IT資格道場

# GCP PDO 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 DevOps Engineer 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 DevOps Engineer 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 DevOps Engineer |
| 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 —— 5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 | Google Cloud 組織の立ち上げと維持 | **20%** | 第1章 |

| セクション | 分野 | 採点対象に占める割合（公式） | 本書 |
|---|---|---|---|
| 2 | アプリケーション・インフラ・機械学習ワークロード向けのCI/CD パイプライン | **25%** | 第2章 |
| 3 | サイトリライアビリティエンジニアリング | **18%** | 第3章 |
| 4 | オブザーバビリティ実践の実装と問題のトラブルシューティング | **25%** | 第4章 |
| 5 | パフォーマンスとコストの最適化 | **12%** | 第5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 Google Cloud 組織の立ち上げと維持(ドメイン1・採点対象の 20%)

この章は「土地の区画整理」にあたります。アプリケーションを載せる前に、組織・フォルダ・プロジェクトをどう切り、ネットワークをどう共有し、誰にどの権限を与え、インフラをどうコードで管理し、開発者にどんな作業環境を配るかを決めます。

Professional 級で問われるのは「Organization とは何か」ではありません。**与えられた要件(組織構造・監査要件・規制・チーム数・移行の制約)に対して、どの構成を選ぶのが最適で、なぜ他の構成が不適か**です。本章は一貫してその形で書きます。



## 1-1 組織全体のリソース階層の設計 [1.1]

### 1-1-1 リソースの整理 —— application-centric・projects・folders [1.1]

Google Cloud のすべてのリソースは、次の木構造のどこかに属します。

```
Organization(組織)          ← Cloud Identity / Google Workspace ドメインに 1 対 1
  └ Folder(フォルダ)          ← 部門・環境・チーム。入れ子は最大 10 階層
      └ Project(プロジェクト) ← リソースを作る最小の入れ物。課金・API・クォータの境界
          └ Resource          ← VM・バケット・GKE クラスタなど
```

押さえるべき性質は 3 つです。

1. **プロジェクトは境界そのもの**です。課金アカウントの紐付け、API の有効化、クォータ、既定のネットワーク、ラベル、そして「削除すれば中身が丸ごと消える」という単位がプロジェクトです。
2. **IAM は上から下へ継承**します。組織で付けたロールはすべてのフォルダ・プロジェクト・リソースに効きます。継承は取り消せません(下位で「引き算」はできない)。制限したいときは IAM ではなく後述の**組織ポリシー (Organization Policy)** か、IAM Conditions を使います。
3. **組織ポリシーも上から下へ継承**しますが、こちらは下位で上書き(`inheritFromParent: false`)が可能です。ここが IAM と決定的に違う点で、頻出の取り違えです。


図 1-1　リソース階層と、IAM / 組織ポリシーの継承の違い

Organizing resources(リソースの整理)の設計軸は、実務では次の 3 つに集約されます。

| 設計軸 | 何を分けるか | 向く場面 | 弱点 |
|---|---|---|---|
| 環境別 | dev / staging / prod をプロジェクト分割 | 本番の権限と課金を確実に隔離したい | アプリ数が増えると横断の把握が難しい |
| 部門別(business-unit-centric) | 部門ごとにフォルダ | 部門で予算と権限が完全に独立 | 共通基盤の重複が起きやすい |

| 設計軸 | 何を分けるか | 向く場面 | 弱点 |
| --- | --- | --- | --- |
| **application-centric** | アプリケーション(サービス)ごとにフォルダを切り、その下に環境別プロジェクト | マイクロサービスが多く、チームがサービス単位で自律している | フォルダ数が増え、共通ポリシーの適用漏れに注意 |

DevOps Engineer 試験で推される既定は **application-centric** と環境別の組み合わせです。すなわち `Org → Folder(部門) → Folder(application) → Project(app-dev / app-staging / app-prod)` 。理由は、①デプロイの単位(アプリ)と権限の単位(プロジェクト)が一致するため CI/CD のサービスアカウント設計が単純になる、②本番プロジェクトだけに厳しい組織ポリシーと Binary Authorization を掛けられる、③アプリを廃止するときフォルダごと消せる、の 3 点です。

**誤答肢の切り方**: 「1 プロジェクトにすべてのアプリと全環境を入れ、ラベルで区別する」は必ず誤りです。ラベルは課金の可視化には効きますが、IAM・クォータ・API 有効化の境界にはならないため、開発者が本番リソースを操作できてしまいます。逆に「開発者 1 人につき 1 プロジェクト」は隔離としては最強ですが、組織ポリシーと課金の管理コストが跳ね上がるため、恒久環境としては不適です(後述の ephemeral environment として自動生成・自動削除するなら妥当)。

**プロジェクトの移動**: プロジェクトは `gcloud beta projects move` でフォルダ間を移動できます。移動すると継承される IAM と組織ポリシーが変わるため、移動前に実効ポリシーを確認します。組織のない(no-org)プロジェクトを後から組織に取り込むこともできますが、既存の IAM は残るので棚卸しが必要です。

### 1-1-2 共有ネットワーク(Shared networking)—— Shared VPC・VPC Network Peering・Private Service Connect [1.1]

複数プロジェクトに分けると、次に「ネットワークをどう共有するか」が問題になります。選択肢は 3 つあり、**どれを選ぶかを問う問題が頻出**です。

| 方式 | 何をするか | 管理主体 | 向く場面 |
|---|---|---|---|
| **Shared VPC** | ホストプロジェクトの VPC サブネットを、同じ組織内のサービスプロジェクトに貸す | ネットワーク管理者が一元管理 | 社内の多数プロジェクトで IP 空間と Firewall を集中管理したい |
| **VPC Network Peering** | 2 つの VPC を対等に接続。RFC1918 のプライベート通信 | 各 VPC の所有者 | 組織をまたぐ、または別組織のパートナーと繋ぐ |
| **Private Service Connect** | サービスの「入口」を消費側 VPC の内部 IP として公開 | 消費側が endpoint、提供側が service attachment | Google API やサードパーティ SaaS、自社共通サービスへプライベート接続 |

**Shared VPC** の要点。ホストプロジェクト(host project)は 1 つ、サービスプロジェクト(service project)は複数。サービスプロジェクトの VM はホストのサブネットに置かれますが、課金と IAM はサービスプロジェクト側です。付与するロールは組織レベルの `roles/compute.xpnAdmin`(Shared VPC を有効化しサービスプロジェクトを紐付ける)と、サブネット単位の `roles/compute.networkUser`(そのサブネットを使わせる)。サブネット単位で `networkUser` を絞ることで、「経理チームのプロジェクトは経理サブネットにしか VM を作れない」を実現できます。

**取り違え注意**: Shared VPC は**同一組織内でしか使えません**。組織をまたぐ、あるいは組織のないプロジェクトと繋ぐ要件が出たら Shared VPC は不適

で、VPC Network Peering が正解になります。

**VPC Network Peering** の要点。ピアリングは**推移しません**(A-B、B-C があっても A-C は通らない)。両側で明示的にピアリングを作る必要があり、CIDR が重複していると作成できません。ルートは交換されますが Firewall ルールは交換されないため、両側で許可が要ります。ハブ&スポークで多数の VPC を繋ぎたい場合、ピアリングでは非推移性がネックになるので、Network Connectivity Center か Shared VPC を検討します。

**Private Service Connect(PSC)** の要点。消費側 VPC に**内部 IP アドレスのエンドポイント**を作り、そこ宛の通信を提供側のサービスへ届けます。提供側は service attachment を公開します。特徴は、①CIDR の重複があっても使える(ピアリングと違う)、②片方向で、消費側から提供側へ向かうだけ、③Google API 向けには `Private Service Connect endpoints for Google APIs` として、`storage.googleapis.com` などを社内 IP で呼べる。Cloud SQL、Memorystore、Vertex AI、パートナー SaaS への接続で使います。

**選択の型**:「別組織の SaaS へプライベートに繋ぎたい」→ PSC。「自社の多数プロジェクトで IP を一元管理」→ Shared VPC。「合併した別組織の VPC と相互通信」→ VPC Network Peering。「IP が重複している 2 拠点を繋ぐ」→ ピアリング不可なので PSC か再設計。


