を超えた要求を意図的に拒否する仕組みです。DynamoDB、Amazon RDS、Kinesis のそれぞれで、原因と対処が違います。

Amazon DynamoDB では、パーティション単位の上限が効きます。テーブルの各パーティションは、最大で読み取り 3,000 read units/秒、書き込み 1,000 write units/秒を提供するよう設計されています。1 read unit は最大 4 KB の項目に対する強い整合性のある読み取り1回（結果整合性なら2回）、1 write unit は最大 1 KB の項目に対する書き込み1回です。項目サイズが 20 KB なら1回の強い整合性の読み取りで 5 read units を消費するので、その項目に対して同時に駆動できるのは毎秒600回までという計算になります。

したがって DynamoDB のスロットリングの主因は「hot partition」、つまりパーティションキーの偏りです。対処は次の順で考えます。

対処	中身
パーティションキーの再設計	全パーティションに均等に負荷が散るキーを選ぶ
write sharding	キーに接尾辞（ランダム値や算出値）を付けて書き込みを散らす
adaptive capacity	偏りをサービス側が吸収する。on-demand mode と provisioned capacity の両方に適用される

対処	中身
on-demand mode	ピークが読めないワークロードで、テーブル単位の上限まで自動でスケールする
exponential backoff	クライアント側で再試行の間隔を指数的に広げる

Amazon RDS では、接続数と IOPS が制限要因になります。Lambda を大量に並列起動すると1接続ずつ張られ、DBの最大接続数を食い潰します。対処は RDS Proxy による接続プーリング、Lambda の reserved concurrency で同時実行を上限で抑えること、そしてバッチサイズを大きくして呼び出し回数自体を減らすことです。「Lambda が RDS の接続上限に当たっている」という設定では、Lambda 側を無制限にスケールさせる肢ではなく、reserved concurrency で天井を作る肢が正解になります。

Amazon Kinesis では、シャードのスループット超過が ProvisionedThroughputExceededException として現れます。GetRecords が 10 MB を返した直後の5秒間、あるいはストリームのプロビジョンドスループットが足りない場合の1秒間、後続の呼び出しが例外を返します。GetShardIterator を頻繁に呼びすぎても同じ例外が出ます。なおシャードイテレータは返却から5分で失効します。

対処は、シャード数を増やす（UpdateShardCount）、on-demand mode に切り替える、パーティションキーを分散させる、読み手側を enhanced fan-out にして共有スループットの取り合いを解消する、の四つです。書き込み側では PutRecords でまとめて送り、失敗したレコードだけを再送します。

Amazon Data Firehose の Direct PUT では、PutRecord と PutRecordBatch を合わせたクォータが、US East（N. Virginia）・US West（Oregon）・Europe（Ireland）で 500,000 レコード/秒・2,000 リクエスト/秒・5

MiB/秒、その他のリージョンで 100,000 レコード/秒・1,000 リクエスト/秒・1 MiB/秒です。取り込み量が超えてスロットリングが起きると、Firehose は自動でスループット上限を引き上げますが、上がりきるまで時間がかかることがあるため、失敗したレコードの再送は続ける必要があります。なお、Kinesis Data Streams をソースにしている場合はこのクォータは適用されず、Firehose は制限なくスケールします。

1-1-10 ストリーミング配信の fan-in と fan-out を管理する [1.1]

fan-in は多数の送り手を少数のストリームに集めること、fan-out は1本のストリームを複数の受け手へ配ることです。

fan-in の設計論点は、パーティションキーの選び方と集約です。送り手が多いほどレコードは小さくなりがちで、Amazon Data Firehose の課金はレコード数に 5 KB（5,120 バイト）単位で切り上げたサイズを掛けたものなので、小さいレコードを大量に送るほど割高になります。同じ 5 MiB を送るのでも、5,000 レコードに分けるより 1,000 レコードにまとめたほうが安く済みます。送り手側での集約（Kinesis Producer Library の aggregation など）が効くのはこのためです。

fan-out の設計論点は、読み手が増えたときにスループットとレイテンシがどうなるかです。

方式	読み取りスループット	メッセージ伝播遅延	配送モデル
shared throughput consumers（enhanced fan-out なし）	1シャードあたり合計 2 MB/秒 固定。複数の読み手が共有する	読み手1つで平均約 200 ms、5つで平均約1,000 ms	GetRecords による pull

方式	読み取りスループット	メッセージ伝播遅延	配送モデル
enhanced fan-out consumers	読み手ごとに1シャードあたり最大 2 MB/秒 を専有。他の読み手の影響を受けない	読み手が1つでも5つでも平均約70 ms	SubscribeToShard で HTTP/2 による push

enhanced fan-out の登録可能なコンシューマ数は、On-demand Standard と Provisioned のストリームで1本あたり最大20、On-demand Advantage mode で最大50です。RegisterStreamConsumer API で登録し、同時に CREATING 状態にできるのは5つまでです。enhanced fan-out にはデータ取得コストとコンシューマ・シャード時間のコストがかかる点も設計判断に効きます。

分岐の型はこうです。「同じストリームを3つ以上のアプリが読み、そのうちの1つが遅延に厳しい」なら enhanced fan-out。「読み手が1つか2つで、遅延要件も緩い」なら shared throughput のままでコストを抑える。「読み手ごとに異なる加工をして別々の宛先に置きたい」なら、Amazon Data Firehose のストリームを複数本立てるか、EventBridge Pipes と event bus の組み合わせで配ります。

もうひとつの fan-out の形が、Amazon SNS のトピックに複数の Amazon SQS キューを購読させる構成です。SNS がファンアウトを担い、各 SQS キューが受け手ごとのバッファになります。ストリーム的な順序や再生が不要で、単純に「1つのイベントを複数の処理系へ配る」だけならこちらのほうが軽い構成になります。

1-1-11 取り込みパイプラインの replayability を説明する [1.1]

replayability（再実行可能性）は、過去のある時点からデータを流し直せるかどうかです。障害復旧、下流のバグ修正後の再処理、新しい集計軸の遡及計算で必要になります。

replay できるかは、保持期間と開始位置の指定手段で決まります。

仕組み	保持期間	開始位置の指定
Kinesis Data Streams	最小24時間、最大 8,760時間（365日）	シャードイテレータの型（TRIM_HORIZON、AT_TIMESTAMP、LATEST など）
DynamoDB Streams	24時間	ストリーム内の位置
Kinesis Data Streams for DynamoDB	最大1年	同上
Amazon MSK	トピックの保持設定に従う	オフセット
Amazon Data Firehose	配信先が使えない場合、DirectPut なら最大24時間データを保持する	—（Firehose 自体は replay の道具ではない）
Amazon S3	オブジェクトが残っている限り無期限	プレフィックス（日付パーティション）を指定して読み直す
AWS DMS	ソースDBのログ保持次第	custom CDC start time または CDC native start point
Amazon EventBridge	archive を作れば保持し、replay で流し直せる	archive と replay の期間指定

設計の定石は、加工前の生データを Amazon S3 に必ず落としておくことです。ストリームの保持期間は有限なので、365日を超える遡及や、ストリームを消してしまった後の再処理に耐えられません。Amazon S3 に生データがあれば、下流のロジックを直したあとで日付パーティション単位に流し直せます。これが「ストリーミングでも S3 に raw layer を持つ」という構成の理由です。

Amazon Data Firehose を replay の道具と誤解しないでください。Firehose が保持するのは、配信先が使えないときの最大24時間分（DirectPut の場合）であって、任意の時点からの流し直しはできません。ソースが Kinesis Data Streams で配信先が使えない場合は、Kinesis Data Streams 側の設定に基づいてデータが保持されます。

replay の際に必ず問題になるのが重複です。Lambda の event source mapping も、Amazon S3 Event Notifications も「少なくとも1回」の配送なので、replay をすれば同じレコードが二度処理されます。したがって replay を前提にするなら、下流を冪等に作るか、書き込み先を上書き可能な形（同じキーで PUT、または MERGE / UPSERT）にしておく必要があります。

1-1-12 stateful と stateless なデータトランザクションを定義する [1.1]

stateless（状態を持たない）処理は、1件のレコードだけを見て結果が決まる処理です。フィールドの型変換、単位換算、フィルタ、マスキングがこれにあたります。どの順で来ても、何度処理しても結果が同じなので、並列化と再試行が容易です。

stateful（状態を持つ）処理は、過去のレコードを覚えていないと結果が決まらない処理です。時間窓での集計、重複排除、セッション化、直前レコードとの差分計算がこれにあたります。状態をどこに置くかが設計の中心になります。

処理の性質	例	実装の置き場所
stateless	型変換、マスキング、フィルタ、フォーマット変換	Lambda、Glue ETL job、Firehose の変換 Lambda
stateful（短い窓）	直近5分の件数、移動平均	Amazon Managed Service for Apache Flink、Lambda の tumbling windows
stateful（長い状態）	ユーザー単位の累積値、重複排除の既視キー	Amazon DynamoDB、Amazon Redshift、Flink の状態バックエンド

AWS Lambda の Kinesis 向け event source mapping には tumbling windows という集約の仕組みがあり、シャード単位で固定長の時間窓の状態を引き継ぎながら処理できます。これが「Lambda で stateful な集計をやる」ときの手段です。それより複雑な窓（スライディングウィンドウ、セッションウィンドウ、複数ストリームの結合）が要るなら、Amazon Managed Service for Apache Flink を選びます。

トランザクションとしての stateful / stateless は、データストア側の話にもつながります。Amazon RDS や Amazon Aurora は複数行にまたがる ACID トランザクションを持ち、途中で失敗すればロールバックできます。Amazon S3 へのオブジェクト書き込みは単一オブジェクト単位で完結し、複数オブジェクトをまとめてロールバックする仕組みはありません。だから S3 のデータレイクで「途中まで書けた」状態を避けたいときは、一時プレフィックスに書いてから最後にマニフェストや Apache Iceberg のようなオープンテーブル形式でコミットする形をとります。

誤答肢の切り方として、「stateless な処理だから順序が保証される」という肢は誤りです。順序の保証はパーティションキーとシャードの構造から来るものであって、処理が stateless かどうかとは独立です。また「stateful だから並列

化できない」も誤りで、状態をキー単位に分割できれば（Kinesis のパーティションキーごと、Flink の keyBy ごと）並列化できます。

1-2 データを変換し処理する [1.2]

変換（transform）は、取り込んだ生データを分析できる形に整える工程です。ここでの問いは常に二つです。「どのエンジンで動かすか」と「どの形式で置くか」。この二つが決まると、性能もコストもほぼ決まります。

1-2-1 パフォーマンス要件に応じたコンテナ利用の最適化 — Amazon EKS、Amazon ECS [1.2]

