IT資格道場

GCP PCD 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)

Google Cloud 公式 Professional Cloud Developer Exam Guide(2026-09-07 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

3点セットの使い方

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、Google Cloud が実施する **Professional Cloud Developer 認定** の合格を目的に作られています。

Professional 級の試験が問うのは、個々の製品の説明ではありません。**要件と制約を読み取り、複数の選択肢の中から最も適切な設計・運用・トラブルシューティングの判断を選べること**です。本書は Exam guide の各セクションに沿って、「どの場面でどれを選ぶか・なぜ他は不適か」を本文に書き込んでいます。

本書は Google Cloud 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	Google Cloud Certified – Professional Cloud Developer
実施	Google Cloud（テストセンター、またはオンライン監督試験）
問題数	**50〜60問**（公式の案内）
試験時間	**120分**
出題形式	**択一（multiple choice）、複数選択（multiple select）**
合格ライン	非公表
受験言語	英語・日本語（日本語提供あり）

出題数・試験時間・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、Google Cloud 公式サイトで最新の情報をご確認ください。

出題範囲の全体像 —— 4つのセクションと本書の章

セクション	分野	採点対象に占める割合（公式）	本書
1	高いスケーラビリティ・セキュリティ・信頼性を備えたクラウドネイティブアプリケーションの設計	**32%**	第1章
2	アプリケーションの構築とテスト	**23%**	第2章
3	デプロイに向けたクラウドネイティブアプリケーションの構成	**24%**	第3章
4	アプリケーションとGoogle Cloud サービスの統合	**21%**	第4章

本書の章の並びは、Exam guide 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、Exam guide のセクション番号です。Google Cloud はセクションごとの割合だけを公表しており、細目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てください。本文では、製品名・機能名・用語を英語表記のまま書いています。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。

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: 要件は何を最優先しているか	可用性／性能／費用／セキュリティ／運用負荷	同じ「移行したい」でも、何を最優先に置くかで正解の製品と構成が変わります
問い2: マネージドで足りるか、自分で管理するか	サーバーレス／マネージド／セルフマネージド	「運用したくない」ならマネージド、「細かな制御が要る」ならセルフマネージド、と制約が構成を決めます
問い3: それは Google の責任か、自分の責任か	責任共有モデル	どの層まで自分で守り・運用するかは全章で一貫して効きます。迷ったらここへ戻ってください

第1章 高いスケーラビリティ・セキュリティ・信頼性を備えたクラウドネイティブアプリケーションの設計 (ドメイン1・採点対象の 32%)

この章は Professional Cloud Developer で最も配点の大きい領域です。問われるのは「Google Cloud のサービスの名前を知っているか」ではなく、要件が並んだ場面を読んで「どれを選び、なぜ他を捨てるか」を判断できるかどうかです。したがって本章では、各サービスの一言定義に続けて、必ず「似ているどれと迷うか」「どの一語が決め手になるか」「他の選択肢がなぜ誤りか」を書きます。この書き方は本章以降も同じです。

1-1 高性能なアプリケーションと API の設計 [1.1]

1-1-1 ユースケースと要件に基づくプラットフォームの選択 [1.1]

Google Cloud でアプリケーションを動かす土台は、大きく Compute Engine、Google Kubernetes Engine(GKE)、Cloud Run の3つです。試験では「この要件ならどれか」を一行で決められる必要があります。まず性格を押さえます。

プラットフォーム	抽象度	課金の単位	ゼロスケール	得意な場面
Compute Engine	仮想マシン (IaaS)	VM の起動時間 (秒単位・1分下限)	しない(インスタンスを止めれば止まる)	OS・カーネルへの依存、既存のライセンス、GPU/TPU の細かい制御、リフト&シフト

プラットフォーム	抽象度	課金の単位	ゼロスケール	得意な場面
Google Kubernetes Engine(GKE)	コンテナオーケストレーション	ノード(Standard)またはリクエストした Pod リソース(Autopilot)	しない(ノードは最小0にできるが制御面は常時)	多数のマイクロサービス、複雑なネットワークポリシー、StatefulSet、サービスメッシュ、独自のスケジューリング
Cloud Run	コンテナのサーバーレス実行	リクエスト処理中の vCPU/メモリ(100 ミリ秒単位)+リクエスト数	する(最小インスタンス 0 が既定)	HTTP/gRPC の API、イベント駆動の処理、トラフィックが不規則、運用担当を置きたくない

判断の順序は次のとおりです。試験の問題文は、この順序のどこかに引っかかる一語を必ず含んでいます。

1. OS レベルの制御、特権が要るドライバ、カーネルモジュール、既存のライセンス持ち込み(BYOL)が要件にあるか。あるなら Compute Engine です。コンテナ化できない、と問題文が明言している場合も同じです。
2. コンテナ化されているとして、Kubernetes 固有の機能(カスタムリソース定義、Operator、DaemonSet、Pod 間の細かいアフィニティ、Kubernetes Network Policy、サービスメッシュのサイドカー注入)が要るか。あるなら GKE です。
3. どちらも要らず、HTTP・gRPC・イベントで駆動されるステートレスなワークロードなら Cloud Run です。運用の手離れとゼロスケールが決め手になります。

よく出る取り違えを3つ挙げます。第一に「Kubernetes の経験がないチーム」「クラスタ管理の負担を減らしたい」という記述は、GKE を捨てて Cloud Run を選ぶ合図です。第二に「Kubernetes は使いたいがノードの管理はしたくない」という記述は Cloud Run ではなく GKE Autopilot です。Autopilot はノードのプロビジョニング、アップグレード、ビンパッキングを Google が受け持ち、課金は Pod が要求した vCPU・メモリ・エフェメラルストレージに対して発生します。第三に「常時一定のトラフィックがあり、コストを最小化したい」場合、リクエストが途切れない前提なら Compute Engine の確約利用割引(CUD)や Spot VM のほうが安くなることがあります。Cloud Run が常に安いわけではありません。

Cloud Run には2つの実行形態があります。サービス(Cloud Run services)は HTTP リクエストや gRPC 呼び出しで駆動される常駐型で、URL を持ちます。ジョブ(Cloud Run jobs)はリクエストを受けず、実行して終了するバッチ型で、タスク数と並列度を指定して走らせます。「日次のデータ変換を実行して終わる処理」はサービスではなくジョブです。ジョブは Cloud Scheduler から起動するのが定石です。

Compute Engine 側では、マネージドインスタンスグループ(MIG)が中心です。インスタンステンプレートから同一構成の VM を複製し、オートスケーラーが CPU 使用率、ロードバランサの処理能力、Cloud Monitoring のカスタム指標、あるいはスケジュールに基づいて台数を増減します。自動修復(autohealing)はヘルスチェックに失敗した VM を作り直します。ここは「VM だがスケールと自己修復が要る」という要件に対する答えです。

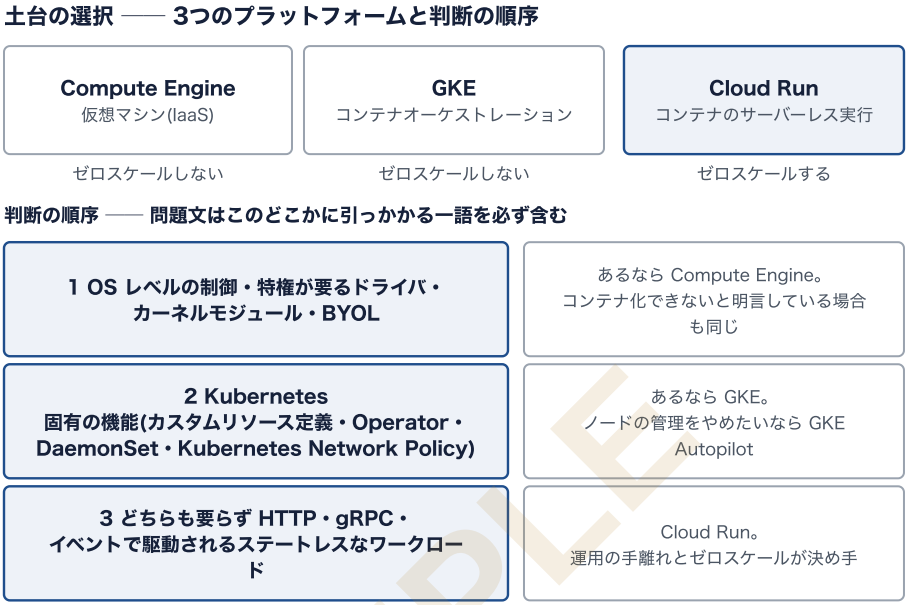


図 1-1　土台の選択 —— 3つのプラットフォームと判断の順序

1-1-2 Cloud Run と GKE へのアプリケーションコンテナの構築・リファクタリング・デプロイ [1.1]

Building, refactoring, and deploying application containers to Cloud Run and GKE は、この objective の実務の中心です。まず構築です。コンテナイメージは Cloud Build で作り、Artifact Registry に保存します。ソースからの最短経路は次の2つです。

- `gcloud run deploy --source .` : ソースを Cloud Build に送り、Dockerfile があればそれを、なければ Google Cloud Buildpacks で自動的にイメー

ジを組み立て、Artifact Registry に置き、Cloud Run にデプロイします。Dockerfile を書かずに済むのが利点です。

- `gcloud builds submit --tag <region>-docker.pkg.dev/<project>/<repo>/<image>:<tag>`：イメージだけを作って Artifact Registry に置きます。GKE へ配る場合はこちらを使い、マニフェストの image を差し替えます。

リファクタリングの観点で問われるのは、既存のアプリケーションをそのままコンテナにしても Cloud Run では動かない部分をどう直すか、です。Cloud Run のコンテナが満たすべき契約(container contract)は次のとおりです。

要件	内容	守らないと起きること
ポート	環境変数 PORT(既定 8080)で待ち受ける	起動チェックに失敗しリビジョンがデプロイされない
バインドアドレス	0.0.0.0 で待ち受ける(localhost では不可)	外部からの接続が届かない
ステートレス	ローカルディスクはインメモリ(tmpfs)扱いで、インスタンス終了時に消える	書き込んだファイルが消える、メモリ課金が増える
状態の外出し	セッション、アップロード、キャッシュは Cloud Storage、Firestore、Memorystore へ	インスタンス間で不整合
起動時間	起動が速いほどコールドスタートの影響が小さい	レイテンシのばらつき

サーバー上のファイルにセッションを書いていた従来型のアプリケーションを Cloud Run に載せる場合、リファクタリングの正解はセッションストアを Memorystore for Redis に外出しすることです。ローカルディスクに書き続け

る、あるいはセッションアフィニティだけで凌ぐ、という選択肢は誤りです。セッションアフィニティはベストエフォートであり、インスタンスが入れ替われば状態は失われます。

GKE へのデプロイでは、Deployment(ステートレス)、StatefulSet(安定したネットワーク ID と永続ボリュームが要る)、DaemonSet(全ノードに1つずつ)、Job/CronJob(実行して終わる)の使い分けが基本です。コンテナ化した Web アプリケーションを配るなら Deployment に Service(ClusterIP)を付け、外部公開は Ingress または Gateway API 経由でロードバランサに繋ぎます。

Cloud Run と GKE のどちらに置くかで迷う場面の切り分けを表にします。

問題文の記述	選ぶべきもの	理由
「リクエストが来ない夜間はコストをゼロにしたい」	Cloud Run	最小インスタンス0でゼロスケールできる
「Pod 間の通信をラベル単位で制限したい」	GKE	Kubernetes Network Policy が要る
「サイドカーで mTLS を強制したい」	GKE(Cloud Service Mesh)	Cloud Run にもサイドカーはあるがメッシュ機能は GKE が本命
「チームに Kubernetes の運用経験がない」	Cloud Run	学習・運用コストが決め手
「複数のバッチ Pod を優先度付きでスケジュールしたい」	GKE	スケジューラの制御が要る
「1つのコンテナを HTTPS で公開するだけ」	Cloud Run	最小の構成で足りる

サーバー上のファイルにセッションを書いていた従来型のアプリケーションは、
セッションストアを Memorystore for Redis に外出しするのが正解。
セッションアフィニティはベストエフォートであり、
インスタンスが入れ替われば状態は失われる。セッションデータの保存手段には使わない。

図 1-2　Cloud Run のコンテナ契約 —— 守らないと何が起きるか

1-1-3 Google Cloud サービスの地理的分散(レイテンシ・リージョナル・ゾーナル) [1.1]

Understanding how Google Cloud services are geographically distributed は、可用性設計の土台です。用語を正確に区別します。

区分	意味	例	障害時の挙動
ゾーナル(zonal)	1つのゾーン内に閉じる	Compute Engine の VM、ゾーナル永続ディスク、ゾーナル GKE クラスタ	ゾーン障害でそのリソースは失われる