図 1-2　共有ネットワークの 3 方式 —— 要件からの選び分け

## 1-1-3 マルチプロジェクトのモニタリングとロギング [1.1]

プロジェクトを分けると、監視とログが散らばります。Multi-project monitoring and logging(マルチプロジェクトのモニタリングとロギング)の道具は 2 系統で、**混同が頻出**です。

**モニタリング側** —— **metrics scope** Cloud Monitoring では、1 つのプロジェクトを **scoping project(スコープ設定プロジェクト)** に指定し、そこに他プロジェクトを **monitored project** として追加すると、1 つのダッシュボードとアラートで横断的に見られます。これを metrics scope と呼びます。制限は、①1 つのプロジェクトは複数の metrics scope に属せる、②1 metrics scope

あたりの監視対象プロジェクト数に上限がある、③scoping project を消すとスコープごと消える。運用の定石は、**監視専用のプロジェクトを 1 つ作って scoping project にする**ことです。アプリ用プロジェクトを scoping project にすると、そのアプリを廃止するとき監視構成まで巻き添えになります。

**ロギング側 —— aggregated sink** Cloud Logging では、組織またはフォルダに **aggregated sink(集約シンク)** を作り、配下の全プロジェクトのログを 1 か所へ流します。宛先は Cloud Logging バケット、BigQuery データセット、Cloud Storage バケット、Pub/Sub トピックです。組織レベルのシンク作成には `roles/logging.configWriter` が要り、`--include-children` を付けて配下を含めます。

**取り違え注意**:「全プロジェクトのログを 1 か所で検索したい」に対して metrics scope を選ぶのは誤りです。metrics scope はメトリクス(と一部の可視化)の話で、ログは aggregated sink が正解です。逆に「全プロジェクトの CPU 使用率を 1 枚のダッシュボードで」に aggregated sink を選ぶのも誤りです。

**ログの重複に注意**: 各プロジェクトの `_Default` シンクは残ったままなので、aggregated sink を足すとログは 2 か所に保存されます。コストを抑えるなら、プロジェクト側の `_Default` に除外フィルタ(exclusion)を掛けるか、組織ポリシーで既定バケットの保持期間を短くします。


図 1-3　マルチプロジェクトの監視とログ ── 2 系統を混同しない

## 1-1-4 IAM ロールと組織レベルのポリシー [1.1]

**Identity and Access Management (IAM) roles** は 3 種類です。

| 種類 | 例 | 使いどころ |
|---|---|---|
| 基本ロール(basic) | Owner / Editor / Viewer | 本番では使わない。プロジェクト全体に効きすぎる |
| 事前定義ロール(predefined) | `roles/run.developer`, `roles/logging.viewer` | 既定の選択。粒度が実務に合っている |
| カスタムロール(custom) | 組織またはプロジェクトに作成 | 事前定義で過不足があるときだけ |

Professional 級では「最小権限を実現する最も適切なロールはどれか」を問われます。**基本ロールが選択肢にあれば、まず誤答を疑う**のが定石です。特に Editor は `iam.serviceAccounts.actAs` を含むため、Editor を持つ人はサービスアカウントになりすまして権限を昇格できます。

**IAM Conditions** は、ロールの付与に条件式(CEL)を付けます。時刻(`request.time`)、リソース名(`resource.name.startsWith(...)`)、リソースタイプで絞れます。「業務時間だけ本番にアクセスさせたい」「`prod-` で始まるバケットだけ読ませたい」に使います。すべての API が条件に対応しているわけではない点は注意です。

**Policy Intelligence / IAM Recommender** は、過去 90 日の実際の使用状況から過剰な権限を検出し、より狭いロールを提案します。棚卸しの主役です。**Policy Troubleshooter** は「この人がこのリソースにこの権限を持つか、なぜ持つ/持たないか」を継承経路つきで説明します。**Policy Simulator** は変更前に影響をシミュレートします。

**組織レベルのポリシー(organization-level policies)** = Organization Policy Service は、IAM とは別の軸で「何をしてよいか」を制約します。IAM が「誰が」を決めるのに対し、組織ポリシーは「そもそも許すか」を決めます。DevOps で頻出の制約(constraint)は次のとおりです。

| 制約 | 効果 |
|---|---|
| `constraints/compute.vmExternalIpAccess` | VM の外部 IP を禁止/許可リスト化 |
| `constraints/iam.disableServiceAccountKeyCreation` | サービスアカウントキー(JSON)の作成を禁止 |
| `constraints/iam.allowedPolicyMemberDomains` | 外部ドメインのアカウント追加を禁止(ドメイン制限共有) |

| 制約 | 効果 |
| --- | --- |
| `constraints/gcp.resourceLocations` | リソースを作れるロケーションを限定(データレジデンシー) |
| `constraints/compute.requireOsLogin` | SSH を OS Login 必須にする |
| `constraints/compute.disableSerialPortAccess` | シリアルポート接続を禁止 |
| `constraints/run.allowedIngress` | Cloud Run の受信元を内部限定にする |

制約には**リスト制約**(許可/拒否の値を列挙)と**ブール制約**(有効/無効)があります。適用は組織・フォルダ・プロジェクトのどこでも可能で、下位で `inheritFromParent` を切って上書きできます。ただし上位で `denyAll` を掛けたものを下位で緩めるには、上位側の設計で例外を許す構成にしておく必要があります。

**custom organization policy** は、CEL で独自の制約を書ける仕組みで、対応リソース(GKE、Compute Engine など)で「マシンタイプは n2 系のみ」といったルールを作れます。

**誤答肢の切り方**:「全プロジェクトで外部 IP を禁止したい」に対し「Firewall ルールで egress を deny」は不適です。Firewall は通信の可否であって外部 IP の付与自体は止められず、プロジェクトごとに設定が要ります。組織ポリシー `compute.vmExternalIpAccess` を組織レベルで掛けるのが正解です。

### 1-1-5 サービスアカウントの作成と管理 [1.1]

**Creating and managing service accounts(サービスアカウントの作成と管理)** は DevOps の中核です。サービスアカウント(SA)は「人ではないアカウント」で、ID であると同時に**リソースでもある**という二重性がポイントです。

- **ID として**: 他のリソースに対してロールを付与される側。`roles/storage.objectViewer` を SA に付ける。
- **リソースとして**: SA 自体に IAM ポリシーがあり、「誰がこの SA を使ってよいか」を決める。`roles/iam.serviceAccountUser`(VM 等に付けて使う)、`roles/iam.serviceAccountTokenCreator`(短命トークンを発行してなりすます)。

**認証方式の優先順位**(この順で選ぶのが正解パターンです)。

1. **アタッチされたサービスアカウント**: VM、GKE、Cloud Run、Cloud Build に SA を紐付け、メタデータサーバー経由で自動的に短命トークンを取得する。鍵ファイルは存在しない。
2. **Workload Identity Federation**: Google Cloud の外(GitHub Actions、オンプレ Kubernetes、AWS、Azure)から、外部 IdP の OIDC/SAML トークンを Google の短命トークンに交換する。**鍵を配らずに外部から Google Cloud を操作する唯一の推奨手段**です。
3. **サービスアカウントキー(JSON)**: 最後の手段。漏洩すると失効させるまで永続的に使えるため、組織ポリシー `iam.disableServiceAccountKeyCreation` で禁止するのが推奨です。

**GKE の Workload Identity Federation for GKE** は、Kubernetes ServiceAccount と Google のサービスアカウントを結び付ける仕組みです。従来のノードのサービスアカウントを Pod が共有する方式(すべての Pod が同じ強い権限を持つ)を置き換えます。

**サービスアカウントの偽装 (impersonation)**: `gcloud --impersonate-service-account=SA_EMAIL` で、人間のアカウントが一時的に SA として実行できます。監査ログには「誰が偽装したか」が残るため、鍵配布より追跡性が高いのが利点です。CI/CD の権限昇格でもこの型を使います。

**設計の型**: パイプラインごと・環境ごとに SA を分けます。1 つの SA を全パイプラインで共有すると、権限の和集合になり最小権限が崩れます。またデフォルトのサービスアカウント(Compute Engine default service account は既定で Editor 相当)は本番で使わず、専用 SA に置き換えます。組織ポリシー `iam.automaticIamGrantsForDefaultServiceAccounts` を無効化しておくと、既定の強い付与を止められます。


