IT資格道場

DOP-C02 学習用テキスト

予備知識ゼロから合格ラインへ ―― 読んで理解する1冊目

無料サンプル(第1章まで)

AWS Certified DevOps Engineer - Professional (DOP-C02) Exam Guide 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、AWS が実施する **AWS Certified DevOps Engineer – Professional（DOP-C02）** に合格することを目的に作られています。

Professional 級が問うのは「サービスを知っているか」ではなく、**状況と要件と制約を読んで、どの設計・どの運用を選ぶか**です。公式の試験ガイドは 6 つのドメイン（SDLC 自動化／構成管理と IaC／回復性の高いクラウドソリューション／監視とログ／インシデントおよびイベント対応／セキュリティとコンプライアンス）から成り、本書はその順番どおりに第1章〜第6章を置いていま

す。各 Task statement の末尾には「状況 → 要件 → 選ぶ設計 → なぜ他の案が劣るか」の場面演習があり、本番の長文シナリオ問題はこの型で解きます。

試験の基本情報

項目	内容
試験名	AWS Certified DevOps Engineer – Professional（**DOP-C02**）
運営	Amazon Web Services
問題数	**75問**（採点対象 65問＋非採点 10問）
出題形式	**択一と複数選択（「2つ選べ」「3つ選べ」）**
試験時間	**180分**
合格ライン	**スケール 100〜1,000 で 750点**

試験の設計図 —— 6 つのドメインと配点

ドメイン	内容	配点	本書の章
1	SDLC 自動化	22%	第1章
2	構成管理と IaC	17%	第2章
3	回復性の高いクラウドソリューション	15%	第3章
4	監視とログ	15%	第4章
5	インシデントおよびイベント対応	14%	第5章
6	セキュリティとコンプライアンス	17%	第6章

※ 公式は Task statement 単位の配点を公表していません。本書の節の厚みは、公式ガイドの Knowledge/Skills の項目数に比例させた**当サイトの推定**です。

4 ステップ学習法

1. 第1章から順に読む（既に SAA/SOA/DVA をお持ちなら、第1章・第2章・第5章の「判断基準」と「落とし穴」から）
2. 各節の「場面演習」を、答えを隠して自分で設計してから読む
3. 問題演習で腕試しをし、間違えた問題の該当節に戻る
4. 頻出度別ノートで直前の総仕上げ

第1章 SDLC自動化(ドメイン1・採点対象の22%)

この章は DOP-C02 で最大の配点を持つ分野です。問われるのは「CI/CD の道具の名前」ではなく、**与えられた制約(停止時間・切り戻し時間・アカウント境界・監査要件・コスト)のもとで、どの構成を選ぶと要件を満たすか**という判断です。

Associate 試験との違いははっきりしています。Associate は「CodeDeploy とは何か」を問い、Professional は「**CodeDeploy の Blue/Green と ECS の Rolling update と Lambda のカナリアのうち、この要件ならどれか。なぜ他の2つでは要件を満たさないか**」を問います。したがって本章では、サービスごとに次の3つをそろえて覚えます。

1. **できること**(機能)
2. **できないこと・制約**(誤答肢はここを突いてきます)
3. **他の候補との分かれ目**(どの条件で選択が切り替わるか)

出題形式は択一と複数選択(「2つ選べ」「3つ選べ」)で、多くが長文シナリオです。**シナリオの最後の1〜2文に「最小の運用負荷で」「ダウンタイムなしで」「最も短時間で切り戻せるように」といった評価軸が置かれ、そこが正解を一意に決めます**。技術的に正しい選択肢が複数並ぶのが普通で、**評価軸に照らして最良のものだけが正解**です。

1-1 CI/CDパイプラインを実装する [1.1]

1-1-1 ソフトウェア開発ライフサイクル(SDLC)の概念・フェーズ・モデル [1.1]

ソフトウェア開発ライフサイクル(SDLC) は、要件からリリース・運用までを一連の工程として扱う考え方です。試験では、**どのフェーズにどの自動化を差し込むか**という形で問われます。フェーズと、そこに対応する AWS の自動化は次のように対応します。

SDLC のフェーズ	何をするか	対応する自動化
計画・要件定義	作るものと受け入れ条件を決める	課題管理・変更管理プロセス(1-1-3、2-1-2)
設計	アーキテクチャとインタフェースを決める	IaC テンプレートの設計(第2章)
実装(コーディング)	コードを書き、バージョン管理へ入れる	プルリクエスト、コードレビュー、静的解析
ビルド	ソースを実行可能な成果物に変換する	AWS CodeBuild(1-1-4)
テスト	単体・統合・受け入れ・セキュリティスキャン	CodeBuild の別ステージ(1-2)
リリース・デプロイ	成果物を環境へ配る	AWS CodeDeploy、AWS CodePipeline(1-1-7、1-4)
運用・監視	稼働を測り、異常時に戻す	CloudWatch アラーム連動のロールバック(第4章・第5章)

SDLC のモデルは3つを対比で押さえます。

モデル	進み方	CI/CD との相性	出題での扱われ方
ウォーターフォール	フェーズを一方向に進め、前工程へ戻らない前提	相性が悪い。統合が最後にまとまって起きる	「統合の失敗が終盤に集中する」症状の原因として登場
アジャイル(スクラム等)	短い反復で動くものを作り続ける	相性が良い。反復ごとにデプロイ可能な成果物が出る	パイプラインの前提として登場
DevOps	開発と運用の責務を一体で持ち、自動化と計測で回す	CI/CD そのものの土台	「文化と自動化の両輪」として問われる

CI と CD の3語の切り分けは、言葉の定義を問う設問としてそのまま出ます。

- **継続的インテグレーション(CI)**……開発者の変更を**1日に何度も共有ブランチへ統合し、そのたびに自動でビルドとテストを走らせる**こと。目的は「壊れていることを早く知る」ことです。
- **継続的デリバリー(Continuous Delivery)**……CI に加え、**いつでも本番へ出せる状態の成果物を常に用意**しておくこと。**本番への反映には人の承認が入ります。**
- **継続的デプロイメント(Continuous Deployment)**……テストが通れば**人の承認なしに本番まで自動で反映**すること。

> **よく出る取り違え**: 「継続的デリバリー=承認なしで本番まで自動反映」は誤りです。承認なしまで行くのが**継続的デプロイメント**。監査上「本番反映の前に必ず責任者の承認が要る」という要件が出たら、答えは**継続的デリバリー**で、AWS CodePipeline の**手動承認アクション**を本番ステージの前に置く構成になります。

DevOps の効果指標(いわゆる4つの指標) も、シナリオの評価軸として登場します。

指標	意味	改善の打ち手の例
デプロイ頻度	どれだけ頻繁に本番へ出せるか	パイプラインの自動化、小さな変更単位
変更のリードタイム	コミットから本番反映までの時間	ビルドキャッシュ、テストの並列化
変更失敗率	本番反映のうち障害を招いた割合	自動テスト、カナリアデプロイ
平均復旧時間(MTTR)	障害から復旧までの時間	自動ロールバック、イミュータブルデプロイ

「**デプロイの失敗が本番利用者に届く前に止まるようにしたい**」ならカナリアと自動ロールバック、「**障害からの復旧を最短にしたい**」ならイミュータブル+切り戻し可能な構成、と評価軸から打ち手が決まります。

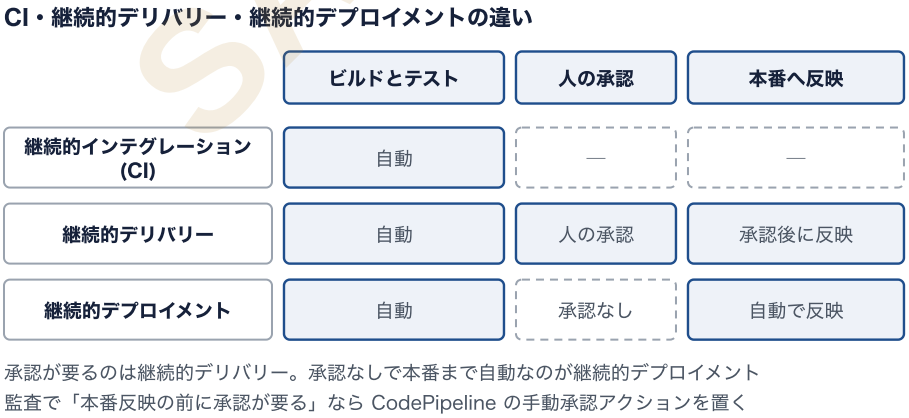


図 1-1　CI・継続的デリバリー・継続的デプロイメントの違い

1-1-2 コード・イメージ・成果物リポジトリの構成 [1.1]

パイプラインが扱う保管先は3種類あり、**混同すると選択肢の切り分けができません**。

種類	何を置くか	AWS の受け皿
コードリポジトリ	ソースコードと履歴	GitHub・GitLab・Bitbucket などの外部 Git(CodePipeline のソースアクション、または CodeConnections 経由で接続)
イメージリポジトリ	コンテナイメージ	**Amazon ECR**(プライベート/パブリック)
成果物(アーティファクト)リポジトリ	ビルド済みのパッケージ・依存ライブラリ	**AWS CodeArtifact**(npm・PyPI・Maven・NuGet・生成物の一般パッケージ)、**Amazon S3**(zip などの中間・最終成果物)

構成で必ず押さえる点を並べます。

- **CodePipeline のアーティファクトストアは Amazon S3 バケット**です。ステージ間の受け渡しはこのバケットを経由し、**入力アーティファクトと出力アーティファクトの名前がアクション定義で一致していないとステージが失敗**します。トラブルシューティングの定番です。
- **アーティファクトストアの暗号化**は既定でも行われますが、**マルチアカウント構成やコンプライアンス要件があるときはカスタマー管理の AWS KMS キー**を指定します。理由は 1-1-6 で扱うとおり、**別アカウントのロールにキーの利用許可(`kms:Decrypt` など)を与えないと、クロスアカウントのデプロイが復号で失敗する**からです。

- **Amazon ECR** は**イメージタグのイミュータビリティ(Tag immutability)** を設定でき、有効にすると**同じタグで別のイメージを上書きできなくなります**。「デプロイした `v1.2` が、いつの間にか中身の違うイメージになっていた」という事故の防止策として出ます。
- **Amazon ECR のライフサイクルポリシー**で古いイメージや未タグのイメージを自動削除し、保管コストを抑えます。**Amazon ECR イメージスキャン**(基本スキャンと拡張スキャン)は脆弱性の検出に使い、拡張スキャンは Amazon Inspector が継続的に再スキャンします。
- **AWS CodeArtifact** はパブリックリポジトリ(npmjs、PyPI など)を**上流リポジトリ**として設定でき、**取得したパッケージを自社ドメインにキャッシュ**します。上流が消えても取得済みのバージョンは残るため、**「外部リポジトリの障害や削除でビルドが止まるのを防ぎたい」の正解**になります。

> **よく出る取り違え**:「外部の npm パッケージの脆弱性を検出したい」は CodeArtifact ではなく**スキャンの話**(Amazon Inspector や CodeBuild 内の SCA ツール)です。CodeArtifact は**配布と依存の統制**であって、脆弱性検査の機能ではありません。

リポジトリの分割方針もシナリオで問われます。

方針	向く状況	落とし穴
モノレポ(1リポジトリに複数アプリ)	共通ライブラリの同時変更が多い	無関係なコミットでも全パイプラインが動く。**変更パスによるトリガーの絞り込み**が要る
マルチレポ(アプリごとに1リポジトリ)	チームが独立している	共通ライブラリのバージョン整合を別途管理する必要がある(CodeArtifact が効く)

コンテナイメージは Amazon ECR、パッケージは AWS CodeArtifact、中間成果物は Amazon S3
CodeArtifact は配布と依存の統制であって、脆弱性検査の機能ではない

図 1-2　パイプラインが扱う3つの保管先

1-1-3 バージョン管理とアプリ環境の紐づけ [1.1]

バージョン管理をどう使ってパイプラインを環境に結び付けるかは、Professional で頻出の設計判断です。

ブランチ戦略とパイプラインの対応を表で押さえます。

戦略	構造	パイプラインの張り方	向く状況
Git Flow	`develop`・`release`・`hotfix` など複数の長命ブランチ	ブランチごとに環境を割り当てる	リリース日が固定で、複数バージョンを並行保守する
GitHub Flow	`main` + 短命のフィーチャーブランチ	`main` へのマージで本番パイプラインが起動	継続的デプロイメントに向く
トランクベース開発	全員が短命ブランチで `main` へ頻繁にマージ	1本のパイプライン + **機能フラグ**で未完成機能を隠す	デプロイ頻度を最大化したい

- **環境の分け方**: 「開発・ステージング・本番をどう表現するか」は、**ブランチで分ける**か**1つの成果物を昇格(promote)させる**かの二択です。**Professional の推奨は後者**です。すなわち、**ビルドは1回だけ行い、同じ成果物を検証→本番へ順に進める**。ブランチごとに再ビルドすると、**本番へ出るバイナリがテストしたバイナリと同一である保証が消えます**。この考え方(build once, deploy many)は、成果物の話(1-3)とデプロイの話(1-4)の両方に効きます。
- **タグとリリース**: リリース対象は**コミットハッシュまたは Git タグで一意に固定**します。「`main` の最新」を本番へ配る設計は、**切り戻し先を特定できない**ため避けます。
- **トリガーの設計**: CodePipeline は**ソースの変更をきっかけに自動起動**します。外部 Git を使う場合は**Webhook(または CodeConnections)による起動が既定で、ポーリングは検知が遅く API 制限にも当たる**ため、「反映が数分遅れる」という症状ではまずポーリング設定を疑います。
- **フィルタリング**: CodePipeline の**トリガー**では、**ブランチ名・ファイルパス・タグ**を条件にして、対象外の変更でパイプラインが動かないようにできます。モノレポの無駄実行を止める打ち手です。
- **プルリクエスト単位の検証**: PR を作った時点でビルドとテストを走らせ、**結果をコミットステータスとして返す**構成にします(1-2-3)。**`main` を壊れた状態にしないための第一の防波堤**です。