区分	意味	例	障害時の挙動
リージョナル (regional)	1リージョン内の複数ゾーンに複製される	リージョナル MIG、リージョナル永続ディスク、Cloud Run、リージョナル GKE クラスタ、リージョナル Cloud Storage バケット	ゾーン障害を吸収する
マルチリージョナル/グローバル	複数リージョンにまたがる	Spanner のマルチリージョン構成、マルチリージョンの Cloud Storage、グローバル外部アプリケーションロードバランサ、Cloud DNS	リージョン障害を吸収しうる

レイテンシ(latency)の設計原則は単純です。ユーザーに近いリージョンで受け、データはできるだけ計算の近くに置きます。同一リージョン内のゾーン間往復はおおむね1ミリ秒未満から数ミリ秒、大陸をまたぐ往復は数十ミリ秒から100ミリ秒超になります。したがって、アプリケーションとデータベースを別のリージョンに置くと、1リクエストで複数回クエリを投げる設計では致命的に遅くなります。試験では「レイテンシが高い原因はどれか」という形で、アプリケーションとデータストアのリージョン不一致が正解になる問題がよく出ます。

分散の設計は次の順に考えます。まず単一ゾーンでよいか(開発環境やバッチならよい場合がある)。次にゾーン障害に耐える必要があるか(リージョナル構成にする)。最後にリージョン障害に耐える必要があるか(複数リージョンに配置し、グローバルロードバランサで振り分ける)。段階を飛ばして最初からマルチリージョンにするとコストとレイテンシ(強整合を保つ場合の書き込み遅延)が跳ね上がるため、要件にない限り誤答になります。

1-1-4 ロードバランサのユースケース [1.1]

Understanding the use cases for load balancers は毎回問われます。Google Cloud のロードバランサは、まず「グローバルか、リージョナルか」「外部か、内部か」「レイヤー7(アプリケーション)か、レイヤー4(ネットワーク)か」の3軸で分類します。

ロードバランサ	レイヤー	範囲	主な用途	決め手になる要件
グローバル外部アプリケーションロードバランサ	L7(HTTP/HTTPS)	グローバル(単一のエニーキャスト IP)	世界中のユーザーに Web/API を公開	世界中のユーザー、URL マップによるパス振り分け、Cloud CDN、Google Cloud Armor、複数リージョンのバックエンド
リージョン外部アプリケーションロードバランサ	L7	1リージョン	特定リージョンでの公開、データ所在地の制約	「このリージョン内に留める」
内部アプリケーションロードバランサ	L7	リージョン(クロスリージョン版もある)	VPC 内のマイクロサービス間の HTTP 振り分け	内部のみ、パスベースのルーティング
外部パススルーネットワークロードバランサ	L4	リージョン	TCP/UDP をそのまま通す、送信元 IP を保持	非 HTTP プロトコル、クライアント IP をそのまま見たい

ロードバランサ	レイヤー	範囲	主な用途	決め手になる要件
内部パススルーネットワークロードバランサ	L4	リージョン	VPC 内の TCP/UDP 分散、次ホップとしての利用	内部の非 HTTP、送信元 IP 保持
プロキシネットワークロードバランサ	L4(プロキシ)	グローバル/リージョン	TCP/SSL のプロキシ終端	TLS 終端したいが HTTP ではない

誤答の切り方を覚えます。「世界中のユーザー」「1つの IP アドレスで受けたい」「Cloud CDN を有効にしたい」「WAF ルールを適用したい」はグローバル外部アプリケーションロードバランサです。Cloud CDN と Google Cloud Armor はアプリケーションロードバランサ(および一部のプロキシ型)に紐づく機能で、パススルー型のネットワークロードバランサには付けられません。逆に「UDP を分散したい」「クライアントの実 IP をアプリケーションで見たい」ならパススルーネットワークロードバランサで、アプリケーションロードバランサは誤りです(アプリケーションロードバランサはプロキシなので送信元は Google のプロキシ IP になり、実クライアント IP は X-Forwarded-For ヘッダーで渡されます)。

Cloud Run と GKE のバックエンドは、サーバーレス NEG(ネットワークエンドポイントグループ)またはコンテナネイティブ負荷分散(GKE の場合は Pod IP を直接エンドポイントにする NEG)でロードバランサに接続します。GKE で「ノードを経由する二重ホップをなくしてレイテンシを下げたい」という要件が出たら、コンテナネイティブ負荷分散(NEG)が答えです。

図 1-3　ロードバランサ —— 3つの軸と決め手になる要件

1-1-5 高性能なコンテンツ配信のためのセッションアフィニティの有効化 [1.1]

Enabling session affinity for performant content delivery は、同じクライアントからのリクエストを同じバックエンドに送る設定です。目的は主に2つあります。バックエンド側のインメモリキャッシュを効かせて応答を速くすること、そしてバックエンドがローカルに持つ一時的な状態を再利用することです。

アプリケーションロードバランサのセッションアフィニティには次の種類があります。

種類	何を鍵にするか	使う場面
CLIENT_IP	クライアント IP アドレス	単純だが NAT 配下の多数ユーザーが同一バックエンドに偏る
GENERATED_COOKIE	ロードバランサが発行する Cookie(GCLB)	HTTP(S) で最も一般的。IP が変わっても維持される
HEADER_FIELD	指定した HTTP ヘッダーの値	マイクロサービスで独自のキーを使う
HTTP_COOKIE	アプリケーションが指定した Cookie	既存のセッション Cookie に合わせる
STRONG_COOKIE_AFFINITY	ロードバランサが発行する強い Cookie	分散の維持を厳密にしたい

Cloud Run にもセッションアフィニティの設定があり、`gcloud run services update <service> --session-affinity` で有効にします。ただし試験で強調されるのは、セッションアフィニティはベストエフォートであるという点です。インスタンスのスケールイン、リビジョンの入れ替え、バックエンドの異常検知が起きれば、同じクライアントが別のインスタンスに回されます。したがって、セッションアフィニティを「セッションデータの保存手段」として使う設計は誤りで、状態は Memorystore や Firestore などの外部ストアに置くのが正解です。セッションアフィニティはあくまで性能最適化(キャッシュヒット率の向上)として扱います。

なお、アフィニティを強く効かせるほど負荷の分散は偏ります。バックエンドの分散度合いとキャッシュ効率のトレードオフである、という言い方が問題文に出たら、それはセッションアフィニティの話です。

1-1-6 キャッシュソリューションの実装(Memorystore) [1.1]

Implementing caching solutions の中心は Memorystore です。Memorystore は Redis、Valkey、Memcached のマネージド版を提供します。

製品	特徴	向く用途
Memorystore for Redis / Redis Cluster	データ構造(リスト、ソート済みセット、ハッシュ)、永続化、レプリケーション、Pub/Sub	セッションストア、レート制限のカウンタ、リーダーボード、ジョブキュー
Memorystore for Valkey	Redis 互換のオープンソース分岐、クラスタ構成	Redis と同じ用途で、クラスタスケールを重視する場合
Memorystore for Memcached	単純なキー・バリューのみ、水平にスケール	純粋な読み取りキャッシュ、大量の単純キャッシュ

選択の決め手は「データ構造や永続化が要るか」です。要るなら Redis、単純なキャッシュだけなら Memcached でも足ります。セッションストアの要件が出たら Redis を選びます。

接続の設計も出題されます。Memorystore は VPC 内のプライベート IP を持ちます。したがって Cloud Run から接続するには、Direct VPC egress(推奨)または Serverless VPC Access コネクタが必要です。「Cloud Run から Memorystore に繋がらない」というトラブルシュートの答えは、多くの場合この VPC 接続の欠落です。GKE からは同じ VPC 内であればそのまま接続できます。

キャッシュ設計の定石も押さえます。キャッシュアサイド(lazy loading)は、まずキャッシュを見て、なければデータベースから読んでキャッシュに書き戻す方式で、最も一般的です。ライトスルーは書き込み時にキャッシュも更新するため整合性は高いがレイテンシが増えます。TTL を設定して古いデータを自然に追い出すこと、キャッシュが一斉に失効して背後のデータベースが飽和す

る事象(キャッシュスタンピード)を避けるために TTL をずらすこと、が実装上の勘所です。

Memorystore 以外のキャッシュ層も区別します。Cloud CDN は静的コンテンツやキャッシュ可能な HTTP レスポンスをエッジに置く仕組みで、アプリケーションロードバランサのバックエンドサービス/バックエンドバケットで有効にします。「画像や JavaScript の配信を世界中で速くしたい」なら Memorystore ではなく Cloud CDN です。この取り違えは頻出です。

1-1-7 API の作成とデプロイ(HTTP REST・gRPC) [1.1]

Creating and deploying APIs では、HTTP REST と gRPC(Remote Procedure Call)の使い分けが問われます。

観点	HTTP REST(JSON)	gRPC
データ形式	JSON(テキスト、人が読める)	Protocol Buffers(バイナリ、小さく速い)
契約の定義	OpenAPI 仕様	.proto ファイル
通信	HTTP/1.1 も可、リクエスト・レスポンス	HTTP/2 前提、単方向・双方向ストリーミング可
ブラウザからの直接呼び出し	容易	制限がある(gRPC-Web が必要)
向く場面	外部公開 API、パブリックな Web/モバイル向け	マイクロサービス間の内部通信、低レイテンシ・高スループット、ストリーミング

選択の決め手は「誰が呼ぶか」です。外部の第三者やブラウザが直接呼ぶなら REST、社内のサービス同士で低レイテンシが要るなら gRPC です。「双方向ストリーミングが必要」という一語が出たら gRPC で確定します。

Cloud Run は HTTP/1.1、HTTP/2、gRPC(単項およびストリーミング)、WebSocket を受け付けます。gRPC を使う場合、Cloud Run では HTTP/2 をエンドツーエンドで有効にする必要があり、`gcloud run deploy --use-http2` を指定します。これを忘れて gRPC が通らない、というのが典型的なトラブルシュート問題です。GKE で gRPC をロードバランサ越しに受ける場合は、バックエンドサービスのプロトコルを HTTP2 にします。

API のバージョニングは、REST ならパスに版を含める(/v1/、/v2/)のが定石で、破壊的変更のときだけ版を上げます。gRPC では .proto のフィールド番号を再利用しない、フィールドを削除せず reserved にする、といった後方互換の規律が答えになります。

1-1-8 アプリケーションのレート制限・認証・オブザーバビリティ (Apigee・Cloud API Gateway) [1.1]

Using application rate limiting, authentication, and observability は、API を「作る」ではなく「管理する」層の話です。

製品	位置づけ	主な機能	向く場面
Apigee	フルスタックの API 管理プラットフォーム	レート制限(Quota / Spike Arrest)、API キー、OAuth 2.0、JWT 検証、リクエスト/レスポンス変換、マネタイズ、開発者ポータル、詳細な分析	外部のパートナーや顧客に API を提供し、収益化・SLA・ポータルまで要る

製品	位置づけ	主な機能	向く場面
Cloud API Gateway	軽量なマネージド API ゲートウェイ	OpenAPI で定義、API キー、JWT/Firebase/Auth0 による認証、Cloud Run/Cloud Functions/App Engine のバックエンドを保護	サーバーレスのバックエンドに単純なゲートウェイを付けたい
Cloud Endpoints	ESP/ESPv2 プロキシによる API 管理	OpenAPI または gRPC 設定、API キー、認証、モニタリング	既存の構成、GKE 上での gRPC 管理

判断は要件の重さで決まります。開発者ポータル、マネタイズ、複雑なポリシーチェーン、多段のプロキシ、といった語が出たら Apigee です。単に Cloud Run の前に API キーと JWT 検証を置きたいだけなら Cloud API Gateway で足り、Apigee はオーバースペック(コスト過大)として誤答になります。

レート制限(rate limiting)の実装先は3層あります。第一に API 管理層(Apigee の Quota ポリシー、Spike Arrest ポリシー、API Gateway の割り当て)。第二にエッジ層(Google Cloud Armor のレート制限ルールで、IP 単位のスロットリングや DDoS 緩和を行う)。第三にアプリケーション層(Memorystore for Redis にカウンタを置くトークンバケット)。「特定の IP からの過剰なリクエストを遮断したい」は Cloud Armor、「顧客ごとの契約プランに応じて月間呼び出し回数を制限したい」は Apigee の Quota、と切り分けます。

Spike Arrest と Quota の違いも押さえます。Quota は一定期間の総呼び出し数の上限(契約に基づく制限)で、Spike Arrest は瞬間的な流量の平滑化(バ

ックエンド保護)です。「バックエンドが急激なトラフィックの山で落ちるのを防ぎたい」は Spike Arrest です。

オブザーバビリティ側では、Apigee が API プロキシ単位のレイテンシ・エラー率・トラフィック分析を持ち、API Gateway と Cloud Endpoints は Cloud Monitoring と Cloud Logging に指標とログを送ります。API の呼び出し回数を消費者ごとに可視化したいなら API キーやデベロッパーアプリ単位で集計できる Apigee が答えになります。

1-1-9 非同期・イベント駆動によるアプリケーションの統合 (Eventarc・Pub/Sub) [1.1]

Integrating applications using asynchronous or event-driven approaches は本試験の主要テーマです。まず Pub/Sub の基礎を固めます。