図 1-4　サービスアカウントの二重性と、認証方式の優先順位

## 1-1-6 データレジデンシー [1.1]

**Data residency(データレジデンシー)** は「データを物理的にどの国・地域に置くか」の要件です。GDPR、国内の金融規制、公共調達などで問われます。DevOps 側で押さえるのは次の 4 点です。

1. `constraints/gcp.resourceLocations`：組織ポリシーで、リソースを作成できるロケーションを `in:asia-northeast1-locations` のように限定します。既存リソースは移動しないので、**新規作成の抑止**である点に注意。
2. **ロケーションの粒度**：ゾーン( `asia-northeast1-a` )、リージョン( `asia-northeast1` )、マルチリージョン( `asia` )、グローバル。Cloud Storage のマルチリージョンは複数国にまたがることがあるため、厳格なデータレジデンシー要件ではリージョナルを選びます。
3. **ログとメトリクスもデータ**：Cloud Logging のログバケットは作成時にリージョンを指定でき(既定は `global` )、後から変更できません。データレジデンシー要件があるなら、`_Default` と `_Required` を含めて**リージョン指定のバケットを先に作る**設計が必要です。Cloud Logging の**リージョン化されたログルーティング**と、Assured Workloads による強制も選択肢です。
4. **Assured Workloads**：規制体系(EU 圏内、日本のリージョン限定など)を選ぶと、データ所在地・サポート担当者の所在地・暗号鍵の制御をまとめて強制するフォルダを作れます。「規制要件を組織全体に確実に効かせたい」なら Assured Workloads、「単にリージョンを縛りたいだけ」なら `gcp.resourceLocations` が正解です。

**CMEK と鍵の所在**：Cloud KMS の鍵も所在地を持ちます。データはリージョン内でも鍵がグローバルだと要件を満たさない場合があるため、鍵リングのロケーションを合わせます。

---

# 1-2 インフラストラクチャの管理 [1.2]

## 1-2-1 IaC ツールとマネージドサービス [1.2]

**Infrastructure-as-code tooling and managed services** の登場人物を整理します。

| ツール | 位置づけ | 状態(state)の持ち方 | 向く場面 |
|---|---|---|---|
| **Terraform** | 宣言的 IaC のデファクト。HCL で記述 | tfstate をリモートバックエンド(GCS)に置く | マルチクラウド・大半の場面 |
| **Infrastructure Manager** | Google のマネージド Terraform 実行サービス | Google 側が state を保持 | state 管理と実行基盤を自前で持ちたくない |
| **Cloud Foundation Toolkit** | Google 公式の Terraform モジュール/ブループリント集 | Terraform に準拠 | 組織の初期構築を推奨構成で立ち上げる |
| **Config Connector** | Kubernetes のカスタムリソースで Google Cloud リソースを管理 | Kubernetes の etcd と継続的な調整ループ | すでに GKE 中心で、K8s の作法に寄せたい |
| **GitOps** | Git を唯一の正とし、差分をエージェントが継続的に適用する運用手法 | Git + クラスタの実状態 | 多数クラスタの構成ドリフトを継続的に是正 |
| **Helm** | Kubernetes マニフェストのテンプレート/パッケージ管理 | リリース情報をクラスタ内に保存 | 同じアプリを環境別の値で配る |

**Infrastructure Manager(Infra Manager)** の要点。Terraform 構成を Cloud Storage、Git リポジトリ、ローカルディレクトリから受け取り、**一時的な Cloud Build 環境**で `terraform init` → `terraform validate` → `terraform apply` を実行します。ログは Cloud Storage にストリーミングされ、一過性の失敗は自動リトライされます。管理単位は **deployment** で、更新のたびに **revision** が積まれます。実行に使うサービスアカウントを指定でき、そのサービスアカウントの権限がそのままインフラ変更の権限になります。**preview**(適用前の差分確認)に対応し、Terraform のバージョンを deployment ごとに指定できます。

**注意点**(公式ドキュメント記載): Terraform の構成値に個人情報・機微情報を含めないこと。構成は保存されログにも出るためです。機微な値は Secret Manager 参照にします。

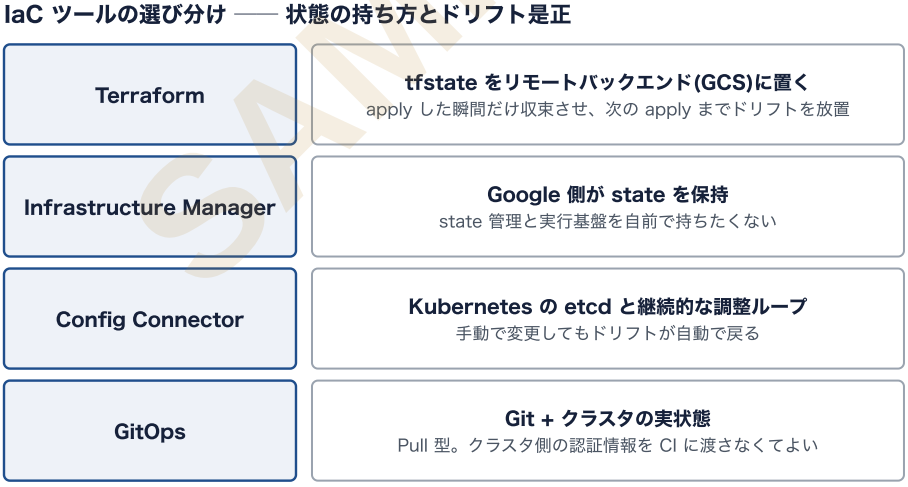
**Terraform を自前で回す場合の定石。**

- state は GCS バックエンドに置き、バケットは**オブジェクトのバージョニングを有効化**する(state 破損からの復旧経路)。
- state ロック。GCS バックエンドはオブジェクトの世代番号でロックします。
- 環境ごとに state を分ける。`terraform workspace` より**ディレクトリ分割**が推奨されることが多いのは、環境ごとにバックエンド・権限・変数を独立させられるためです。
- `terraform plan` の結果を人がレビューし、`apply` はマージ後に自動、という **plan/apply 分離**が CI/CD の基本形。
- サービスアカウント偽装(`impersonate_service_account`)で、実行者は plan 権限のみ、apply は昇格 SA、という分離ができます。

**Config Connector** の要点。GKE のアドオンとして入れると、`SQLInstance`、`PubSubTopic`、`IAMPolicyMember` といった CRD で Google Cloud リソースを宣

言できます。Kubernetes のコントローラが継続的に実状態を望ましい状態へ寄せるため、**手動で変更してもドリフトが自動で戻る**のが Terraform との最大の違いです。Terraform は `apply` した瞬間だけ収束させ、次の apply までドリフトを放置します。「クラスタ外のリソースも含めて継続的にドリフトを是正したい」→ Config Connector、「一度きりの構築とレビュー可能な差分」→ Terraform、という切り分けになります。

**GitOps** の要点。Git を Single Source of Truth とし、クラスタ内のエージェント(Config Sync、Argo CD、Flux)が Git をポーリングして差分を適用します。特徴は、①**Pull 型**なのでクラスタ側の認証情報を CI に渡さなくてよい、②Git のレビューと履歴がそのまま変更管理になる、③ロールバックは Git の revert。Google の実装は **Config Sync**(GKE Enterprise / fleet の機能)です。




一度きりの構築とレビュー可能な差分 → Terraform。継続的にドリフトを是正したい → Config Connector。
Google の GitOps 実装は Config Sync(GKE Enterprise / fleet の機能)。
Infrastructure Manager は一時的な Cloud Build 環境で terraform init → validate → apply を実行する。

図 1-5　IaC ツールの選び分け —— 状態の持ち方とドリフト是正

**Helm** の要点。`values.yaml` を環境ごとに差し替えて同じ Chart を配ります。Kustomize との比較は 1-3 で扱います。

### 1-2-2 推奨プラクティスとブループリントに沿ったインフラ変更 [1.2]

**Making infrastructure changes using Google-recommended practices and blueprints(Google 推奨プラクティスとブループリントを用いたインフラ変更)** の骨子です。