> **落とし穴**: 「開発・検証・本番でそれぞれ CodeBuild が別々にビルドする」構成は、選択肢としてもっともらしく見えますが、**同一性の保証が無い**ため Professional では劣った案として切ります。正しくは**1回ビルドして成果物を昇格**させ、環境差は**設定の外出し**(Parameter Store・AppConfig・環境変数)で吸収します。

1-1-4 ビルドプロセスの設定 —— AWS CodeBuild [1.1]

AWS CodeBuild は、ソースを取得してコマンドを実行し、成果物を出力するマネージドのビルドサービスです。**サーバーを持たず、ビルドごとにクリーンな環境が立ち上がる**のが要点です。

`buildspec.yml` のフェーズは暗記対象です。

フェーズ	用途	特記
`install`	ランタイムやツールの導入	`runtime-versions` でランタイムを指定
`pre_build`	ログイン・依存取得	ECR への `docker login`、CodeArtifact のトークン取得はここ
`build`	本体のビルド・テスト	ここでの**シェルの終了コードがビルドの成否**になる(1-2-5)
`post_build`	イメージのプッシュ・後処理	**`build` が失敗しても `post_build` は実行される**(重要)
`artifacts`	出力する成果物の指定	`files`・`base-directory`・`discard-paths`
`reports`	テストレポートの指定	JUnit・Cucumber・カバレッジ形式を取り込む(1-2-6)
`cache`	キャッシュ対象パス	ローカルキャッシュと S3 キャッシュ

フェーズ	用途	特記
`env`	環境変数	`variables`・`parameter-store`・`secrets-manager`

CodeBuild で問われる設計判断を並べます。

- **VPC 内でのビルド**: RDS やプライベートな内部サービスへ接続してテストする必要があるなら、**CodeBuild プロジェクトを VPC に接続**します。このとき**プライベートサブネット + NAT ゲートウェイ、または VPC エンドポイント**が要ります。**パブリックサブネットに置いてもパブリック IP は付かず、インターネットへ出られない**のが定番の詰まり所です。
- **ビルドの高速化**: ①**キャッシュ**(依存ライブラリ・Docker レイヤー)、②**コンピュートタイプの引き上げ**、③**ビルドのバッチ(batch build)による並列実行**、④**Lambda コンピュートタイプ**(短時間・軽量なビルドを低レイテンシで)。「ビルド時間を短縮したい」の第一手は**依存関係のキャッシュ**です。
- **Docker イメージのビルド**: 従来は**特権モード(privileged mode)** の有効化が必要でした。「CodeBuild で `docker build` が `Cannot connect to the Docker daemon` で失敗する」の定番原因です。
- **同時実行の制御**: プロジェクト単位で**同時ビルド数の上限**を設定できます。「テスト用の共有リソースを壊さないよう、同時に1本だけ走らせたい」の解になります。
- **ビルドの再現性**: **ビルド環境イメージのバージョンを固定**し、`latest` を使わないこと。「昨日は通ったビルドが今日は落ちる」の原因として出ます。
- **ログ**: ビルドログは **Amazon CloudWatch Logs** と **Amazon S3** の双方に出力できます。長期保管と監査は S3、リアルタイム調査と Logs Insights は CloudWatch Logs、と役割で分けます。

CodeBuild と他のビルド手段の使い分けも出ます。

手段	選ぶ条件
CodeBuild	AWS 上で完結、従量課金、VPC 接続、IAM ロールで権限を渡せる
自前の EC2 上の Jenkins	既存のジョブ資産が大量にある、独自プラグイン依存。**運用負荷は最大**
EKS/ECS 上のセルフホストランナー	特殊なハードウェア要件やライセンス要件がある

「**運用負荷を最小にしつつビルドを自動化したい**」という評価軸なら、答えは**CodeBuild**です。Jenkins の可用性向上や自動スケールを自前で組む案は、要件を満たしても「運用負荷が最小」の軸で負けます。

1-1-5 ビルドとデプロイのシークレット管理 [1.1]

パイプラインが扱う機密は、**外部サービスのトークン・データベースの資格情報・署名鍵**などです。置き場所は2つで、**選択の基準がそのまま設問になります**。

	AWS Secrets Manager	AWS Systems Manager Parameter Store
主目的	機密情報の保管と**自動ローテーション**	設定値と機密情報の保管
ローテーション	**Lambda による自動ローテーションを標準搭載**。RDS・Redshift・DocumentDB は組み込みで対応	自動ローテーションの機能は無い(自前で組む)

	AWS Secrets Manager	AWS Systems Manager Parameter Store
暗号化	常に AWS KMS で暗号化	`SecureString` 型を選んだときに KMS で暗号化
料金	シークレットごとの月額 + API 呼び出し	**標準パラメータは追加料金なし**。アドバンストは有料
クロスアカウント	**リソースベースポリシーで別アカウントへ共有できる**	パラメータにリソースポリシーは付けられない(共有は別手段)
クロスリージョン	**レプリケーション機能あり**	無い
サイズ上限	大きめ(64KB)	標準4KB、アドバンスト8KB

判断の基準はこうです。

- **ローテーションが要件に書かれていたら Secrets Manager**。「90日ごとにデータベースのパスワードを自動で変更したい」は Secrets Manager 一択です。
- **単なる設定値(エンドポイント名・機能の有効無効・AMI ID)なら Parameter Store**。無料で扱え、`AWS::SSM::Parameter::Value<String>` として CloudFormation から直接参照もできます。
- **クロスアカウントで同じ資格情報を共有するなら Secrets Manager**。リソースベースポリシーで相手アカウントのロールに `secretsmanager:GetSecretValue` を許可し、**KMS キーポリシーにも相手を許可**します(KMS 側を忘れるのが定番の失敗)。

パイプラインでの使い方も具体的に押さえます。

- CodeBuild の `buildspec.yml` では `env/secrets-manager` と `env/parameter-store` を書くと、**ビルド開始時に値が環境変数へ注入**されます。プロジェクトのサービスロールに `secretsmanager:GetSecretValue` / `ssm:GetParameters` と、必要なら `kms:Decrypt` が要ります。
- **ビルドログへの出力を避ける**: 環境変数に入れた機密を `echo` すると CloudWatch Logs に残ります。CodeBuild は既知のシークレット参照をマスクしますが、**加工した値まではマスクされません**。「ログに資格情報が出ていた」という設問では、**シークレットを平文の環境変数(variables)に直書きしていた**のが原因として問われます。
- **やってはいけない設計**: ①ソースコードや `buildspec.yml` に平文で書く、②CodePipeline のアクション設定に平文で埋める、③コンテナイメージに焼き込む、④EC2 のユーザーデータに書く。**いずれも「監査で機密が露出した」型の設問の誤答肢**です。
- **そもそも長期の資格情報を持たない**のが上位の解です。AWS リソースへのアクセスは**IAM ロール**(CodeBuild サービスロール、CodeDeploy サービスロール、インスタンスプロファイル)で渡します。外部サービス(GitHub など)へは**OIDC による短期認証**を使い、アクセスキーをシークレットに置かない構成が最良の案として出ます。

> **よく出る取り違え**: 「Parameter Store の `SecureString` は自動でローテーションされる」は誤りです。**ローテーションの標準機能を持つのは Secrets Manager だけ**です。逆に「設定値の保管に必ず Secrets Manager を使う」も、コストの軸で劣る案として切られます。

1-1-6 単一アカウントとマルチアカウントのパイプライン展開パターン [1.1]

Professional の主戦場です。**「ツールアカウント」に CI/CD の実体を置き、対象環境は別アカウント**という構成が基本形になります。

構成の型は次のとおりです。

要素	置き場所	要点
AWS CodePipeline 本体、CodeBuild	**ツール(共有サービス)アカウント**	パイプラインの定義と実行はここに集約
アーティファクトストア (Amazon S3)	ツールアカウント	**バケットポリシーで対象アカウントへ読み取りを許可**
暗号化キー(AWS KMS)	ツールアカウント	**カスタマー管理キーが必須**。キーポリシーで対象アカウントのロールに `kms:Decrypt` を許可
デプロイ先(開発・検証・本番)	**環境ごとに別アカウント**	各アカウントに**クロスアカウントロール**を用意し、ツールアカウントの `PipelineRole` が引き受ける

マルチアカウントにする理由は3つで、設問の前提として出ます。

1. **障害の波及範囲(ブラストラディウス)を切る**。検証環境の事故が本番に及ばない。
2. **権限の分離**。本番アカウントの権限を開発者が常時持たない状態を作れる。

3. **サービスクォータとコストの分離**。アカウント単位の上限と請求の可視化。

クロスアカウントで必ず点検する4点は、トラブルシューティングの設問でそのまま使えます。

1. **パイプラインのサービスロール**に、対象アカウントのロールへの `sts:AssumeRole` が許可されているか。
2. **対象アカウントのロールの信頼ポリシー**に、ツールアカウント(のロール)が principal として書かれているか。
3. **アーティファクトの S3 バケットポリシー**が対象アカウントに `s3:GetObject` を許可しているか。**さらに、アップロード時に `bucket-owner-full-control` が付いているか**。
4. **KMS キーポリシー**が対象アカウントのロールに `kms:Decrypt` と `kms:DescribeKey` を許可しているか。**既定の AWS 管理キーではクロスアカウント共有ができない**ため、ここが「クロスアカウントデプロイが `AccessDenied` で失敗する」の最頻出原因です。

マルチリージョンのパイプラインでは、**リージョンごとにアーティファクトバケットが必要**です。CodePipeline は他リージョンのアクションを実行するとき、そのリージョンのバケットに成果物を複製します。「別リージョンへの展開ステージだけ失敗する」なら、**当該リージョンのアーティファクトバケットと KMS キーの不足**を疑います。

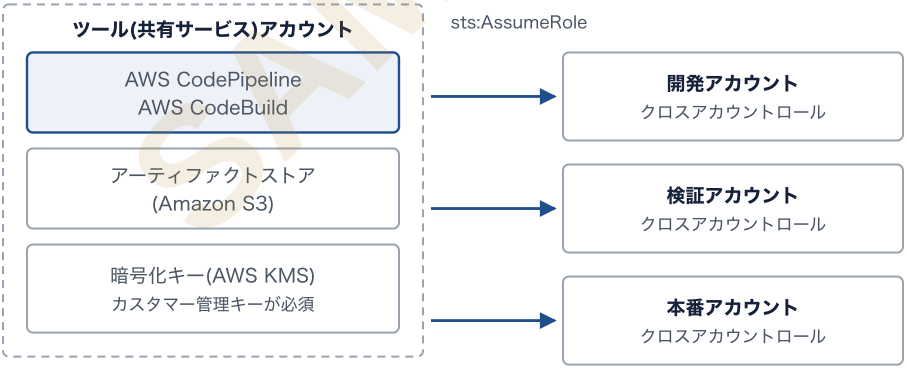
パイプライン自体の管理方法も判断対象です。

方式	内容	向く状況
手作業でコンソール定義	再現性が無い	実験のみ。設問では常に劣位

方式	内容	向く状況
CloudFormation/CDK でパイプラインを定義	パイプラインも IaC で管理	標準解
CDK Pipelines(自己変更型)	パイプライン自身の定義変更をパイプラインが適用する	パイプラインの構成変更が頻繁な大規模組織
CloudFormation StackSets との併用	多数アカウントへ同じパイプライン基盤を配る	組織全体への展開(2-1-5)

> **落とし穴**:「開発者が本番アカウントの管理者ロールを引き受けてデプロイする」案は動きますが、**権限分離の軸で必ず負けます**。正解は「**人ではなくパイプラインのロールだけが本番を変更できる**」構成です。

ツールアカウントに CI/CD を置くマルチアカウント構成



クロスアカウントで点検する4点
① パイプラインのサービスロールに sts:AssumeRole の許可があるか
② 対象アカウントのロールの信頼ポリシーにツールアカウントが書かれているか
③ アーティファクトの S3 バケットポリシーが s3:GetObject を許可しているか
④ KMS キーポリシーに kms:Decrypt と kms:DescribeKey の許可があるか

図 1-3　ツールアカウントに CI/CD を置くマルチアカウント構成

1-1-7 デプロイ戦略の決定 —— AWS CodeDeploy [1.1]

AWS CodeDeploy は、**成果物を対象へ配って、決められた手順で切り替える**役割です。ビルドはしません。対象は3種類あり、**選べる戦略が対象ごとに違う**のが最重要点です。

コンピュートプラットフォーム	選べるデプロイタイプ	主な設定
EC2/オンプレミス	**In-place(その場更新)** と **Blue/Green**	`AllAtOnce` / `HalfAtATime` / `OneAtATime` / カスタム。**CodeDeploy エージェントが必須**
Amazon ECS	**Blue/Green のみ**	`AllAtOnce` / `Linear` / `Canary` 。エージェント不要
AWS Lambda	**Blue/Green のみ**(エイリアスの重み付け)	`AllAtOnce` / `Linear` / `Canary` 。エージェント不要

ここが分かれ目です。

- **ECS と Lambda では In-place を選べません。**「ECS サービスを In-place で更新したい」という選択肢は、それだけで誤りです(ECS 自体のローリング更新は ECS のデプロイコントローラー側の機能で、CodeDeploy のデプロイタイプとは別物です)。
- **EC2 の Blue/Green は、新しいインスタンス群を Auto Scaling グループから起動し、ロードバランサーの向き先を差し替えます。**したがって **ELB(Application Load Balancer など)が前提**で、置き換え後の旧環境を**指定時間だけ残して即時切り戻し可能**にできます(`TerminateBlueInstancesOnDeploymentSuccess` の待機時間)。