Pub/Sub はグローバルなメッセージングサービスで、パブリッシャーがトピックにメッセージを publish し、サブスクライバーがサブスクリプション経由で受け取ります。配信の性質は次のとおりです。

概念	内容	設計上の含意
配信保証	少なくとも1回(at-least-once)が既定	受信側は冪等に作る。message_id や業務キーで重複排除する
正確に1回	リージョナルトピックで exactly-once delivery を有効にできる	重複を避けたいが、スループットと引き換え
順序	既定は順序保証なし。ordering key を付け、サブスクリプションで順序指定を有効にすると同一キー内で順序保証	順序が要るなら ordering key。ただし同一リージョンでのパブリッシュが前提

概念	内容	設計上の含意
プル型サブスクリプション	受信側が pull する。ストリーミングプルが一般的	受信側がスループットを制御できる。GKE のワーカーなどに向く
プッシュ型サブスクリプション	Pub/Sub が HTTP エンドポイントに POST する	Cloud Run に向く。エンドポイントは認証を要求できる
ack 期限	既定10秒、最大600秒。期限内に ack しないと再配信	長い処理では ack 期限を延長するか、modifyAckDeadline を使う
デッドレタートピック	最大配信回数を超えたメッセージを別トピックへ	毒メッセージで無限ループするのを防ぐ
再試行ポリシー	即時再試行または指数バックオフ	バックエンドの過負荷を避けるなら指数バックオフ
メッセージ保持	既定7日(最大31日)、シークで過去に巻き戻せる	障害後の再処理に使う

Pub/Sub の代表的な誤答パターンを挙げます。「重複したメッセージが処理されている」の対策として順序指定(ordering key)を選ぶのは誤りです。順序は重複を防ぎません。正解は受信側の冪等化、または exactly-once delivery の有効化です。「処理が10分かかるのに何度も同じメッセージが来る」は ack 期限の不足で、ack 期限の延長が正解です。サブスクリプションを作り直す、メッセージ保持期間を延ばす、は誤りです。

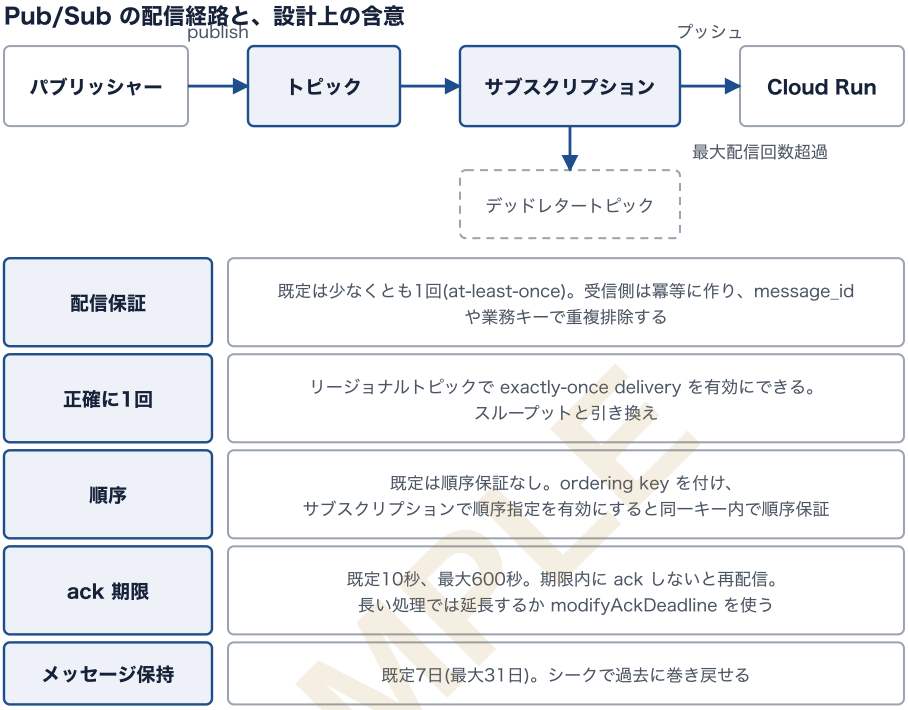


図 1-4　Pub/Sub の配信経路と、設計上の含意

次に Eventarc です。Eventarc は Google Cloud のイベントを標準の CloudEvents 形式で受け取り、Cloud Run、GKE、Workflows などのターゲットに配送するルーター役です。イベント源は3種類あります。

1. Cloud Audit Logs 経由 : Google Cloud の 100 を超えるサービスの操作(たとえば Cloud Storage へのオブジェクト作成、BigQuery のジョブ完了)をイベントにする。
2. Pub/Sub 経由 : 既存のトピックに来たメッセージをそのままターゲットに配送する。

3. 直接イベント : Cloud Storage などが直接送るイベント。

Eventarc と Pub/Sub の使い分けは頻出です。「Google Cloud のサービスで起きた出来事(オブジェクトがアップロードされた、テーブルが更新された)を引き金にしたい」なら Eventarc、「自分のアプリケーション同士でメッセージをやり取りしたい」なら Pub/Sub です。Eventarc は内部で Pub/Sub を使いますが、フィルタ(イベントタイプ、リソース名)とCloudEvents 形式への正規化を肩代わりしてくれる点が価値です。

非同期にする設計理由も本文で押さえます。呼び出し側と処理側を疎結合にする(片方が落ちてもメッセージが残る)、負荷の山を吸収する(バッファリング)、複数の下流に同報する(ファンアウト)、処理時間の長い仕事を切り離してレスポンスを速く返す、の4つです。同期呼び出しでチェーンを組むと、末端の遅延が全体のレイテンシに積み上がり、1か所の障害が全体を止めます。試験で「サービス間の結合を弱めたい」「トラフィックの急増でバックエンドが落ちる」と書かれていたら、答えは Pub/Sub を挟むことです。

1-1-10 ワークロードのリソース要件の定義 [1.1]

Defining resource requirements for workloads は、プラットフォームごとに設定の場所が違います。

Cloud Run では、リビジョン単位で CPU(0.08 vCPU から最大8 vCPU)、メモリ(最大32 GiB)、同時実行数(concurrency、既定80、最大1000)、リクエストタイムアウト(既定300秒、最大3600秒)、最小インスタンス数、最大インスタンス数を設定します。ここで最重要なのが同時実行数です。1インスタンスが同時に扱うリクエスト数を上げるとインスタンス数が減ってコストが下がりますが、1リクエストあたりのメモリ・CPU を食う処理では OOM や遅延を招きます。逆に同時実行数を1にすると、リクエストごとに専有されるため安全ですがコス

トが上がります。「メモリ不足でコンテナが落ちる」なら、メモリを増やすか同時実行数を下げるのが正解です。

CPU 割り当てのモードも重要です。既定は「リクエスト処理中のみ CPU を割り当てる」で、レスポンスを返した後のバックグラウンド処理は CPU が絞られます。バックグラウンドで非同期処理を続けたい、あるいは起動時の初期化を安定させたい場合は、CPU を常時割り当てる(CPU always allocated)設定にします。「レスポンスを返した後に走らせている集計処理が途中で止まる」の答えがこれです。

GKE では Pod のコンテナごとに requests と limits を指定します。requests はスケジューラがノードを選ぶ基準であり、limits は上限です。CPU の limits を超えるとスロットリングされ、メモリの limits を超えると OOMKilled になります。requests を設定しないと BestEffort クラスになり、ノードのリソースが逼迫したとき最初に退避(eviction)されます。安定性が要るワークロードは requests と limits を同値にして Guaranteed クラスにします。GKE Autopilot では requests がそのまま課金対象になるため、過大な requests はコスト増に直結します。

Compute Engine ではマシンタイプの選択が要件定義そのものです。汎用(E2、N2、N4、C4)、コンピューティング最適化(C2、C3、H3)、メモリ最適化(M シリーズ)、アクセラレータ最適化(A シリーズ、G シリーズ)を用途で選び、必要ならカスタムマシンタイプで vCPU とメモリを個別に指定します。「メモリだけ大量に要る」ならメモリ最適化かカスタムマシンタイプで、汎用の大型を選んで vCPU まで増やすのは無駄です。

1-1-11 コストとリソース使用量の最適化 [1.1]

Optimizing for cost and resource usage は、選択肢のうち「同じ要件を満たしつつ安いもの」を選ぶ問題として出ます。手札を整理します。

手法	対象	効き方	注意点
Spot VM	Compute Engine、GKE のノード	大幅な割引(変動制)	いつでも停止されうる。中断耐性のあるバッチ向け
確約利用割引(CUD)	Compute Engine、Cloud Run など	1年または3年の使用量確約で割引	使わなくても課金される。定常負荷向け
継続利用割引(SUD)	Compute Engine	月内の稼働時間が長いと自動で割引	申し込み不要
適切なマシンタイプ選択(rightsizing)	Compute Engine	推奨事項(Recommender)に従って縮小	Active Assist の推奨を使う
最小インスタンス0	Cloud Run	アイドル時のコストが消える	コールドスタートが発生する
同時実行数の引き上げ	Cloud Run	インスタンス数が減る	メモリ・CPU の逼迫に注意
GKE Autopilot	GKE	使っていないノードの余剰分を払わない	Pod の requests が課金基準
クラスタオートスケーラー/HPA/VPA	GKE	需要に応じて縮む	縮退の速度と安定性のトレードオフ
ストレージクラスの選択とライフサイクル	Cloud Storage	アクセス頻度に応じて安いクラスへ	早期削除料金がある
BigQuery のパーティション・クラスタリング	BigQuery	スキャン量が減る	設計時に決める

Cloud Run のコストの考え方は、リクエストを処理している時間の vCPU・メモリ×インスタンス数です。したがって「トラフィックが1日数回しかない API のコストを最小化したい」なら最小インスタンス0(既定)にします。逆に「コールドスタートによる初回の遅延を無くしたい」なら最小インスタンスを1以上にしますが、その分アイドル課金が発生します。この2つはトレードオフで、問題文がレイテンシとコストのどちらを優先しているかで答えが変わります。

1-1-12 ゾーン・リージョンのフェイルオーバーを支えるデータレプリケーション [1.1]

Understanding data replication to support zonal and regional failover models は、可用性の要件から製品と構成を選ぶ問題です。

データストア	ゾーン障害への耐性	リージョン障害への耐性	仕組み
Cloud SQL	高可用性(HA)構成で別ゾーンにスタンバイを同期複製し、自動フェイルオーバー	クロスリージョンのリードレプリカを昇格(手動)、または災害復旧構成	同期(HA)/非同期(リードレプリカ)
AlloyDB	リージョナルクラスタが複数ゾーンに冗長化、自動フェイルオーバー	クロスリージョンレプリケーション	同期/非同期
Spanner	リージョナル構成で3ゾーンにレプリカ	マルチリージョン構成でリージョン障害に耐える	Paxos による同期複製。外部整合性
Bigtable	クラスタはゾーナル。インスタンス内に複数クラスタを置く	複数リージョンにクラスタを置きレプリケーション	非同期。結果整合

データストア	ゾーン障害への耐性	リージョン障害への耐性	仕組み
Firestore	リージョナル(複数ゾーン)またはマルチリージョン	マルチリージョン構成	同期(マルチリージョンは強整合)
Cloud Storage	リージョンバケットは複数ゾーンに冗長	デュアルリージョン/マルチリージョンバケット	冗長書き込み。強整合(オブジェクト単位)
永続ディスク	リージョナル永続ディスクは2ゾーンに同期複製	スナップショットを別リージョンへ	同期(2ゾーン)

フェイルオーバーの設計語彙も押さえます。RPO(目標復旧時点、どれだけのデータ損失を許すか)と RTO(目標復旧時間、どれだけの停止を許すか)です。同期レプリケーションは RPO をほぼゼロにできますが、書き込みレイテンシが増えます。非同期レプリケーションはレイテンシが小さい代わりに、フェイルオーバー時に直近の書き込みを失う可能性があります。「データ損失を一切許容できない」なら同期複製(Cloud SQL HA、Spanner)、「多少の損失は許すがコストを抑えたい」なら非同期のクロスリージョンレプリカです。

アプリケーション側の対応も出題されます。リージョナルフェイルオーバーを行う構成では、アプリケーションを複数リージョンにデプロイし、グローバル外部アプリケーションロードバランサでヘルスチェックに基づき振り分けます。Cloud Run はリージョナルサービスなので、複数リージョンに同じサービスをデプロイし、サーバーレス NEG を各リージョンに作ってグローバルロードバランサのバックエンドにまとめるのが定石です。「Cloud Run をマルチリージョン化したい」の答えはこれで、Cloud Run 単体でマルチリージョンにする設定は存在しません。

1-1-13 Cloud Run・GKE の新サービスに対するトラフィック分割戦略 [1.1]

Using traffic splitting strategies (gradual rollouts, rollbacks, A/B testing) on a new service on Cloud Run or GKE は、リリース戦略の理解を問います。まず戦略の定義を揃えます。