**ブループリント**とは、Google が公開する「この構成なら推奨に沿っている」という実装済みテンプレートです。中心は **Cloud Foundation Toolkit** と、その上位の **enterprise foundation blueprint**(組織・フォルダ・共有 VPC・ログ集約・組織ポリシー・CI/CD をまとめて構築する Terraform 一式)です。ゼロから設計するのではなく、ブループリントを起点に組織要件で差分を当てるのが推奨の進め方です。

**インフラ変更の推奨フロー**。

1. 変更を **Git のプルリクエスト**として出す。インフラも「レビューされるコード」にする。
2. CI で `terraform fmt -check`、`terraform validate`、**静的解析**(tfsec / Checkov / `gcloud terraform vet`)を回す。`gcloud terraform vet` は、Terraform の plan を組織ポリシー的な制約(Policy Library の constraint)に照らして違反を検出します。
3. `terraform plan` の出力を PR にコメントし、人がレビューする。
4. マージで **staging に自動適用**、本番は**手動承認**を挟む。
5. 適用後に検証(スモークテスト)を行い、失敗したら Git を revert して再適用する。

**変更の安全化テクニック**。

- **段階適用**: 影響範囲の小さいプロジェクト・リージョンから順に適用する。
- `prevent_destroy` : 消えると致命的なリソース(本番 DB、state バケット)に `lifecycle { prevent_destroy = true }` を付ける。plan で破壊が出たらエラーになります。
- `terraform import` / **インポートブロック**: 手作業で作られた既存リソースを state に取り込み、以後 IaC の管理下に置く。
- **ドリフト検出**: 定期的に `terraform plan -detailed-exitcode` を Cloud Scheduler + Cloud Build で回し、差分が出たら通知する。
- **モジュール化とバージョン固定**: モジュールソースに Git タグを指定し、プロバイダも `required_providers` で固定する。固定しないと、無関係な変更で意図しない差分が出ます。

**誤答肢の切り方**:「本番の設定を急いで直したいのでコンソールで変更した」は、たとえ即時性の要件があっても Professional 級では誤答です。正解は「IaC を直して適用する」「緊急時はブレークグラス手順を用い、直後に IaC へ反映してドリフトを解消する」。

## 1-2-3 スクリプトによる自動化 —— Python・Go [1.2]

**Automation with scripting (e.g., Python, Go)** は、宣言的 IaC でカバーしきれない「手続き的な作業」を埋める役割です。

**どちらを使うか**。

| 言語 | 強み | 典型用途 |
| --- | --- | --- |
| **Python** | クライアントライブラリが厚く、記述量が少ない | 運用スクリプト、Cloud Run functions、データ整形、レポート生成 |
| **Go** | 単一バイナリで配布でき、起動が速く並行処理に強い | CLI ツール、Kubernetes のカスタムコントローラ、高頻度に呼ばれる関数 |

**書き方の原則。**

- **クライアントライブラリを使う**(`google-cloud-*` for Python、`cloud.google.com/go/*` for Go)。`gcloud` をシェルから叩くより、エラーハンドリングとリトライが確実です。
- **認証は Application Default Credentials (ADC)** に任せる。ローカルは `gcloud auth application-default login`、Google Cloud 上は付与されたサービスアカウントが自動で使われます。コードに鍵のパスを書かない。
- **冪等性**: 同じスクリプトを 2 回流しても結果が変わらないようにする。作成前に存在確認する、`If-Match` /etag を使う、など。
- **指数バックオフ付きリトライ**: `429 RESOURCE_EXHAUSTED`、`503 UNAVAILABLE` は再試行対象。`400`、`403` は再試行しても無駄。
- **実行基盤**: 定期実行は Cloud Scheduler → Pub/Sub → Cloud Run functions、あるいは Cloud Run jobs。イベント駆動は Eventarc。長時間の一括処理は Cloud Run jobs か Batch。
- **監査**: スクリプトが行った変更は Cloud Audit Logs に「そのサービスアカウントが行った」として残ります。人の操作と区別するため、自動化専用の SA を使います。

**取り違え注意**:「大量の VM を毎日棚卸しして停止したい」に対して「Terraform で毎日 apply」は不適です。Terraform は望ましい状態の宣言であって、動的な条件判断(最終利用日で判定など)には向きません。Recommender の idle VM insight を読み、条件に合うものを停止する Python スクリプトを Cloud Scheduler で回すのが妥当です。

---

## 1-3 Google Cloud・ハイブリッド・マルチクラウド環境での CI/CD アーキテクチャスタックの設計 [1.3]

### 1-3-1 Cloud Build による継続的インテグレーション [1.3]

**Continuous integration (CI) with Cloud Build** は、この試験の中核サービスです。

**構造**: `cloudbuild.yaml` に **steps** を並べ、各 step が **builder イメージ**(コンテナ)として実行されます。step 同士は `/workspace` ディレクトリを共有します。既定は逐次実行で、`waitFor: ['-']` を書くと並列に走らせられます。

```
steps:
- name: 'gcr.io/cloud-builders/go'
  args: ['test', './...']
- name: 'gcr.io/cloud-builders/docker'
  args: ['build','-t','${_REGION}-
docker.pkg.dev/$PROJECT_ID/app/api:$SHORT_SHA','.']
- name: 'gcr.io/cloud-builders/docker'
  args: ['push','${_REGION}-
docker.pkg.dev/$PROJECT_ID/app/api:$SHORT_SHA']
options:
  logging: CLOUD_LOGGING_ONLY
  machineType: 'E2_HIGHCPU_8'
```

**押さえる要素。**

- **代入変数**: `$PROJECT_ID` 、`$BUILD_ID` 、`$COMMIT_SHA` 、`$SHORT_SHA` 、`$BRANCH_NAME` 、`$TAG_NAME` 。ユーザー定義は `_` 始まり( `_REGION` )。
- **artifacts / images**: ビルド成果物を Artifact Registry や Cloud Storage へ自動で上げる宣言。
- **timeout**: ビルド全体と step ごとに設定。既定 10 分は長いビルドで足りず、失敗の典型原因です。
- **サービスアカウント**: 既定の Cloud Build サービスアカウントではなく、**ユーザー指定のサービスアカウント**を使うのが推奨です(最小権限)。既定 SA は広い権限を持ちがちで、そのまま本番デプロイに使うと過剰付与になります。

- **ログの保存先**: 既定はユーザー指定バケット。ユーザー指定 SA を使う場合は `logging: CLOUD_LOGGING_ONLY` などの明示が必要になることがあります。
- **キャッシュ**: Kaniko キャッシュ、または前回イメージを pull して `--cache-from` に渡す。ビルド時間短縮の定番。
- **private pools(プライベートプール)**: 既定のビルドワーカーは Google 管理の共有プールで、顧客 VPC には入れません。**VPC 内のプライベートリソース(オンプレの Git、内部 IP のみの GKE、Cloud SQL のプライベート IP)にアクセスするビルドは private pool が必須**です。頻出の判断ポイントです。
- **Cloud Build repositories(2nd gen)**: GitHub、GitHub Enterprise、GitLab(セルフホスト含む)への接続を Secret Manager 経由で管理します。

**トリガー(CI/CD pipeline triggers)**: push、pull request、タグ、手動、Pub/Sub、Webhook。ブランチ・タグを正規表現で絞れます。含めるファイル・除外するファイル(`includedFiles` / `ignoredFiles`)を指定すると、モノレポで関係ない変更のビルドを避けられます。

**ビルドの品質ゲート**: ユニットテスト → 静的解析 → コンテナイメージのビルド → 脆弱性スキャン(Artifact Analysis) → 署名(Binary Authorization の attestation) → Artifact Registry へ push、という並びが典型です。