- `appspec.yml` が CodeDeploy の指示書です。EC2 では `files`(コピー先)と `hooks`(ライフサイクルイベント)、ECS と Lambda では**リソース定義(タスク定義 ARN、Lambda の関数バージョン)**を書きます。**EC2 用の `appspec.yml` は YAML でも JSON でも書けますが、ECS/Lambda 用も同様**です。
- **ライフサイクルイベント(EC2)**の順序は頻出です: `ApplicationStop` → `DownloadBundle` → `BeforeInstall` → `Install` → `AfterInstall` → `ApplicationStart` → `ValidateService`。Blue/Green ではこれに `BeforeBlockTraffic` / `BlockTraffic` / `AfterBlockTraffic` と `BeforeAllowTraffic` / `AllowTraffic` / `AfterAllowTraffic` が加わります。**`DownloadBundle` と `Install` と `AllowTraffic` / `BlockTraffic` はフックスクリプトを書けません**(CodeDeploy が実行する固定処理)。
- **`ApplicationStop` は「新しいリビジョンではなく、直前にデプロイ済みのリビジョンのスクリプト」が実行されます。**初回デプロイでは実行されません。「`ApplicationStop` で失敗し続けてデプロイが進まない」の原因は、**過去のリビジョンに壊れたスクリプトが残っていること**です。
- **自動ロールバック**は、①デプロイ失敗時、②**指定した Amazon CloudWatch アラームが発報したとき**に設定できます。ロールバックは「元に戻す」のではなく、**直前に成功したリビジョンを新しいデプロイとして再適用**します。この挙動の理解は第5章のインシデント対応にもつながります。
- **`ValidateService` フック**に検証スクリプトを置くと、**トラフィックを流す前に自前のヘルスチェックを走らせ、失敗したらロールバック**できます。Lambda/ECS では `BeforeAllowTraffic` / `AfterAllowTraffic` に **Lambda の検証関数(フック)**を割り当てます。

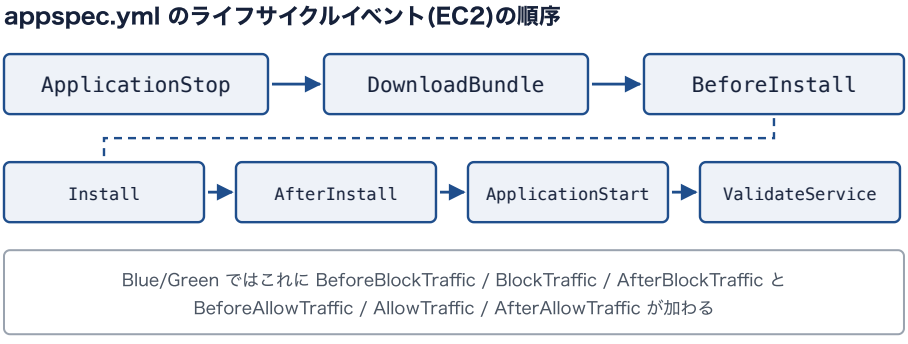


図 1-5　appspec.yml のライフサイクルイベント(EC2)の順序

デプロイ設定(DeploymentConfig)の選び方は、要件語との対応で覚えます。

要件の書かれ方	選ぶ設定
「ダウンタイムを許容し、最速で全台へ」	`AllAtOnce`
「容量を落とさずに順次更新したい」	`HalfAtATime` / `OneAtATime`(**In-place では容量が一時的に減る**点に注意)
「少数の利用者にだけ先に出して様子を見たい」	**カナリア**(ECS/Lambda の `Canary10Percent5Minutes` など)
「一定割合ずつ均等に移していきたい」	**リニア**(`Linear10PercentEvery1Minute` など)
「切り戻しを数秒で終わらせたい」	**Blue/Green**(旧環境を残す)

> **よく出る取り違え**:「In-place でも切り戻しは一瞬」は誤りです。In-place のロールバックは**旧リビジョンの再デプロイ**なので、**所要時間はデプロイ**

> **1回分**かかります。「切り戻し時間を最短に」という評価軸が出たら**Blue/Green** です。

AWS CodeDeploy —— 対象で選べるデプロイタイプが違う

EC2/オンプレミス	Amazon ECS	AWS Lambda
In-place Blue/Green	Blue/Green のみ	Blue/Green のみ (エイリアスの重み付け)
CodeDeploy エージェントが必須	エージェント不要	エージェント不要
デプロイ設定 AllAtOnce HalfAtATime OneAtATime	デプロイ設定 AllAtOnce Linear Canary	デプロイ設定 AllAtOnce Linear Canary

ECS と Lambda では In-place を選べない。ECS を In-place で更新する選択肢は誤り
切り戻し時間を最短にしたいなら Blue/Green。In-place の切り戻しは旧リビジョンの再デプロイ

図 1-4　AWS CodeDeploy —— 対象で選べるデプロイタイプが違う

1-1-8 パイプラインを束ねる —— AWS CodePipeline の構造 [1.1]

AWS CodePipeline は、ステージとアクションで工程を並べるオーケストレーターです。**自分ではビルドもデプロイもせず、他サービスを呼びます。**

- **ステージ**は順に実行され、**同一ステージ内のアクションは** `runOrder` **が同じなら並列**、違えば順に実行されます。「テストを並列化して時間を短縮したい」なら**同じ** `runOrder` **を割り当てる**のが答えです。
- **アクションタイプ**は Source / Build / Test / Deploy / Approval / Invoke の6種。**Invoke アクションから AWS Lambda や AWS Step Functions を呼べる**ので、標準アクションに無い処理はここで差し込みます(1-2-7)。
- **手動承認アクション**は Amazon SNS トピックへ通知でき、**承認待ちの間パイプラインは停止**します。承認の有効期限は7日間です。

- **変数(variables)** をアクション間で受け渡せます。CodeBuild の `exported-variables` で出した値を、後段のアクション設定で `#{BuildVariables.IMAGE_TAG}` のように参照する使い方が定番です。
- **ステージのロールバック**機能により、失敗したステージを直前の成功時のアーティファクトで再実行できます。
- **実行モード**: 既定の `SUPERSEDED`(後続の実行が先行を追い越す)のほか、`QUEUED`(順に実行)と `PARALLEL`(並列)が選べます。「デプロイの順序を必ず守りたい」なら `QUEUED` です。

通知は **AWS User Notifications / CodeStar Notifications** か、**Amazon EventBridge のパイプライン状態変更イベント**で行います。「パイプラインが失敗したら運用チームへ通知したい」の標準解は、**EventBridge ルール(`CodePipeline Pipeline Execution State Change`)→ Amazon SNS** です。

1-1-9 場面演習 [1.1]

場面A: 監査要件のある本番反映。 状況 —— 金融系の企業で、`main` へのマージから本番反映まで自動化したい。ただし監査部門から「**本番へ出る前に、変更内容を責任者が承認した記録が必ず残ること**」を求められている。ビルド成果物は検証環境で使ったものと同一である必要がある。

要件の要は「承認記録」と「同一成果物」です。設計はこうなります。

1. Source(Git)→ Build(**AWS CodeBuild で1回だけビルド**、成果物を Amazon S3 のアーティファクトストアへ)。
2. Deploy to Staging(**AWS CodeDeploy** で検証アカウントへ)。
3. **手動承認アクション**(Amazon SNS で責任者へ通知。承認は CloudTrail に記録される)。
4. Deploy to Production(**同じ入力アーティファクト**を本番アカウントへ)。

なぜ他が劣るか —— 「本番ステージで再度 CodeBuild を走らせる」案は、**本番に出るバイナリが検証したものと同一である保証を失います**。「承認を Slack の会話で行う」案は**記録が AWS 側に残らず**、監査要件を満たしません。「継続的デプロイメントにして承認を省く」案は要件そのものに反します。

場面B: クロスアカウントデプロイが AccessDenied で止まる。 状況 —— ツールアカウントの CodePipeline から本番アカウントへデプロイするステージだけが `AccessDenied` で失敗する。パイプラインのロールには本番アカウントのデプロイロールへの `sts:AssumeRole` があり、本番側の信頼ポリシーにもツールアカウントが入っている。アーティファクトバケットのバケットポリシーも本番アカウントに読み取りを許可している。

残る原因は**暗号化キー**です。アーティファクトストアが**AWS 管理キー**のままだと、**別アカウントのロールは復号できません**。対処は、①ツールアカウントで**カスタマー管理の AWS KMS キー**を作る、②**キーポリシーで本番アカウントのデプロイロールに `kms:Decrypt` と `kms:DescribeKey` を許可**する、③CodePipeline のアーティファクトストアにそのキーを指定する、の3点です。

「本番アカウントに管理者権限を与える」案は権限分離を壊し、症状(復号の失敗)にも当たっていません。「バケットを本番アカウント側に移す」案は、パイプラインの書き込み側で同じ問題が起きるだけです。

場面C: ビルド時間が伸び続けている。 状況 —— 依存パッケージが 800 個ある Node.js アプリのビルドが 18 分かかる。うち 11 分が依存の取得。パブリックの npm レジストリが不安定な日はビルドが失敗する。

打ち手は2つを組み合わせます。①**CodeBuild のキャッシュ**(S3 キャッシュまたはローカルキャッシュ)で `node_modules` を再利用する。② **AWS CodeArtifact** に npmjs を上流リポジトリとして登録し、**取得済みパッケー**

ジを自社ドメインにキャッシュする。これで外部レジストリの障害からも切り離せます。

「コンピュートタイプを最大にする」案は取得時間(ネットワーク待ち)には効きにくく、コストだけ増えます。「依存を減らす」案はアプリ側の改修で、設問の評価軸(運用の変更で短期に解決)に合いません。

1-2 CI/CDパイプラインに自動テストを統合する [1.2]

1-2-1 テストの種類 —— 単体・統合・受け入れ・UI・セキュリティスキャン [1.2]

テストは**種類ごとに「何を検証するか」「どれだけ速いか」「どこで壊れたと分かるか」が違います**。試験では「この目的にはどのテストか」「このテストをどのステージへ置くか」を問われます。

テストの種類	何を検証するか	速さ	実行の主体
ユニットテスト(単体テスト、unit tests)	関数やクラス単体の振る舞い。外部依存はモックに置き換える	最速(秒〜分)	CodeBuild(コミット直後)
統合テスト(インテグレーションテスト、integration tests)	複数の部品やサービス間の連携。実際のデータベースやキューに接続する	中(分)	CodeBuild(VPC 接続あり)、または検証環境デプロイ後
受け入れテスト(アクセプタンステスト、acceptance tests)	業務要件を満たすか。利用者の視点でのシナリオ	遅い(分〜十数分)	検証環境へのデプロイ後

テストの種類	何を検証するか	速さ	実行の主体
ユーザーインターフェイステスト(UIテスト、user interface tests)	画面の操作を通した動作。ブラウザ自動操作	最も遅く不安定	検証環境デプロイ後。**Amazon CloudWatch Synthetics のカナリアや AWS Device Farm** を使う
セキュリティスキャン(security scans)	脆弱性・機密の混入・設定不備	種類による	ビルド段階と成果物段階の両方

セキュリティスキャンはさらに細分され、Professional ではここまで問われます。

略称	何をするか	AWS 側の受け皿
SAST(静的解析)	ソースコードを実行せずに脆弱な書き方を検出	Amazon CodeGuru Reviewer、CodeBuild 内の任意ツール
SCA(依存関係の検査)	使っているライブラリの既知脆弱性を検出	CodeBuild 内のツール、Amazon Inspector(成果物側)
DAST(動的解析)	動いているアプリへ攻撃的な入力を送って検査	検証環境デプロイ後に CodeBuild から実行
シークレットスキャン	コードに紛れ込んだ資格情報の検出	ビルド前(pre-commit フック)とビルド内の両方
イメージスキャン	コンテナイメージの OS パッケージの脆弱性	**Amazon ECR イメージスキャン(拡張スキャンは Amazon Inspector)**

略称	何をするか	AWS 側の受け皿
IaC スキャン	テンプレートの設定不備	CodeBuild 内のツール、**AWS CloudFormation Guard**

> **よく出る取り違え**:「ユニットテストで外部 API との連携を確認する」は誤りです。**外部との連携を確かめるのは統合テスト**。ユニットテストは依存を切り離すからこそ速く、失敗の原因も一点に絞れます。

1-2-2 テストをパイプラインのどの段階に置くか [1.2]

配置の原則は**シフトレフト**、つまり「**速くて壊れた場所が特定しやすいテストほど前へ**」です。理由は2つ —— ①開発者へのフィードバックが早いほど修正が安く済む、②高価なテストを走らせる前に落とせば計算資源が無駄にならない。

推奨の並びは次のとおりです。

段階	置くテスト	失敗したときの意味
コミット/プルリクエスト時	静的解析、シークレットスキャン、ユニットテスト、SCA	コードそのものの欠陥。マージを止める
ビルド段階	ユニットテストの再実行、コードカバレッジ、IaC スキャン、イメージスキャン	成果物を作る資格が無い
検証環境へのデプロイ後	統合テスト、契約テスト、受け入れテスト、DAST	組み合わせたときの欠陥
本番前(ステージング)	負荷テスト、パフォーマンスベンチマーク、UIテスト	規模と体験の欠陥

段階	置くテスト	失敗したときの意味
本番デプロイ中	カナリアの監視、合成監視(Synthetics)	実トラフィックでの欠陥。**自動ロールバックの引き金**

判断の基準を明示します。

- **UIテストと負荷テストを PR ごとに全部走らせる**設計は、**PR のフィードバック時間を壊す**ため誤りです。これらは**夜間実行やマージ後、あるいは変更の性質に応じた条件付き実行**にします。
- **本番でしか再現しない問題**(実データ量、実トラフィック)に対しては、**本番へ小さく出して測る**(カナリア + 監視)のが正解です。テストで完全に防ぐ設計は不可能で、**検知と切り戻しの速さで補う**のが Professional の考え方です。
- **テストピラミッド**: ユニットを最も多く、統合を中程度、UIを最少に。**逆ピラミッド(UIテストばかり)は、遅くて壊れやすく、原因が分からない**という症状で設問に登場します。