戦略	何をするか	目的
段階的ロールアウト (gradual rollout)/カナリア	新版に少量(1%、5%、10%…)のトラフィックを流し、指標を見ながら増やす	影響範囲を限定して不具合を早期に検知
ブルーグリーン	新旧2系統を並行稼働させ、一気に切り替える	切り戻しが速い。ただし2倍のリソースが要る
ロールバック	前の版にトラフィックを戻す	障害からの即時復旧
A/B テスト	属性(ヘッダー、Cookie、地域)で振り分け、機能の効果を比較	事業上の意思決定。ランダム割合ではなく条件で分ける場合が多い
ローリングアップデート	旧 Pod を少しずつ新 Pod に置き換える	GKE の Deployment の既定

Cloud Run では、デプロイのたびに不変(immutable)のリビジョンが作られ、トラフィックはリビジョン間で百分率で分割できます。要点は次のとおりです。

- `gcloud run deploy --no-traffic --tag canary` : 新リビジョンを作るがトラフィックは流さず、タグ付き URL(canary---service-xxx.a.run.app)だけで検証できます。
- `gcloud run services update-traffic <svc> --to-revisions <rev>=10,<prev>=90` : 10% を新リビジョンに流します。

- `gcloud run services update-traffic <svc> --to-latest`：最新リビジョンに100%流します。
- ロールバックは `--to-revisions <old-rev>=100` で、前のリビジョンに戻すだけです。再デプロイもビルドも不要で、これが Cloud Run のロールバックが速い理由です。

A/B テストで「特定のヘッダーを持つリクエストだけ新版へ」といった条件分岐が要る場合、Cloud Run のトラフィック分割は百分率のみなので不足します。その場合はタグ付き URL を用意し、前段のグローバル外部アプリケーションロードバランサの URL マップ(ヘッダーベースのルーティング、重み付きトラフィック分割)で振り分けます。

GKE では、Deployment のローリングアップデートが既定で、maxSurge(一時的に増やせる Pod 数)と maxUnavailable(同時に落としてよい Pod 数)で速度を制御します。`kubectl rollout undo deployment/<name>` でロールバックし、`kubectl rollout status` で進行を確認します。より細かいカナリアや A/B は、Deployment を2つ(stable と canary)用意してレプリカ数の比で割合を作るか、Cloud Service Mesh(Istio 系)や Gateway API の重み付きルーティングで制御します。

判断のポイントを表にします。

問題文	正解
「新版を本番トラフィックなしで検証したい」	Cloud Run のタグ付きリビジョン(--no-traffic --tag)
「不具合が出たら即座に前の版へ戻したい」	Cloud Run: update-traffic で前リビジョンへ100%。GKE: kubectl rollout undo
「5% から徐々に増やしたい」	トラフィック分割による段階的ロールアウト

問題文	正解
「日本からのユーザーだけ新機能を見せたい」	ロードバランサの URL マップまたはサービスメッシュによる条件ルーティング
「新版のリソースを別に確保して一気に切り替えたい」	ブルーグリーン

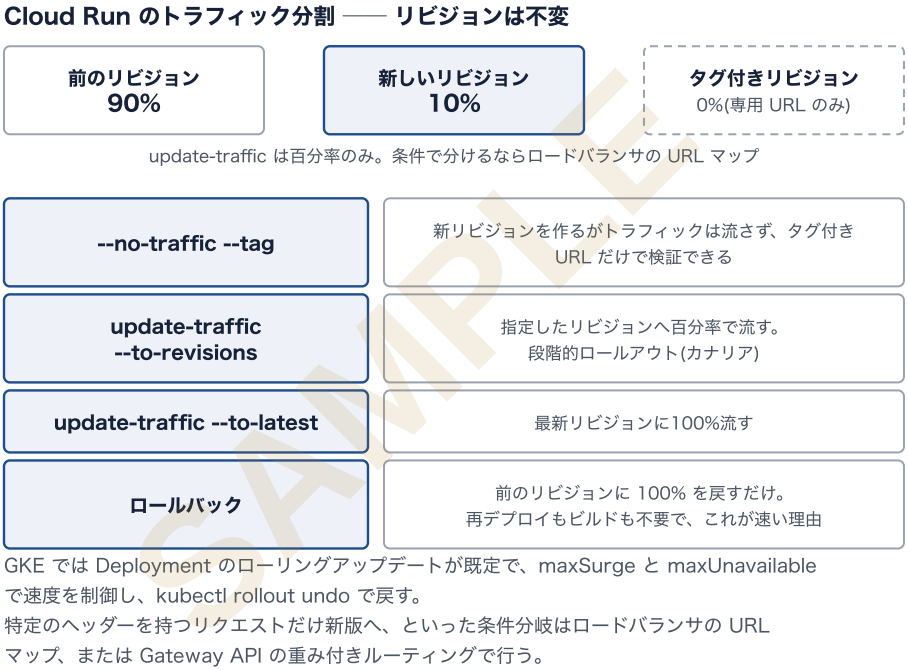


図 1-6　Cloud Run のトラフィック分割 —— リビジョンは不変

1-1-14 Workflows・Eventarc・Cloud Tasks・Cloud Scheduler によるアプリケーションサービスのオーケストレーション [1.1]

Orchestrating application services with Workflows, Eventarc, Cloud Tasks, and Cloud Scheduler は、4つの製品の役割を混同しないことが全てです。

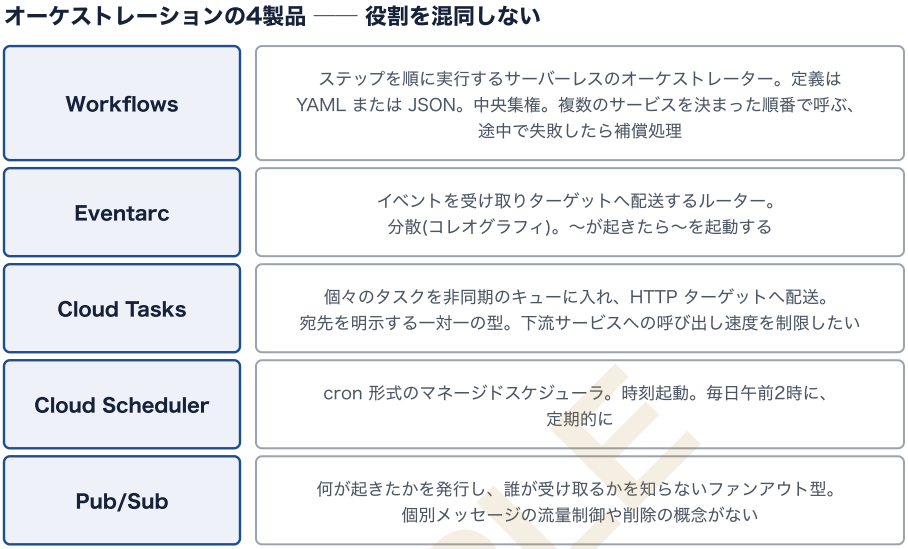
製品	一言で	制御の向き	代表的な要件語
Workflows	YAML/JSON で定義したステップを順に実行するサーバーレスのオーケストレーター。条件分岐、ループ、並列、リトライ、エラーハンドリング、コネクタによる Google Cloud API 呼び出し	中央集権(オーケストレーション)	「複数のサービスを決まった順番で呼ぶ」「途中で失敗したら補償処理」「長時間(最大1年)の待機」
Eventarc	イベントを受け取りターゲットへ配送するルーター	分散(コレオグラフィ)	「〜が起きたら〜を起動する」「Cloud Storage にファイルが置かれたら」
Cloud Tasks	個々のタスクを非同期のキューに入れ、HTTP ターゲットへ配送。レート制御、同時実行制御、リトライ、スケジュール指定、重複排除	キューイング	「処理をキューに積んで後で実行」「下流サービスへの呼び出し速度を制限したい」「タスク単位でリトライしたい」
Cloud Scheduler	cron 形式のマネージドスケジューラ。HTTP、Pub/Sub、App Engine をターゲットにできる	時刻起動	「毎日午前2時に」「定期的に」

最も間違えやすいのは Pub/Sub と Cloud Tasks の区別です。Pub/Sub は publisher が「何が起きたか」を発行し、誰が受け取るかを知らないファンアウト型です。Cloud Tasks は呼び出し元が「誰に何をさせるか」を知っていて、宛先を明示してタスクを積む一対一の型です。したがって「特定のエンドポイント

への呼び出しを毎秒N件に制限したい」「個別のタスクを後で実行するようスケジュールしたい」「特定のタスクを削除したい」は Cloud Tasks です。Pub/Sub には個別メッセージの流量制御や削除の概念がありません。

Workflows と Eventarc の区別も頻出です。処理の全体像を1か所で見たい、順序と失敗時の分岐を明示したい、なら Workflows です。各サービスが自律的にイベントに反応する構成でよいなら Eventarc です。Workflows は Cloud Run、Cloud Run functions、Google Cloud の各 API(BigQuery、Cloud Storage、Vertex AI など)をコネクタ経由で呼べ、ステップ間の変数の受け渡し、try/retry/except による例外処理、parallel による並列分岐を持ちます。

組み合わせの定石も覚えます。Cloud Scheduler が毎日決まった時刻に Workflows を起動し、Workflows が複数の Cloud Run サービスを順に呼び、途中の重い処理は Cloud Tasks でキューに逃がし、外部で起きた出来事は Eventarc が拾って別の Cloud Run を起動する、という形です。



組み合わせの定石 —— Cloud Scheduler が毎日決まった時刻に Workflows を起動し、Workflows が複数の Cloud Run サービスを順に呼び、
重い処理は Cloud Tasks でキューに逃がし、外部で起きた出来事は Eventarc が拾って別の Cloud Run を起動する。

図 1-5　オーケストレーションの4製品 —— 役割を混同しない

場面	正解	誤答になりやすいもの
毎晩23時にレポート生成のジョブを走らせる	Cloud Scheduler → Cloud Run job	Cloud Tasks(時刻起動の道具ではない)
注文確定後に在庫、決済、通知を順に呼び、決済失敗ならキャンセル処理	Workflows	Pub/Sub(順序と補償処理を書けない)
ユーザー登録直後にメール送信を後回しにし、送信先 API の1秒あたり呼び出し数を制限	Cloud Tasks	Pub/Sub(宛先ごとのレート制御ができない)

場面	正解	誤答になりやすいもの
Cloud Storage にファイルが置かれたら変換処理を起動	Eventarc → Cloud Run	Cloud Scheduler(ポーリングは非効率で誤り)
1つの出来事を3つの独立したサービスに同報	Pub/Sub(複数サブスクリプション)	Cloud Tasks(宛先ごとにタスクを積む必要があり非効率)

1-2 セキュアなアプリケーションの設計 [1.2]

この objective は「守り」の設計です。Professional 級では、機能の名前だけでなく「どの脅威に対してどの製品が効くか」「最小権限をどう実装するか」を選べることが求められます。

1-2-1 データ保持・組織ポリシーの実装(Cloud Storage のオブジェクトライフサイクル管理と保持ポリシー) [1.2]

Implementing data retention and organization policies は、データを「いつまで残し、いつ消し、どこに置かせないか」を仕組みで縛る話です。

Cloud Storage Object Lifecycle Management は、バケットに付けるルールで、条件に合ったオブジェクトを自動的に別のストレージクラスへ移すか削除します。ルールは条件(condition)とアクション(action)の組です。

条件の例	意味
age	オブジェクト作成からの日数
createdBefore	指定日より前に作成された
matchesStorageClass	現在のストレージクラスが一致する

条件の例	意味
numNewerVersions	より新しい版がN個以上ある(オブジェクトのバージョニング利用時)
daysSinceNoncurrentTime	非現行版になってからの日数
matchesPrefix / matchesSuffix	名前の前方・後方一致

アクション	意味
SetStorageClass	Standard → Nearline → Coldline → Archive へ移行
Delete	削除する
AbortIncompleteMultipartUpload	未完了のマルチパートアップロードを中止

ストレージクラスの選択基準は「アクセス頻度」と「最小保存期間」です。

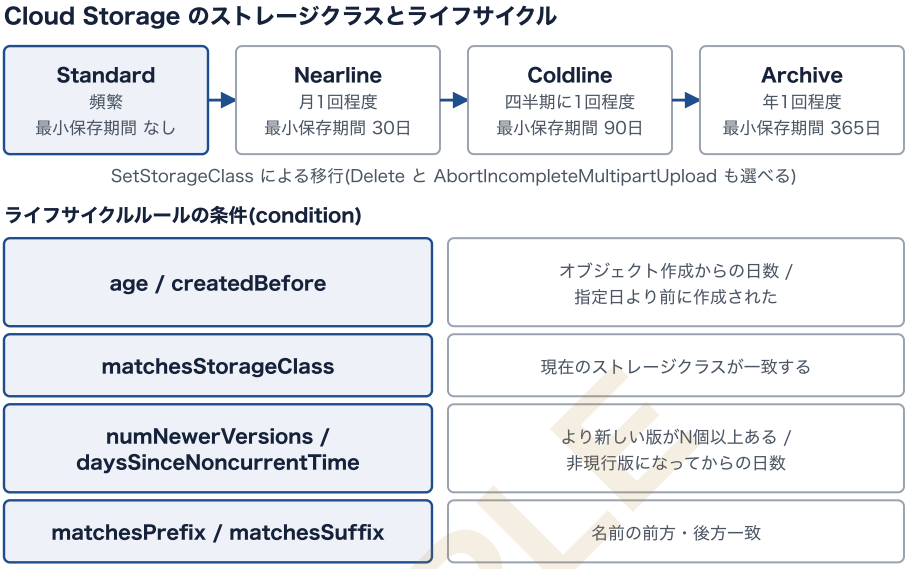
クラス	想定アクセス頻度	最小保存期間	取り出し料金
Standard	頻繁	なし	なし
Nearline	月1回程度	30日	あり
Coldline	四半期に1回程度	90日	あり(Nearline より高い)
Archive	年1回程度	365日	あり(最も高い)