データ処理をコンテナで動かす選択肢が Amazon EKS（Elastic Kubernetes Service）と Amazon ECS（Elastic Container Service）です。データエンジニアリングの文脈では、既存のコンテナ化された処理を持ち込むとき、あるいは Spark を Kubernetes 上で動かすときに出てきます。

選択肢	性質	向く場面
Amazon ECS	AWS ネイティブのオーケストレータ。学習コストが小さい	AWS だけで完結する定型のコンテナ処理
Amazon EKS	Kubernetes 互換。エコシステムがそのまま使える	既存の Kubernetes 資産、Spark on Kubernetes
AWS Batch	ジョブキューとコンピュート環境を管理し、ECS や EKS 上で実行	大量のバッチジョブのキューイングと優先度制御
AWS Fargate	ECS / EKS のサーバーレスな実行基盤。EC2 を管理しない	インスタンス管理をしたくない、断続的な処理

コンテナ利用の最適化で問われるのは次の点です。第一に、タスク定義のvCPU とメモリの割り当てです。過大に割り当てると空きリソースに課金され、過小だとタスクが OOM で落ちます。第二に、EC2 起動タイプと AWS Fargate の選択です。常時高負荷なら EC2 のほうが単価で有利になりやすく、断続的なら Fargate のほうが空き時間の課金が無い分だけ有利になります。第三に、コンテナイメージのサイズです。イメージが大きいほど起動が遅くなるので、Amazon ECR に置くイメージは多段ビルドで小さくします。第四に、スケーリングです。ECS Service Auto Scaling や Kubernetes の Horizontal Pod Autoscaler で、キューの深さやCPU使用率に応じて台数を増減させます。

判断の型はこうです。「Spark の ETL を動かしたい」だけなら AWS Glue か Amazon EMR が先です。コンテナを選ぶのは、既にコンテナ化された独自処理がある、Kubernetes のスケジューリングやオペレータが要る、といった理由があるときだけです。要件に何も書かれていないのに Amazon EKS を選ぶ肢は、運用の重さを持ち込むだけなので切れます。

1-2-2 さまざまなデータソースへ接続する — JDBC、ODBC [1.2]

JDBC（ Java Database Connectivity ）と ODBC（ Open Database Connectivity）は、リレーショナルデータベースへ標準的な方法で接続するためのインターフェイスです。JDBC は Java 系のアプリケーション、ODBC は言語非依存（Windows のBIツールなどでよく使われる）という違いがありますが、データエンジニアリングの文脈では「標準的なドライバでDBに繋ぐ口」として同じ扱いになります。

AWS のどこで出てくるかを整理します。

場所	使われ方
AWS Glue connections	JDBC 接続を登録し、crawlers と ETL jobs から RDS・Aurora・Redshift・オンプレDBに繋ぐ
Amazon Athena	JDBC ドライバと ODBC ドライバで BI ツールから Athena にクエリを投げる
Amazon Redshift	JDBC / ODBC で BI ツールや ETL ツールから接続する
Amazon EMR	Spark の JDBC データソースで外部DBを読む

AWS Glue で JDBC 接続を作るときの要点は四つです。第一に、接続先が VPC 内にあるなら、Glue のジョブをその VPC のサブネットに配置する Network 接続の設定が要ります。第二に、認証情報は AWS Secrets Manager に置き、接続定義から参照します。第三に、セキュリティグループは Glue の ENI からDBへの通信を許可する必要があり、Glue の ENI 同士が通信できる自己参照ルールも要ります。第四に、Amazon S3 など VPC 外のサービスへ出るには、VPC endpoints か NAT Gateway が要ります。

JDBC ソースからの増分読み取りには job bookmarks が使えますが、既定でプライマリキーをブックマークキーとして使い、それが欠番なく連続的に増減していることが前提でした。UUID のような順序を持たないキーではこの前提が崩れるので、スクリプトで明示的にブックマークキー（更新タイムスタンプ列など）を指定します。大文字小文字を区別する列名は使えません。

誤答肢の切り方として、「オンプレミスの Oracle から日次で差分だけ取りたい」に対して「毎回 full load で読み直す Glue job」という肢は、ソースDBへの負荷とジョブ時間の両方で劣ります。job bookmarks を有効にするか、AWS DMS の CDC を使うのが正解の方向です。

1-2-3 複数ソースのデータを統合する（Integrate data from multiple sources） [1.2]

複数ソースの統合は、形式・粒度・キー・時刻の四つを揃える作業です。

揃えるもの	ありがちなずれ	対処
形式	片方が CSV、片方が JSON、片方が Parquet	変換して Apache Parquet に寄せる
粒度	片方が明細、片方が日次集計	細かい側に合わせるか、集計側の粒度で丸める
キー	顧客IDの体系が違う（前ゼロ、大文字小文字、別採番）	正規化ルールを決め、マッピングテーブルを持つ
時刻	タイムゾーンが混在、記録時刻と発生時刻が別	UTC に統一し、event time と ingestion time を両方持つ

AWS での実装は三系統あります。第一に、AWS Glue の ETL job で複数のデータソースを読み、join して1つのテーブルに書く形。第二に、Amazon Athena の federated query で、S3 上のテーブルと外部データソース（RDS など）を1つの SQL で結合する形。第三に、Amazon Redshift に COPY で集約し、DWH 内で結合する形です。

どれを選ぶかは、結合の頻度とデータ量で決めます。日次で大量に結合してから使い回すなら Glue で事前結合して Parquet に落とす、たまにアドホックに結合するだけなら Athena の federated query、DWH の中で BI から繰り返し使うなら Redshift に寄せる、という順です。

AWS Glue Data Catalog は、この統合の土台になります。複数ソースのテーブル定義を1か所にまとめると、Athena も Amazon EMR も Amazon Redshift

Spectrum も同じメタデータを参照できます。ここに crawlers でスキーマを起こしておくのが、統合の第一歩です。

1-2-4 データ処理のコストを最適化する（Optimize costs while processing data）[1.2]

コスト最適化の効き目は、次の順で大きくなります。

第一に、読むデータ量を減らすこと。Amazon Athena はスキャンしたデータ量で課金されるため、列指向形式（Apache Parquet、ORC）への変換とパーティション分割が最も効きます。100列のうち3列しか読まないクエリなら、Parquet は該当列だけを読みます。パーティションを WHERE 句で指定すれば、そのパーティションのデータだけがスキャンされます。

第二に、圧縮すること。gzip、Snappy、ZSTD などで圧縮すればスキャン量と転送量が減ります。ただし gzip は分割できないため、Spark や Athena の並列読み取りが効きにくくなります。並列性が要るなら Snappy 圧縮の Parquet が定番です。

第三に、コンピュートの選び方。

手段	効き方
AWS Glue の Flexible execution class	急がないジョブを安く回す。実際に動いたワーカーの時間で課金される
Amazon EMR の Spot Instances	task nodes を Spot にする。HDFS を持たないので終了しても失われるデータが無い
Amazon EMR の transient cluster	処理が終わったらクラスタを終了する。常時起動をやめる
Amazon Redshift Serverless	使った分だけ。断続的なワークロード向け

手段	効き方
AWS Lambda のメモリ調整	メモリに比例して CPU が割り当てられるため、増やしたほうが総額が下がることがある

Amazon EMR で Spot Instances を使うときの原則は明快です。primary node が終了するとクラスタが終わるので、突然の終了が許容できる場合だけ Spot にします。core nodes は HDFS でデータを保持するため、Spot にすると部分的なデータ損失のリスクを負います。task nodes は HDFS の永続データを持たないので、終了してもデータは失われず、クラスタへの影響は最小限です。したがって定番は「primary と core は On-Demand、task は Spot」です。

なお Amazon EMR リリース 5.19.0 以降は YARN node labels を使い、application master プロセスが core nodes 上でだけスケジュールされるよう既定で設定されているため、task nodes の Spot Instance が終了してもジョブが失敗しにくくなっています。ただし Amazon EMR 6.x 系以降では YARN node labels 機能は既定で無効で、application master は core と task の両方で動きます。有効化するには yarn.node-labels.enabled を true に、yarn.node-labels.am.default-node-label-expression を CORE に設定します。

第四に、小さいファイルを作らないこと。Amazon Data Firehose の課金はレコード数に 5 KB 単位で切り上げたサイズを掛けるので、細かいレコードを大量に送ると割高になります。また S3 上に小さいファイルが大量にあると、Athena も Spark もファイルを開くオーバーヘッドで遅くなります。バッファリングして適度な大きさ（一般に数十MB〜数百MB）にまとめます。

Amazon Data Firehose の buffer interval のヒントは60秒から900秒の範囲です。遅延を詰めたいなら短く、コストとファイルサイズを優先するなら長く

設定します。

第五に、コストを見えるようにすること。AWS Cost Explorer で内訳を見て、AWS Budgets で閾値を超えたときに通知します。タグでジョブやチーム単位に費用を割り当てられるようにしておくと、どのパイプラインが高いのかが分かります。

1-2-5 要件に応じたデータ変換サービスを選ぶ — Amazon EMR、AWS Glue、Lambda、Amazon Redshift [1.2]

これはドメイン1で最も出題されやすい対比です。四つを一枚の表で覚えます。

サービス	実行モデル	得意	限界
AWS Lambda	イベント駆動・関数単位	小さいファイルの即時加工、軽い変換	タイムアウト 900 秒、メモリ最大 10,240 MB、/tmp 最大 10,240 MB
AWS Glue	サーバーレスの Spark（および Python shell）	定期 ETL、Data Catalog 連携、job bookmarks による増分	Spark の細かいチューニングは EMR ほど自由でない
Amazon EMR	クラスタ上の Spark / Hive など	大規模処理、Spark 設定の作り込み、既存 Hadoop 資産	クラスタの管理と起動時間、コスト設計が要る
Amazon Redshift	DWH 内の SQL（ELT）	すでに DWH にあるデータの集計・結合	生の半構造化ファイルの前処理には向かない

選び方の順序はこうです。まず「データはどこにあるか」。すでに Amazon Redshift にあるなら、外に出さず SQL で変換する（ELT）のが最速です。

Amazon S3 にあるなら Glue か EMR。1ファイルが小さくイベント駆動なら Lambda。

次に「どれくらいの規模か」。数MBから数百MBの単発なら Lambda で足ります。数GB以上、あるいは複数ファイルの join があるなら Glue。数TB規模、あるいは Spark のパラメータを細かく詰める必要があるなら EMR です。

最後に「運用の重さをどこまで許容するか」。サーバーレスで済ませたいなら Glue、クラスタを持ってでも制御したいなら EMR です。

誤答肢の切り方の典型を挙げます。「毎晩 500 GB の CSV を Parquet に変換する」に Lambda を当てる肢は 900秒のタイムアウトで切れます。「S3 に届いた 2 MB の JSON を即座に整形して DynamoDB に入れる」に Amazon EMR を当てる肢は、クラスタ起動のオーバーヘッドが処理時間を上回るので切れます。「すでに Redshift にある注文テーブルと顧客テーブルを結合して集計する」に Glue を当てる肢は、UNLOAD して読んで書き戻すという無駄な往復が入るので切れます。