1-2-3 プルリクエストやマージのときにビルド・テストを走らせる [1.2]

「壊れたコードが共有ブランチへ入らない」ようにするのが目的です。仕組みは次の形になります。

1. 開発者がプルリクエストを作る、または既存の PR を更新する。
2. その**イベント**を受けて **AWS CodeBuild** のビルドが起動する。
3. ビルドの結果(成功/失敗)が **コミットステータス**として PR に返る。
4. **必須チェックが成功しないとマージできない**ようにリポジトリ側で保護する(ブランチ保護ルール)。

起動の経路は押さえておきます。

- **CodeBuild のウェブフック**: GitHub・GitLab・Bitbucket と CodeBuild を直接つなぎ、`PULL_REQUEST_CREATED` / `PULL_REQUEST_UPDATED` / `PUSH` などの**イベントタイプでフィルタ**します。さらに `HEAD_REF` (ブランチ名) や `FILE_PATH` (変更ファイル) でも絞れます。**パイプライン全体を動かさずビルドだけ回したい PR 検証には、この直結が最短**です。
- **Amazon EventBridge 経由**: リポジトリのイベントをルールで受け、CodeBuild や AWS Lambda や AWS Step Functions を起動します。**複数の処理を分岐させたいときはこちら**。
- **CodePipeline のトリガー**: マージ(`main` への push)を起点にパイプライン全体を起動します。PR 単位の検証には重すぎます。

設計上の要点を挙げます。

- **PR 検証のビルドは本番用の成果物を作らない**。作った成果物をアーティファクトストアへ置くと、**マージされていないコードが本番へ流れる経路**を作ってしまいます。PR 検証は**検証だけ**にとどめます。
- **フォークからの PR には資格情報を渡さない**。ウェブフックの設定でフォーク元の PR を対象にすると、**third-party のコードが CodeBuild のサービスロールで動きます**。「外部コントリビューターの PR からシークレットが漏れた」は典型的な事故で、対策は**フォーク PR では権限の無い別プロジェクトを使う**ことです。
- **同時実行の抑制**: 同じ PR を連続で更新すると古いビルドが無駄に走ります。**キューイングと同時ビルド上限**で制御します。

> **落とし穴**: 「マージ後にだけテストする」設計は、**壊れた `main` を全員が引く**状態を招きます。設問では「開発者の生産性が落ちている」「`main` が頻

> 繁に壊れる」という症状で現れ、正解は**PR 時点での自動検証とブランチ保護**です。

1-2-4 負荷テスト・ストレステスト・パフォーマンスベンチマーク [1.2]

規模に対する振る舞いを測るテストです。用語の切り分けが問われます。

用語	目的
負荷テスト(load test)	**想定される負荷**をかけて、応答時間とエラー率が基準内に収まるかを見る
ストレステスト(stress test)	**想定を超える負荷**をかけて、どこで壊れるか・壊れ方が安全かを見る
パフォーマンスベンチマーク	バージョン間で**同じ条件**の測定値を比較し、性能の劣化(リグレッション)を検出する
ソークテスト(耐久テスト)	長時間かけ続けて、メモリリークや接続枯渇を検出する
スパイクテスト	急激な立ち上がりに対するスケールの追従を見る

AWS 上での組み方は次のようになります。

- **負荷の発生源**: AWS Fargate 上のコンテナで負荷生成ツールを並列に走らせる構成が標準です。**AWS Lambda を多数並列に呼ぶ**方法もあります。**AWS Step Functions の Map 状態**で分散実行を束ねると、開始・待機・集計を1つのワークフローにできます。
- **測定対象**: Amazon CloudWatch のメトリクス(レイテンシ、エラー率、CPU、コネクション数)と、**CloudWatch Logs Insights** によるログ集計。分散トレースは **AWS X-Ray**。

- **パイプラインへの組み込み**: 本番前ステージで **CodeBuild の Test アクション**として負荷テストを起動し、**結果のしきい値を超えたらビルドを失敗させてパイプラインを止める**のが基本形です。判定を外部に任せたい場合は **Invoke アクションから Lambda を呼び、CloudWatch のメトリクスを取得して合否を返す**構成にします。
- **どこに負荷をかけるか**: 本番と同等の構成を持つ**専用の負荷テスト環境**が原則です。本番へ直接かけると実利用者に影響します。ただし、**本番同等の環境をずっと持つとコストが高い**ため、**IaC でテスト直前に作り、終わったら壊す**(使い捨て環境)のが Professional の定石です。
- **AWS の許容ポリシー**: 自分のリソースに対する負荷テストは概ね許容されますが、**分散型サービス拒否(DDoS)に相当する行為は許されません。**試験では「本番に対して無制限の負荷をかける」案は常に誤答です。

> **判断の基準**:「毎回のコミットで負荷テストを回すべきか」は**否**です。**時間とコストが大きい**ため、**夜間バッチかリリース候補に対してのみ**走らせ、日常はユニット・統合テストで守ります。ただし**性能が主要な要件である場合は、ベンチマークの数値をリリースの受け入れ条件にする**という判断もあり、シナリオの評価軸次第で答えが変わります。

1-2-5 終了コードでアプリケーションの健全性を測る [1.2]

プロセスの終了コード(exit code) は、自動化の世界で「成功/失敗」を伝える最も基本的な信号です。**0 が成功、0 以外が失敗**という規約に、AWS の各サービスが乗っています。

場所	終了コードの扱われ方
CodeBuild の `buildspec.yml`	各フェーズのコマンドの**終了コードが 0 以外ならそのフェーズは失敗**し、ビルド全体が `FAILED` になる。テストランナーが失敗を 0 以外で返す前提が成り立つ
CodeDeploy のフックスクリプト	フックのスクリプトが**0 以外を返すとそのライフサイクルイベントが失敗**し、デプロイが失敗(設定次第で自動ロールバック)する
Amazon ECS のコンテナ	**コンテナの終了コードが `essential` なコンテナで 0 以外だとタスク全体が停止**。停止理由に終了コードが記録される。`137` は SIGKILL(OOM やタイムアウトによる強制終了)、`139` はセグメンテーション違反
AWS Lambda	例外の送出が失敗を意味する。非同期呼び出しは既定で2回再試行され、その後 **Lambda の送信先(destination)** または **デッドレターキュー(DLQ)** へ
AWS Batch / Systems Manager Run Command	ジョブ・コマンドの成否判定に終了コードを使う

設計上のポイントを明示します。

- **失敗を握りつぶさない**。シェルスクリプトで `command || true` と書いたり、パイプでつないだ先の終了コードだけを見たりすると、**テストが落ちてもビルドが緑になります**。`set -e` と `set -o pipefail` を入れるのが定石で、「テストは失敗しているのにパイプラインが通ってしまう」設問の答えがここです。
- **`post_build` は `build` が失敗しても走る**ので、`post_build` の中で無条件に `docker push` すると、**テストに落ちたイメージが公開されます**。

`CODEBUILD_BUILD_SUCCEEDING` 環境変数(1 が成功)で分岐させます。

- **ヘルスチェックとの関係**: 終了コードは「起動したプロセスが正常に終わったか」を表すだけです。**動き続けるアプリの健全性は、ELB のヘルスチェック・ECS のコンテナヘルスチェック・CodeDeploy の** `ValidateService` **フック**で測ります。**この2つを混ぜた選択肢は誤り**です。
- **ECS の** `stopCode` **と終了コード**: タスクが落ちたとき、`ExitCode` が 0 以外ならアプリ側の異常、`stoppedReason` が `Essential container in task exited` なら必須コンテナの終了、`OutOfMemoryError` ならメモリ不足、と切り分けます(第5章のトラブルシューティングでも使います)。

1-2-6 ユニットテストとコードカバレッジの自動化 [1.2]

コードカバレッジは、テストがコードのどれだけを通ったかの割合です。**品質そのものではなく、テストの網羅の目安**という位置づけを外すと設問を落とします。

CodeBuild のテストレポート機能が要点です。

- `buildspec.yml` の `reports` **セクション**にレポートグループ名とファイルパス、`file-format` を書くと、CodeBuild がテスト結果を取り込み、コンソールで**合格数・失敗数・実行時間・傾向**を表示します。
- 対応形式は、テスト結果が **JUnitXml・NUnitXml・NUnit3Xml・Cucumber JSON・TestNGXml・VisualStudioTrx**、カバレッジが **JACOCOXML・SIMPLECOV・CLOVERXML・COBERTURAXML**。
- **レポートグループ**は結果を蓄積する入れ物で、**保持期間(既定30日、最大未エクスポートで一定期間)** の設定と、**Amazon S3 へのエクスポート**が可能です。

- カバレッジのしきい値を割ったらビルドを落としたい場合は、**テストツール側でしきい値を設定して 0 以外の終了コードを返させる**のが基本です(CodeBuild のレポート機能はしきい値でビルドを落とす機能ではありません)。ここは**混同されやすい点**です。

カバレッジの扱いで問われる判断を挙げます。

- **「カバレッジ 100% を必須にする」は良い設計ではない**。テストのためのテストが増え、**意味のない断言(アサーションの無いテスト)** を誘発します。実務的には**新規に追加・変更した行のカバレッジ(差分カバレッジ)にしきい値を置く**のが妥当な解として出ます。
- **カバレッジが高いのに障害が減らない**という症状は、**アサーションが不足している**か**統合の層が薄い**ことを示します。答えは「カバレッジ目標を上げる」ではなく「**統合テスト・契約テストを足す**」です。
- **フレーキー(不安定)なテスト**は、パイプラインの信頼を壊します。**再実行で通るテストを放置しない**、**隔離して原因を直す**、が正しい対処で、「失敗したら自動で3回まで再試行する」だけの案は問題を隠す劣った案として扱われます。

1-2-7 パイプラインから AWS サービスを呼んでテストする [1.2]

CodePipeline の標準アクションに無い処理は、**サービスを直接呼び出して差し込みます**。この「呼び出しの選択肢」が設問になります。

呼び出し方	使いどころ	特徴
Invoke アクション → AWS Lambda	短時間の判定・外部 API 呼び出し・スモークテスト	**最長15分**。結果は `PutJobSuccessResult` / `PutJobFailureResult` でパイプラインへ返す

呼び出し方	使いどころ	特徴
Invoke アクション → AWS Step Functions	長時間・多段階・並列・待機を含む処理	15分の壁を越えられる。**分散マップで大量並列**。人手の承認待ちも表現可能
Test アクション → AWS CodeBuild	任意のコマンドでのテスト実行	VPC 接続、レポート取り込み、成果物出力ができる
Deploy アクション → AWS CloudFormation	テスト用スタックの作成と削除	**使い捨てのテスト環境**を作る定石
Amazon EventBridge	パイプラインの状態変化を外部処理へ渡す	疎結合。通知・記録・別ワークフロー起動
AWS Systems Manager Automation	インスタンス群への操作を伴う検証	ランブックとして手順を定義

判断の分かれ目を明示します。

- **処理が15分を超えるなら Lambda ではなく Step Functions**(または CodeBuild)。「Lambda で60分の負荷テストを回す」は、**タイムアウト上限に反するので誤り**です。
- **Lambda の Invoke アクションはパイプラインへ結果を返す責任がアプリ側にある**。`PutJobSuccessResult` を呼ばずに関数が終わると、**パイプラインは待ち続け、やがてタイムアウトします**。「Invoke ステージが1時間止まっている」の原因はほぼこれです。
- **テスト環境の作成と破棄**: 使い捨て環境は **CloudFormation のスタック作成 → テスト → スタック削除**をステージとして並べます。**削除ステージを必ず入れる**こと(入れ忘れがコスト超過の原因として出ます)。
- **合成監視の使い分け**: **Amazon CloudWatch Synthetics のカナリア**は、**本番の外形監視**にも**デプロイ直後のスモークテスト**にも使えます。パ

イプラインから起動して結果を判定に使う構成が出ます。

1-2-8 場面演習 [1.2]

場面A: テストが落ちているのにデプロイされる。状況 —— CodeBuild でユニットテストを実行しているのに、失敗したコードが検証環境へ流れることがある。`buildspec.yml` の `build` フェーズは `npm test | tee test.log` という1行で、`post_build` で `docker build` と `docker push` を行っている。

原因は2つあります。①**パイプでつないだため、終了コードが `tee` のもの(常に0)になっている。**②**`post_build` は `build` が失敗しても実行される**ため、テストに落ちたイメージが push されている。

対処は、① `set -o pipefail` を入れる(または `tee` を使わずログを別に出す)、② `post_build` の先頭で `CODEBUILD_BUILD_SUCCEEDING` を確認し、0 なら push をスキップする、の2点です。「テストの本数を増やす」「レポートグループを作る」は症状の原因に当たっていません。

場面B: PR のフィードバックが遅い。状況 —— すべての PR で、ユニットテスト・統合テスト・UIテスト・負荷テストの全部を1つの CodeBuild プロジェクトで直列に実行しており、結果が返るまで 55 分かかる。開発者が待ちきれずマージしてしまう。

設計を段階で分けます。①PR 時は**静的解析・シークレットスキャン・ユニットテスト**だけ(数分)。同一 `runOrder` にして並列化する。②マージ後のパイプラインで**統合テスト**。③本番前ステージで**UIテストと負荷テスト**を実行、あるいは夜間のスケジュール実行(**Amazon EventBridge のスケジュールルール**)に回す。④**ブランチ保護で PR の必須チェックを①に限定**する。