最小保存期間より早く削除・移行すると早期削除料金が発生します。したがって「30日で消すログを Coldline に置く」は誤りで、Standard か Nearline のまま削除するのが正解です。また、Autoclass を有効にすると、アクセスパターンに応じて Google が自動でクラスを切り替え、早期削除料金も取り出し料金

も発生しません。「アクセスパターンが読めない」という記述があればAutoclass が正解になります。

保持ポリシー(retention policy)は、バケット内のオブジェクトを一定期間削除・上書きできなくする仕組みです。法規制で「7年間は改変不可」といった要件があるときに使います。重要なのは lock です。保持ポリシーをロック(use and lock retention policies)すると、そのポリシーは二度と短縮も解除もできなくなり、バケットは保持期間が切れるまで削除できません。これが監査対応の決め手になります。試験では「管理者であっても削除できないようにしたい」という要件でロック済み保持ポリシーが正解になり、IAM で削除権限を外す、という選択肢は「管理者が付け直せる」ため誤りになります。

個別のオブジェクトを固定したい場合はオブジェクト保持ロック(オブジェクト単位の保持設定)またはイベントベースのホールド、訴訟対応などで無期限に固定したい場合は訴訟ホールド(temporary/event-based hold)を使います。ホールドが掛かっているオブジェクトはライフサイクルルールでも削除されません。



最小保存期間より早く削除・移行すると早期削除料金が発生する。30日で消すログを Coldline に置くのは誤り。
アクセスパターンが読めない場合は Autoclass。
ロックした保持ポリシーは二度と短縮も解除もできず、管理者でも削除できない。

図 1-7　Cloud Storage のストレージクラスとライフサイクル

組織ポリシー(Organization Policy Service)は、組織・フォルダ・プロジェクトに対する制約(constraint)です。よく問われるものを挙げます。

制約	効果
constraints/gcp.resourceLocations	リソースを作れるリージョンを限定する(データ所在地の要件)
constraints/storage.uniformBucketLevelAccess	バケットに均一なバケットレベルアクセスを強制する(ACL を無効化)
constraints/storage.publicAccessPrevention	バケットの公開を禁止する

制約	効果
constraints/iam.disableServiceAccountKeyCreation	サービスアカウントキーの作成を禁止する
constraints/iam.allowedPolicyMemberDomains	外部ドメインのプリンシパルへの付与を禁止する
constraints/compute.requireOsLogin	OS Login を強制する
constraints/run.allowedIngress	Cloud Run のイングレス設定を制限する

「開発者が誤ってバケットを公開しても実際には公開されないようにしたい」は publicAccessPrevention、「サービスアカウントの JSON キーを作らせたくない」は disableServiceAccountKeyCreation です。IAM のロールを調整する選択肢より、組織ポリシーによる構造的な禁止のほうが正解になる場面が多い、と覚えます。

1-2-2 脆弱性を特定しサービスとリソースを保護するセキュリティ機構 (Identity-Aware Proxy・Web Security Scanner) [1.2]

Using security mechanisms that identify vulnerabilities and protect services and resources では、2つの製品が名指しされています。

Identity-Aware Proxy(IAP)は、アプリケーションの前段で ID を検証し、認可されたユーザーだけを通すゲートです。VPN を張らずに「誰であるか」に基づいてアクセスを制御する、いわゆるゼロトラスト(BeyondCorp)の実装です。

保護対象	使い方
App Engine、Cloud Run、GKE、Compute Engine の背後にあるアプリケーション (HTTPS ロードバランサ経由)	IAP を有効化し、IAP-secured Web App User ロールを付与した Google ID だけを通す

保護対象	使い方
VM への SSH、RDP	IAP TCP 転送(IAP-secured Tunnel User ロール)。VM に外部 IP を付けずに接続できる

IAP の要点は3つです。第一に、認証済みユーザーの情報はアプリケーションに署名付きヘッダー(x-goog-iap-jwt-assertion)で渡され、アプリケーションはこの JWT を検証してユーザーを識別できます。第二に、IAP を有効にしただけではロードバランサをバイパスする経路が残るため、バックエンドが直接アクセスできないようにする(Cloud Run ならイングレスを内部+ロードバランサに限定する、GKE/VM ならファイアウォールでロードバランサの IP レンジのみ許可する)必要があります。「IAP を有効にしたのに直接 URL でアクセスできてしまう」の答えがこれです。第三に、外部 IP を持たない VM に安全に SSH したい、という要件は IAP TCP 転送が正解で、踏み台 VM に外部 IP を付ける、Cloud VPN を張る、は「より複雑・より高コスト」として誤答になります。

Web Security Scanner は、App Engine、Compute Engine、GKE 上で動く Web アプリケーションをクロールして、既知の脆弱性を検出するスキャナです。検出できるのはクロスサイトスクリプティング(XSS)、Flash インジェクション、混在コンテンツ、平文での機密送信、古い脆弱なライブラリの利用、クリアテキストパスワードなどです。Security Command Center の一機能として提供され、マネージドスキャン(本番向けに読み取り操作のみ)とカスタムスキャン(任意の URL、認証情報を与えられる)があります。

注意点として、Web Security Scanner はテスト環境で実行するのが推奨です。実際にフォームを送信しリンクをクリックするため、本番でカスタムスキャンを走らせるとデータの変更やメール送信が起こりえます。試験では「本番に影響を与えずに Web アプリケーションの脆弱性を継続的に検査したい」に対

して、ステージング環境でのスキャン、あるいはマネージドスキャンが正解になります。

この2つと混同しやすい製品を並べます。

製品	役割
Google Cloud Armor	エッジでの WAF。OWASP Top 10 のプレコンフィグルール、IP 許可・拒否、レート制限、DDoS 緩和
Identity-Aware Proxy	ID に基づくアクセス制御(誰が入れるか)
Web Security Scanner	アプリケーション自身の脆弱性検出
Artifact Analysis	コンテナイメージ・パッケージの脆弱性スキャン
VPC Service Controls	API 単位のデータ持ち出し防止の境界

「SQL インジェクションの試行をブロックしたい」は Cloud Armor、「社内の特定グループだけに管理画面を見せたい」は IAP、「デプロイ前のイメージに既知の CVE がないか調べたい」は Artifact Analysis です。

1-2-3 脆弱性への対応と解決(Artifact Analysis・Security Command Center) [1.2]

Responding to and resolving vulnerabilities, including those identified by Artifact Analysis and Security Command Center は、検出から是正までの流れを問います。

Artifact Analysis(旧称 Container Analysis)は、Artifact Registry に置かれたコンテナイメージ、および Maven、npm、Go、Python などのパッケージを

スキャンし、脆弱性(vulnerability occurrence)のメタデータを生成します。動作は2種類です。

種類	いつ走るか	用途
自動スキャン(オンプッシュ)	イメージが Artifact Registry に push された時	CI で作った直後に検出する
継続的分析	新しい脆弱性情報が公開された時に既存イメージの結果を更新	「昨日は安全だったイメージが今日は危険」を検知する

是正の流れは、ベースイメージを更新して再ビルドする、依存ライブラリを固定版から修正版に上げる、そして Binary Authorization で「重大な脆弱性のあるイメージはデプロイさせない」という関門を作る、という順です。試験では「脆弱なイメージが本番にデプロイされるのを防ぐ」に対して Binary Authorization が正解で、Artifact Analysis 単体は「検出はするが止めない」ため不十分です。この違いは頻出です。

Security Command Center(SCC)は、組織全体のセキュリティ態勢を一望するダッシュボードです。扱うのは次の3種類です。

種類	内容	例
誤設定(misconfiguration)	Security Health Analytics による構成の検査	公開バケット、外部 IP を持つ VM、過剰な IAM 権限、無効化された監査ログ
脆弱性(vulnerability)	Web Security Scanner、Artifact Analysis 由来	XSS、CVE
脅威(threat)	Event Threat Detection、Container Threat Detection による実行時の検知	不審な IAM 付与、暗号通貨マイニング、コンテナ内の異常なバイナリ実行

SCC には Standard と Premium(および Enterprise)のティアがあり、Event Threat Detection 、Container Threat Detection 、Security Health Analytics の全検出器、コンプライアンスレポート(CIS、PCI DSS、NIST など)は上位ティアの機能です。「組織全体の PCI DSS 準拠状況をレポートしたい」は SCC の Premium 以上が答えです。

検出結果(findings)への対応は、SCC 上でミュート、修正、Pub/Sub 経由での外部連携(チケット起票、Chronicle/SIEM 連携)という形を取ります。「検出をワークフローに繋いで自動で是正したい」なら SCC の findings を Pub/Sub に出し、Cloud Run 関数で是正処理を走らせる、という構成が正解です。

1-2-4 アプリケーションのシークレット・認証情報・暗号鍵の保存・アクセス・ローテーション(Secret Manager・Cloud KMS・Workload Identity Federation) [1.2]

Storing, accessing, and rotating application secrets, credentials, and encryption keys は、実装の詳細まで問われます。

Secret Manager は、API キー、パスワード、証明書などの機密文字列を保管するサービスです。要点を整理します。

概念	内容
シークレットとバージョン	シークレットは入れ物、実際の値はバージョン。バージョンは不変で、更新は新バージョンの追加
アクセス	roles/secretmanager.secretAccessor をサービスアカウントに付け、`versions/latest` または固定のバージョン番号で取得

概念	内容
ローテーション	ローテーション期間を設定すると、指定周期で Pub/Sub トピックに通知が届く。実際の値の入れ替えは自分の処理で行う
レプリケーション	自動(Google が選ぶ)またはユーザー管理(リージョンを明示)。データ所在地の要件があれば後者
暗号化	既定は Google 管理の鍵。顧客管理暗号鍵(CMEK)も指定可
有効期限	シークレットに expire_time を設定して自動削除できる

実装上の勘所は3つです。第一に、シークレットを環境変数に入れるかファイルとしてマウントするかです。Cloud Run と GKE はどちらも可能ですが、環境変数はプロセス一覧やログに漏れる危険があり、また値はインスタンス起動時に固定されます。ファイルとしてマウントすると、Cloud Run では新しいリビジョンを作らずに `latest` を参照して更新を拾える場面があります。ローテーションを頻繁に行うなら、実行時に API で取得するのが最も確実です。第二に、シークレットの値をコードやコンテナイメージに埋め込まない、Git に置かない、という原則です。第三に、`versions/latest` を使うと自動的に新しい値を拾えますが、壊れた値を入れると即座に全インスタンスが影響を受けるため、段階的に切り替えたい場合はバージョン番号を明示します。

Cloud Key Management Service(Cloud KMS)は暗号鍵そのものを管理します。

概念	内容
キーリングとキー	リージョンに属するキーリングの中にキーを作る。キーリングもキーも削除できない(バージョンは破棄予定にできる)
目的	対称暗号化、非対称暗号化、非対称署名、MAC
保護レベル	SOFTWARE、HSM(FIPS 140-2 Level 3)、EXTERNAL(外部キーマネージャ)、EXTERNAL_VPC
ローテーション	対称鍵は自動ローテーション期間を設定できる。新バージョンが主バージョンになり、以後の暗号化に使われる。古いバージョンは復号のために残る
CMEK	顧客管理暗号鍵。Cloud Storage、BigQuery、Cloud SQL、Pub/Sub などのサービスのデータをこの鍵で暗号化する
CSEK	顧客指定暗号鍵。鍵を Google に預けず、リクエストごとに渡す(Cloud Storage、Compute Engine)

「規制で暗号鍵を自分で管理し、ローテーション周期を決め、いつでも失効させたい」は CMEK です。「鍵素材を Google Cloud に一切保存したくない」は CSEK または Cloud External Key Manager(EKM)です。「特に要件がない」なら Google 管理の既定の暗号化で足り、CMEK を選ぶのは過剰(運用負担とコスト)として誤答になります。