AWS Glue の中でも、job type を選び分けます。Spark（job command が glueetl）は分散処理、Spark Streaming（gluestreaming）はストリーミング ETL、Python shell（pythonshell）は分散処理を必要としない小さなスクリプトです。「Spark が要らない軽いスクリプトを Glue で回したい」なら Python shell が正解で、ここで Spark ジョブを選ぶと DPU の無駄になります。

図 1-5　データ変換サービスの選び分け —— 居場所と規模と運用の重さ

1-2-6 フォーマット間の変換 ― .csv から Apache Parquet へ [1.2]

データレイクで最もよく行われる変換が、行指向のテキスト形式（.csv、JSON）から列指向のバイナリ形式（Apache Parquet、ORC）への変換です。

形式	構造	得意	不得意
.csv	行指向・テキスト	人が読める、どこでも扱える	型が無い、スキャン量が多い、圧縮効率が低い
JSON	行指向・テキスト	ネスト構造を表現できる	冗長、スキャン量が多い
Apache Avro	行指向・バイナリ	スキーマ進化に強い、行単位の書き込みが速い	列だけを読む用途には向かない

形式	構造	得意	不得意
Apache Parquet	列指向・バイナリ	列だけ読める、圧縮が効く、述語プッシュダウン	1行の追記には向かない
ORC	列指向・バイナリ	Parquet と同様。Hive 系で実績が厚い	同上

.csv から Apache Parquet への変換で得られるものは三つです。スキャン量の削減（必要な列だけ読む）、圧縮率の向上（同じ型の値が並ぶので圧縮が効く）、述語プッシュダウン（フッタの統計情報で、条件に合わないデータブロックを読み飛ばす）。Athena の課金がスキャン量である以上、これは性能改善であると同時にコスト削減です。

実装手段は三つあります。

手段	やり方
AWS Glue ETL job	DynamicFrame / DataFrame で読み、format="parquet" で書く。パーティション列を指定して書き分ける
Amazon Athena の CTAS	CREATE TABLE AS SELECT で、Parquet や ORC などのストレージ形式に変換した新しいテーブルを作る
Amazon Data Firehose	配信中に record format conversion で Parquet や ORC に変換する（AWS Glue Data Catalog のスキーマを参照する）

Amazon Athena の CTAS は、SELECT の結果から新しいテーブルを作り、データファイルを指定した S3 の場所に置きます。用途は、生データを何度も読み直さずに済むテーブルを一度で作ること、Apache Iceberg のような別のテ

ーブル形式へ移すこと、Parquet や ORC のようなストレージ形式へ変換してクエリ性能を上げコストを下げること、既存テーブルから必要なデータだけの複製を作ることです。なお Service Quotas の扱いでは、CTAS は DML として扱われます。

CTAS には作成できるパーティション数の制限（100パーティション）があり、それを超える場合は CTAS で最初の分を作ってから INSERT INTO で追加していく手順を取ります。この「CTAS + INSERT INTO」の組み合わせは、Athena だけで ETL を回すときの定石です。

1-2-7 変換の失敗と性能問題をトラブルシュートする [1.2]

よくある失敗と、その切り分けを型で持っておきます。

症状	よくある原因	対処
Glue ジョブが out of memory で落ちる	データの偏り（skew）、小さいファイルの大量リスト、ドライバへの collect	ワーカータイプを G.2X 以上や R 系へ、S3 file lister を使う、パーティション数を調整する
Glue の job bookmarks が同じデータを再処理する	S3 のファイルが更新された、transformation_ctx を変えずにパスを変えた	更新時刻を確認する、パス変更時は文脈も見直す
Spark ジョブの一部タスクだけ極端に遅い	キーの偏りによる data skew	キーに salt を足す、broadcast join に切り替える、事前に集計する
Athena のクエリが遅い・高い	パーティションが効いていない、小さいファイルが多い、CSV のまま	パーティションを WHERE で指定する、Parquet に変換する、ファイルをまとめる

症状	よくある原因	対処
Redshift の COPY が遅い	1つの大きなファイルを単一で読んでいる	ファイルを分割して並列ロードする、マニフェストを使う
Lambda がタイムアウトする	1回の処理量が多すぎる	バッチサイズを下げる、メモリを増やす、Glue や EMR へ移す
Glue のジョブが起動時に失敗する	VPC のIPアドレス枯渇	サブネットのIPを増やす、同時実行数を絞る、job run queuing を使う

性能を見る道具も押さえます。AWS Glue では Job metrics を有効にすると Amazon CloudWatch にメトリクスが出ます。Job observability metrics で追加のメトリクスが取れます。Continuous logging を有効にしないと、ログはジョブ完了後にしか見られません。Spark UI を有効にすると、ステージとタスクの偏りを目で確認できます。Delay notification threshold（分）を設定すると、RUNNING・STARTING・STOPPING が想定より長引いたときに通知が出ます。

障害設定の問題では「ジョブが走っている最中の状況が分からない」という記述が出たら Continuous logging、「どのステージで詰まっているか知りたい」なら Spark UI、「終わらないジョブに気づきたい」なら Delay notification threshold、と対応させます。

data skew（データの偏り）は、変換処理の性能問題として繰り返し出ます。特定のキーにレコードが集中すると、そのキーを担当するタスクだけが長時間動き、他のタスクは終わっているのにジョブが終わりません。対処は、キーにランダムな接尾辞を足して分散させてから再集計する、小さい側のテーブルを

broadcast join でブロードキャストする、事前に集計して結合対象を小さくする、パーティション数を増やす、の四つです。

1-2-8 AWS のサービスでデータを他システムに提供する data APIs を作る [1.2]

作ったデータを他システムに使ってもらう口が data APIs です。AWS での作り方は主に三つです。

構成	中身	向く場面
Amazon API Gateway + AWS Lambda + データストア	REST や HTTP の API を公開し、Lambda が DynamoDB や Redshift を読む	外部アプリからの少量・低遅延の参照
Amazon Redshift Data API	HTTP で Redshift に SQL を投げる。持続接続が不要	サーバーレスな分析処理から DWH を叩く
Amazon Athena の API / JDBC / ODBC	S3 上のデータに SQL で問い合わせる	データレイクをそのまま参照させる

Amazon API Gateway 経由の API を作るときの論点は、認証（IAM 認証、Amazon Cognito、API キー）、スロットリング、キャッシュ、そしてバックエンドとの同時実行の釣り合いです。API Gateway の既定のスロットル上限は毎秒 10,000リクエスト、Lambda の既定の同時実行数は 1,000 です。この差があるため、API Gateway が受けたリクエスト量を Lambda が捌ききれないことが起こり得ます。想定トラフィックに合わせて Lambda の同時実行クォータの引き上げを申請するか、API Gateway 側でスロットルをかけます。

もうひとつの落とし穴が、集計済みでないデータを毎回計算させる API です。参照のたびに Athena で数十GBをスキャンする API は、遅くて高くつきます。

設計としては、集計結果を DynamoDB や Amazon Redshift のマテリアライズドビューに事前計算して置き、API はそれを読むだけにします。

1-2-9 データの volume、velocity、variety を定義する — structured data と unstructured data [1.2]

データの性質を測る三つの軸が volume（量）、velocity（速度）、variety（多様性）です。

軸	何を測るか	設計への効き方
volume	総量と増加率。GB か TB か PB か	ストレージ階層、パーティション設計、処理エンジンの選択
velocity	到着の速さと必要な鮮度。日次か、分単位か、秒単位か	バッチかストリーミングか、Kinesis か Glue か
variety	形式の幅。1種類か、多数の異なる形式か	スキーマ管理、Data Catalog、変換の分岐

variety を語るときに出てくるのが、structured data（構造化データ）、semi-structured data（半構造化データ）、unstructured data（非構造化データ）の区別です。

種類	例	置き場所の定石
structured data	RDBのテーブル、CSV、Parquet。行と列が定義済み	Amazon RDS、Amazon Aurora、Amazon Redshift、S3 上の Parquet
semi-structured data	JSON、XML、Avro、ログ。タグや階層はあるが固定スキーマではない	Amazon S3、Amazon DynamoDB、Amazon OpenSearch Service

種類	例	置き場所の定石
unstructured data	画像、音声、動画、自由文の文書	Amazon S3。メタデータだけ別のストアに持つ

判断の型として、volume が大きく velocity が低い（日次で大量）ならバッチ処理（AWS Glue、Amazon EMR）、volume が中程度で velocity が高い（秒単位で到着）ならストリーミング（Kinesis Data Streams、Amazon Managed Service for Apache Flink）、variety が高いなら AWS Glue crawlers と Data Catalog でスキーマを起こしてから扱う、となります。

unstructured data については、そのままでは SQL で分析できません。画像や音声はいったん特徴量やメタデータに変換してから、structured または semi-structured の形で分析対象にします。この「変換して初めて分析できる」という点が、非構造化データを扱う問題の要点です。

1-2-10 large language models（LLMs）をデータ処理に組み込む [1.2]

Amazon Bedrock は、複数の提供元の foundation model へ API 経由で安全にアクセスできるフルマネージドのサービスです。データ処理の文脈で LLM を使う場面は、次のようなものです。

用途	中身
非構造化テキストの構造化	自由文の問い合わせ内容から、カテゴリ・製品名・緊急度を抽出して列にする
分類とタグ付け	ログや文書に分類ラベルを付ける
要約	長文のレポートを短い要約列にする

用途	中身
埋め込み（embeddings）の生成	テキストをベクトルに変換し、類似検索の対象にする

組み込み方の型は二つです。ひとつは、AWS Glue の ETL job や AWS Lambda の中から Amazon Bedrock の API を呼んで、レコードごとに推論させる形。もうひとつは、大量のレコードをまとめて処理する batch inference の形です。件数が多いほど、レコードごとの同期呼び出しは遅くて高くつくため、バッチ側に寄せます。

設計上の注意は四つです。第一に、スループットとスロットリングです。LLM の呼び出しは1件あたりの時間が長く、数百万件を1件ずつ同期で呼ぶと現実的な時間で終わりません。第二に、コストです。トークン量に比例するため、入力を必要最小限に絞ります。第三に、決定性がないことです。同じ入力でも出力が揺れうるので、結果を検証する仕組み（スキーマ検証、値の範囲チェック）を後段に置きます。第四に、機微データの扱いです。個人情報を含むテキストをそのまま推論に送らないよう、事前にマスキングします。