「コンピュートタイプを上げる」だけでは、UIテストと負荷テストの本質的な所要時間は縮みません。「テストを減らす」は品質を下げるだけで、**配置の問題**

を解いていません。

場面C: 本番だけで起きる性能劣化。状況 —— 検証環境では応答 120 ミリ秒のAPIが、本番リリース後に 900 ミリ秒へ悪化した。検証環境のデータ件数は本番の 1/500。

打ち手は3段構えです。①**本番同等のデータ量を持つ負荷テスト環境**を CloudFormation で使い捨てに作り、**パフォーマンスベンチマーク**をリリース候補ごとに測って前バージョンと比較する。②本番反映は**カナリア**にして、**Amazon CloudWatch アラーム(p99 レイテンシ)を CodeDeploy の自動ロールバックに紐づける。**③**AWS X-Ray** で遅い区間を特定する。

「検証環境のインスタンスサイズを本番と同じにする」だけでは、**データ量に起因する劣化(インデックス不足、N+1)は再現しません。**「本番に出してから様子を見る」だけでは、**検知と切り戻しの仕組みが無い**ため評価軸を満たしません。

1-3 成果物(アーティファクト)を構築し管理する [1.3]

1-3-1 成果物のユースケースと安全な管理 [1.3]

成果物(アーティファクト) とは、ビルドの結果として生まれる**配布可能な形のもの**です。試験に出る種類は次のとおりです。

成果物の種類	具体例	主な保管先
アプリケーションパッケージ	zip、jar、war、Lambda のデプロイパッケージ	**Amazon S3**
コンテナイメージ	Docker イメージ	**Amazon ECR**

成果物の種類	具体例	主な保管先
ライブラリパッケージ	npm、PyPI、Maven、NuGet、gem	**AWS CodeArtifact**
マシンイメージ	AMI、コンテナイメージ	**EC2 Image Builder** の出力
IaC テンプレート	CloudFormation テンプレート、CDK の合成結果、SAM パッケージ	Amazon S3、Service Catalog
Helm チャート、OCI アーティファクト	Kubernetes 向けの配布物	**Amazon ECR**(OCI 対応)

成果物のユースケースは3つです。①**同一性の担保**(検証で動かしたものと同じものを本番へ)、②**再現性**(過去の任意のバージョンを再デプロイできる)、③**分離**(ビルド環境とデプロイ環境を切り離す)。

安全な管理で問われる観点を並べます。

- **不変性(イミュータビリティ)**: **一度公開した成果物は書き換えない**。Amazon ECR の**タグイミュータビリティ**、Amazon S3 の**バージョニング**と**オブジェクトロック**で担保します。「同じバージョン番号で中身が変わっていた」型の事故の対策です。
- **暗号化**: S3 は SSE-S3 か SSE-KMS(**クロスアカウント共有ならカスタマー管理キーが実質必須**)、ECR も KMS による保管時の暗号化を選べます。転送時は TLS。
- **アクセス制御**: 読み取りと書き込みを分けます。**パイプラインのビルドロールだけが書き込み(push)でき、デプロイロールは読み取り(pull)のみ**が原則です(1-4-5)。

- **来歴と署名**:「その成果物が本当に自社のパイプラインから出たものか」を確かめる仕組み。**AWS Signer** によるコード署名(Lambda のコード署名設定、AWS IoT、コンテナイメージ)を使い、**署名されていない成果物のデプロイを拒否**できます。**Lambda のコード署名設定**は、署名が無い、または改ざんされたパッケージのデプロイを拒否します。
- **監査**: **AWS CloudTrail** に成果物の取得・登録の API 呼び出しが記録されます。S3 のデータイベント、ECR の API 呼び出しが調査の起点です。
- **脆弱性の継続検査**: **Amazon ECR の拡張スキャン(Amazon Inspector 連携)** は、**すでに保管済みのイメージも新しい脆弱性情報が出るたびに再評価**します。基本スキャンは push 時などの検査で、継続的な再評価は行いません。ここが選択肢の分かれ目になります。

> **落とし穴**:「成果物を Git リポジトリにコミットして管理する」案は、**リポジトリの肥大化と、バイナリの差分管理の非効率**を招くため誤答です。コードはコードリポジトリ、成果物は成果物リポジトリ、と役割を分けます。

1-3-2 成果物の作り方と生成の手段 [1.3]

生成の主体は次の3つで、選択の基準が違います。

手段	何を作るのに向くか	選ぶ理由
AWS CodeBuild	アプリのパッケージ、コンテナイメージ、テストレポート	標準解。VPC 接続、キャッシュ、レポート、任意のランタイム
AWS Lambda	軽量な変換・生成(テンプレートの合成、マニフェストの書き換え、メタデータの付与)	**短時間で完結する生成処理**をパイプラインに差し込むとき

手段	何を作るのに向くか	選ぶ理由
EC2 Image Builder	**AMI とコンテナイメージ**	OS 層まで含む「ゴールデンイメージ」の生成と配布(1-3-6)

CodeBuild で成果物を出す設定を具体的に押さえます。

- `artifacts` セクションの `files` で出力対象を指定します。`base-directory` を指定すると、そこからの相対パスで収集されます。`discard-paths: yes` はディレクトリ構造を捨てて平坦化します(**デプロイ先でパスが合わない**という不具合の原因になりがちで、設問にも出ます)。
- `secondary-artifacts` で複数の成果物を別々に出力できます。CodePipeline 側では**出力アーティファクト名で受け取ります**。
- `name` に `$(date +%Y-%m-%d)` などの動的な値を入れるとファイル名を変えられます。ただし**CodePipeline 経由の場合、名前は CodePipeline が管理**します。
- **パイプライン外からの参照**では、CodeBuild プロジェクトの**アーティファクト設定で S3 バケットを直接指定**します。

AWS Lambda を使う生成の典型例は、**CloudFormation カスタムリソース**や**CodePipeline の Invoke アクション**の中で、成果物にメタデータ(コミットハッシュ、ビルド番号、署名)を付与したり、環境ごとのマニフェストを差し替えたりする処理です。「15分を超えない、状態を持たない、小さな変換」であることが選択の条件です。

成果物に埋め込む情報もよく問われます。**バージョン番号、Git のコミットハッシュ、ビルド ID、ビルド日時**を成果物のメタデータやタグに残しておくと、**本番で動いているものがどのコミットから来たかを追跡**できます。これは第5章の障害調査で効いてきます。

1-3-3 成果物のライフサイクル [1.3]

作って終わりではなく、保持と削除まで設計するのが Professional の観点です。

フェーズ	決めること	実現手段
生成	いつ・何を作るか	CodeBuild、Image Builder
保管	どこに・どの暗号化で	S3、ECR、CodeArtifact
昇格(promotion)	検証を通ったものだけを次の環境へ	パイプラインのステージ、リポジトリの分離(dev/prod)
保持	いつまで残すか	S3 ライフサイクルルール、ECR ライフサイクルポリシー、CodeArtifact のドメイン管理
削除	何を消すか	未タグイメージ、古いビルド、失敗した成果物

具体的な設計判断を挙げます。

- **Amazon ECR のライフサイクルポリシー**:「未タグのイメージは1日で削除」「`dev-` で始まるタグは直近30個だけ残す」といったルールを**優先順位付きで**書きます。**本番で使っているタグを消さないよう、除外条件を先に置く**のが要点です。「本番タスクの起動が `image not found` で失敗する」の原因として、**ライフサイクルポリシーが本番イメージを消した**というシナリオが出ます。
- **Amazon S3 のライフサイクル**: CodePipeline のアーティファクトバケットは放置すると際限なく増えます。**バージョニング + 非現行バージョンの失効ルール**で保持期間を決めます。**監査要件がある場合は、削除ではなく S3 Glacier 系のストレージクラスへの移行**を選びます。

- **保持期間の決め方**:「**切り戻し可能でなければならない期間**」と「**監査で提示を求められる期間**」の長いほうに合わせます。「直近のビルド成果物だけ残す」案は、**過去バージョンへの切り戻しができなくなる**ため、切り戻し要件がある設問では誤りです。
- **CodeArtifact のパッケージのバージョン**: 一度公開したバージョンは**上書きしない**。ステータス(`Published` / `Unlisted` / `Archived` / `Disposed`)で扱いを変えます。脆弱性が見つかったバージョンは**削除せず利用不可にする**のが安全側の判断です(既存のビルドの再現性を壊さないため)。
- **クロスリージョンの複製**: 災害対策や別リージョンへのデプロイのために、**ECR のクロスリージョン/クロスアカウントレプリケーション**(レジストリのレプリケーション設定)と、**S3 のレプリケーション**を使います。「別リージョンのデプロイが遅い/失敗する」の対策として出ます。

1-3-4 成果物リポジトリの作成と設定 —— CodeArtifact・S3・ECR [1.3]

AWS CodeArtifact は、npm・PyPI・Maven・NuGet・SwiftPM・Ruby などの**パッケージマネージャ互換のプライベートリポジトリ**です。

構造を押さえます。

- **ドメイン(domain)**: 複数のリポジトリをまとめる最上位。**同じアセットはドメイン内で1回だけ保管**されるため、リポジトリを増やしてもストレージが重複しません。**KMS キーはドメイン単位**で指定します。
- **リポジトリ(repository)**: パッケージの入れ物。**上流リポジトリ(upstream)** を最大10段まで連鎖させられます。
- **外部接続(external connection)**: npmjs、PyPI、Maven Central などへの接続。**1つのリポジトリに設定できる外部接続は1つ**です。**外部から取得**

したパッケージはドメインにキャッシュされ、以後は外部に依存しません。

- **認証**: `aws codeartifact get-authorization-token` で**一時トークン(既定12時間、最大12時間)** を取得し、パッケージマネージャに設定します。CodeBuild では `pre_build` で取得するのが定番です。**IAM で認可され、ユーザー名/パスワードの長期資格情報は不要**です。

設計の判断基準はこうです。

要件	解
社内ライブラリを複数チームへ配りたい	CodeArtifact のリポジトリ + 上流連鎖
外部レジストリの障害・削除からビルドを守りたい	CodeArtifact の外部接続によるキャッシュ
承認したパッケージのバージョンだけを使わせたい	**上流を持たない「承認済み」リポジトリ**を作り、そこだけを開発者に向ける
コンテナイメージを保管したい	**Amazon ECR**(CodeArtifact はコンテナイメージのレジストリではない)
ビルドの中間成果物をステージ間で渡したい	**Amazon S3**(CodePipeline のアーティファクトストア)

> **よく出る取り違え**: 「CodeArtifact に Docker イメージを保管する」は誤りです。**コンテナイメージは Amazon ECR**。逆に「ECR に npm パッケージを置く」も設問では誤答として扱われます(ECR は OCI アーティファクトを扱えますが、パッケージマネージャ互換の配布は CodeArtifact の役割です)。

Amazon ECR の設定項目で問われるもの。

- **プライベートレジストリとパブリックレジストリ**(Amazon ECR Public Gallery)。
- **リポジトリポリシー**(リソースベース)で**別アカウントからの pull を許可**します。クロスアカウントのデプロイでは、**ECR リポジトリポリシー + 相手側ロールの `ecr:GetAuthorizationToken` (これはリソース `*` に対する IAM ポリシー側で必要)** の両方が要ります。ここが「別アカウントの ECS タスクがイメージを取得できない」の定番原因です。
- **プルスルーキャッシュ(pull through cache)**: 外部の公開レジストリのイメージを ECR にキャッシュします。CodeArtifact の外部接続と同じ発想で、**外部レジストリのレート制限や障害からビルドを守る**打ち手です。
- **VPC エンドポイント**: プライベートサブネットのタスクが ECR からイメージを取得するには、`ecr.api` ・ `ecr.dkr` **のインターフェースエンドポイントと、レイヤー実体のための `com.amazonaws.<region>.s3` ゲートウェイエンドポイント(または NAT)** が要ります。**S3 側を忘れて「イメージの pull が途中で止まる」**が頻出のトラブルです。

Amazon S3 の設定項目で問われるもの。

- **バージョニング**(切り戻し・誤削除対策)、**サーバーサイド暗号化**、**バケットポリシー**、**ブロックパブリックアクセス**。
- `bucket-owner-full-control` : 別アカウントが書き込んだオブジェクトの所有権。**S3 のオブジェクト所有者設定(Bucket owner enforced)** を使えば ACL に依存せず解決できます。

1-3-5 成果物を生成するビルドツールの設定 —— CodeBuild・Lambda [1.3]

コンテナイメージを作る典型的な `buildspec.yml` の流れは、そのまま設問の材料になります。

1. `pre_build`：`aws ecr get-login-password | docker login --username AWS --password-stdin <account>.dkr.ecr.<region>.amazonaws.com`
2. `build`：`docker build -t $REPO:$IMAGE_TAG .`
3. `post_build`：成功時のみ `docker push`、そして `imagedefinitions.json`（ECS の標準デプロイ用）または `imageDetail.json`（ECS の Blue/Green 用）を出力
4. `artifacts`：上記 JSON とタスク定義テンプレートを成果物として出力

押さえる点を挙げます。

- `imagedefinitions.json` は `[{"name":"<コンテナ名>","imageUri":"<イメージURI>"}]` の形式で、**CodePipeline の ECS 標準デプロイアクション**が読み取ります。コンテナ名がタスク定義のものと一致していないと失敗します。
- `imageDetail.json` は `{"ImageURI":"<イメージURI>"}` で、**ECS の Blue/Green（CodeDeploy）アクション**が使います。**2つを取り違える**のが定番の誤りです。
- **イメージタグに `latest` を使わない。コミットハッシュやビルド ID をタグにする**ことで、どのタスクがどのコードで動いているかが特定でき、切り戻し先も一意になります。
- **CodeBuild のサービスロール**には、ECR への `ecr:GetAuthorizationToken`（リソース `*`）、

