株式会社デュクシ

Think only pleasant thoughts
Survive just by having fun

ECSとCodePipelineでウェブアプリケーションを構築する

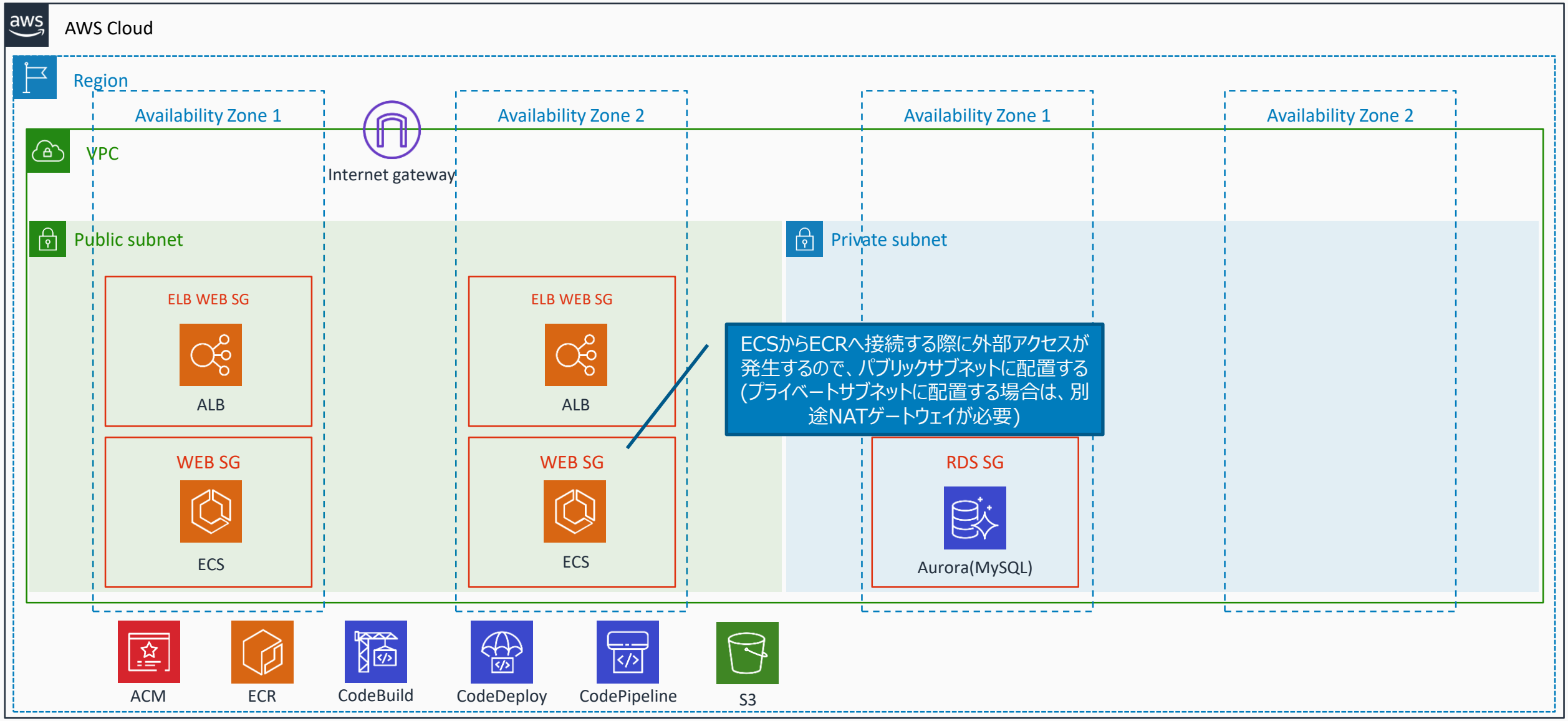
著者: 丸山浩之

目次

目的

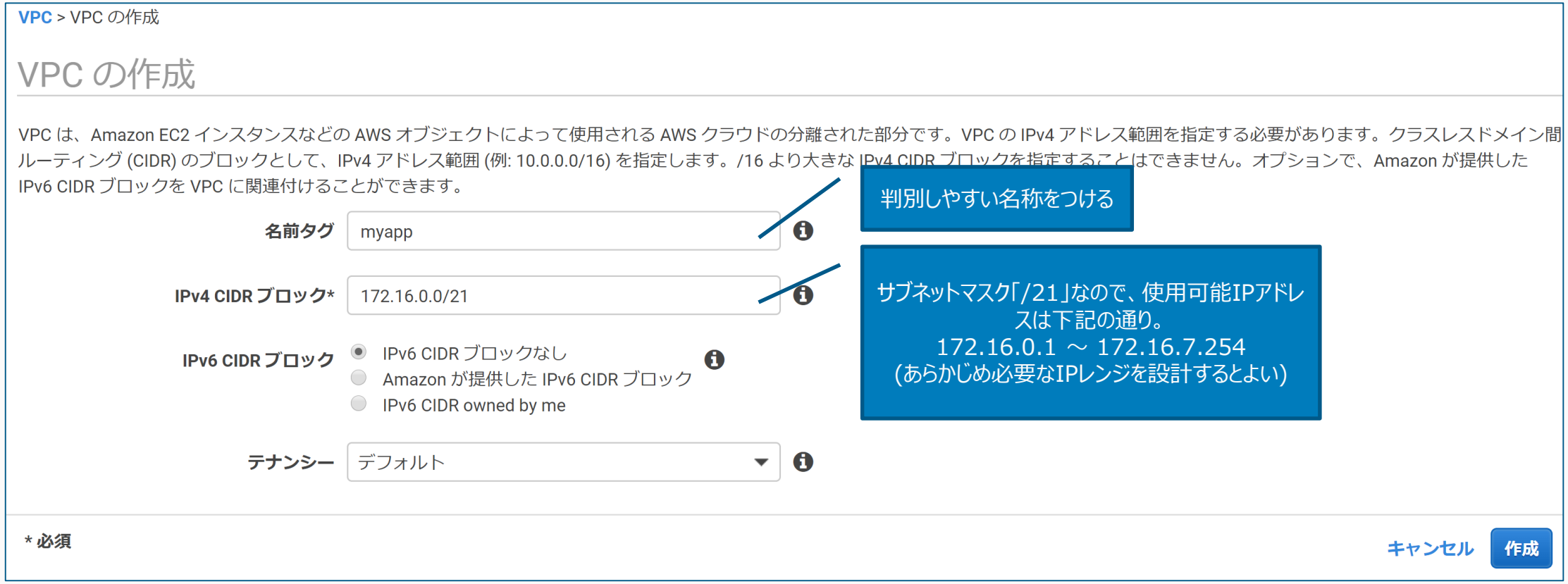
1. CakePHP 3.8.10で制作したウェブアプリケーションを、 ECS(Fargate) と CodePipleline を利用して公開する
2. 公開する際は、ACMを利用してSSL対応する

全体構成図


AWS Cloud
Region
Availability Zone 1
Availability Zone 2
Availability Zone 1
Availability Zone 2
VPC
Internet gateway
Public subnet
Private subnet
ELB WEB SG
ALB
ELB WEB SG
ALB
WEB SG
ECS
WEB SG
ECS
RDS SG
Aurora(MySQL)
ECSからECRへ接続する際に外部アクセスが発生するので、パブリックサブネットに配置する(プライベートサブネットに配置する場合は、別途NATゲートウェイが必要)
ACM
ECR
CodeBuild
CodeDeploy
CodePipeline
S3

VPC編

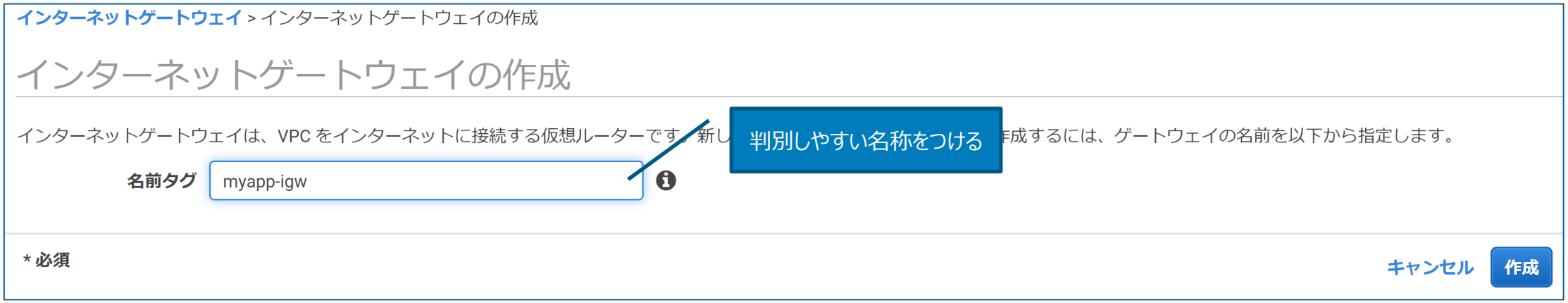
VPC作成


VPC > VPC の作成
VPC の作成