埋め込み（embeddings）とベクトル検索は、LLM を使った検索や推薦の土台になります。テキストをベクトルに変換して保存し、問い合わせもベクトルに変換して近いものを探す、という流れです。ベクトルの保存先としては Amazon OpenSearch Service や Amazon Aurora PostgreSQL が使われます。

誤答肢の切り方として、「数値の集計を LLM にやらせる」肢は誤りです。集計は SQL や Spark が正確かつ安価にできる仕事であって、LLM を使う理由がありません。LLM が効くのは、ルールで書き下せない自然言語の解釈です。この線引きが問われます。

1-3 データパイプラインをオーケストレーションする [1.3]

オーケストレーションは、複数の処理を「正しい順序で、条件に応じて、失敗しても立て直せる形で」つなぐ仕事です。個々のジョブが正しく動いても、つなぎ方が悪ければパイプラインは止まります。

1-3-1 ETL パイプラインのワークフローを組む — Lambda、EventBridge、Amazon MWAA、AWS Step Functions、AWS Glue workflows [1.3]

五つの手段を、表現力と運用の重さで並べます。

手段	表現できること	重さ	向く場面
Lambda	関数1つ分の処理。次の処理を自分で呼ぶ	最軽量	単段の処理、他サービスの起動役
EventBridge	イベントや時刻で起動する。順序の表現は持たない	軽い	起動のきっかけ、疎結合な連携
AWS Glue workflows	Glue の crawlers と jobs をトリガーでつなぐ	軽い	Glue だけで完結するパイプライン
AWS Step Functions	分岐・並列・再試行・待機・エラー処理を状態機械で書く	中	AWS サービスをまたぐ多段のパイプライン
Amazon MWAA（Apache Airflow）	DAG による依存関係、豊富なオペレータ、バックフィル	重い	大規模で複雑な依存、既存の Airflow 資産

AWS Glue workflows は、Glue の crawler と job をトリガーで連結する仕組みです。「crawler が終わったら job を起動し、その job が終わったら次の job

を起動する」という直列や、条件トリガーによる分岐が書けます。Glue の中だけで閉じるなら、これで十分です。逆に Amazon Redshift への COPY や Amazon EMR のステップ実行が混ざるなら、Step Functions か MWAA が要ります。

AWS Step Functions は Amazon States Language（ASL）でワークフローを定義します。状態機械の型は Standard と Express の二つで、作成後に変更できません。

観点	Standard Workflows	Express Workflows
最大実行時間	1年	5分
実行セマンティクス	exactly-once（正確に1回）	非同期は at-least-once、同期は at-most-once
実行履歴	Step Functions API で取得可能。完了後90日間保持。コンソールで視覚的にデバッグできる	Step Functions は履歴を保持しない。Amazon CloudWatch Logs でのロギングを有効にする必要がある
課金	状態遷移の数	実行回数・実行時間・メモリ消費
サービス統合	すべての統合パターンに対応	Job-run（.sync）と Callback（.waitForTaskToken）のパターンには対応しない
Distributed Map	対応	非対応
Activities	対応	非対応

選び分けの原則は、冪等かどうかです。Standard は exactly-once なので、Amazon EMR クラスタの起動や決済処理のような非冪等な処理のオーケス

トレーションに向きます。Express は at-least-once なので、DynamoDB への PUT のように何度やっても同じ結果になる冪等な処理に向きます。ETL パイプラインの多くは Standard です。5分を超える処理を含むなら、そもそも Express は選べません。

AWS Glue job の完了を待ってから次に進みたいときは、Job-run（.sync）の統合パターンを使います。これは Express Workflows では使えないため、「Glue job の完了を待ってから Redshift に COPY する」というパイプラインは Standard Workflows になります。ここが選択肢の分かれ目になります。

Amazon MWAA は Apache Airflow をマネージドで動かすサービスです。Apache Airflow のバージョンを選ぶだけでセットアップが済み、ワーカー（タスクを実行するコンピュートリソース）の最小数と最大数を設定すると、需要に応じて自動でスケールします。Apache Airflow のワーカーとスケジューラは AWS Fargate のコンテナとして、環境の Amazon VPC のプライベートサブネットに接続して動きます。各環境は AWS が管理する専用の Apache Airflow メタデータベースを持ち、VPC エンドポイント経由で安全にアクセスします。

Apache Airflow の webserver へのアクセスは Public network と Private network の二つのアクセスモードから選びます。どちらの場合も、ユーザーのアクセス制御は IAM で定義したポリシーで行います。データは AWS Key Management Service で自動的に暗号化されます。環境のメトリクスと、有効にすれば Apache Airflow のログが自動的に Amazon CloudWatch へ送られるため、タスクの遅延やワークフローのエラーを追加のツール無しで確認できます。

MWAA は Amazon Athena、AWS Batch、Amazon CloudWatch、Amazon DynamoDB、AWS DataSync、Amazon EMR、AWS Fargate、Amazon EKS、

Amazon Data Firehose、AWS Glue、AWS Lambda、Amazon Redshift、Amazon SQS、Amazon SNS、Amazon SageMaker AI、Amazon S3 などとのオープンソース統合をサポートし、多数の組み込みオペレータとコミュニティ製オペレータが使えます。

Step Functions と MWAA の選び分けは、次の観点です。

観点	AWS Step Functions	Amazon MWAA
定義方法	ASL（JSON / YAML）または AWS CDK	Python の DAG コード
課金	状態遷移や実行単位。動かなければほぼ課金されない	環境が起動している時間
依存関係	状態機械としての順序・分岐・並列	DAG による任意の依存グラフ、バックフィル、SLA
移行性	AWS 固有	Apache Airflow なのでポータビリティが高い
立ち上げの速さ	即時	環境の作成に時間がかかる

「たまにしか動かないパイプラインで、コストを抑えたい」なら Step Functions。「毎日数百のタスクが依存関係を持って動き、既に Airflow の DAG がある」なら MWAA。この対比が問われます。

図 1-6　オーケストレーション手段の階段 —— 表現力と運用の重さ

1-3-2 performance、availability、scalability、resiliency、fault tolerance を満たすパイプラインを組む [1.3]

五つの性質を、パイプラインの具体策に落とします。

性質	意味	具体策
performance	決められた時間内に終わること	列指向形式とパーティション、並列度（シャード数、ワーカー数、ParallelizationFactor）、不要なデータを読まない
availability	必要なときに使えること	マルチAZ構成のデータストア、サーバーレスサービスの利用、単一障害点を作らない

性質	意味	具体策
scalability	量が増えても対応できること	on-demand mode、auto scaling、シャードやワーカーの増減
resiliency	障害から復帰できること	再試行、チェックポイント、job bookmarks、CDC の開始点指定
fault tolerance	一部が壊れても全体が止まらないこと	dead-letter queue、失敗先（on-failure destination）、部分失敗の切り分け

fault tolerance の具体で押さえるべきは、失敗したレコードをどこへ逃がすかです。Lambda の event source mapping には、失敗したバッチの情報を送る on-failure destination（Amazon SQS キューや Amazon SNS トピック）を設定できます。バッチ全体を捨てずに失敗レコードを二分探索で絞り込む設定（bisect on function error）、最大レコード経過時間（maximum record age）、最大再試行回数（maximum retry attempts）も併せて設定します。これらを設定しないと、1件の毒レコードでシャードの処理が止まり続けます。

Amazon Data Firehose では、配信先が使えないとき、ソースが DirectPut なら最大24時間データを保持します。Amazon Redshift と OpenSearch Service への配信のリトライ期間（retry duration）は0秒から7,200秒の範囲で設定します。変換用の Lambda が失敗した場合、Firehose はエラー用の S3 バケットへレコードを配送します。Amazon MSK をソースにする場合、Lambda を有効にしていると 6 MB、無効なら 10 MB を超えるレコードはエラー用の S3 バケットへ送られ、CloudWatch のメトリクスが発行されます。

resiliency の要は、途中から再開できることです。AWS Glue の job bookmarks、AWS DMS の CDC recovery checkpoint、Kinesis のシャードイテレータ、Apache Airflow のタスク単位の再実行が、それぞれの層での「途中から」の仕組みです。設計時には「どの粒度でやり直せるか」を必ず決めます。1日分をまとめて処理する設計だと、23時間目の失敗で最初からやり直しになります。時間単位のパーティションに分ければ、失敗した1時間分だけを再実行できます。

availability について、EMR の HDFS レプリケーション係数も押さえます。既定のレプリケーション係数は、core node が10以上のクラスタで3、4から9のクラスタで2、3以下のクラスタで1です。レプリケーション係数を下げるとHDFS の冗長性が落ち、失われたブロックからの復旧能力が下がります。

1-3-3 サーバーレスワークフローを実装し保守する [1.3]

サーバーレスワークフローは、サーバーを管理せずにパイプラインを回す構成です。中核は AWS Step Functions、AWS Lambda、Amazon EventBridge、AWS Glue、Amazon Athena、Amazon Redshift Serverless です。

典型的な構成を一つ、順序どおりに書きます。

1. Amazon S3 の raw プレフィックスにファイルが置かれる。
2. Amazon S3 Event Notifications が Amazon EventBridge にイベントを送る。
3. EventBridge のルールが AWS Step Functions の Standard Workflow を起動する。
4. Step Functions が AWS Glue crawler を起動し、完了を待つ。
5. Step Functions が AWS Glue ETL job を Job-run（.sync）で起動し、CSV を Apache Parquet に変換して curated プレフィックスへ書く。

6. Step Functions が Amazon Redshift へ COPY を実行する（Amazon Redshift Data API を使えば持続接続が不要）。
7. 成功なら Amazon SNS で完了通知、失敗なら Catch から Amazon SNS でアラートを出す。

保守で効く設定は次のとおりです。

設定	効果
Step Functions の Retry と Catch	一時的な失敗を吸収し、恒久的な失敗だけを通知に回す
Lambda の reserved concurrency	下流のDBを守るための上限
Lambda の dead-letter queue と on-failure destination	失敗イベントを失わない
Glue の Job run queuing	クォータ待ちで即失敗させない
CloudWatch Logs へのロギング	Express Workflows では必須。Standard でも有効にすると詳細が追える
Step Functions の Distributed Map	大量のS3オブジェクトを並列処理する。Standard Workflows のみ

Standard Workflows は、同じ名前で実行中のワークフローがあると、新しい実行を開始せず冪等な応答を返します。これは二重起動を防ぐ仕組みとして使えます。Express Workflows では冪等性が自動管理されないため、同じ名前で複数起動すると並行実行になります。Standard Workflows の実行履歴データは90日後に削除され、その後は同じワークフロー名を再利用できます。

1-3-4 通知サービスでアラートを送る — Amazon SNS、Amazon SQS [1.3]

Amazon SNS（Simple Notification Service）は pub/sub の通知サービス、Amazon SQS（Simple Queue Service）はメッセージキューです。名前が並んで出るので、役割の違いを固定しておきます。