`ecr:BatchCheckLayerAvailability`・`ecr:PutImage`・`ecr:InitiateLayerUpload`・`ecr:UploadLayerPart`・`ecr:CompleteLayerUpload`(対象リポジトリ)が要ります。

- **AWS Lambda のデプロイパッケージ**: zip かコンテナイメージ。zip は 50MB(直接アップロード)、S3 経由で 250MB(解凍後)の制限があり、**大きな依存は Lambda レイヤーへ切り出すかコンテナイメージ(最大10GB)**を選びます。**AWS SAM の** `sam build` **と** `sam package` がこの成果物生成を担い、`sam package` **は成果物を S3 へアップロードし、テンプレートのローカルパスを S3 の URI へ書き換えます**。

1-3-6 EC2 インスタンスとコンテナのイメージ生成を自動化する —— EC2 Image Builder [1.3]

EC2 Image Builder は、**AMI とコンテナイメージの作成・テスト・配布をパイプラインとして自動化**するサービスです。「ゴールデンイメージ(標準イメージ)を毎月のパッチ適用込みで作り直したい」という要件の標準解です。

構成要素は暗記対象です。

要素	役割
コンポーネント(component)	インストールやテストの手順を YAML(AWSTOE 形式)で書いた部品。ビルド用とテスト用がある
イメージレシピ(image recipe)	**ベースイメージ + コンポーネントの並び + ストレージ設定**。AMI 向け
コンテナレシピ(container recipe)	ベースイメージ + コンポーネント + Dockerfile テンプレート。**出力先は Amazon ECR**

要素	役割
インフラストラクチャ設定	**どこでビルドするか**(インスタンスタイプ、サブネット、セキュリティグループ、インスタンスプロファイル、SNS 通知先、ログ出力先の S3)
ディストリビューション設定	**どこへ配るか**(リージョン、共有先アカウント/組織、AMI の起動許可、KMS キー、ライセンス設定)
イメージパイプライン	上記を束ね、**スケジュール実行**(cron)や手動実行を行う

動きの順序は「**ビルド段階 → テスト段階 → 配布**」です。**テスト段階では、作成したイメージから新しいインスタンスを起動して検証**し、**成功した場合にだけ配布**されます。テストに失敗したイメージは配布されません。

設計判断のポイントを並べます。

- **依存関係の自動更新**: パイプラインの `pipelineExecutionStartCondition` を `EXPRESSION_MATCH_AND_DEPENDENCY_UPDATES_AVAILABLE` にすると、**ベースイメージやコンポーネントに更新があるときだけ**スケジュール実行されます。無駄なビルドを避けたい要件の答えです。
- **共有**: **AWS Resource Access Manager(RAM)** でコンポーネントやレシピを組織内へ共有し、**ディストリビューション設定で AMI を他アカウントへ配布**します。「組織全体で同じ標準 AMI を使わせたい」の解です。
- **セキュリティ強化**: AWS 提供の**強化(hardening)コンポーネント**(STIG など)をレシピに含められます。
- **パッチ適用の位置づけ**: **Image Builder は「新しいイメージを作り直す」イミュータブルなパッチ運用、AWS Systems Manager Patch Manager は「動いているインスタンスに当てる」ミュータブルなパッチ運**

用です(1-4-3、2-3-3)。**どちらを選ぶかは、インスタンスを作り直せるかどうか**で決まります。

- **料金**: Image Builder 自体の利用料は無く、**ビルドに使う EC2・EBS スナップショット・S3・ECR の実費のみ**です。
- **既存資産との対比**:「Packer で AMI を作っている運用を、運用負荷を下げつつ AWS へ寄せたい」なら Image Builder が答えになります。Packer を EC2 上の Jenkins で回し続ける案は、**運用負荷の軸で負けます**。

> **落とし穴**:「Image Builder で作った AMI を、既存の Auto Scaling グループへ自動反映したい」場合、**Image Builder は起動テンプレートを更新できます**(ディストリビューション設定で起動テンプレートの新バージョンを作成)。その後の入れ替えは **Auto Scaling のインスタンスリフレッシュ**が担当します(1-4-3)。**Image Builder だけで既存インスタンスが入れ替わるわけではありません**。

1-3-7 場面演習 [1.3]

場面A: 本番のタスクが image not found で起動しない。 状況 —— ECS のサービスがスケールアウトしようとしたとき、`CannotPullContainerError` でタスクが起動しない。数週間前にデプロイしたイメージで、ECR のライフサイクルポリシーは「作成から30日を超えたイメージを削除」となっている。

原因は、**ライフサイクルポリシーが、いま本番で使っているイメージを削除した**ことです。対策は2つ。①ライフサイクルポリシーを**タグの接頭辞で保護**する(`prod-` で始まるタグは対象外、または直近 N 個だけ残す)。②**タグのイミュータビリティ**を有効にし、**デプロイ時に必ず新しいタグを付ける**運用にすることで、「動いているイメージが消える」経路を減らします。

「タスク定義のイメージを `latest` に変える」案は、**どのイメージが動いているか分からなくなる**ので逆効果です。「ECR のスキャンを有効にする」は無関係です。

場面B: ビルドは通るが、本番だけ依存ライブラリのバージョンが違う。 状況 —— 開発者のローカル、検証環境、本番でライブラリのマイナーバージョンが揃わず、本番でだけ例外が出る。各環境が個別に npmjs からパッケージを取得している。

解は、**AWS CodeArtifact の承認済みリポジトリに一本化**することです。①ドメインとリポジトリを作り、外部接続で npmjs を上流にする。②開発者・CodeBuild のすべてが CodeArtifact を向くように設定する。③**ロックファイルをリポジトリにコミットして固定**する。④さらに根本的な対策として、**1回だけビルドした成果物(コンテナイメージ)を全環境へ昇格**させ、環境ごとの再ビルドをやめます。

「各環境で `npm install` の前にバージョンを確認するスクリプトを入れる」案は、**ずれを検出するだけで防いでいません。**

場面C: 全社の標準 AMI を毎月更新したい。 状況 —— 30 のアカウントに散らばる EC2 が、それぞれ別々の AMI から起動している。監査から「OS のパッチ水準を揃え、CIS の強化設定を全社共通にせよ」と指示された。運用負荷は最小にしたい。

設計は次のとおりです。①共有サービスアカウントに **EC2 Image Builder のイメージパイプライン**を作り、ベース AMI に**強化コンポーネント + 社内エージェントの導入コンポーネント**を重ねたレシピを組む。②**スケジュールを毎月にし、依存更新があるときだけ実行**する条件を付ける。③**テスト段階**で起動確認と設定検査を行う。④**ディストリビューション設定**で AWS Organizations の組織 ID を指定して、**全アカウント・必要リージョンへ AMI**

を配布する。⑤各アカウントの起動テンプレートを新 AMI で更新し、**Auto Scaling のインスタンスリフレッシュ**で入れ替える。

「各アカウントで Patch Manager を回す」案も水準は揃いますが、**起動時点のイメージがばらばらのまま**で、新規インスタンスは毎回パッチ適用の待ち時間を抱えます。「アカウントごとに Packer を回す」案は運用負荷が 30 倍になります。

1-4 インスタンス・コンテナ・サーバーレス環境向けのデプロイ戦略を実装する [1.4]

1-4-1 プラットフォーム別のデプロイ手法 —— EC2・ECS・EKS・Lambda [1.4]

同じ「ブルー/グリーン」という言葉でも、プラットフォームによって実現の仕方と制約が違います。ここを表で持っておくと、選択肢の切り分けが速くなります。

プラットフォーム	主なデプロイ手法	切り替えの実体	押さえる制約
Amazon EC2	CodeDeploy の In-place / Blue/Green、Auto Scaling のインスタンスリフレッシュ、CloudFormation の `UpdatePolicy` によるローリング更新	**ロードバランサーのターゲット登録の入れ替え**、または新しい Auto Scaling グループへの差し替え	**CodeDeploy エージェント**が要る。In-place は容量が一時的に減る

プラットフォーム	主なデプロイ手法	切り替えの実体	押さえる制約
Amazon ECS	ECS のローリング更新(既定)、**CodeDeploy による Blue/Green**、外部デプロイコントローラー	**タスクセット**と**ロードバランサーのターゲットグループの差し替え**	`minimumHealthyPercent` と `maximumPercent` で更新中の容量が決まる
Amazon EKS	Kubernetes の Deployment のローリング更新、Blue/Green(Service の向き先変更)、カナリア(サービスメッシュや Ingress の重み)	Pod の入れ替え、Service/Ingress のルーティング	**CodeDeploy は EKS を直接サポートしない**。Kubernetes 側の仕組みか、Argo Rollouts などを使う
AWS Lambda	**エイリアスの重み付け**によるカナリア/リニア、**CodeDeploy によるトラフィックシフト**、AWS SAM の `DeploymentPreference`	**バージョンとエイリアス**。エイリアスが指す2つのバージョンへ重みで振り分ける	**重み付けは同一関数の2バージョン間のみ**。`$LATEST` は重み付けの対象にできない

それぞれの要点を掘ります。

Amazon EC2 の場合。

- **Auto Scaling のインスタンスリフレッシュ**は、**起動テンプレートの新バージョンに合わせて既存インスタンスを段階的に入れ替える**機能です。`MinHealthyPercentage`(維持する健全な割合)、`InstanceWarmup`(ウォームアップ時間)、**チェックポイント**(一定割合で一時停止して確認)、**スキップマッ

チング(すでに新しい設定のインスタンスは入れ替えない)を設定できます。**これがイミュータブルなデプロイの中心的な道具**です。

- **CloudFormation の** `UpdatePolicy`: Auto Scaling グループの更新方法を `AutoScalingRollingUpdate`(既存グループを順次入れ替え)、`AutoScalingReplacingUpdate`(グループごと作り直す)から選びます。後者は**イミュータブル**、前者は**ローリング**です。`MinInstancesInService`・`MaxBatchSize`・`PauseTime`・`WaitOnResourceSignals` が設定項目で、`WaitOnResourceSignals: true` **にすると** `cfn-signal` **が送られるまで次のバッチへ進みません**。「新しいインスタンスがアプリの起動を終える前に次のバッチへ進んでしまう」の対処がこれです。
- **AWS Elastic Beanstalk** も EC2 系のデプロイ手法として登場します。**All at once / Rolling / Rolling with additional batch / Immutable / Blue/Green(環境の URL スワップ)** の5通りで、**Immutable は新しい Auto Scaling グループに全台を作ってから入れ替える**、**Blue/Green は別環境を作って CNAME をスワップする**という違いを押さえます。

Amazon ECS の場合。

- **ローリング更新**では、`minimumHealthyPercent`(既定100)と `maximumPercent`(既定200)が更新中のタスク数を決めます。`maximumPercent` **が 100 のままだと新しいタスクを追加で立てられず、先に古いタスクを止めることになります**。「更新中に容量が落ちる」「更新が進まない」の原因として問われます。
- **サーキットブレーカー(deployment circuit breaker)**: 新しいタスクが繰り返し起動失敗したときに、**デプロイを自動的に失敗させ、**`rollback: true` **なら直前の安定版へ戻します**。「失敗したデプロイが延々と再試行を続ける」の対処です。

- **CodeDeploy による Blue/Green** では、**本番リスナーとテストリスナー**を分けられます。テストリスナーへ先に新タスクセットを出し、`AfterAllowTestTraffic` **フック(Lambda)で検証**してから本番トラフィックを移す構成が取れます。旧タスクセットは**指定時間だけ残せる**ので、切り戻しが即時です。

Amazon EKS の場合。

- **Deployment のローリング更新**は `maxSurge` と `maxUnavailable` で速度と可用性を調整します。`readinessProbe` が正しくないと、**準備できていない Pod へトラフィックが流れます**。
- **カナリア**は、Ingress コントローラーの重み付け、**AWS App Mesh** のような仕組み、または Argo Rollouts / Flagger で実現します。**CodeDeploy は使えません**。
- **クラスターのアップグレード順序**は、**コントロールプレーンを先、データプレーン(ノードグループ)を後**です。逆順や複数バージョンの飛び越しは不可です。

AWS Lambda の場合。

- **バージョン**は不変のスナップショット、**エイリアス**は名前付きのポインタです。**エイリアスの** `RoutingConfig` **で2つのバージョンへ重みを配分**します。
- **AWS SAM** では `AutoPublishAlias`(デプロイ時に新バージョンを発行してエイリアスを更新)と `DeploymentPreference`(`Canary10Percent5Minutes`、`Linear10PercentEvery1Minute`、`AllAtOnce` など)を書くだけで、**裏で CodeDeploy によるトラフィックシフト**が構成されます。`Alarms` に CloudWatch アラームを、`Hooks` に `PreTraffic` / `PostTraffic` の検証用 Lambda を指定できます。

- **プロビジョニング済み同時実行**を使っている場合、**新バージョンには別途プロビジョニングが必要**です。「カナリア中に新バージョンだけコールドスタートが多発する」の原因になります。

> **判断の基準**:「切り替えの粒度をトラフィックの割合で細かく制御したい」なら **Lambda か ECS の CodeDeploy Blue/Green**。「インスタンス単位でしか制御できない」EC2 の In-place では、**1台未満の粒度は作れません**(ただし ALB の加重ターゲットグループを併用すれば割合制御は可能で、この応用も出題されます)。

1-4-2 アプリケーションのストレージパターン —— EFS・S3・EBS [1.4]

デプロイ方式の選択は、アプリが状態をどこに置いているかで縛られます。ここが Professional らしい出題点です。

ストレージ	性質	デプロイ上の意味
Amazon EBS	**単一のインスタンスにアタッチするブロックストレージ**(io2 のマルチアタッチは例外)	インスタンスを作り直すと**中身は消える**(スナップショットを取らない限り)。**イミュータブルなデプロイと相性が悪い**状態の置き場
Amazon EFS	**複数のインスタンス/コンテナから同時にマウントできる共有ファイルシステム**(NFS)	インスタンスを入れ替えても**共有データが残る**。**イミュータブル/ブルーグリーンを可能にする鍵**。Lambda と ECS/Fargate からもマウント可能