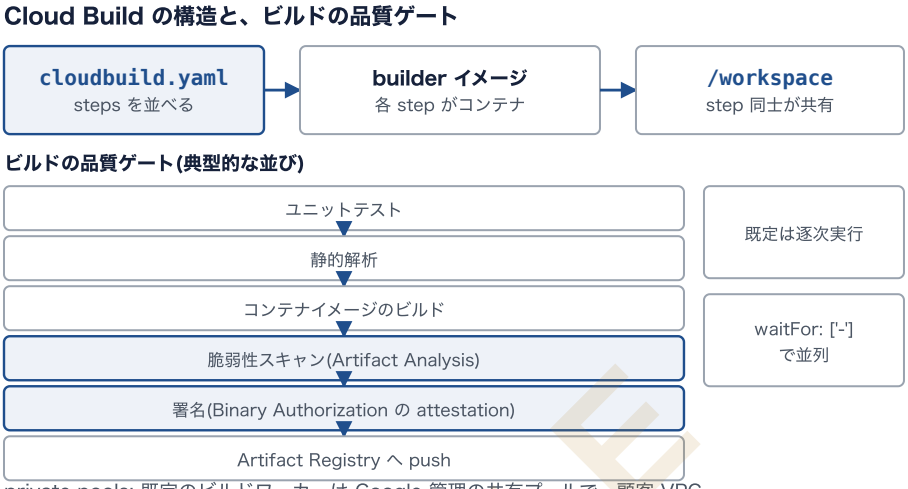



図 1-6　Cloud Build の構造と、ビルドの品質ゲート

## 1-3-2 Cloud Deploy による継続的デリバリー —— Skaffold と Kustomize [1.3]

### Continuous delivery (CD) with Cloud Deploy, including Kustomize and Skaffold。

**Cloud Deploy** はマネージドの継続的デリバリーサービスです。概念の階層を正確に覚えます。

| 概念 | 意味 |
|---|---|
| **delivery pipeline(デリバリーパイプライン)** | ターゲットの並び(昇格順序)を定義した最上位リソース |

| 概念 | 意味 |
| --- | --- |
| **target(ターゲット)** | デプロイ先環境。GKE、Cloud Run、GKE Enterprise の attached cluster、custom target |
| **release(リリース)** | 特定のソース/イメージから**レンダリング済みマニフェスト**を作った不変のスナップショット |
| **rollout(ロールアウト)** | ある release をある target へ実際に適用する行為 |
| **phase / job** | rollout の中の段階と個別作業(deploy、verify、predeploy/postdeploy) |

**動作の 2 段階。**

1. **render(レンダリング)**: `skaffold render` により、target ごとにマニフェストを確定させる。ここでイメージタグが解決され、環境差分が当たる。
2. **deploy(デプロイ)**: 確定済みマニフェストを target に適用する。

**なぜ render と deploy を分けるか**が Professional 級の問いです。release 作成時に全ターゲット分のマニフェストを確定させておくことで、**dev で検証したものと同じ成果物が prod に上がることを保証**します。「prod へ昇格するときに再ビルドする」設計は、ビルド間の非決定性(依存の更新、`latest` タグ)によって dev と prod で違う成果物になるリスクがあるため誤りです。

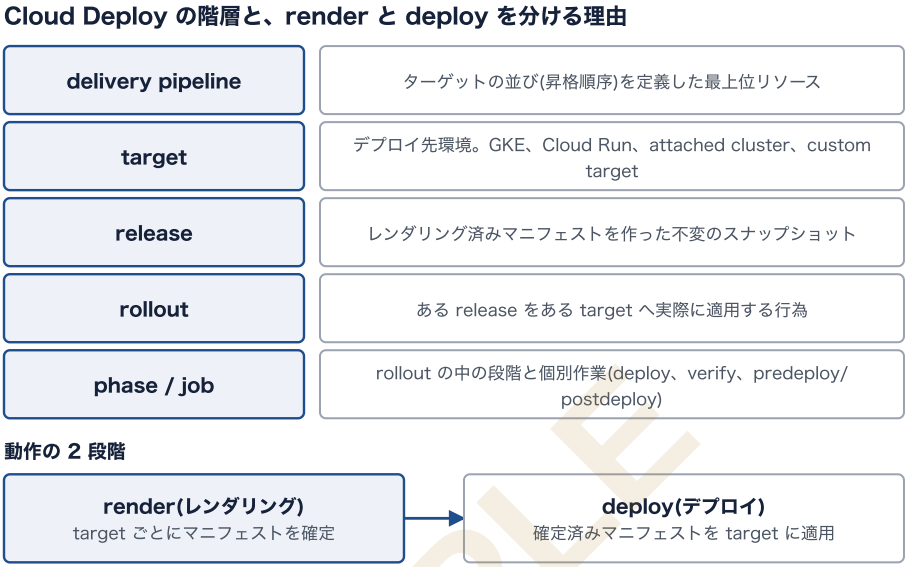



図 1-7　Cloud Deploy の階層と、render と deploy を分ける理由

**Skaffold** は Cloud Deploy が内部で使うツールで、`skaffold.yaml` に「どうビルドし、どうマニフェストを生成し、どう検証するか」を書きます。ローカル開発では `skaffold dev` でファイル変更を検知して再ビルド・再デプロイするループを回せます。

**Kustomize** はマニフェストの環境差分をパッチで当てる仕組みです。`base/` に共通マニフェストを置き、`overlays/dev`、`overlays/prod` で `namePrefix`、`replicas`、`images`、`patchesStrategicMerge` を上書きします。

**Kustomize と Helm の使い分け。**

| | Kustomize | Helm |
|---|---|---|
| 方式 | 素の YAML にパッチを重ねる | テンプレート言語で YAML を生成 |
| 学習コスト | 低い(YAML のまま) | 高い(テンプレート構文) |
| 配布 | 配布の仕組みは持たない | Chart として repository で配布・バージョン管理 |
| 向く場面 | 自社アプリの環境差分 | サードパーティ製品の導入、外部への配布 |

Cloud Deploy はどちらもサポートします( `skaffold.yaml` の `manifests` で `kustomize` または `helm` を指定)。

**Cloud Deploy の主要機能。**

- **canary deployment**: `strategy.canary` で `percentages: [25, 50, 100]` のように段階を定義。GKE では Gateway API または Service ベース、Cloud Run では traffic split を使います。
- **approval(承認)**: target に `requireApproval: true` を付けると、その target への rollout 開始前に人の承認が必要になります。**本番ターゲットに承認を掛けるのが定石**です。
- **deploy hooks(predeploy / postdeploy job)**: DB マイグレーション、キャッシュのウォームアップ、外部への通知などをフェーズ前後で実行します。
- **verify(検証)**: デプロイ後に `skaffold verify` でスモークテストを走らせ、失敗したら rollout を失敗扱いにします。
- **rollback(ロールバック)**: 最後に成功した release に対して rollout を起こし直します。再ビルドは不要で、レンダリング済みマニフェストが残ってい

るため高速です。

- **automation(自動化)**: `promote-release` ルールで「dev で成功して 10 分経ったら staging へ自動昇格」、`advance-rollout` ルールで「canary の次フェーズへ自動前進」を定義できます。
- **parallel deployment / multi-target**: 1 つの target を「複数の子ターゲットの集合」として定義し、複数リージョン・複数クラスタへ同時にデプロイします。
- **custom target**: GKE / Cloud Run 以外(Terraform、他社基盤、オンプレ)へ、独自のコンテナで render/deploy を実装して届けます。**マルチクラウドへの CD を Cloud Deploy で統一したい**という要件はここに当たります。
- **通知**: render、deploy、承認要求などのイベントを Pub/Sub に発行し、Slack やチケットへ流せます。

### 1-3-3 Artifact Registry の構成 [1.3]

**Artifact Registry configuration**。Container Registry の後継で、コンテナイメージだけでなく Maven、npm、Python、Go、Apt、Yum、Helm chart、汎用ファイルを扱います。

**構成要素**。

- **repository(リポジトリ)**: 形式(docker、maven など)、モード、ロケーション(リージョン/マルチリージョン)を持つ単位。ホスト名は `LOCATION-docker.pkg.dev/PROJECT/REPO/IMAGE:TAG`。
- **モード 3 種**:
  - **standard**: 自分で push する通常のリポジトリ。

    - **remote**: Docker Hub、Maven Central、PyPI などの**上流をキャッシュする代理**。外部への依存を減らし、レート制限とネットワーク断の影響を緩和します。
    - **virtual**: 複数のリポジトリ(standard と remote)を**1 つのエンドポイントに束ねる**。優先順位を付けて、社内版を先に、なければ remote を、という解決ができます。