観点	Amazon SNS	Amazon SQS
モデル	publish / subscribe。1つのメッセージを複数の購読者へ	producer / consumer。1つのメッセージを1つの受け手が取る
配送	push（購読者へ送りつける）	pull（受け手がポーリングする）
保持	購読者が受け取れなければ再試行のうえ失敗	キューに保持され、受け手の都合で取り出せる
用途	人への通知（Eメール、SMS）、複数システムへのファンアウト	処理のバッファリング、負荷平準化、非同期の受け渡し

パイプラインでの使い分けはこうです。ジョブの成功・失敗を担当者に知らせるなら Amazon SNS。処理待ちのタスクをためて、下流が自分のペースで処理するなら Amazon SQS。両方を組み合わせて、SNS トピックに複数の SQS キューを購読させれば、1つのイベントを複数の処理系にバッファ付きで配れます。

失敗の受け皿としての使い方も重要です。Lambda の dead-letter queue、Step Functions の失敗時通知、Lambda event source mapping の on-failure destination は、いずれも Amazon SQS か Amazon SNS を指定します。Amazon S3 Event Notifications の送信先としては、SQS の標準キュー

は使えますが FIFO キューは直接指定できず、Amazon EventBridge を経由させる必要があります。

アラート設計で問われるのは、何をトリガーにするかです。Amazon CloudWatch のアラームをメトリクス（Glue のジョブ失敗、Kinesis の IteratorAge 、Lambda の Errors と Throttles 、Step Functions の ExecutionsFailed）に対して設定し、そのアラームのアクションとして Amazon SNS トピックに publish するのが標準形です。「ジョブが失敗したことに翌朝まで気づかなかった」という設定の問題では、この CloudWatch アラーム + SNS の組み合わせが正解になります。

一方、通知を出しすぎないことも設計です。一時的な失敗で毎回アラートを鳴らすと、本当の障害が埋もれます。Step Functions の Retry で自動復帰する範囲は通知せず、Catch に落ちた恒久的な失敗だけを通知する、という段構えにします。

1-4 プログラミングの概念を適用する [1.4]

この Task statement は、コードの書き方そのものではなく、データエンジニアが日常的に使う工学的な習慣を問います。試験の対象外タスクに「プログラミング言語固有の構文の知識を示すこと」が明記されているとおり、細かい文法は問われません。問われるのは、どの道具をどの場面で使うかです。

1-4-1 取り込みと変換のランタイムを縮めるコード最適化 [1.4]

ランタイムを縮める手は、効き目の大きい順にこうです。

手	中身	効く場面
読む量を減らす	パーティションプルーニング、列の絞り込み、述語プッシュダウン	ほぼ全ての処理。最初に検討する
形式を変える	.csv から Apache Parquet へ。適切な圧縮	S3 上の大量データ
ファイルサイズを整える	小さいファイルをまとめる。大きすぎるファイルは分割する	Spark、Athena、Redshift の COPY
並列度を上げる	シャード数、ワーカー数、ParallelizationFactor、Spark のパーティション数	スループット不足
偏りを解消する	data skew に対する salt、broadcast join	一部タスクだけ遅い
往復を減らす	1件ずつのAPI呼び出しをバッチ化する（PutRecords、BatchWriteItem）	大量の小さい書き込み
不要な shuffle を避ける	集計を先にする、結合キーで事前にパーティションする	Spark の重い join と groupBy
キャッシュする	繰り返し使う中間結果を永続化する、マテリアライズドビューを使う	同じ計算を何度もする処理

Amazon Redshift の COPY で押さえる原則は、並列ロードです。単一の大きなファイルを1本で読ませるより、複数のファイルに分割したほうがスライスが並列に読み込みます。マニフェストファイルを使えば、読み込むファイルの一覧を明示的に指定できます。COPY の入力1行の最大サイズは 4 MB です。

COPY のオプションでランタイムに効くものを挙げます。COMPUPDATE は圧縮エンコーディングを自動で決めるかどうか、STATUPDATE は統計情報を更新するかどうかを制御します。MAXERROR は許容するエラー行数の上限で、これを超えるとロードが失敗します。NOLOAD は実際にはロードせずデータの検証だけを行うので、大量ロードの前の確認に使えます。COPY はソースとして Amazon S3、Amazon EMR、SSH 接続経由のリモートホスト、Amazon DynamoDB を扱え、認証はクラスタにアタッチした IAM ロールの参照が推奨です。データ形式は固定幅、文字区切り、CSV、JSON、Avro、Parquet、ORC などに対応し、GZIP、BZIP2、LZOP、ZSTD の圧縮が使えます。既定の区切り文字はパイプ（|）です。なお Amazon Redshift Spectrum の外部テーブルは読み取り専用なので、COPY の対象にはできません。

Lambda のランタイム最適化では、メモリ設定が鍵です。Lambda はメモリ量に比例して CPU を割り当て、1,769 MB で1 vCPU 相当になります。CPU バウンドな処理では、メモリを増やすと実行時間が短くなり、単価が上がっても総額が下がることがあります。「メモリを減らせば安くなる」とは限らない、という点が問われます。

初期化コードの置き場所も効きます。ハンドラの外で作った接続やクライアントは、同じ実行環境が再利用される限り引き継がれます。ハンドラの中で毎回 SDK クライアントを作り直すと、その分の時間が毎回かかります。

1-4-2 同時実行と性能要件に合わせて Lambda functions を設定する [1.4]

concurrency（同時実行）は、その関数が同時に処理している実行中リクエストの数です。制御の仕組みは二つあります。

仕組み	何をするか	課金
reserved concurrency	その関数に割り当てる同時実行数の上限であり下限。他の関数はこの分を使えない	追加料金なし
provisioned concurrency	あらかじめ初期化しておく実行環境の数。コールドスタートを減らす	追加料金あり

reserved concurrency は、上限と下限の両方として働きます。重要な関数に確実に枠を確保する使い方と、下流のリソース（データベース接続など）を守るために上限を掛ける使い方の両方ができます。設定できるのは「未予約のアカウント同時実行数から100を引いた値」までで、残りの100は reserved concurrency を使わない関数のために残されます。アカウントの同時実行上限が 1,000 なら、1つの関数に 1,000 全部を予約することはできません。関数を意図的に止めたいときは reserved concurrency を0にします。

provisioned concurrency は、初期化済みの実行環境をあらかじめ確保しておく仕組みで、コールドスタートのレイテンシを減らし、2桁ミリ秒の応答を狙えます。Web やモバイルのように利用者がリクエストを起こす対話的なワークロードで効きます。データ処理パイプラインのような非同期のワークロードはレイテンシに敏感でないことが多く、通常は不要です。

アカウント全体の既定クォータは、同時実行数 1,000（数万まで引き上げ可能）です。関数ごとのスケーリング上限は、10秒ごとに 1,000 の実行環境です。同期呼び出しでは、実行環境1つあたり毎秒10リクエストまで処理でき、総呼び出し上限は同時実行上限の10倍になります。非同期呼び出しでは実行環境あたりのリクエスト数に上限がなく、総呼び出し上限は利用可能な同時実行数だけで決まります。

必要な同時実行数の見積もりは、次の式です。

Concurrency = (1秒あたりの平均リクエスト数) × (平均リクエスト処理時間・秒)

CloudWatch の Invocation メトリクスから毎秒のリクエスト数を、Duration メトリクスから平均処理時間を取ります。実際に動いている関数なら、ConcurrentExecutions メトリクスでピークを直接確認できます。上流と下流のスループット制約も併せて確認します。Lambda は負荷に応じてシームレスにスケールしますが、上流と下流が同じ能力を持つとは限りません。API Gateway の既定スロットルは毎秒10,000リクエスト、Lambda の既定同時実行数は 1,000 という不一致がその例です。

Kinesis や DynamoDB Streams からの呼び出しでは、同時実行数はシャード数と ParallelizationFactor で決まります。100 シャードに ParallelizationFactor 2 なら最大200です。拡張機能を使っていると、次のバッチの処理と並走してシャード数より高い同時実行が一時的に観測されることがあります。

そのほかの構成値も押さえます。関数のタイムアウトは最大900秒、環境変数は合計で 4 KB、レイヤーは5つまで、リソースベースのポリシーは 20 KB、コンテナイメージのコードパッケージは最大 10 GB (非圧縮、全レイヤー込み)、.zip のデプロイパッケージは Lambda API や SDK 経由のアップロードで 50 MB (圧縮時)、レイヤーとカスタムランタイムを含む展開後の内容が 250 MB です。それより大きいファイルは Amazon S3 経由でアップロードします。実行環境あたりのネットワーク帯域は 625 Mbps です。ファイルディスクリプタと実行プロセス・スレッドの上限はそれぞれ 1,024 です。

1-4-3 データエンジニアリングの言語とフレームワークを使う — Python、SQL、Scala、R、Java、Bash、PowerShell [1.4]

言語ごとに、AWS のどこで使うかを対応させます。

言語	主な使い場所
Python	AWS Glue の ETL スクリプト（PySpark）と Python shell job、AWS Lambda、Apache Airflow の DAG、boto3 による SDK 呼び出し、Amazon EMR の PySpark
SQL	Amazon Athena、Amazon Redshift、Amazon EMR の Hive と Spark SQL、Amazon RDS と Aurora
Scala	AWS Glue の ETL スクリプト、Amazon EMR の Spark。Spark 本体と同じ言語で書ける
Java	Amazon MSK のクライアント、Kinesis Client Library（KCL）、AWS Lambda、Amazon EMR
R	統計解析。Amazon EMR や Amazon SageMaker AI のノートブックで使う
Bash	Amazon EMR のブートストラップアクション、AWS CLI を並べた運用スクリプト、CI/CD のステップ
PowerShell	Windows 環境での運用、AWS Tools for PowerShell によるリソース操作

AWS Glue のスクリプトは Python と Scala で書けます。AWS Glue のバージョンによって使える Apache Spark と Python のバージョンが決まり、バージョンを指定せずに作ったジョブは既定のバージョンになります。Python shell

job（job command は pythonshell）は Spark を使わない小さなスクリプト向けです。

SQL は、この試験全体で最も出番の多い言語です。Amazon Athena では Presto / Trino 系の SQL、Amazon Redshift では PostgreSQL 由来の SQL を使います。Amazon Redshift では、Python の UDF が2026年6月30日以降サポートされなくなる点にも触れておきます。

言語選択そのものを問う問題は少なく、「Spark の資産があるので Scala か Python を選ぶ」「Kafka のクライアントを書くので Java」「集計は SQL で済ませる」といった形で、他の判断とセットで出てきます。分散処理が不要な小さな処理に PySpark のジョブを立てる肢は、DPU の無駄という理由で切れます。