ストレージ	性質	デプロイ上の意味
Amazon S3	オブジェクトストレージ。HTTP API でアクセス	**アップロードされたファイル・静的コンテンツ・成果物の置き場の第一候補**。インスタンスから完全に切り離せる

設計の原則は「**アプリケーションサーバーをステートレスにする**」ことです。具体的にはこうなります。

- **セッション状態**は、インスタンスのローカルディスクではなく **Amazon DynamoDB や Amazon ElastiCache** に置く。ローカルに置くと、**入れ替えのたびにログアウトが発生**します。
- **利用者がアップロードしたファイル**は **Amazon S3** に置く。EBS に置くと、**インスタンスを作り直せなくなります**。
- **複数台から同時に読み書きする必要のある作業領域**(レガシーアプリの共有ディレクトリ、コンテンツ管理システムのメディアフォルダ)は **Amazon EFS**。「アプリの改修なしでブルー/グリーンを可能にしたい」という要件で登場します。
- **ログ**はローカルに書き捨てず、**CloudWatch エージェントや Fluent Bit(ECS の FireLens)で外部へ送る**。「インスタンスを終了したら障害時のログが消えていた」の対策です。
- **一時的な高速作業領域**が要るなら EBS(gp3 / io2)やインスタンスストア。**インスタンスストアは停止で消える**点に注意。

> **よく出る取り違え**: 「EBS ボリュームを複数のインスタンスで共有して、ブルー/グリーンのデータ引き継ぎに使う」は原則として誤りです。**共有が要るなら Amazon EFS**。EBS のマルチアタッチは io2 系かつ同一 AZ、かつク

> ラスター対応ファイルシステムが前提という強い制約があり、汎用の解にはなりません。

コンテナでのストレージも問われます。ECS/Fargate では**バインドマウント(タスク内で共有)**、**Amazon EFS ボリューム(タスク間で共有・永続)**、**Fargate の一時ストレージ(既定20GB、最大200GB。タスク終了で消える)**の3つを切り分けます。EKS では **Amazon EBS CSI ドライバー**(単一 Pod 向け永続ボリューム)と **Amazon EFS CSI ドライバー**(複数 Pod 共有)を使い分けます。

1-4-3 ミュータブルなデプロイとイミュータブルなデプロイ [1.4]

この対比が第1章の背骨です。

	ミュータブル(可変)	イミュータブル(不変)
やること	**稼働中のサーバーの中身を書き換える**	**新しいサーバーを作って古いものを捨てる**
具体例	CodeDeploy の In-place、Systems Manager Patch Manager、構成管理エージェントによる収束	Auto Scaling のインスタンスリフレッシュ、EC2 Image Builder の新 AMI、Blue/Green、コンテナイメージの入れ替え
利点	速い。追加のインスタンス費用がかからない	**構成ドリフトが起きない**。テストしたものと同じものが動く。**切り戻しが確実**
欠点	**構成ドリフト**が蓄積する。失敗すると中途半端な状態が残る	一時的に**二重の容量**が要る。**状態を外に出す設計が前提**

	ミュータブル(可変)	イミュータブル(不変)
切り戻し	旧リビジョンを**再デプロイ**(時間がかかる)	**古い側へ向き先を戻すだけ**(速い)

構成ドリフト(configuration drift) とは、**手作業や個別のパッチ適用の積み重ねで、同じはずのサーバー同士が少しずつ違う状態になること**です。症状は「あるインスタンスだけエラーが出る」「再構築したら動かない」。イミュータブルは、この問題を**構造的に起こさない**のが最大の価値です。

判断の基準を明示します。

状況	選ぶ方
迅速な切り戻しが要件	**イミュータブル**
監査で「本番の構成が定義と一致していること」を求められる	**イミュータブル**(+ AWS Config によるドリフト検出)
インスタンスにローカルの永続状態がある(改修できない)	ミュータブル、または **EFS へ状態を移してからイミュータブル**
起動に時間がかかる(数十分のウォームアップ)	ミュータブルか、**事前ウォームアップ付きのイミュータブル**
コストが厳しく、追加容量を出せない	ミュータブル(またはローリング)
緊急のセキュリティパッチを既存台へ即時適用	**ミュータブル(Patch Manager)**。その後、次のイメージへ取り込む

> **Professional の考え方**: 実務では両方を使い分けます。**日常のリリースはイミュータブル、緊急のパッチはミュータブルで当てて、次のイメージビルドで恒久化する**。この「二段構え」を答えさせる設問が出ます。「イミュー

> タブルを採用したので Patch Manager は不要」という選択肢は、**緊急対応の穴**を作るため誤りです。

ミュータブルなデプロイとイミュータブルなデプロイ

	ミュータブル(可変)	イミュータブル(不変)
やること	稼働中のサーバーの 中身を書き換える	新しいサーバーを作って 古いものを捨てる
具体例	In-place、 Patch Manager	インスタンスリフレッシュ Blue/Green、コンテナ
利点	速い。追加のインスタンス 費用がかからない	構成ドリフトが起きない 切り戻しが確実
欠点	構成ドリフトが蓄積する	一時的に二重の容量が要る
切り戻し	旧リビジョンを再デプロイ	古い側へ向き先を戻すだけ

構成ドリフト = 手作業やパッチの積み重ねで、同じはずのサーバーが少しずつ違う状態になること
日常のリリースはイミュータブル、緊急のパッチはミュータブルで当てて次のイメージへ取り込む

図 1-6　ミュータブルなデプロイとイミュータブルなデプロイ

1-4-4 コードを配る道具 —— CodeDeploy と Image Builder [1.4]

役割の違いをはっきりさせます。

	AWS CodeDeploy	EC2 Image Builder
配るもの	**アプリケーションのリビジョン**(コードと設定)	**マシンイメージ**(AMI、コンテナイメージ)
対象	稼働中の EC2/オンプレミス、ECS サービス、Lambda 関数	新しく作るイメージ

	AWS CodeDeploy	EC2 Image Builder
位置づけ	**ミュータブルにもイミュータブルにも使える**(In-place / Blue/Green)	**イミュータブル運用の源**(OS 層ごと作り直す)
切り替え制御	デプロイ設定(AllAtOnce / Linear / Canary)、フック、自動ロールバック	起動テンプレートの更新まで。入れ替えは Auto Scaling が担当

組み合わせの型を覚えます。

1. **OS 層まで作り直すイミュータブル**: Image Builder が AMI を作る → 起動テンプレートを更新 → **Auto Scaling のインスタンスリフレッシュ**で入れ替え。**アプリの入れ替えごとに AMI を焼く**ので所要時間は長いが、最も再現性が高い。
2. **AMI は共通、アプリだけ配る**: 共通の AMI で起動 → **CodeDeploy がアプリのリビジョンを配る**。デプロイが速く、AMI のビルド時間が要らない。**ただし OS 層のドリフトは Patch Manager や Image Builder による定期更新で別に管理**する。
3. **コンテナ**: CodeBuild がイメージを作る → ECR へ push → **CodeDeploy(ECS Blue/Green)または ECS のローリング更新**で入れ替え。**イミュータブルが既定**になる。

その他の配布手段も候補として挙がります。

- **AWS Systems Manager Run Command / State Manager**: エージェント経由でコマンドや設定を配る。**アプリのデプロイにも使えるが、デプロイ設定・自動ロールバック・ライフサイクルフックのような仕組みが無い**ため、アプリのリリースには CodeDeploy が優ります。

- **AWS Systems Manager Distributor**: **ソフトウェアパッケージ(エージェント類)を定義してインスタンスへ配布・更新**する仕組み。「監視エージェントを全インスタンスへ配り、バージョンを揃えたい」の解です。
- **AWS AppConfig**: **コードを配らずに設定と機能フラグだけを配る**。「再デプロイなしで機能を出したり引っ込めたりしたい」の解で、**デプロイ戦略(Linear / Exponential)・ベイクタイム・CloudWatch アラームによる自動ロールバック**を持ちます。

> **判断の基準**:「アプリの再デプロイなしで振る舞いを変えたい」なら **AWS AppConfig の機能フラグ**。「新しいバイナリを配りたい」なら CodeDeploy。「OS も含めて新品にしたい」なら Image Builder。**この3択の切り分けが設問の中心**です。

1-4-5 成果物リポジトリへのアクセス権限の設定 —— IAM と CodeArtifact [1.4]

「誰が何を push でき、誰が pull できるか」を分けるのが原則です。

IAM の基本形を整理します。

主体	必要な権限	与え方
CodeBuild のビルド	ECR への push、S3 の成果物書き込み、CodeArtifact の読み書き	**CodeBuild サービスロール**
CodeDeploy	対象への操作(EC2 のタグ読み取り、ECS の更新、Lambda の更新)	**CodeDeploy サービスロール**
EC2 インスタンス	S3 の成果物読み取り、ECR の pull、SSM の操作	**インスタンスプロファイル (IAM ロール)**

主体	必要な権限	与え方
ECS タスク	ECR の pull(**タスク実行ロール**)、アプリからの AWS API 呼び出し(**タスクロール**)	2つのロールを取り違えないこと
開発者	CodeArtifact の読み取り、必要なら push	IAM ロール(IAM Identity Center 経由)

CodeArtifact の権限は2層で考えます。

- **ドメインポリシー**(リソースベース): ドメイン全体に対する許可。**組織内の別アカウントへリポジトリの利用を許可**するときに使います。
- **リポジトリポリシー**(リソースベース): 個別リポジトリの読み書き。
- **IAM ポリシー**(アイデンティティベース): 主体側の許可。`codeartifact:GetAuthorizationToken` と `sts:GetServiceBearerToken` の両方が要る点が頻出です。`sts:GetServiceBearerToken` **を忘れて認証トークンが取得できない**というのが定番のトラブルです。
- **KMS**: ドメインの暗号化キーがカスタマー管理キーなら、**利用側に** `kms:Decrypt` **が要ります**。

Amazon ECR の権限も同じ構造です。

- `ecr:GetAuthorizationToken` **はリソース** `*` **に対して**しか書けません(リポジトリ単位に絞れない)。
- **pull には** `ecr:BatchGetImage` **と** `ecr:GetDownloadUrlForLayer`、**push には** `ecr:PutImage` **と各種 Layer 系**が要ります。
- **クロスアカウントの pull** は、**リポジトリポリシーで相手アカウントを許可 + 相手側の IAM ポリシー**の両方が要ります。

- **AWS Organizations 全体へ許可**するなら、リポジトリポリシーの条件に `aws:PrincipalOrgID` を使います。

Amazon S3 の成果物バケットでは、**バケットポリシー・IAM ポリシー・KMS キーポリシー**の3点が揃っている必要があります(1-1-6)。

> **落とし穴**:「ECS のタスクがイメージを pull できない」場合、**タスクロールではなくタスク実行ロール**を見ます。タスクロールはアプリが AWS API を呼ぶためのもので、**イメージの取得はタスク実行ロールの仕事**です。この取り違えは頻出です。

1-4-6 デプロイエージェントの設定 —— CodeDeploy エージェント [1.4]

CodeDeploy エージェント(CodeDeploy agent) は、EC2/オンプレミスのデプロイで**インスタンス側に常駐して指示を取りに行くプログラム**です。**ECS と Lambda では不要**です。

押さえる点を並べます。

- **導入方法**: ユーザーデータでのインストール、**AWS Systems Manager の `AWS-ConfigureAWSPackage` ドキュメントによる導入と自動更新**(State Manager のアソシエーションで定期実行すると常に最新に保てる)、AMI への焼き込み。**運用負荷が最小なのは Systems Manager 経由**です。
- **通信方向**: エージェントが**アウトバウンドで CodeDeploy と S3 へ接続**します。インバウンドのポート開放は不要です。プライベートサブネットからは **NAT ゲートウェイ、または `codedeploy`・`codedeploy-commands-secure`・`s3` の VPC エンドポイント**が要ります。

- **必要な権限**: インスタンスプロファイルに、**リビジョンを置いた S3 バケットからの** `s3:GetObject` と、**AWS が提供するリビジョンバケットへのアクセス**が要ります。
- **インスタンスの識別**: **タグ**(EC2)または **Auto Scaling グループ名**でデプロイグループの対象を決めます。「新しく起動したインスタンスに最新のアプリが入っていない」場合、**Auto Scaling グループをデプロイグループの対象にしていない**のが原因です。**CodeDeploy は Auto Scaling のライフサイクルフックを使い、スケールアウトで増えたインスタンスへ自動的に最新リビジョンを配ります。**
- **よくある失敗と切り分け**:

症状	見るところ
インスタンスが `Pending` のまま進まない	エージェントが動いているか(`codedeploy-agent status`)、エンドポイントへ到達できるか
`DownloadBundle` で `AccessDenied`	インスタンスプロファイルの S3 権限、**KMS キーの許可**
`ApplicationStop` で毎回失敗	**前回リビジョンのスクリプトが壊れている。**デプロイ設定でこのイベントの失敗を無視する(`ignoreApplicationStopFailures`)か、手動でエージェントのデプロイ履歴を掃除する
`ValidateService` で失敗	検証スクリプトの終了コード、アプリの起動待ち時間

症状	見るところ
ログはどこか	インスタンス上の `/var/log/aws/codedeploy-agent/codedeploy-agent.log` と、デプロイごとのスクリプトログ。**CloudWatch エージェントで収集して集中管理**するのが推奨

1-4-7 ブルー/グリーンとカナリア —— デプロイ方式の全体像 [1.4]