- **IAM**: `roles/artifactregistry.reader` (pull)、`roles/artifactregistry.writer` (push)、`roles/artifactregistry.admin` 。**リポジトリ単位で付与できる**ため、本番リポジトリへの push を CI の SA だけに限定できます。
- **CMEK**: リポジトリ作成時に顧客管理鍵を指定できます(後から変更不可)。
- **immutable tags(タグの不変化)**: 有効にすると既存タグの上書きを禁止できます。`v1.2.3` が別のイメージにすり替わる事故を防ぐため、本番用リポジトリでは有効にするのが推奨です。
- **cleanup policies(クリーンアップポリシー)**: 「タグなしイメージを 30 日で削除」「最新 10 バージョンを保持」といった条件で自動削除します。**dry run モードで先に影響を確認**してから有効化するのが定石です。ストレージコスト削減の主役です。
- **Artifact Registry と Artifact Analysis の連携**: push 時に自動で脆弱性スキャンが走ります(2-4 で詳述)。

**設計の型**: 環境ごとにリポジトリを分け( `dev-images` / `prod-images` )、CI は dev へ push、承認済みイメージだけを prod リポジトリへコピーする、という**プロモーション型**が最小権限に合います。あるいは 1 リポジトリで運用し、

Binary Authorization の attestation で本番デプロイ可否を制御する型もあります。

**取り違え注意**:「Docker Hub のレート制限でビルドが不安定」→ remote repository。「複数のチームのリポジトリを開発者に 1 つの URL で見せたい」→ virtual repository。「古いイメージでストレージ費用が膨らむ」→ cleanup policy。


図 1-8　Artifact Registry のリポジトリ モード 3 種と、選ぶ場面

## 1-3-4 広く使われるサードパーティツール —— Git・Jenkins・Argo CD・Packer・kpt [1.3]

**Widely used third-party tooling**。Google Cloud のマネージドサービスだけで完結しない現場は多く、既存資産をどう繋ぐかが問われます。

**Git**: ブランチ戦略が設計判断として出ます。

| 戦略 | 内容 | 向く場面 |
|---|---|---|
| trunk-based development | 短命ブランチを 1 日以内に main へマージ。未完成機能は feature flag で隠す | 高頻度デプロイ、CI/CD が整っている |
| GitFlow | develop / release / hotfix ブランチを長期に維持 | リリースが低頻度でバージョン管理が厳格 |
| 環境ブランチ | dev / staging / prod ブランチを持つ | 素朴だが、ブランチ間のドリフトが起きやすく非推奨寄り |

Google が推すのは **trunk-based development** です。「デプロイ頻度を上げたい」「マージ競合が多い」という課題には、長寿命ブランチを廃してトランクベースへ、が定番の答えです。

**Jenkins**: 既存の Jenkins 資産を Google Cloud で動かす場合、GKE 上に Jenkins を置き、**Kubernetes plugin で agent を Pod として動的に起動**する構成が推奨です。常時起動の VM agent より安く、スケールします。認証は Workload Identity Federation for GKE でサービスアカウントキーを排除します。Jenkins から Google Cloud を操作する場合も、鍵ファイルではなく Workload Identity Federation を使います。

**Argo CD**: GitOps の代表実装。クラスタ内のエージェントが Git を監視し、差分を適用します。Cloud Deploy との違いは、**Argo CD は「望ましい状態への継続的同期」、Cloud Deploy は「昇格順序を持つリリースの前進」**という点です。Argo CD は自動同期・自己修復(self-heal)でドリフトを戻し、Cloud Deploy は release/rollout という監査可能な単位と承認ゲートを持ちます。Google のマネージド GitOps は **Config Sync**(fleet 機能)で、Argo CD はそれを自前で運用する選択肢に当たります。

**Packer**: VM イメージ(golden image)を宣言的にビルドするツールです。Compute Engine 向けには、ベースイメージから VM を起動 → プロビジョナ(shell、Ansible)を実行 → **カスタムイメージとして保存**します。これを**immutable infrastructure(不変インフラ)** に使います。すなわち、稼働中の VM に設定変更を当てるのではなく、新しいイメージを焼いてマネージドインスタンスグループのインスタンステンプレートを差し替え、rolling update で入れ替えます。Cloud Build の step から Packer を呼んで「イメージビルドの CI」を組むのが典型です。

**kpt**: Kubernetes の構成パッケージを Git から取得し、**関数(kpt functions)で変換して適用する**ツールです。テンプレート言語を使わず、実際の YAML を操作する点が Helm と異なります。Config Sync や Cloud Foundation Toolkit の一部で使われます。「テンプレートを増やさずにパッケージ化された構成を配りたい」という文脈で出ます。

### 1-3-5 CI/CD ツールのセキュリティ [1.3]

**Security of CI/CD tooling**。パイプラインは本番への最短経路なので、攻撃対象そのものです。

**脅威と対策の対応表**。

| 脅威 | 対策 |
|---|---|
| ビルド SA の権限過剰 | パイプライン専用 SA + 環境別の最小権限。既定 SA を使わない |
| 鍵ファイルの漏洩 | サービスアカウントキーを組織ポリシーで禁止し、Workload Identity Federation に置換 |

| 脅威 | 対策 |
| --- | --- |
| 依存パッケージの汚染 | Artifact Registry の remote repository で上流を固定・キャッシュ。ロックファイルとハッシュ検証 |
| 改ざんされたイメージのデプロイ | Binary Authorization で署名済みイメージのみ許可 |
| ビルド環境からの情報流出 | private pool + VPC Service Controls。外部ネットワークアクセスを制限 |
| パイプライン定義の改ざん | `cloudbuild.yaml` をコードレビュー必須にし、CODEOWNERS で保護 |
| シークレットのログ出力 | Secret Manager から実行時に取得。ビルドログに出さない( `secretEnv` を使う) |
| 誰が何をデプロイしたか不明 | Cloud Audit Logs + Cloud Deploy の rollout 履歴 |

**環境分離の原則**: CI(ビルド)と CD(デプロイ)の権限を分けます。ビルド用 SA は Artifact Registry への push 権限のみ、デプロイ用 SA は本番クラスタへの適用権限のみ、とし、1 つの SA に両方を持たせません。さらに **CI/CD 自体を専用プロジェクトに置き**、アプリの本番プロジェクトとは別にします。

**VPC Service Controls** は、Artifact Registry、Cloud Build、Cloud Storage などを境界(perimeter)で囲み、境界外への API 経由のデータ持ち出しを止めます。「ビルドがイメージを外部プロジェクトへ push できてしまう」を構造的に塞ぐ手段です。

## 1-4 複数環境(ステージング・本番等)の管理 [1.4]

### 1-4-1 一時的な環境の管理 [1.4]

**Managing ephemeral environments(一時的な環境の管理)** は、プルリクエストごとに使い捨ての環境を作り、マージまたはクローズで自動的に消す運用です。

**なぜ必要か**: 共有のステージング環境は「誰かの壊れた変更で全員が止まる」ボトルネックになります。PR 単位で独立した環境を持てば、レビュアーは実物を触って確認でき、テストは他の変更に干渉されません。

**実装パターン**。

| 粒度 | 実装 | 長所 / 短所 |
| --- | --- | --- |
| Namespace 単位 | 共有 GKE クラスタに `pr-123` namespace を作る | 速くて安い / クラスタ全体の設定(CRD、ノードプール)は共有され、完全な隔離にならない |
| クラスタ単位 | PR ごとに GKE Autopilot クラスタを作る | 隔離が強い / 作成に時間がかかり高価 |
| プロジェクト単位 | PR ごとにプロジェクトを自動作成 | 最も強い隔離。IAM とクォータまで検証できる / 作成・削除が重く、組織ポリシーの整備が要る |
| リビジョン単位 | Cloud Run の tag 付きリビジョンで `pr-123---service-xxx.run.app` を発行 | 極めて軽量・低コスト / コンテナ 1 個で完結するアプリ向け |

**必ず組み込む要素**。

- **自動削除**: PR クローズのイベント(GitHub webhook → Cloud Build トリガー)で環境を破棄。加えて**TTL による保険**(作成日ラベルを見て N 日で消す定期ジョブ)を入れます。消し忘れが FinOps の主要な浪費源だからです。
- **命名と識別**: `pr-<番号>` を namespace / プロジェクト ID / ラベルに一貫して付け、費用配分とクリーンアップの鍵にします。
- **データ**: 本番データをコピーしない。マスクした少量のシードデータを流し込みます。個人情報の複製は規制違反になり得ます。
- **クォータ**: 一時環境の作成でプロジェクトのクォータを食い潰さないよう、専用プロジェクト/フォルダに閉じ込め、上限を設定します。