1-4-4 ソフトウェアエンジニアリングのベストプラクティスを使う — version control、testing、logging、monitoring [1.4]

四つを、データパイプラインの文脈に落とします。

version control（バージョン管理）は、ETL スクリプト・IaC テンプレート・DAG・SQL をすべてリポジトリに置くことです。効果は、変更履歴が残ること、レビューができること、壊れたときに戻せることです。AWS では AWS CodePipeline、AWS CodeBuild、AWS CodeDeploy がこの流れを支えます。Amazon MWAA では DAG とサポートファイルを Amazon S3 のバケットに置くため、リポジトリから S3 への同期を CI に組み込みます。

testing（テスト）は、三層で考えます。

層	何を検証するか	道具
ユニットテスト	変換関数が期待どおりの出力を返すか	各言語のテストフレームワーク、AWS SAM CLI のローカル実行

層	何を検証するか	道具
統合テスト	サービスをつないだときに動くか	本番と同じ構成をIaCで別環境に作って実行
データ品質テスト	流れてきたデータ自体が正しいか	AWS Glue Data Quality、AWS Glue DataBrew

AWS Glue Data Quality は、オープンソースの DeeQu を土台にした、マネージドでサーバーレスなデータ品質の仕組みです。Data Quality Definition Language（DQDL）というドメイン固有言語で品質ルールを書きます。入口は二つあり、AWS Glue Data Catalog（すでにカタログ登録済みのデータセットを評価する）と AWS Glue ETL jobs（データレイクに載せる前に不良データを弾く、先回りの品質チェック）です。25以上の組み込みルールタイプが用意され、ルールの推奨（recommend rules）機能もあります。ルールを評価すると、通過したルールの割合としてデータ品質スコアが出ます。ETL jobs 側では、品質チェックに失敗したレコードを特定できます。

制限も押さえます。1つのルールセットに書けるルールは 2,000 まで、ルールセットのサイズは 65 KB まで、統計情報はアカウントあたり 100,000 まで（最大2年間保持）です。ネストやリスト型のデータソースは評価できないため、事前にフラット化が要ります。

logging（ロギング）は、Amazon CloudWatch Logs を中心に組みます。AWS Glue では Continuous logging を有効にしないとジョブ完了後にしかログが見られません。Step Functions の Express Workflows は Step Functions 自身が履歴を保持しないため、CloudWatch Logs へのロギングが必須です。Amazon MWAA は環境のメトリクスと、有効化すれば Apache Airflow のログを CloudWatch へ送ります。

monitoring（監視）は、メトリクスとアラームです。AWS Glue の Job metrics と Job observability metrics、Kinesis の IteratorAge、Lambda の Errors・Throttles ・ ConcurrentExecutions ・ Duration 、Step Functions の ExecutionsFailed を見て、CloudWatch アラームから Amazon SNS で通知します。API 呼び出しの記録は AWS CloudTrail が担当します。

1-4-5 Infrastructure as Code（IaC）でデータエンジニアリングのソリューションをデプロイする [1.4]

Infrastructure as Code（IaC）は、インフラをコードとして記述し、繰り返し同じ構成を作れるようにする手法です。手作業のコンソール操作に比べ、再現性・レビュー可能性・環境の複製が段違いです。

AWS の主な選択肢は二つです。

道具	書き方	特徴
AWS CloudFormation	JSON または YAML の宣言的テンプレート	AWS ネイティブ。スタック単位で作成・更新・削除。ドリフト検出、変更セット
AWS CDK（Cloud Development Kit）	TypeScript、Python、Java などのプログラミング言語	ループや条件でテンプレートを生成できる。最終的に CloudFormation テンプレートへ合成される

選び分けは、宣言的に書きたいか、プログラム的に書きたいかです。似たリソースを大量に生成する（100個のGlue jobを設定違いで作る、など）なら AWS CDK が有利です。構成が固定的で、コードを書ける人が限られるなら AWS CloudFormation のテンプレートのほうが読みやすくなります。AWS CDK は最終的に CloudFormation を経由するため、両者は排他ではありません。

データエンジニアリングで IaC に載せる対象は、AWS Glue のジョブとクローラと接続、AWS Step Functions の状態機械、Amazon EventBridge のルールとスケジュール、AWS Lambda の関数とイベントソースマッピング、Amazon Kinesis のストリーム、Amazon S3 のバケットとライフサイクル設定、IAM ロールとポリシー、Amazon Redshift のクラスタです。AWS Glue Data Quality のルールセットも AWS CloudFormation でサポートされています。Amazon AppFlow も AWS::AppFlow::ConnectorProfile と AWS::AppFlow::Flow で CloudFormation から作れます。

反復可能なリソースデプロイ（repeatable resource deployment）の利点は、開発・検証・本番の三環境を同じテンプレートからパラメータ違いで作れることです。「検証環境では動いたのに本番で動かない」という事故の大半は、手作業で作った構成の差から来ます。IaC はその差を消します。

誤答肢の切り方として、「本番だけコンソールで微調整する」肢は IaC の前提を壊します。次にテンプレートから更新をかけたとき、その微調整は消えます。変更はテンプレート側で行うのが原則です。

1-4-6 AWS SAM でサーバーレスのデータパイプラインをパッケージ・デプロイする [1.4]

AWS SAM（AWS Serverless Application Model）は、Infrastructure as Code でサーバーレスアプリケーションを構築するためのオープンソースのフレームワークです。短縮構文で CloudFormation のリソースとサーバーレス専用のリソースを宣言し、デプロイ時にインフラへ変換されます。

構成要素は二つです。

要素	中身
AWS SAM template	CloudFormation の拡張。サーバーレスリソースを簡略な構文で定義する
AWS SAM CLI	開発・ローカルテスト・デプロイを助けるコマンドラインツール

sam init でプロジェクトディレクトリを作ると、SAM テンプレート、アプリケーションコード、その他の設定ファイルが生成されます。AWS SAM で素早く定義できるのは、AWS Lambda 関数、Lambda durable functions、Amazon API Gateway の API、Amazon DynamoDB テーブルなどのサーバーレスリソースです。

AWS SAM の主な利点は次のとおりです。少ないコードでインフラを定義できること。SAM CLI で、作成・ビルド・デプロイ・テスト・監視という開発ライフサイクル全体を管理できること。AWS SAM connectors でリソース間の権限を宣言すると、意図に必要な IAM 権限へ変換されること。sam sync でローカルの変更をクラウドへ自動同期し、開発とクラウドでのテストを速められること。SAM CLI で Lambda 関数をローカルでテストし、API Gateway のエンドポイントを模擬し、デプロイ前にデバッグできること。

他の IaC 道具との使い分けはこうです。サーバーレスリソースの定義を簡潔にしつつ CloudFormation との互換を保ちたいなら、CloudFormation の代わりに SAM を使います。インフラを手続き的にではなく宣言的に書きたいなら、AWS CDK の代わりに SAM を使います。両方を組み合わせて、CDK アプリケーションのローカルテストに SAM CLI の機能を使うこともできます。SAM のリソースは同じテンプレート内で標準の CloudFormation リソースと並べて書けるため、既存の CloudFormation テンプレートにサーバーレス部品を足す使い方もできます。

データパイプラインの文脈では、「Lambda 関数と Step Functions の状態機械と DynamoDB テーブルを一式でデプロイする」という場面が典型です。ここで「コンソールで一つずつ作る」肢や「3つの別々のテンプレートに分ける」肢は、依存関係の管理という点で劣ります。

1-4-7 Lambda functions からストレージボリュームを使いマウントする [1.4]

Lambda が使えるストレージは三層あります。

層	容量	性質
/tmp のエフェメラルストレージ	512 MB から 10,240 MB まで 1 MB 刻み	実行環境ごとのローカル。実行環境が再利用される間は残るが、永続ではない
Amazon EFS のマウント	ファイルシステムの容量に従う（自動拡張）	複数の関数・複数の同時実行から共有できる永続ストレージ
Amazon S3 Files のマウント	S3 バケットの容量	S3 のオブジェクトを、read や write といった標準的なファイルシステム操作でファイルとして扱える

Lambda 関数は、ファイルシステムをローカルディレクトリにマウントするよう設定できます。サポートされるファイルシステムのタイプは Amazon Elastic File System（Amazon EFS）と Amazon S3 Files です。重要な制約として、1つの Lambda 関数は Amazon EFS か Amazon S3 Files のどちらか一方しか使えず、両方は使えません。既に一方が設定されている関数に他方を設定するには、先に既存のものを外す必要があります。

使い分けの型はこうです。1回の実行の中だけで使う作業ファイル（ダウンロードした .csv を展開して変換する、など）なら /tmp で足ります。10 GB を超えるデータや、複数の実行で共有したい状態（機械学習モデル、参照用の辞書ファイル）なら Amazon EFS です。S3 上のオブジェクトを、SDK ではなくファイルとして扱いたいライブラリを使う場合は Amazon S3 Files が候補になります。

Amazon EFS を使う場合、Lambda 関数を VPC に接続する必要があります。VPC 接続の Lambda は elastic network interface（ENI）を消費し、ENI のクォータは VPC あたり既定で500、これは Amazon EFS など他のサービスと共有です。大量の同時実行を想定する場合は、この枠と VPC のIP在庫を確認します。

誤答肢の切り方として、「Lambda の /tmp に 50 GB のファイルを展開する」肢は 10,240 MB の上限で切れます。「複数の Lambda 関数が同じ中間ファイルを共有する」に /tmp を当てる肢も、/tmp が実行環境ごとに独立している点で誤りです。

1-4-8 継続的インテグレーションと継続的デリバリー（CI/CD）を説明する [1.4]

CI/CD は、コードの変更を自動でビルド・テスト・デプロイまで運ぶ仕組みです。データパイプラインに当てはめると、次の流れになります。

1. 開発者が ETL スクリプト、IaC テンプレート、DAG をリポジトリにコミットする（implementation）。
2. パイプラインが起動し、AWS CodeBuild が依存関係を入れてユニットテストを実行する（testing）。
3. IaC テンプレートを検証し、検証環境へデプロイする。

4. 検証環境でサンプルデータを流し、統合テストとデータ品質チェックを実行する。
5. 通過したら本番環境へデプロイする（deployment）。AWS CodeDeployで段階的に切り替える。
6. デプロイ後のメトリクスを CloudWatch で監視し、閾値を割ったらロールバックする。

データパイプライン特有の論点は三つあります。

第一に、テスト用のデータをどうするかです。本番データをそのまま検証環境に置くと、個人情報の扱いが問題になります。マスキングしたサブセットを用意するのが定石です。

第二に、スキーマの変更をどう扱うかです。テーブルに列を足す変更は、既存のクエリを壊さない場合が多いですが、列の型を変える、列を消す変更は下流を壊します。CI の中でスキーマの互換性を検証する段を置きます。