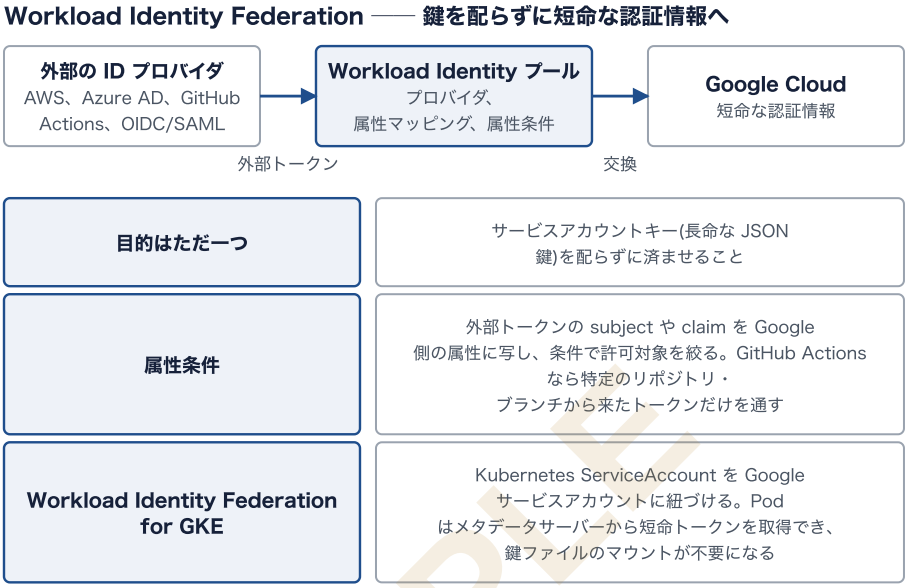
Secret Manager と Cloud KMS の使い分けは頻出です。Secret Manager は「秘密の値そのものを預ける」、Cloud KMS は「値を暗号化する鍵を預ける」場所です。API キーやデータベースのパスワードは Secret Manager、アプリケ

ーションが自前で暗号化するためのマスターキーは Cloud KMS です。「パスワードを Cloud KMS で暗号化して Cloud Storage に置く」という選択肢は、Secret Manager がある以上わざわざ複雑にしているため誤答になります。

Workload Identity Federation(WIF)は、外部の ID プロバイダ(AWS、Azure AD、GitHub Actions、オンプレミスの OIDC/SAML プロバイダ、Kubernetes)が発行したトークンを、Google Cloud の短命な認証情報に交換する仕組みです。目的はただ一つ、サービスアカウントキー(長命な JSON 鍵)を配らずに済ませることです。

構成要素は、Workload Identity プール、そのプール内のプロバイダ(OIDC または SAML の設定)、そして属性マッピングと属性条件です。外部トークンの subject や claim を Google 側の属性に写し、条件で許可対象を絞ります。たとえば GitHub Actions なら、特定のリポジトリ・ブランチから来たトークンだけを通す条件を書きます。

GKE 内のワークロードについては Workload Identity Federation for GKE を使い、Kubernetes ServiceAccount を Google サービスアカウントに紐づけます。これによって Pod はメタデータサーバーから短命トークンを取得でき、鍵ファイルのマウントが不要になります。「GKE の Pod が Cloud Storage にアクセスする最も安全な方法は」の答えはこれで、サービスアカウントキーを Secret として置く、ノードのサービスアカウントに広い権限を付ける、はいずれも誤りです。



GKE の Pod が Cloud Storage にアクセスする最も安全な方法はこれ。キーを Secret として置く、ノードのサービスアカウントに広い権限を付ける、はいずれも誤り。組織ポリシー constraints/iam.disableServiceAccountKeyCreation でキーの作成そのものを禁止できる。

図 1-8　Workload Identity Federation —— 鍵を配らずに短命な認証情報へ

1-2-5 Google Cloud サービスへの認証(Application Default Credentials・JWT・OAuth 2.0・Cloud SQL Auth Proxy・AlloyDB Auth Proxy・Identity Platform・WIF) [1.2]

Authenticating to Google Cloud services は、認証方式の使い分けを整理する項目です。

Application Default Credentials(ADC)は、クライアントライブラリが認証情報を探す標準の順序です。順序は次のとおりです。

1. 環境変数 GOOGLE_APPLICATION_CREDENTIALS が指すファイル(サービスアカウントキーまたは外部アカウント構成)
2. gcloud のユーザー認証情報(`gcloud auth application-default login` で作られる)
3. 実行環境に紐づくサービスアカウント(Compute Engine、GKE、Cloud Run、Cloud Run functions のメタデータサーバー)

実装の原則は「コードに認証情報を書かない」です。ローカル開発では `gcloud auth application-default login`、Google Cloud 上では実行環境に付いたサービスアカウントを ADC が自動で拾う、外部からは Workload Identity Federation の構成ファイルを ADC に読ませる、という三本立てです。「ローカルでは動くが Cloud Run で権限エラーになる」というトラブルシュートは、Cloud Run サービスに割り当てたサービスアカウントの IAM ロール不足が原因です。

`gcloud auth login` と `gcloud auth application-default login` の違いも問われます。前者は gcloud コマンド自身が使う認証、後者はアプリケーションのクライアントライブラリが ADC として使う認証です。「コードから API を呼ぶのに認証エラーになる」なら後者が必要です。

OAuth 2.0 と JWT の位置づけを整理します。

方式	何を証明するか	使う場面
OAuth 2.0(3-legged)	エンドユーザーが自分のデータへのアクセスをアプリケーションに委任した	ユーザーの Google Drive や Gmail のデータを扱う。同意画面が必要
OAuth 2.0(2-legged、サービスアカウント)	アプリケーション自身が主体	サーバー間の呼び出し。Google Cloud API の大半

方式	何を証明するか	使う場面
ID トークン(OIDC の JWT)	呼び出し元が誰であるか	Cloud Run の認証済み呼び出し、IAP、API Gateway の認証
アクセストークン(OAuth 2.0)	何にアクセスしてよいか	Google Cloud API の呼び出し
自己署名 JWT	サービスアカウントが自らの鍵で署名した主張	一部の Google API で、トークンエンドポイントを介さず直接使う

Cloud Run のサービス間呼び出しでは、呼び出し元が自分のサービスアカウントの ID トークン(audience を呼び出し先の URL にしたもの)をメタデータサーバーから取得し、Authorization: Bearer ヘッダーに付けます。呼び出し先には roles/run.invoker を呼び出し元サービスアカウントに付与します。この形が「Cloud Run から Cloud Run を安全に呼ぶ」の正解です。API キーを使う、`--allow-unauthenticated` にして IP で絞る、は誤りです。API キーは呼び出し元プロジェクトの特定には使えても、認証(誰であるか)の手段ではありません。

Cloud SQL Auth Proxy と AlloyDB Auth Proxy は、データベースへの接続を安全にするローカルプロキシです。効能は3つあります。第一に、IAM の権限(roles/cloudsql.client)で接続を認可するため、ネットワーク的な許可リスト(承認済みネットワーク)の管理が不要になります。第二に、接続は自動的に TLS で暗号化され、証明書の管理が要りません。第三に、データベースインスタンスに公開 IP を付けずにプライベート IP のみで運用できます。Cloud SQL では IAM データベース認証を併用すると、データベースのパスワードそのものを廃してサービスアカウントの ID で認証できます。

Cloud Run から Cloud SQL に繋ぐ経路は、Cloud SQL 接続機能(内蔵の Cloud SQL Auth Proxy による Unix ソケット接続)を使う方法と、Direct VPC egress または Serverless VPC Access でプライベート IP に繋ぐ方法があります。GKE では Cloud SQL Auth Proxy をサイドカーコンテナとして動かすのが定石です。「接続文字列にパスワードを書かず、公開 IP も付けずに接続したい」という要件は、Auth Proxy と IAM データベース認証の組み合わせが正解です。

Identity Platform は、アプリケーションのエンドユーザー(顧客)の認証を担う CIAM(顧客 ID 管理)サービスです。メール・パスワード、電話番号、Google、Facebook、Apple などのソーシャルログイン、SAML/OIDC の企業 ID 連携、多要素認証、マルチテナントに対応します。Firebase Authentication の上位版という位置づけです。

ここで区別すべきは「誰の ID か」です。

対象	使うもの
自社の従業員・Google Cloud の操作者	Cloud Identity / Google Workspace の ID と IAM
自社アプリケーションの一般顧客	Identity Platform(または Firebase Authentication)
外部クラウド・CI で動くワークロード	Workload Identity Federation
Google Cloud 上のワークロード	サービスアカウント(+ ADC)
社内向け Web アプリへのアクセス制御	Identity-Aware Proxy

「モバイルアプリのユーザーが SNS アカウントでログインできるようにしたい」は Identity Platform で、IAM や IAP は従業員向けなので誤りです。

1-2-6 サービスアカウント向け IAM ロールによるクラウドリソースの保護 [1.2]

Securing cloud resources using Identity and Access Management (IAM) roles for service accounts は、IAM の基礎と、サービスアカウント特有の論点を扱います。

IAM の許可ポリシーは、リソースに対して「誰(principal)に、どのロール(role)を」束ねたバインディングの集合です。ロールには3種類あります。

種類	例	使い方
基本ロール	Owner、Editor、Viewer	プロジェクト全体に広く効く。本番では使わない
事前定義ロール	roles/run.invoker、roles/storage.objectViewer、roles/pubsub.publisher	実務の標準。最小権限に近いものを選ぶ
カスタムロール	権限を個別に列挙	事前定義ロールでは広すぎる場合

ポリシーは継承されます。組織 → フォルダ → プロジェクト → リソースの順に下位へ継承され、上位で付けた権限を下位で剥がすことはできません(拒否ポリシー、IAM Deny を使う場合を除く)。したがって最小権限の設計は「上位で広く付けない」ことから始まります。

サービスアカウント特有の論点は次のとおりです。

1. サービスアカウントは ID であると同時にリソースでもあります。したがって「サービスアカウントに付けるロール」(そのサービスアカウントが何をできるか)と「サービスアカウントに対して付けるロール」(誰がそのサービスアカウントを使えるか)の2種類があります。後者の代表が

roles/iam.serviceAccountUser(そのサービスアカウントを付けてリソースを起動できる)と roles/iam.serviceAccountTokenCreator(そのサービスアカウントのトークンを発行できる=なりすませる)です。

2. 既定のサービスアカウント(Compute Engine の既定サービスアカウント、App Engine の既定サービスアカウント)は広い権限(Editor)を持つことがあるため、本番では専用のサービスアカウントを作って必要なロールだけを付けます。「Cloud Run が既定のサービスアカウントで動いていて権限が過大」という指摘に対する正解はこれです。
3. サービスアカウントキー(JSON)は長命な認証情報で、漏洩リスクが最も高い資産です。原則として作らず、Workload Identity Federation、実行環境に紐づくサービスアカウント、あるいはサービスアカウントの権限借用(impersonation)で代替します。組織ポリシー iam.disableServiceAccountKeyCreation で作成そのものを禁止できます。
4. 権限借用(impersonation)は、あるプリンシパルが別のサービスアカウントとして短命なトークンを取得する仕組みです。`gcloud --impersonate-service-account=` や `gcloud auth application-default login --impersonate-service-account=` で使えます。開発者にキーを配らずに本番相当の権限で検証させたいときの正解です。

IAM Conditions(条件付きロールバインディング)も押さえます。時刻(request.time)、リソース名(resource.name)、リソースタイプなどの条件でロールの効力を絞れます。「特定の接頭辞を持つバケットにだけ読み取りを許したい」「期間限定で権限を付与したい」は IAM Conditions が正解です。

権限が過剰かどうかの点検には IAM Recommender(Policy Intelligence)を使います。過去90日の使用実績から、使われていない権限を外す推奨を出

します。「最小権限に近づけたいが、どのロールを外してよいか分からない」の答えです。

1-2-7 セキュアなサービス間通信の実装(Cloud Service Mesh・Kubernetes Network Policy・Direct VPC egress・プライベートサービス接続) [1.2]

Incorporating secure service-to-service communications は、通信経路をどう閉じるかの設計です。

Cloud Service Mesh は、GKE および Compute Engine 上のワークロードに、サイドカープロキシ(Envoy)またはプロキシレス gRPC を通じてトラフィック管理、可観測性、セキュリティを提供するマネージドのサービスメッシュです。セキュリティ面では、相互 TLS(mTLS)を自動で有効にしてサービス間通信を暗号化し、ワークロードの ID(SPIFFE ID)に基づく認可ポリシーで「どのサービスがどのサービスを呼べるか」を制御します。加えて、重み付きルーティングによるカナリア、リトライ、タイムアウト、サーキットブレーカー、障害注入といったトラフィック制御も担います。「アプリケーションのコードを変えずにサービス間通信を暗号化し、呼び出しの許可をサービス単位で制御したい」は Cloud Service Mesh が正解です。

Kubernetes Network Policy は、Pod 間の通信をラベルセレクタで許可・拒否するネットワーク層 (L3/L4) の制御です。GKE では有効化が必要で、Dataplane V2(eBPF ベース)を使うクラスタでは既定で利用でき、ポリシーのログも取れます。既定では全 Pod が相互に通信できるため、まず全拒否のポリシーを置き、必要な経路だけを許可するのが定石です。