**Cloud Deploy との関係**: 一時環境は Cloud Deploy の target として登録するより、Cloud Build から直接 `skaffold run` で立てることが多いです。Cloud Deploy は「決まった昇格順序を持つ恒久環境」に向いています。

### 1-4-2 構成とポリシーの管理 [1.4]

**Managing configuration and policy(構成とポリシーの管理)**。環境が増えるほど「環境差分をどう表現し、逸脱をどう防ぐか」が問題になります。

**構成(configuration)の管理**。

- 環境差分は **Kustomize の overlay** か **Helm の values** で表現し、環境ごとにマニフェストを丸ごと複製しない。複製するとドリフトが必ず起きます。
- アプリの設定値は **ConfigMap / Parameter Manager**、機密値は **Secret Manager** に置き、イメージには焼き込まない。同じイメージが全環境で動くことが「dev で検証したものが prod で動く」保証の前提です。

- **Config Sync** を使うと、Git のディレクトリ構成をそのまま複数クラスタへ同期でき、`ClusterSelector` で環境ごとに適用対象を切り替えられます。

**ポリシー(policy)の管理**。3 層で考えます。

| 層 | 道具 | 例 |
|---|---|---|
| Google Cloud のリソース | **Organization Policy**(制約) | 外部 IP 禁止、リージョン限定、SA キー禁止 |
| Kubernetes のマニフェスト | **Policy Controller**(Gatekeeper ベース。GKE Enterprise の機能) | 特権コンテナ禁止、リソース制限必須、許可レジストリのみ |
| デプロイされるイメージ | **Binary Authorization** | 署名済みイメージのみ許可 |

**Policy Controller** の要点。制約テンプレート(ConstraintTemplate)と制約(Constraint)で、Admission Webhook としてマニフェストを検査します。**dryrun / warn / deny** の enforcementAction を持ち、まず dryrun で違反件数を測り、是正してから deny に上げるのが安全な導入手順です。既製のポリシーバンドル(CIS ベンチマーク、PCI-DSS、NIST)を適用できます。さらに **CI 側でも同じポリシーを事前チェック**(`kpt` / `gator` によるシフトレフト)すると、クラスタに届く前に落とせます。

**誤答肢の切り方**:「本番クラスタに特権コンテナを作らせたくない」に対して「IAM で開発者からクラスタ管理者ロールを外す」は不十分です(正当なデプロイ権限を持つ CI が作れてしまう)。Policy Controller の制約で構造的に禁止するのが正解です。

### 1-4-3 エンタープライズ規模の GKE クラスタ管理 —— fleets [1.4]

**Managing Google Kubernetes Engine (GKE) clusters across an enterprise (e.g., fleets)。**

**fleet(フリート)** は、複数の Kubernetes クラスタを 1 つの論理グループとして扱う仕組みです。クラスタは GKE だけでなく、他クラウドの GKE(GKE on AWS/Azure)や、**attached cluster** として登録した任意の CNCF 準拠クラスタも含められます。fleet は 1 プロジェクトに 1 つ(fleet host project)です。

**fleet が前提にする「sameness(同一性)」**。fleet 内では、同名の namespace は同じチームのものとみなされます。この前提の上に fleet 機能が乗ります。

| fleet 機能 | 役割 |
|---|---|
| **Config Sync** | Git から全クラスタへ構成を継続的に同期(GitOps) |
| **Policy Controller** | 全クラスタへポリシーを一括適用し違反を集約表示 |
| **Cloud Service Mesh** | クラスタ横断のサービス間通信、mTLS、トラフィック管理、SLO |
| **Multi Cluster Ingress / Gateway** | 複数クラスタ・複数リージョンへ 1 つの外部エンドポイントから振り分け |
| **Multi-cluster Services (MCS)** | クラスタをまたいだ Service の名前解決 |
| **fleet team management** | fleet namespace とチームスコープで、チーム単位に権限とリソースを割り当て |
| **Workload Identity Pool(fleet 共通)** | fleet 全体で共通の Workload Identity を使う |

**設計の型**:「複数リージョンにクラスタを置いて可用性を上げたい。構成とポリシーは一元管理したい」→ fleet + Config Sync + Policy Controller + Multi Cluster Gateway。「オンプレの Kubernetes も同じ管理下に置きたい」→ attached cluster として fleet に登録。

**Autopilot と Standard の選択**も設計判断として出ます。Autopilot はノードの管理・スケール・セキュリティ設定を Google が持ち、Pod 単位課金です。運用負荷を最小にしたい・ノードの細かい制御が不要ならば Autopilot。DaemonSet で特殊なエージェントを入れる、GPU/特殊マシンタイプを細かく制御する、ノードの OS を触る必要があるなら Standard です。

### 1-4-4 安全なパッチ適用とアップグレード [1.4]

**Safe and secure patching and upgrading practices**。

**GKE のアップグレード**。

- **リリースチャンネル**: Rapid / Regular / Stable / Extended。新機能を早く試すなら Rapid、本番の既定は Regular か Stable。チャンネルに登録すると自動アップグレードが有効になります。**本番のバージョンを完全に止めることは推奨されず**、代わりにメンテナンスウィンドウと除外で制御します。
- **maintenance window / maintenance exclusion**: アップグレードを実施してよい時間帯を指定し、繁忙期(年末セールなど)は exclusion で一定期間止めます。exclusion には「マイナーアップグレードのみ停止」「すべての自動アップグレードを停止」など種類と最大期間があります。
- **コントロールプレーンとノードは別**: 先にコントロールプレーンが上がり、その後ノードプールが上がります。バージョンスキューの制約(ノードはコントロールプレーンより新しくできない、差は 2 マイナーまで)があります。

- **ノードアップグレード戦略**:
  - **surge upgrade**: 新ノードを `maxSurge` 台先に作り、古いノードを `maxUnavailable` 台ずつ drain して置換。既定でコストと速度のバランスが良い。
  - **blue-green upgrade**: 新バージョンのノードプールをまるごと作り、ワークロードを移してから旧プールを消す。**問題があれば即座に旧プールへ戻せる**のが利点で、コストは一時的に倍近くになります。重要な本番で選ばれます。
- **PodDisruptionBudget(PDB)**: drain 時に同時に落としてよい Pod 数を制限します。PDB を設定していないと、アップグレードで全レプリカが同時に落ちる可能性があります。逆に PDB が厳しすぎる( `minAvailable` がレプリカ数と同じ)と drain が永久に進まず、アップグレードがタイムアウトします。**両方向のトラブルが出題されます**。
- **検証の順序**: 開発 → ステージング → 本番の順にアップグレードし、各段階でスモークテストを回します。fleet を使う場合は「カナリアクラスタ」を 1 つ作り、そこだけ先に新バージョンにする型が有効です。

**VM のパッチ適用**。

- **VM Manager** の **OS patch management** で、パッチジョブをスケジュール実行します。対象を絞り込み(ラベル、ゾーン)、事前/事後スクリプト、再起動の要否( `RebootConfig` : DEFAULT / ALWAYS / NEVER)、ロールアウト方式(ゾーンごと、割合)を指定できます。**patch compliance** レポートで未適用の VM を把握します。
- **OS Config agent** が VM に必要です。
- ただし DevOps の推奨は **immutable infrastructure** です。稼働中の VM にパッチを当てるのではなく、Packer で新しいイメージを焼き、インス

タンステンプレートを差し替えて MIG の **rolling update**(`--max-surge`、`--max-unavailable`)で入れ替えます。差し戻しはテンプレートを戻して再度 rolling update するだけです。

**コンテナイメージのパッチ**: ベースイメージを定期的に再ビルドし、Artifact Analysis の脆弱性スキャン結果で緊急度を判断します。**distroless イメージ**や最小構成のベースを使うと、そもそも脆弱性の面積が減ります。

**誤答肢の切り方**:「本番 GKE のアップグレードで一切のダウンタイムを避け、問題時は即戻したい」→ blue-green upgrade。ここで「surge upgrade で `maxUnavailable: 0`」も無停止ではありますが、**旧バージョンへの即時復帰**という要件には blue-green が合致します。要件のどの語に対応するかで選びます。