第三に、ロールバックの単位です。コードは戻せますが、書き込んでしまったデータは戻りません。だからこそ、生データを Amazon S3 に残し、出力先を日付やバージョンでパーティション分けして、間違ったバージョンの出力を捨てて作り直せるようにしておきます。

AWS のサービス対応は、AWS CodePipeline がパイプライン全体の流れ、AWS CodeBuild がビルドとテストの実行、AWS CodeDeploy がデプロイ、AWS CloudFormation と AWS CDK と AWS SAM がインフラの定義です。AWS SAM は SAM テンプレートを使って、ステージングと本番の環境に必要な CloudFormation インフラを自動生成するデプロイパイプラインを構築できます。

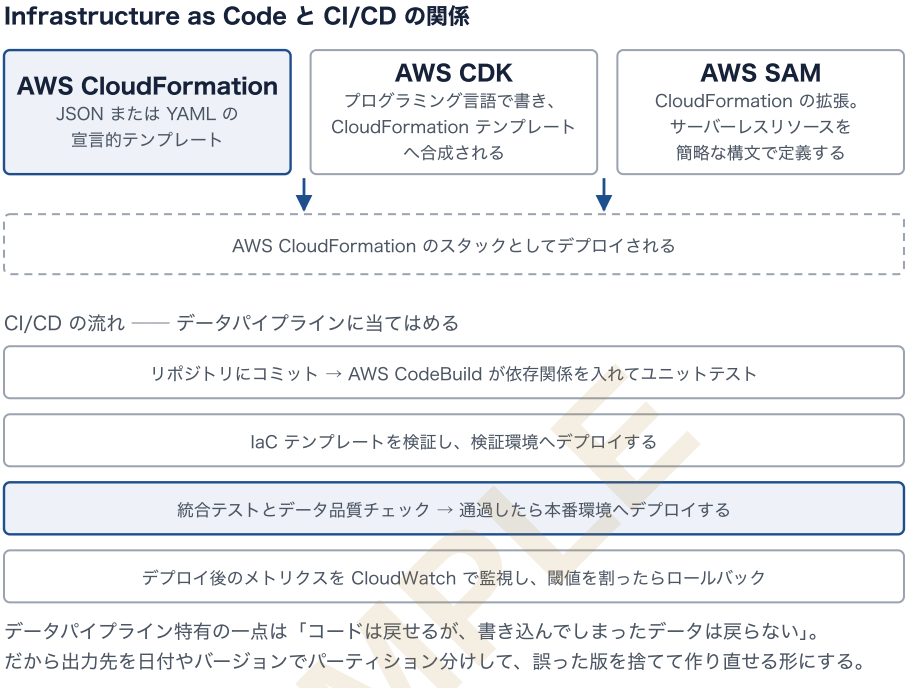

図 1-8　Infrastructure as Code と CI/CD の関係

1-4-9 分散コンピューティング（distributed computing）を定義する [1.4]

distributed computing（分散コンピューティング）は、1台では終わらない処理を複数のマシンに分けて同時に行い、結果をまとめる方式です。データエンジニアリングでは Apache Spark と Apache Hadoop がその代表です。

基本の語彙を押さえます。

用語	意味
ノード	処理を行う個々のマシン

用語	意味
パーティション	データを分割した単位。1つのタスクが1つのパーティションを処理する
タスク	1つのパーティションに対する処理の単位
shuffle	パーティションをまたいでデータを再配置する処理。ネットワークとディスクを使うので重い
データローカリティ	処理をデータのある場所に近づけること
skew	パーティション間でデータ量が偏ること。全体の完了が遅い側に引きずられる

Amazon EMR のクラスタは三つのノードタイプで構成されます。

ノードタイプ	役割	特徴
primary node	タスクを割り当てクラスタを制御する	終了するとクラスタが終わる。計算能力はあまり要らない
core node	タスクを処理し、HDFS にデータを保存する	処理能力とストレージ容量の両方が要る
task node	タスクを処理するがデータは保存しない	処理能力だけが要る。増減が容易

ノードの追加方法は、instance groups（同じインスタンスタイプを手動で追加、または CloudWatch メトリクスに基づく自動スケーリング）と instance fleets（On-Demand と Spot の目標容量を指定する）の二つの構成があります。インスタンスの購入オプション（On-Demand と Spot）は、クラスタ実行中には変更できません。primary と core を変えるにはクラスタを終了して作り

直す必要があり、task nodes なら新しい task instance group や instance fleet を作って古いものを外せます。

Spot Instances を使うときの挙動も違います。core instance group を Spot にすると、要求した台数すべてが確保できるまでインスタンスグループが起動しません。6台要求して5台しか確保できなければ、6台揃うかクラスタを終了するまで待ち続けます。一方 task instance group を Spot にすると、確保できた分だけで起動し、あとから残りを追加します。この非対称性が設計判断に効きます。

図 1-7　Amazon EMR のクラスタ構成 —— 三つのノードタイプ

分散処理でよく問われるのが「なぜ遅いのか」です。原因はほぼ三つに集約されます。shuffle が多すぎる（結合や集計の設計）、パーティションが偏っている（skew）、小さいファイルが多すぎてタスク数だけが膨らんでいる。それぞれ、

結合順序と broadcast join の検討、キーの salt 化、ファイルの集約が対処になります。

1-4-10 データ構造とアルゴリズムを説明する — graph data structures、tree data structures [1.4]

データエンジニアリングで出てくるデータ構造は、次の四つを押さえれば足ります。

データ構造	性質	AWS での現れ方
ハッシュ（key/value）	キーから値へ一定時間で到達する	Amazon DynamoDB のパーティションキー、Amazon MemoryDB
tree data structures（木構造）	階層を表す。範囲検索や順序付き走査ができる	B木系のデータベースインデックス、JSON のネスト構造、S3 のプレフィックス階層
graph data structures（グラフ構造）	ノードとエッジで関係を表す。多対多の関係と経路探索に強い	Amazon Neptune、ワークフローの DAG（有向非巡回グラフ）
列指向の配置	同じ列の値をまとめて置く	Apache Parquet、ORC、Amazon Redshift の内部形式

tree data structures は、インデックスの話につながります。B木系のインデックスは、キーの範囲検索（WHERE date BETWEEN ...）と順序付きの走査を高速にします。一方でキーの全体一致だけならハッシュのほうが速く、DynamoDB のパーティションキーはこの発想です。JSON の入れ子も木構造で、これをフラット化して列に展開する処理（AWS Glue の Relationalize トランスフォームなど）が ETL の定番です。

graph data structures は、関係そのものが分析対象になる場面で使います。ソーシャルグラフ、不正検知（口座と取引の結びつき）、推薦、ネットワーク経路、系譜がその例です。リレーショナルデータベースで多段の結合を繰り返す代わりに、グラフデータベース（Amazon Neptune）で辿るほうが自然に書け、深い階層の探索が速くなります。

もうひとつの graph の顔が、パイプラインの依存関係です。Apache Airflow の DAG（Directed Acyclic Graph、有向非巡回グラフ）は、タスクをノード、依存をエッジとするグラフです。「非巡回」であることが重要で、循環があるとどのタスクからも始められません。AWS Step Functions の状態機械も、実質的には状態の遷移グラフです。

アルゴリズム側では、次の考え方が効きます。ソートとマージは大量データの結合の基礎で、Spark の sort-merge join がこれにあたります。ハッシュ結合は片方が小さいときに速く、broadcast join はその極端な形です。二分探索の考え方は、Lambda の event source mapping にある「関数エラー時にバッチを二分して失敗レコードを絞り込む（bisect on function error）」設定に現れます。集合演算（重複排除、差分）は、増分処理でどのレコードが新しいかを判定する処理そのものです。

試験でここが問われるときは、実装の詳細ではなく「どの構造がこの要件に合うか」という形になります。多対多の関係を何段も辿るなら graph、階層と範囲検索なら tree、キー一発の参照なら key/value、列単位の集計なら列指向、という対応で切ります。「顧客同士の紹介関係を5段先まで辿って影響度を出したい」という設定に対して、リレーショナルの自己結合を5回重ねる肢より Amazon Neptune を使う肢が適切、という形です。

1-5 章末の整理と誤答肢の切り方 [1.1/1.2/1.3/1.4]

ここまでの内容を、試験本番で使える形に畳み直します。公式の Skills は英語で書かれているため、以降は英語の言い回しをそのまま並べて、日本語の判断基準と対応させます。

1-5-1 取り込み（1.1）の要点一覧 [1.1]

Skill の言い回し	判断の核	落とす肢
Read data from streaming sources	遅延・順序・保持期間で選ぶ。Kinesis / MSK / DynamoDB Streams / DMS / Glue / Redshift	Kafka 互換が要件なのに Kinesis Data Streams を選ぶ肢
Read data from batch sources	データの居場所と1回あたりの量で選ぶ。S3 / Glue / EMR / DMS / Redshift / Lambda / AppFlow	500 GB の変換に Lambda（900秒上限）を当てる肢
Implement appropriate configuration options for batch ingestion	増分か全量か、ワーカーと DPU、実行クラス、再試行とタイムアウト、同時実行	増分要件に job bookmarks を使わず毎回全量を読む肢
Consume data APIs	認証情報は Secrets Manager、ページング、backoff、Redshift Data API	100 MB を Lambda の同期レスポンス（6 MB 上限）で返す肢
Set up schedulers by using Amazon EventBridge, Apache Airflow, or time-based schedules for jobs and crawlers	依存関係が要るかどうか。schedulers は起動役、依存は Step Functions か MWAA	多段の依存に cron の schedulers を三つ並べる肢

Skill の言い回し	判断の核	落とす肢
Set up event triggers	S3 Event Notifications と EventBridge。at least once、数秒〜1分以上	「必ず1回だけ届く」と書いてある肢、同一バケットへ書き戻してループする肢
Call a Lambda function from Kinesis	event source mapping、batching window（最大5分）、ParallelizationFactor（1〜10）、6 MB のバッチ上限	冪等性を考慮せず「重複は起きない」とする肢
Create allowlists for IP addresses to allow connections to data sources	送信元を固定する。VPC 接続 + NAT Gateway + Elastic IP、security groups、PrivateLink	VPC 非接続の Lambda で IP addresses の allowlists に応える肢
Implement throttling and overcoming rate limits	DynamoDB はパーティション設計、RDS は接続数、Kinesis はシャード数	RDS の接続枯渇に Lambda を無制限にスケールさせる肢
Manage fan-in and fan-out for streaming data distribution	読み手の数と遅延要件。enhanced fan-out は最大20（On-demand Advantage は50）	読み手5つで遅延要件が厳しいのに shared throughput のままにする肢
Describe replayability of data ingestion pipelines	保持期間と開始位置の指定手段。生データを S3 に残す	Amazon Data Firehose を任意時点の replay の道具として説明する肢
Define stateful and stateless data transactions	過去のレコードを要るかどうか。窓・重複排除・セッションは stateful	「stateless だから順序が保証される」とする肢

allowlists と IP addresses について補足します。SaaS 側が接続元の IP addresses の登録を求めるとき、AWS 側で固定できるのは NAT Gateway に紐づく Elastic IP だけです。したがって順序は、(1) Lambda や AWS Glue を