この2つの違いは頻出です。Network Policy は IP とポートの層で、誰が誰に接続できるかを決めます。暗号化も、HTTP のパスやメソッド単位の認可もできません。Cloud Service Mesh は L7 で、mTLS による暗号化とアイデンティ

ティに基づく認可を行います。「通信を暗号化したい」なら Network Policy では不足です。「単純に名前空間をまたぐ通信を止めたい」なら Network Policy で十分で、メッシュの導入は過剰です。

Direct VPC egress は、Cloud Run のインスタンスに VPC 内の IP アドレスを直接割り当て、コネクタを介さずに VPC 内部のリソース(Memorystore、Cloud SQL のプライベート IP、内部ロードバランサ、オンプレミス)へ通信できるようにする機能です。従来の Serverless VPC Access コネクタと比べ、コネクタ用の VM が不要でレイテンシとコストが下がり、スループットの上限がインスタンス数に応じて伸びます。設定では VPC ネットワーク、サブネット、そして下り(egress)の範囲(すべてのトラフィックを VPC 経由にするか、プライベート IP 宛てだけにするか)を選びます。「Cloud Run から VPC 内のデータベースに繋ぎたい、コネクタの管理はしたくない」は Direct VPC egress が正解です。

プライベートサービス接続(private service connectivity)は、Google のマネージドサービスや第三者のサービスに、公開インターネットを経由せずアクセスするための仕組み群です。区別して覚えます。

仕組み	何をするか	使う場面
Private Service Connect(PSC)	自分の VPC 内に、対象サービスへ向かうエンドポイント(内部 IP)を作る	Google API へ内部 IP でアクセス、他テナントのサービスを内部 IP で消費、公開したサービスを他テナントに提供
Private Service Access(サービスネットワーキング)	VPC ピアリングで Google 管理の VPC と接続	Cloud SQL、Memorystore、AlloyDB のプライベート IP

仕組み	何をするか	使う場面
Private Google Access	サブネットで有効化し、外部 IP のない VM が Google API の外部 IP に到達できるようにする	外部 IP なしの VM から Cloud Storage などへ
VPC Service Controls	サービス境界を作り、境界外へのデータ持ち出しを API レベルで防ぐ	内部関係者による持ち出しの防止、データ流出対策

「外部 IP を持たない VM から Cloud Storage にアクセスしたい」は Private Google Access、「Google API へのアクセスを自分の VPC の内部 IP 経由に固定したい」は Private Service Connect、「機密データが境界の外のプロジェクトへコピーされるのを防ぎたい」は VPC Service Controls です。VPC Service Controls はネットワークではなく API 呼び出しの境界である、という点が誤答を切る鍵になります。

1-2-8 最小権限でのサービス実行 [1.2]

Running services with least privileged access は、これまでの内容を実装の型に落とす項目です。原則を並べます。

1. ワークロードごとに専用のサービスアカウントを作ります。複数のサービスで1つのサービスアカウントを共有すると、権限が和集合になり最小権限を保てません。
2. 付けるロールは事前定義ロールから最小のものを選びます。roles/editor や roles/owner を付ける選択肢は原則すべて誤答です。事前定義ロールでも広すぎる場合はカスタムロールを作ります。
3. コンテナ側でも最小権限を適用します。GKE では Pod のセキュリティコンテキストで runAsNonRoot: true、allowPrivilegeEscalation: false、

readOnlyRootFilesystem: true、不要な Linux capability の drop を設定します。特権コンテナ(privileged: true)は原則使いません。GKE Autopilot はこうした制約を既定で強制します。

4. ノードレベルでは、GKE のノードのサービスアカウントに広い権限を持たせず、Pod ごとに Workload Identity Federation for GKE で必要な Google サービスアカウントを紐づけます。
5. Cloud Run では `--service-account` で専用のサービスアカウントを指定し、呼び出しは `--no-allow-unauthenticated` にして roles/run.invoker を持つプリンシパルだけに限定します。イングレスは `--ingress internal` または `internal-and-cloud-load-balancing` で絞ります。
6. 権限の付与は期間や条件で絞れないか検討します(IAM Conditions)。恒久的な昇格ではなく、必要なときだけ権限借用で昇格する形が望ましい構成です。

トラブルシュートの型も押さえます。403 の権限エラーが出たとき、確認すべきは「どのプリンシパルが」「どのリソースに対して」「どの権限を」必要としているかの3点です。Policy Troubleshooter を使うと、特定のプリンシパルが特定のリソースに対して特定の権限を持つかどうかと、その根拠となるバインディングを表示できます。「なぜアクセスできないのか調べたい」の答えです。

1-2-9 Binary Authorization によるアプリケーション成果物の保護 [1.2]

Securing application artifacts using Binary Authorization は、デプロイ時の関門(admission control)です。仕組みは次のとおりです。

1. ポリシーを定義します。ポリシーは既定ルール(defaultAdmissionRule)と、クラスタや Cloud Run サービス単位の例外ルールから成ります。評価

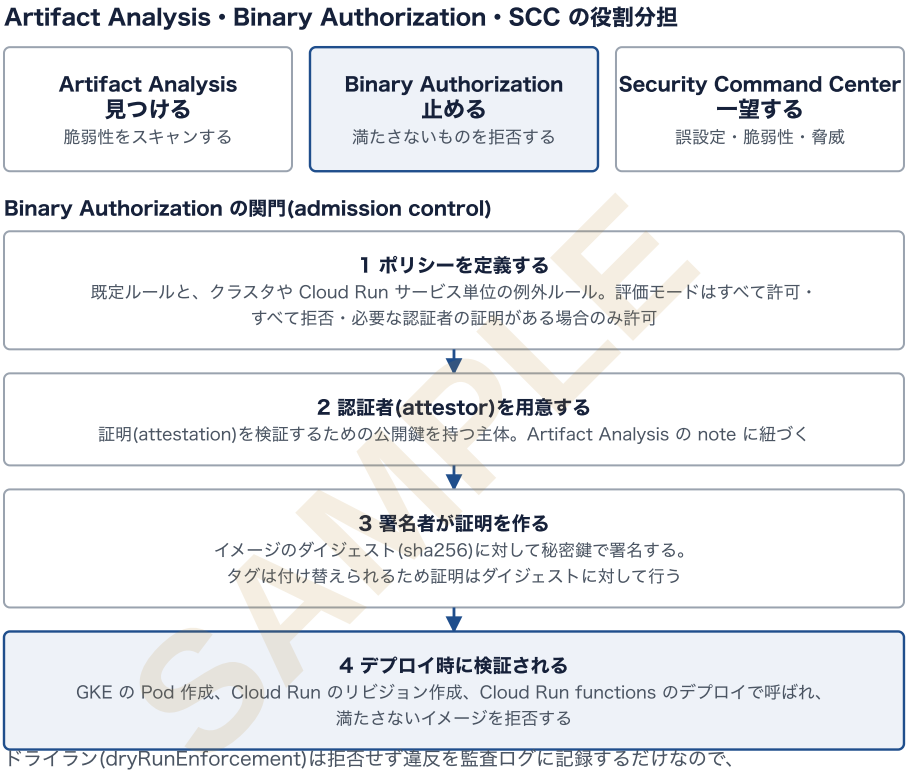
モードは、すべて許可、すべて拒否、または「必要な認証者(attestor)の証明がある場合のみ許可」です。

2. 認証者(attestor)は、証明(attestation)を検証するための公開鍵を持つ主体です。認証者は Artifact Analysis の note に紐づきます。
3. 署名者(たとえば脆弱性スキャンに合格したことを確認する CI のステップ、あるいは QA の承認プロセス)が、イメージのダイジェストに対して秘密鍵で署名し、証明を作ります。
4. デプロイ時(GKE の Pod 作成、Cloud Run のリビジョン作成、Cloud Run functions のデプロイ)に Binary Authorization が呼ばれ、ポリシーを満たさないイメージのデプロイを拒否します。

重要な点を挙げます。第一に、イメージはタグではなくダイジェスト(sha256)で識別されます。タグは付け替えられるため、証明はダイジェストに対して行います。「latest タグを使っていて、意図しないイメージがデプロイされた」の再発防止策はダイジェスト指定と Binary Authorization です。第二に、緊急時のためにブレークグラス(breakglass)があり、ラベルを付けることでポリシーを迂回できますが、その事実は監査ログに記録されます。第三に、ドライラン(dryRunEnforcement)モードでは、拒否せずに違反を監査ログに記録するだけなので、ポリシー導入時の影響調査に使えます。第四に、許可リストのイメージパターン(exempt images)を使うと、特定のレジストリパス配下のイメージを検査対象から外せます。

Cloud Build の provenance(来歴)と組み合わせると、「Cloud Build で作られたイメージであること」自体を条件にできます。SLSA のビルドレベルの考え方で、承認済みのビルドパイプライン以外で作られたイメージを本番に入れない、という統制です。この組み合わせは 2.2 のビルドの節でも扱います。

まとめると、Artifact Analysis が「見つける」、Binary Authorization が「止める」、Security Command Center が「一望する」役割です。この三者の役割分担が問われたら、この一行で切り分けます。



ドライラン(dryRunEnforcement)は拒否せず違反を監査ログに記録するだけなので、
ポリシー導入時の影響調査に使える。
緊急時のブレークグラス(breakglass)はポリシーを迂回できるが、
その事実は監査ログに記録される。

図 1-9　Artifact Analysis・Binary Authorization・SCC の役割分担

1-3 データの保存とアクセス [1.3]

1-3-1 データ量と性能要件に基づく適切なストレージシステムの選択 [1.3]

Selecting the appropriate storage system based on the volume of data and performance requirements は、この試験で最も繰り返し問われる判断です。まず全体像を1枚の表にします。

製品	種類	規模の目安	整合性	得意なこと	苦手なこと
Cloud Storage	オブジェクトストレージ	無制限	オブジェクト単位で強整合	画像・動画・バックアップ・データレイク・静的サイト	部分更新、低レイテンシのトランザクション
Cloud SQL	マネージドRDBMS(MySQL、PostgreSQL、SQL Server)	単一インスタンスで数十TB程度まで	強整合	既存のSQLアプリケーション、トランザクション	水平方向の書き込みスケール
AlloyDB	PostgreSQL互換の高性能RDBMS	大規模OLTP＋分析	強整合	PostgreSQL互換のまま高いスループット、列指向エンジンによる分析高速化	Spannerのようなグローバル書き込み分散

製品	種類	規模の目安	整合性	得意なこと	苦手なこと
Spanner	水平分散のリレーショナル DB	ペタバイト級	強整合(外部整合性)	グローバルに一貫した OLTP、無停止のスケール、99.999% の SLA(マルチリージョン)	小規模用途にはコスト過大
Bigtable	ワイドカラム型 NoSQL	ペタバイト級、毎秒数百万の読み書き	行単位で強整合(単一クラスタ)	時系列、IoT、金融データ、大量書き込み、低レイテンシの単一行アクセス	複数行のトランザクション、SQL による複雑な結合、二次インデックス
Firestore	ドキュメント型 NoSQL	大規模	強整合(Native モード)	モバイル・Web のバックエンド、リアルタイム同期、オフライン対応	大規模な集計分析、複雑な結合
Memorystore	インメモリ	GB 〜 TB	強整合(単一ノード)	キャッシュ、セッション、レート制限のカウンタ	永続的な真実の保管
BigQuery	分析用データウェアハウス	ペタバイト級	強整合	SQL による大規模分析、BI、ML	1行ずつの低レイテンシ更新、OLTP

製品	種類	規模の目安	整合性	得意なこと	苦手なこと
Filestore	マネージド NFS	TB 級	POSIX	複数 VM/Pod からの共有ファイルシステム	オブジェクト的な大規模配信

判断の手順は次のとおりです。

1. 構造化されたデータで SQL とトランザクションが要るか。要らないなら、ファイルなら Cloud Storage、キー・バリューの大量アクセスなら Bigtable、アプリのドキュメントなら Firestore。
2. SQL が要るとして、既存の MySQL/PostgreSQL/SQL Server をそのまま動かすなら Cloud SQL。PostgreSQL 互換のまま性能を大きく上げたいなら AlloyDB。
3. 単一インスタンスの限界を超える書き込みスループット、あるいは複数リージョンにまたがる強整合が要るなら Spanner。
4. 分析クエリ(集計、結合、大量スキャン)が主なら BigQuery。

誤答を切る一語を覚えます。「グローバルに強整合」「無停止で水平にスケールする RDBMS」は Spanner。「毎秒数百万の書き込み」「時系列」「ミリ秒未満の単一行読み取り」は Bigtable。「モバイルアプリでオフラインでも使え、変更がリアルタイムに反映される」は Firestore。「ペタバイトを SQL で分析」は BigQuery。「既存の MySQL アプリを最小の変更で移す」は Cloud SQL です。

1-3-2 構造化データベース(AlloyDB・Spanner)と非構造化データベース(Bigtable・Firestore)の適切なスキーマ設計 [1.3]

Designing appropriate schemas for structured databases and unstructured databases は、製品ごとに設計原則が異なります。