方式ごとの性質を1つの表にまとめます。設問はこの表の1行を言い換えて出てきます。

方式	やり方	ダウンタイム	追加コスト	切り戻し速度	影響範囲
All at once(一括)	全台を同時に入れ替える	**あり**	なし	遅い(再デプロイ)	全利用者
ローリング	一定数ずつ入れ替える	なし(容量は一時的に減る)	なし	遅い	一部→全体へ拡大
ローリング + 追加バッチ	先に予備を足してから入れ替える	なし(容量も維持)	小(バッチ分)	遅い	一部
イミュータブル	新しいグループを全数作ってから切り替える	なし	**大(一時的に2倍)**	速い	全利用者(切替後)

方式	やり方	ダウンタイム	追加コスト	切り戻し速度	影響範囲
ブルー/グリーン	旧環境(ブルー)を残したまま新環境(グリーン)を作り、向き先を切り替える	なし	**大(一時的に2倍)**	**最速**	全利用者(切替後)
カナリア	ごく一部のトラフィックだけ新版へ流し、問題なければ拡大	なし	小〜中	速い	**ごく一部から**
リニア	一定割合ずつ均等に新版へ移す	なし	小〜中	速い	段階的

選び方の要件語との対応を明確にします。

シナリオに書かれる言葉	選ぶ方式
「ダウンタイムを一切許容できない」	ブルー/グリーン、イミュータブル、ローリング(追加バッチ付き)
「問題があれば数分以内に完全に元へ戻したい」	**ブルー/グリーン**
「新機能の影響を一部の利用者で先に確認したい」	**カナリア**
「デプロイ中も容量を落とせない」	ブルー/グリーン、イミュータブル、ローリング + 追加バッチ

シナリオに書かれる言葉	選ぶ方式
「追加のインフラ費用をかけられない」	ローリング(ブルー/グリーンは除外)
「本番のトラフィックで実際の性能を測ってから広げたい」	**カナリア + CloudWatch アラーム連動のロールバック**
「データベースのスキーマ変更を伴う」	**後方互換の二段階リリース**(下記)

データベースを伴うデプロイは Professional の頻出テーマです。ブルー/グリーンでアプリだけ切り替えても、**データベースが1つなら新旧のアプリが同じスキーマを同時に見る**ことになります。答えは**拡張・縮退(expand and contract)** です。

1. **拡張**: 新しい列やテーブルを**追加だけ**する(既存アプリは無視できる)。
2. **両対応のアプリを出す**: 新旧どちらのスキーマでも動くコードをデプロイする。
3. **データを移行**する。
4. **縮退**: 新アプリだけが動いていることを確認してから、**古い列やテーブルを削除**する。

「デプロイと同時に列を削除する」案は、**切り戻したときに旧アプリが動かなくなる**ため誤りです。**切り戻し可能性を壊す変更は、リリースと分ける**という原則で答えます。

トラフィックの切り替え手段も候補として問われます。

手段	使いどころ	注意点
Application Load Balancer の加重ターゲットグループ	EC2・ECS のカナリア/ブルーグリーン	割合を細かく制御できる。**リスナールールの重みを変えるだけで即時切替**

手段	使いどころ	注意点
Amazon Route 53 の加重ルーティング	環境ごと・リージョンごとの切り替え	**DNS の TTL とクライアントのキャッシュ**により切替が即時ではない。「即座に全トラフィックを戻したい」要件では ALB 側の切替に劣る
Amazon API Gateway のステージ変数とカナリアリリース	API のカナリア	ステージ単位で割合を指定
Lambda のエイリアス重み付け	Lambda のカナリア	同一関数の2バージョン間
Amazon CloudFront + Lambda@Edge / CloudFront Functions	エッジでの振り分け	Cookie やヘッダーによる固定化ができる

> **よく出る取り違え**: 「Route 53 の加重ルーティングでブルー/グリーンを行えば、切り戻しは即座」は誤りです。**DNS のキャッシュが残るため、一部の利用者は旧レコードを使い続けます。即時性が要件なら ALB のリスナー/ターゲットグループの切り替え**を選びます。

1-4-8 デプロイの問題を切り分ける [1.4]

症状から原因へたどる表を持っておくと、トラブルシューティングの設問が速く解けます。

症状	疑う原因	確認する場所
CodeDeploy のデプロイが `Failed` だがログが無い	エージェント未導入・停止、権限不足	エージェントの状態、インスタンスプロファイル、CloudWatch Logs

症状	疑う原因	確認する場所
ECS のタスクが起動と停止を繰り返す	ヘルスチェックの失敗、コンテナの終了コードが 0 以外、メモリ不足	`stoppedReason`、`ExitCode`、タスクのログ、ターゲットグループのヘルスチェック設定
ECS のデプロイがずっと `IN_PROGRESS`	`maximumPercent` が足りない、サブネットの IP 不足、ENI 上限、ECR から pull できない	サービスイベント、サブネットの空き IP、VPC エンドポイント
ALB のターゲットが `unhealthy` になる	ヘルスチェックのパス・ポート・成功コード、**セキュリティグループ**、アプリの起動時間	ターゲットグループのヘルスチェック、SG のルール、`HealthCheckGracePeriod`
Blue/Green の切り替え後に 5xx が急増	新環境の設定不備、コネクションプールの枯渇、依存サービスの権限不足	CloudWatch メトリクス、**自動ロールバックのアラーム設定**
Lambda のカナリアで新バージョンだけ遅い	コールドスタート、**プロビジョニング済み同時実行が旧バージョンにしか無い**	Lambda のメトリクス、X-Ray
CloudFormation のスタック更新がロールバックする	依存リソースの作成失敗、`cfn-signal` **が来ない**、IAM 権限不足	スタックのイベントタブの「ステータスの理由」
デプロイは成功したのに古いコードが動いている	**キャッシュ**(CloudFront、ブラウザ)、`latest` タグの使い回し	CloudFront の無効化、イメージタグの設計
Auto Scaling で増えたインスタンスだけ古い	**Auto Scaling グループをデプロイグループの対象にしていない**、起動テンプレートが古い AMI	CodeDeploy のデプロイグループ設定、起動テンプレートのバージョン

切り分けの順序は決まっています。①**どのライフサイクルイベント/ステージで止まったか**を特定する、②**そのイベントのログ**を見る、③**権限・ネットワーク・ヘルスチェック**の3点を順に確かめる。**「まず再実行する」は Professional の答えになりません。**

ロールバックの設計も忘れずに。**CodeDeploy の自動ロールバック(デプロイ失敗時 + CloudWatch アラーム発報時)、ECS のサーキットブレーカー、CloudFormation のスタック更新失敗時の自動ロールバックと** `--disable-rollback` **、AppConfig のベイクタイム中のアラーム連動ロールバック。「異常を検知したら人手を介さず戻る」構成にできているか**が評価軸になります。

1-4-9 場面演習 [1.4]

場面A: 切り戻しを5分以内にしたい EC2 アプリ。 状況 —— ALB の背後にある Auto Scaling グループ(20台)で動く Java アプリ。現在は CodeDeploy の In-place(`HalfAtATime`)でデプロイしており、不具合時の切り戻しに 25 分かかる。セッションはローカルディスクに保存している。追加コストは一時的なら許容できる。

設計はこうなります。①**セッションを Amazon ElastiCache か DynamoDB へ外出し**して、インスタンスをステートレスにする(これをやらないとブルー/グリーンで利用者がログアウトします)。②CodeDeploy のデプロイタイプを **Blue/Green** に変更し、**新しい Auto Scaling グループを作って ALB のターゲットグループを差し替える**構成にする。③**旧環境の終了を60分待機**する設定にして、その間は**ターゲットグループを戻すだけで切り戻し**が済むようにする。④**CloudWatch アラーム(5xx 率、レイテンシ)を自動ロールバックに紐づける。**

「In-place のまま `AllAtOnce` にする」案は切り戻しを速くしません(むしろダウンタイムが出ます)。「Route 53 で切り替える」案は **DNS キャッシュのため即**

時性を満たしません。

場面B: レガシー CMS の共有ディレクトリ。状況 —— EC2 上の CMS が、`/var/www/uploads` に利用者のアップロードファイルを保存している。ブルー/グリーンにしたいが、新環境にファイルが無い。アプリのコードは改修できない。

解は **Amazon EFS** です。①EFS ファイルシステムを作り、既存のファイルを移行する。②起動テンプレートのユーザーデータ(またはイメージ)で `/var/www/uploads` に EFS をマウントする。③これで**新旧どちらの環境からも同じファイルが見える**ため、ブルー/グリーンが成立します。

「S3 に移す」案はアプリ改修が要る(要件に反する)ため使えません。「EBS スナップショットから新環境のボリュームを作る」案は、**切り替え後に旧環境へ書き込まれたファイルが失われます。**

場面C: 新機能の影響を段階的に確認したい API。状況 —— API Gateway + Lambda の API。新機能をまず社内利用者だけに出し、次に 10% の一般利用者、問題が無ければ全体へ広げたい。エラー率が 1% を超えたら自動で戻したい。

設計は、① **Lambda のエイリアスに重み付け**(AWS SAM の `DeploymentPreference` を `Canary10Percent10Minutes` に)、② **`Alarms` に CloudWatch のエラー率アラームを指定**して、超えたら CodeDeploy が自動ロールバックする、③社内利用者だけへの先行公開は、AWS AppConfig の機能フラグ**で利用者属性による分岐を入れる、④さらに **`Hooks` の `PreTraffic` 関数**で、トラフィックを流す前にスモークテストを走らせる。

「API Gateway のステージを分けて社内用の URL を配る」案も先行公開はできますが、**同一の本番経路で段階的に広げる要件**には合いません。「デプロイ

後に手動で監視して問題があれば戻す」案は、**自動ロールバックの要件を満たしていません**。

場面D: ECS のデプロイが終わらない。状況 —— Fargate の ECS サービス(希望タスク数 6)を更新したところ、40 分経ってもデプロイが完了しない。サービスイベントには新しいタスクの起動と停止が繰り返し記録されている。

確認の順序は、①**停止したタスクの** `stoppedReason` **と** `ExitCode`(アプリが起動直後に落ちていないか、`CannotPullContainerError` でないか)、②**ターゲットグループのヘルスチェック**(パス・ポート・`HealthCheckGracePeriod` がアプリの起動時間より短くないか)、③**サブネットの空き IP と ENI 上限**、④**タスク実行ロールの ECR/Secrets Manager の権限**。

恒久対策は **ECS のデプロイサーキットブレーカー(** `rollback: true` **)** を有効にして、**失敗が続くデプロイを自動で止めて直前の安定版へ戻す**ことです。「希望タスク数を増やす」「タイムアウトを延ばす」は原因に当たっていません。

第1章の試験ポイント [1.1/1.2/1.3/1.4]

- **CI / 継続的デリバリー / 継続的デプロイメント**の3語。**承認が要るのは継続的デリバリー**。
- **build once, deploy many**。環境ごとに再ビルドしない。環境差は Parameter Store・AppConfig・環境変数で吸収する。
- **AWS CodeBuild** = ビルドとテスト。`buildspec.yml` のフェーズ、`post_build` **は失敗時も走る**、VPC 接続、キャッシュ、レポート。
- **AWS CodeDeploy** = 配布と切り替え。**EC2 は In-place と Blue/Green、ECS と Lambda は Blue/Green のみ**。EC2 だけ**エージェントが必要**。

- `appspec.yml` **のライフサイクルイベント順**と、`ApplicationStop` **は前回リビジョンのスクリプトが動く**こと。
- **AWS CodePipeline** = オーケストレーション。**アーティファクトストアは S3、クロスアカウントはカスタマー管理の KMS キーが必須**、手動承認、`runOrder` で並列化、Invoke で Lambda / Step Functions。
- **シークレット**: ローテーションが要るなら **AWS Secrets Manager**、単なる設定値なら **AWS Systems Manager Parameter Store**。**長期のアクセスキーを持たず、IAM ロールで渡す**のが最良。
- **テストの配置**: 速いものを前へ。**PR ではユニット・静的解析・シークレットスキャン、負荷テストと UI テストは後段か夜間**。
- **終了コード 0 が成功**。`set -o pipefail` と `CODEBUILD_BUILD_SUCCEEDING` を忘れない。
- **成果物**: パッケージは **AWS CodeArtifact**、イメージは **Amazon ECR**、中間成果物は **Amazon S3**。**外部レジストリ障害への備えは CodeArtifact の外部接続 / ECR のプルスルーキャッシュ**。
- `imagedefinitions.json` **(ECS 標準デプロイ)と** `imageDetail.json` **(ECS Blue/Green)** の使い分け。
- **EC2 Image Builder** = レシピ・コンポーネント・インフラ設定・ディストリビューション設定・パイプライン。**ビルド → テスト → 配布**の順で、**テストに合格したものだけ配布**される。
- **ミュータブル対イミュータブル。切り戻しの速さと構成ドリフトの有無**が分かれ目。日常はイミュータブル、緊急パッチはミュータブル。
- **ストレージ**: 共有が要るなら **Amazon EFS**、オブジェクトは **Amazon S3**、単一インスタンス専用は **Amazon EBS**。**ステートレス化がイミュータブルの前提**。

- **切り替え手段**: 即時性が要るなら **ALB の加重ターゲットグループ**。**Route 53 は DNS キャッシュのため即時ではない**。
- **データベースを伴うリリースは拡張・縮退の二段階**。切り戻し可能性を壊す変更をリリースと同時に行わない。
- **自動ロールバック**: CodeDeploy(失敗時 + CloudWatch アラーム)、ECS のサーキットブレーカー、CloudFormation の自動ロールバック、AppConfig のベイクタイム。

サンプルはここまでです

続き(第2章〜巻末)は、DOP-C02 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。

DOP-C02 学習用テキスト(無料サンプル)

版
2026年7月版(AWS Certified DevOps Engineer - Professional (DOP-C02) Exam Guide 準拠)

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

お問い合わせ
info@shikaku-doujou.com