VPC のプライベートサブネットに接続する、(2) そのサブネットのルートを NAT Gateway に向ける、(3) NAT Gateway の Elastic IP を相手に伝えて allowlists に登録してもらう、となります。逆向きの接続（相手から AWS へ入ってくる）なら、security groups の受信ルールで送信元 IP addresses を絞ります。

schedulers と event triggers の対比も押さえます。schedulers は「時刻が来たから動かす」、event triggers は「何かが起きたから動かす」です。データが届く時刻が読めるなら schedulers、届くタイミングが不定なら event triggers を選びます。両方を組み合わせて、event triggers で即時処理しつつ、schedulers で日次の取りこぼし確認を回す構成もよく使われます。

1-5-2 変換（1.2）の要点一覧 [1.2]

Skill の言い回し	判断の核	落とす肢
Optimize container usage for performance needs	container usage の最適化は vCPU とメモリの割り当て、EC2 と Fargate、イメージサイズ、スケーリング	要件が無いのに Amazon EKS を持ち込む肢
Connect to different data sources（JDBC、ODBC）	Glue connections、VPC とセキュリティグループ、Secrets Manager	差分要件に毎回 full load で読み直す肢
Integrate data from multiple sources	形式・粒度・キー・時刻の四つを揃える。Data Catalog が土台	結合のたびに federated query で数十GBをスキャンする肢
Optimize costs while processing data	読む量 → 形式 → コンピュート → ファイルサイズ → 可視化の順	Amazon EMR の core nodes を Spot にする肢

Skill の言い回し	判断の核	落とす肢
Implement data transformation services based on requirements	requirements（規模・居場所・運用の重さ）から transformation services を選ぶ。EMR / Glue / Lambda / Redshift	Redshift 内で完結する集計に Glue の往復を挟む肢
Transform data between formats	.csv から Apache Parquet へ。列指向・圧縮・述語プッシュダウン。CTAS は100パーティション	分割できない gzip の .csv のまま並列読み取りを期待する肢
Troubleshoot and debug common transformation failures and performance issues	out of memory、skew、小さいファイル、bookmarks の誤動作。Continuous logging と Spark UI で debug する	遅い原因を測らずにワーカー数だけ増やす肢
Create data APIs to make data available to other systems by using AWS services	API Gateway + Lambda、Redshift Data API、Athena。事前集計して読ませる	参照のたびに数十GBをスキャンする API を作る肢
Define volume, velocity, and variety of data	structured data / semi-structured data / unstructured data と、量・速度・多様性	非構造化データをそのまま SQL で集計できるとする肢
Integrate large language models (LLMs) for data processing	非構造化テキストの構造化・分類・要約・embeddings。batch inference に寄せる	数値集計を LLM にやらせる肢

transformation services の選択は、この試験で最も反復して問われます。決め手を一行にすると、「データが今どこにあるか」「1回あたりどれだけ処理するか」「どこまで運用を持てるか」の三つです。この三つが揃えば、Amazon

EMR、AWS Glue、Lambda、Amazon Redshift のどれかに自動的に落ちます。

container usage について一点補足します。AWS Batch はジョブキューとコンピュート環境を管理し、Amazon ECS や Amazon EKS の上でジョブを実行します。大量のバッチジョブに優先度とキューイングが要るときは、ECS や EKS を直接叩くより AWS Batch を挟むほうが素直です。

1-5-3 オーケストレーション（1.3）の要点一覧 [1.3]

Skill の言い回し	判断の核	落とす肢
Use orchestration services to build workflows for data ETL pipelines	orchestration services の表現力の階段。Lambda → EventBridge → Glue workflows → Step Functions → MWAA	Glue job の完了待ちが要るのに Express Workflows を選ぶ肢
Build data pipelines for performance, availability, scalability, resiliency, and fault tolerance	五つの性質それぞれに具体策を当てる。特に「途中から再開できる粒度」	1日分を一括処理し、失敗したら最初からやり直す設計
Implement and maintain serverless workflows	Step Functions の Retry と Catch、Distributed Map、reserved concurrency、job run queuing	Express Workflows でロギングを設定せず履歴を追おうとする肢
Use notification services to send alerts	notification services は SNS（push・複数へ）と SQS（pull・バッファ）。CloudWatch アラーム経由で alerts を send する	S3 Event Notifications を SQS FIFO キューへ直接送る肢

orchestration services を選ぶときの決定木を、一つの流れにまとめます。第一に、処理が1段だけなら Lambda か EventBridge で足ります。第二に、Glue の crawlers と jobs だけをつなぐなら AWS Glue workflows です。第三に、複数の AWS サービスをまたぎ、分岐と再試行が要るなら AWS Step Functions です。第四に、DAG による複雑な依存、バックフィル、既存の Apache Airflow 資産があるなら Amazon MWAA です。

pipelines を build するときに最後まで残る論点が、失敗の受け止め方です。一時的な失敗（スロットリング、ネットワークの瞬断）は Retry で吸収し、恒久的な失敗（データ形式の不正、権限不足）だけを Catch から notification services に回して alerts を send します。この二段構えができていないと、アラートが多すぎて誰も見なくなるか、少なすぎて障害に気づけないかのどちらかになります。

1-5-4 プログラミングの概念（1.4）の要点一覧 [1.4]

Skill の言い回し	判断の核	落とす肢
Optimize code to reduce runtime for data ingestion and transformation	runtime を縮める順序は、読む量 → 形式 → ファイルサイズ → 並列度 → 偏り → 往復 → shuffle	メモリを減らせば必ず安くなるとする肢
Configure Lambda functions to meet concurrency and performance needs	reserved concurrency（上限かつ下限、追加料金なし）と provisioned concurrency（コールドスタート対策、追加料金あり）	予約可能量が「未予約分マイナス100」である点を無視する肢

Skill の言い回し	判断の核	落とす肢
Use programming languages and frameworks for data engineering	data engineering の programming languages と frameworks は Python、SQL、Scala、R、Java、Bash、PowerShell	分散処理が不要な小さな処理に PySpark ジョブを立てる肢
Use software engineering best practices for data engineering	software engineering の best practices は version control、testing、logging、monitoring の四本	本番だけコンソールで微調整する肢
Use Infrastructure as Code (IaC) to deploy data engineering solutions	IaC で data engineering の solutions を再現可能にする。CloudFormation と AWS CDK	環境ごとに別々の手順書を持つ肢
Use AWS SAM to package and deploy serverless data pipelines	AWS SAM で serverless の pipelines を package して deploy する。Lambda functions、Step Functions、DynamoDB tables を一式で	3つのリソースを別々のテンプレートに分ける肢
Use and mount storage volumes from within Lambda functions	Lambda から storage volumes を mount する。/tmp（512〜10,240 MB）、Amazon EFS、Amazon S3 Files。EFS と S3 Files は排他	50 GB を /tmp に展開する肢、/tmp を関数間で共有できるとする肢
Use infrastructure as code (IaC) for repeatable resource deployment	同じ infrastructure を三環境に複製する。AWS CloudFormation と AWS CDK	手作業で作った構成の差を「環境差」で片付ける肢

Skill の言い回し	判断の核	落とす肢
Describe continuous integration and continuous delivery (CI/CD)	continuous integration でテストし、continuous delivery で pipelines をデプロイする。CodePipeline / CodeBuild / CodeDeploy	書き込んだデータもコードと同じようにロールバックできるとする肢
Define distributed computing	分散の語彙（ノード、パーティション、タスク、shuffle、skew）と EMR の三ノード	core instance group を Spot にしても即座に部分起動すると考える肢
Describe data structures and algorithms	tree data structures（階層・範囲検索）と graph data structures（多対多・経路）。algorithms はソート・マージ・ハッシュ結合・二分探索	5段の関係探索を自己結合5回で組む肢

AWS SAM の位置づけをもう一度整理します。AWS SAM template は AWS CloudFormation の拡張であり、AWS SAM CLI は開発・ローカルテスト・デプロイを助けるコマンドラインツールです。sam build でアーティファクトを package し、sam deploy で CloudFormation スタックとして deploy します。sam local invoke で Lambda functions をローカルで実行し、sam sync でローカルの変更をクラウドへ継続的に同期します。定義できる serverless の要素には Lambda functions、Amazon API Gateway の API、Amazon DynamoDB tables が含まれ、AWS SAM connectors を使えばリソース間の権限が必要な IAM 権限へ変換されます。

storage volumes の mount について、選択の順序を最後に固定します。1回の実行内で完結する作業ファイルなら /tmp。複数の実行や複数の関数で共有する永続データなら Amazon EFS を mount する。S3 のオブジェクトをファ

イルとして扱いたいなら Amazon S3 Files を mount する。ただし Amazon EFS と Amazon S3 Files は同時に使えず、切り替えるには既存の設定を外す必要があります。Amazon EFS を使うには VPC 接続が必要で、ENI のクォータ（VPC あたり既定500、Amazon EFS など他サービスと共有）が効いてきます。

continuous integration と continuous delivery を data pipelines に適用するとき、通常のアプリケーションと違うのは「データは戻せない」という一点です。だから implementation・testing・deployment のうち testing の段で、マスキングしたサブセットに対する統合テストとデータ品質チェック（AWS Glue Data Quality の DQDL ルール）を必ず通します。そして出力先を日付やバージョンでパーティション分けし、誤った版を捨てて作り直せる形にしておきます。これが data pipelines における実質的なロールバック手段です。

最後に、この章全体を貫く判断の型をもう一度置きます。要件を読んだら、遅延・件数・順序・再実行・コストの五つに分解する。分解できたら、取り込みは 1-1 の表、変換は 1-2-5 の表、オーケストレーションは 1-3-1 の表に当てる。数字（900秒、6 MB、1 MB/秒、2 MB/秒、24時間、365日、100パーティション、1,000 同時実行）で切れる肢は数字で切り、残った肢は運用の重さで切る。この順序を守れば、ドメイン1の34%はほぼ取り切れます。

サンプルはここまでです

続き(第2章〜巻末)は、AWS DEA-C01 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。

AWS DEA-C01 学習用テキスト(無料サンプル)

版

2026年7月版(AWS 公式 Exam Guide（DEA-C01 v1.1・2026年9月6日取得）準拠)

発行

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

お問い合わせ

info@shikaku-doujou.com