---

## 1-5 安全なクラウド開発環境の整備 [1.5]

### 1-5-1 クラウド開発環境の構成と管理 —— Cloud Workstations・Cloud Shell [1.5]

**Configuring and managing cloud development environments (e.g., Cloud Workstations, Cloud Shell)。**

**Cloud Shell** は、コンソールから即座に開ける無料のシェル環境です。特徴は、① `gcloud`、`kubectl`、`terraform`、`docker` などが最初から入っている、② `$HOME` 配下 5 GB が永続化される、③一定時間で切断され、`$HOME` 以外は初期化される、④**Cloud Shell Editor** で簡易的な IDE として使える、⑤自分の認証情報でそのまま API を叩ける。用途は「ちょっとした調査・単発の作業」です。

制約: マシンスペックが固定で小さい、長時間の連続稼働に向かない、VPC 内のプライベートリソースへ直接は届かない(Cloud Shell は Google 管理のネットワークにあります)。組織のポリシーで無効化されることもあります。

**Cloud Workstations** は、チーム標準の開発環境をマネージドで配る仕組みです。三層構造で覚えます。

| 層 | 意味 |
|---|---|
| **workstation cluster** | リージョンと VPC を決める器。ここでネットワーク(プライベート IP)を決める |
| **workstation configuration** | テンプレート。マシンタイプ、ディスク、**コンテナイメージ**、IDE、タイムアウト、サービスアカウント |
| **workstation** | 開発者ごとの実体。一時的な Compute Engine VM 上で動く |

**なぜ Cloud Shell ではなく Cloud Workstations か**(この比較が出ます)。

| 要件 | Cloud Shell | Cloud Workstations |
|---|---|---|
| 環境を全員で統一したい | 不可(各自が入れる) | **config でコンテナイメージを固定** |
| VPC 内のプライベートリソースに繋ぎたい | 不可 | **クラスタが VPC 内。プライベート IP 対応** |
| マシンスペックを選びたい | 不可 | **可(GPU も)** |
| ソースコードを端末に落としたくない | 一応可 | **可。VPC-SC と併用して持ち出しを封じる** |
| 無料 | **無料** | 稼働時間で課金 |

Cloud Workstations の運用ポイント。

- **idle timeout / running timeout**: 一定時間操作がなければ自動停止し、課金を止めます。**FinOps の観点で必ず設定**します。停止しても永続ホームディレクトリ(Persistent Disk)のファイルは残り、それ以外のランタイムデータは消えます。
- **config の更新は再起動で反映**されます。ベースイメージを更新すれば、全開発者の環境が次回起動時に揃います。
- **アクセス方法**: ブラウザの Code OSS、ローカルの VS Code、JetBrains IDE、SSH。
- **セキュリティ**: プライベート IP のみにしてインターネット経由の到達を断ち、VPC Service Controls の境界に入れ、IAM( `roles/workstations.user` )で利用者を絞ります。

### 1-5-2 必要なツールでの環境ブートストラップ [1.5]

**Bootstrapping environments with required tooling (e.g., custom images, IDE, Cloud SDK)**。

**custom images(カスタムイメージ)** が中心です。Cloud Workstations の base image に、社内の CA 証明書、`gcloud`(Cloud SDK)、`kubectl`、`terraform`、言語ランタイム、リンタ、社内 CLI を焼き込んだイメージを Artifact Registry に置き、config から参照します。

```
FROM us-central1-docker.pkg.dev/cloud-workstations-
images/predefined/code-oss:latest
RUN apt-get update && apt-get install -y terraform kubectl
COPY corp-ca.crt /usr/local/share/ca-certificates/
RUN update-ca-certificates
```

**押さえる点**。

- イメージは**バージョン付きタグで参照**します。`latest` を指すと、いつの間にか全員の環境が変わって再現しない不具合が出ます。
- ベースイメージの更新は CI で定期的に再ビルドし、脆弱性スキャンを通します。開発環境も供給網の一部です。
- **IDE**: Code OSS(ブラウザ)、JetBrains、ローカル VS Code のリモート接続。IDE の拡張(Cloud Code、Gemini Code Assist)をイメージに含めて配ります。
- **Cloud SDK(gcloud CLI)**: `gcloud components install` ではなく、イメージビルド時に固定バージョンを入れるほうが再現性が高いです。認証は開発者本人の ADC か、workstation に付けたサービスアカウントを使います。
- **dotfiles / 初期化スクリプト**: 個人設定は起動時スクリプトで引き込み、共通設定はイメージに入れる、という分離が管理しやすい形です。
- **Cloud Code** は VS Code / JetBrains の拡張で、Skaffold を裏で使って GKE / Cloud Run へのローカル開発ループ、YAML 補完、ログ閲覧を提供します。開発者の内側のループを速くする道具です。

## 1-5-3 開発と運用への AI の活用 —— Gemini Code Assist・Gemini Cloud Assist・Gemini CLI [1.5]

**Leveraging AI to assist with development and operations**。3 つの役割の違いを押さえます。

| 製品 | 使う場所 | 何をするか |
| --- | --- | --- |
| **Gemini Code Assist** | IDE(VS Code、JetBrains、Cloud Workstations、Cloud Shell Editor) | コード補完、コード生成、説明、テスト生成、コードレビューへのコメント |
| **Gemini Cloud Assist** | Google Cloud コンソール | 設計の相談、リソースの調査、**ログ・メトリクス・トレースの分析**、障害調査(Investigations)、コスト最適化の示唆 |
| **Gemini CLI** | ターミナル | 対話的にコマンドやスクリプトを生成・実行し、リポジトリ全体を対象に作業する |

**DevOps 文脈での使いどころ。**

- **Code Assist**: `cloudbuild.yaml` 、Terraform、Kubernetes マニフェストの生成と説明。プルリクエストの一次レビュー。ただし**生成結果は必ず人がレビューし、ポリシーチェック(Policy Controller、`gcloud terraform vet` )を通す**という前提を崩しません。
- **Cloud Assist**: 「このアラートが鳴った原因は」「このログの急増は何か」を自然言語で尋ね、関連するログエントリ・メトリクス・変更イベントを横断的に示します。障害対応の初動を速くする用途で、4 章のログ・メトリクス・トレース分析と組み合わせて出題されます。
- **Gemini CLI**: 運用スクリプトの雛形作成、既存リポジトリの調査、繰り返し作業の自動化。

**セキュリティ上の前提**: Gemini for Google Cloud は、顧客のコードやプロンプトを基盤モデルの学習に使わない旨がエンタープライズ向けに定められて

います。とはいえ、機密コードを扱う組織では VPC Service Controls とデータ所在地の要件を確認し、利用可否と対象プロジェクトを IAM で制御します。

**取り違え注意**:「本番のインシデントで、複数プロジェクトのログとメトリクスから原因の候補を素早く挙げたい」→ Gemini Cloud Assist(コンソール、運用側)。「開発者の IDE で Terraform の書き方を補完したい」→ Gemini Code Assist。用途の場所で切り分けます。

---

### 章末チェック —— この章で「選べる」ようになるべき判断

1. 組織構造の要件を読み、application-centric なフォルダ/プロジェクト構成を提案できる。
2. Shared VPC / VPC Network Peering / Private Service Connect を要件(同一組織か、CIDR 重複か、SaaS か)で選び分けられる。
3. 横断監視は metrics scope、横断ログは aggregated sink、と即答できる。
4. 最小権限のロールを選び、基本ロールとサービスアカウントキーを誤答として切れる。
5. Workload Identity Federation が必要な場面(外部 CI からの認証)を判別できる。
6. Terraform / Infrastructure Manager / Config Connector / GitOps を、ドリフト是正の要否と管理主体で選び分けられる。
7. Cloud Build の private pool が必要な条件(VPC 内リソースへの到達)を言える。
8. Cloud Deploy の render/deploy 分離、承認、canary、rollback、automation、custom target を説明できる。

9. Artifact Registry の standard / remote / virtual と cleanup policy を要件で選べる。
10. GKE の surge upgrade と blue-green upgrade、maintenance exclusion、PDB のトラブルを判別できる。
11. Cloud Shell と Cloud Workstations を、統一性・VPC 到達性・スペックの要件で選び分けられる。
12. Gemini Code Assist と Gemini Cloud Assist の用途を場所で切り分けられる。

---

## サンプルはここまでです

続き(第2章〜巻末)は、GCP PDO 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 PDO 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 DevOps Engineer 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