AlloyDB は PostgreSQL 互換なので、スキーマ設計の原則も PostgreSQL に準じます。正規化してから、読み取りパターンに応じてインデックスを張ります。AlloyDB 固有の要素として、列指向エンジン(columnar engine)があり、分析クエリの対象となる列をメモリ上の列形式で保持して高速化します。行の主キー設計、外部キーによる整合性、パーティションテーブルによる大規模テーブルの分割、といった一般的な設計判断がそのまま通用します。

Spanner のスキーマ設計は独特で、試験で最も問われるのはホットスポットの回避とインターリーブです。

設計要素	内容	なぜそうするか
主キーの選び方	単調増加する値(タイムスタンプ、連番)を主キーの先頭にしない	キー空間の末尾にすべての書き込みが集中し、スプリットが偏る(ホットスポット)
回避策	UUID(ランダム)を使う、ハッシュ値を先頭に付ける、キーの順序を入れ替える、シャード ID を先頭に付ける	書き込みをキー空間全体に散らす
インターリーブ	子テーブルを親テーブルの行の近くに物理的に配置する(INTERLEAVE IN PARENT)	親子を一緒に読むクエリで結合が速くなる。親子を1トランザクションで更新するのが自然

設計要素	内容	なぜそうするか
セカンダリインデックス	検索に使う列に張る。STORING 句で追加列を持たせるとインデックスだけで応答できる	インデックスからベーステーブルへの参照(バックジョイン)を避ける
NULL_FILTERED インデックス	NULL の行を含めない	インデックスの容量削減

「Spanner に書き込みを増やしたら性能が頭打ちになった」の原因はほぼホットスポットで、対策は主キー設計の見直しです。ノードを増やす、という選択肢は根本原因を解決しないため誤りになる場面が多いです。

Bigtable のスキーマ設計はさらに特徴的です。

設計要素	内容
行キーが唯一のインデックス	検索は行キーの完全一致か範囲スキャンのみ。二次インデックスはない
行キーの設計	よく使うクエリの範囲スキャンが効くように、フィールドを連結して作る(例: deviceId#reversedTimestamp)
ホットスポット回避	先頭にタイムスタンプやシーケンシャルな ID を置かない。フィールドプロモーション(識別子を先頭に)やソルティングを使う
列ファミリー	一緒に読む列をまとめる。数は少なく(数十まで)保つ
疎な設計	値のないセルは容量を消費しない。列を大量に持つ横長の設計が可能
単一行トランザクション	1行内の更新は原子的。複数行にまたがるトランザクションはない

設計要素	内容
ガベージコレクション	列ファミリー単位で、バージョン数または経過時間のポリシーを設定

時系列データの定石は、行キーを「エンティティ ID + タイムスタンプ(必要なら逆順)」にして、あるデバイスの一定期間のデータが連続した行として並ぶようにすることです。逆に「タイムスタンプ + デバイス ID」にすると、同時刻の書き込みが1つのタブレットに集中してホットスポットになります。この対比はほぼ必ず出ます。

Firestore のスキーマ設計は、コレクション、ドキュメント、サブコレクションの階層と、クエリの制約から決まります。

設計要素	内容
ドキュメントの上限	1ドキュメントは1 MiB まで。大きなデータは分割するか Cloud Storage に置いて参照を持つ
書き込みレート	1ドキュメントへの持続的な書き込みはおよそ毎秒1回が目安。カウンタは分散カウンタ(シャード)にする
インデックス	単一フィールドは自動。複合クエリには複合インデックスが必要
クエリの制約	クエリはインデックスに支えられ、返す件数に比例したコスト。範囲・不等号は1フィールドに限られる(複数不等号は制約あり)
非正規化	結合がないため、表示に必要なデータをドキュメントに複製して持つのが一般的

設計要素	内容
サブコレクション対配列	要素数が多い、または個別にクエリしたいならサブコレクション。少数で常に一緒に読むなら配列やマップ
シーケンシャルな ID	ドキュメント ID に単調増加値を使うとホットスポットになる。自動 ID(ランダム)を推奨

Firestore には Native モードと Datastore モードがあり、新規のモバイル・Web アプリケーションはリアルタイムリスナーとオフライン対応を持つ Native モードを選びます。「多数のクライアントに変更をリアルタイムで配信したい」は Firestore の Native モードのリスナーが正解です。

非構造化という語について、公式の topics は Bigtable と Firestore を非構造化データベースの例として挙げています。厳密にはこれらは半構造化ですが、試験の文脈では「固定スキーマを持たない、行ごとに列構成が違ってよいデータベース」という意味で読みます。

1-3-3 AlloyDB・Bigtable・Cloud SQL・Spanner・Cloud Storage の結果整合と強整合レプリケーションの含意 [1.3]

Understanding the implications of eventual and strongly consistent replication は、整合性モデルがアプリケーションの作りに与える影響を問います。

製品	既定の整合性	結果整合になる場面	アプリケーション側の対処
Cloud SQL	プライマリへの読み書きは強整合	リードレプリカからの読み取りはレプリケーション遅延の分だけ古い	直後に読み返す必要がある処理はプライマリから読む。レプリカ遅延の指標を監視する
AlloyDB	プライマリインスタンスは強整合	読み取りプールインスタンスは若干の遅延	同上。読み取り専用の分析はプールへ回す
Spanner	強整合(外部整合性)	ステイル読み取り(exact staleness / bounded staleness)を明示的に選んだ場合	レイテンシを下げたい読み取りでステイル読み取りを使う。書き込み直後の読み返しは強い読み取り
Bigtable	単一クラスタ内は行単位で強整合	レプリケーション構成で複数クラスタにまたがると結果整合	アプリケーションプロファイルで単一クラスタルーティングにすると読み書きの整合が保てる。マルチクラスタルーティングは可用性優先で結果整合
Cloud Storage	オブジェクトの書き込み・上書き・削除は強整合(書いた直後に読める)	バケット・オブジェクトの一覧(list)は結果整合になる場合がある。キャッシュされたオブジェクトは Cache-Control 次第で古い	一覧に依存する処理を避ける。更新頻度の高いオブジェクトは Cache-Control: no-cache を設定

製品	既定の整合性	結果整合になる場面	アプリケーション側の対処
Firestore	Native モードは強整合	Datastore モードのクエリの一部は結果整合	強整合が要るならキーによる取得または祖先クエリ

設計上の含意を3つ押さえます。第一に、リードレプリカへの読み取り振り分けは読み取りスループットを増やしますが、書き込み直後の読み返し(read-your-own-writes)が壊れます。ユーザーが更新した直後にその内容を表示する画面は、プライマリから読むか、更新結果をクライアント側で保持します。第二に、強整合を広い範囲で保つほど書き込みレイテンシが上がります。Spanner のマルチリージョン構成では、書き込みが地理的に離れたレプリカの合意を待つため、単一リージョンより遅くなります。第三に、Bigtable のマルチクラスタルーティングは可用性(クラスタ障害時に自動で別クラスタへ)と引き換えに整合性を落とします。「片方のクラスタに書いて、もう片方から即座に読むと古い値が返る」ことを前提に設計します。

1-3-4 Cloud Storage オブジェクトへのアクセスを許可する署名付き URL の作成 [1.3]

Creating signed URLs to grant access to Cloud Storage objects は、実装の細部まで問われます。

署名付き URL(signed URL)は、有効期限付きで特定のオブジェクトに対する特定の操作(GET、PUT など)を許可する URL です。URL を知っていれば Google アカウントを持たないユーザーでもアクセスできます。用途は2つです。第一に、限定公開のオブジェクトを一時的にダウンロードさせる場合。第二に、クライアントから直接 Cloud Storage にアップロードさせ、アプリケーションサーバーを経由させない場合です。後者はサーバーの帯域とレイテンシを

節約でき、Cloud Run のリクエストタイムアウトや上限サイズの制約も回避できるため、「大きなファイルのアップロードを効率化したい」の正解になります。

作成方法は次のとおりです。

- `gsutil signurl -d 10m <key.json> gs://bucket/object` または `gcloud storage sign-url`
- クライアントライブラリの generate_signed_url / getSignedUrl
- 署名にはサービスアカウントの秘密鍵、または IAM Service Account Credentials API の signBlob 権限 (roles/iam.serviceAccountTokenCreator)が必要です

鍵ファイルを置かずに署名したい場合、実行中のサービスアカウントに自分自身への roles/iam.serviceAccountTokenCreator を付け、クライアントライブラリの IAM 署名機能を使います。これが Cloud Run や GKE 上で署名付き URL を作るときの推奨形です。「キーファイルを配布せずに署名付き URL を発行したい」の答えです。

制約と注意点を挙げます。有効期限は最大7日(V4 署名の場合)です。署名付き URL は失効させることができないため、期限は必要最小限にします。署名したサービスアカウント自身がそのオブジェクトへの権限を持っていなければ、URL を使ってもアクセスは失敗します。つまり署名付き URL は権限を作り出すのではなく、既にある権限を一時的に貸し出す仕組みです。

似た仕組みとの区別も押さえます。

仕組み	用途
署名付き URL	期限付きで個別オブジェクトへの読み書きを外部に許可

仕組み	用途
署名付きポリシードキュメント	HTML フォームからのアップロードで、ファイルサイズや名前の条件を課す
allUsers への objectViewer 付与	恒久的な一般公開。期限も条件も付かない
Cloud CDN の署名付き URL / 署名付き Cookie	CDN のエッジで配信するコンテンツへのアクセス制御。多数のファイルをまとめて許可するなら署名付き Cookie

「動画の全セグメントに対して1回の認可でアクセスさせたい」なら署名付き Cookie、「1つのファイルを1回だけダウンロードさせたい」なら署名付き URL です。

1-3-5 分析・AI/ML ワークロードのための BigQuery へのデータ書き込み [1.3]

Writing data to BigQuery for analytics and AI/ML workloads は、開発者の立場から「どうデータを入れるか」を問います。取り込み経路を整理します。

経路	特徴	向く場面
バッチ読み込みジョブ(load job)	Cloud Storage の CSV/JSON/Avro/Parquet/ORC を読み込む。無料	日次・時間次のバッチ取り込み
BigQuery Storage Write API	gRPC のストリーミング書き込み。exactly-once(ストリームとオフセット)、コミット済み/保留中/バッファ型のストリーム	リアルタイム取り込みの標準。従来の insertAll(tabledata.insertAll)より高スループット・低コスト
従来のストリーミング挿入(insertAll)	REST の行単位挿入	既存実装。新規は Storage Write API を推奨

経路	特徴	向く場面
Dataflow	Apache Beam のパイプラインで変換しつつ書き込む	ETL/ELT、Pub/Sub からのストリーミング処理
Pub/Sub の BigQuery サブスクリプション	Pub/Sub から直接テーブルへ	変換不要でそのまま入れる場合。コードなし
Datastream	データベースの変更データキャプチャ(CDC)	Cloud SQL などからのレプリケーション
外部テーブル / BigLake	データを Cloud Storage に置いたままクエリ	取り込まずに分析したい

選択の決め手は「変換が要るか」「リアルタイム性が要るか」「コードを書きたいか」です。Pub/Sub のメッセージを加工せずそのまま BigQuery に入れるだけなら BigQuery サブスクリプションが最も単純で安価です。加工や結合、遅延データの扱いが要るなら Dataflow です。アプリケーションから直接書くなら Storage Write API です。

書き込み先の設計も押さえます。パーティション分割テーブル(取り込み時間、DATE/TIMESTAMP 列、または整数範囲)にすると、クエリのスキャン量が減り、コストと時間が下がります。クラスタリングを併用すると、指定列で並べ替えられブロックのプルーニングが効きます。「クエリのコストを下げたい」の答えは、まずパーティション分割とクラスタリング、次に SELECT * をやめて必要な列だけ選ぶこと、そして必要ならマテリアライズドビューです。パーティション有効期限を設定すると古いパーティションが自動削除され、保持ポリシーとコストの両方に効きます。

AI/ML の観点では、BigQuery ML により SQL だけでモデルの学習と推論ができます。CREATE MODEL でロジスティック回帰、ブースティングツリー、行列分解、時系列(ARIMA_PLUS)、k-means などを学習し、ML.PREDICT で推論

します。Vertex AI と連携して既存のモデルを呼び出したり、生成 AI のモデルを SQL から使うこともできます。「データを動かさずに機械学習をしたい」「SQL しか書けないアナリストにモデルを使わせたい」は BigQuery ML が正解です。

冪等性の扱いも実装上の要点です。Storage Write API のコミット済みストリームではオフセットを指定することで重複挿入を検出でき、exactly-once のセマンティクスが得られます。ストリーミングで重複が入りうる前提なら、書き込み後に重複排除するビュー(ROW_NUMBER で1件に絞る)を用意する方法もあります。「Pub/Sub の at-least-once 配信で BigQuery に重複が入る」への対処はこの2つです。

サンプルはここまでです

続き(第2章〜巻末)は、GCP PCD 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。

GCP PCD 学習用テキスト(無料サンプル)

版
2026年7月版(Google Cloud 公式 Professional Cloud Developer Exam Guide（2026-09-07 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