VPC は、Amazon EC2 インスタンスなどの AWS オブジェクトによって使用される AWS クラウドの分離された部分です。VPC の IPv4 アドレス範囲を指定する必要があります。クラスレスドメイン間ルーティング (CIDR) のブロックとして、IPv4 アドレス範囲 (例: 10.0.0.0/16) を指定します。/16 より大きな IPv4 CIDR ブロックを指定することはできません。オプションで、Amazon が提供した IPv6 CIDR ブロックを VPC に関連付けることができます。
名前タグ
myapp
IPv4 CIDR ブロック*
172.16.0.0/21
IPv6 CIDR ブロック
IPv6 CIDR ブロックなし
Amazon が提供した IPv6 CIDR ブロック
IPv6 CIDR owned by me
テナンシー
デフォルト
* 必須
キャンセル
作成
判別しやすい名称をつける
サブネットマスク「/21」なので、使用可能IPアドレスは下記の通り。
172.16.0.1 ～ 172.16.7.254
(あらかじめ必要なIPレンジを設計するとよい)

サブネット作成

1. サブネットは、同一の役割で使用するものを、異なるアベイラビリティーゾーンで2つ以上作成する(リージョンごとに使用できるAZは異なる)
2. 今回は、ALB向けとECSやDB向けを作成するため、下記の6つを作成する
    1. myapp-subnet-public-1 / myapp-subnet-public-2
    2. myapp-subnet-ecs-1 / myapp-subnet-ecs-2
    3. myapp-subnet-rds-1 / myapp-subnet-rds-2

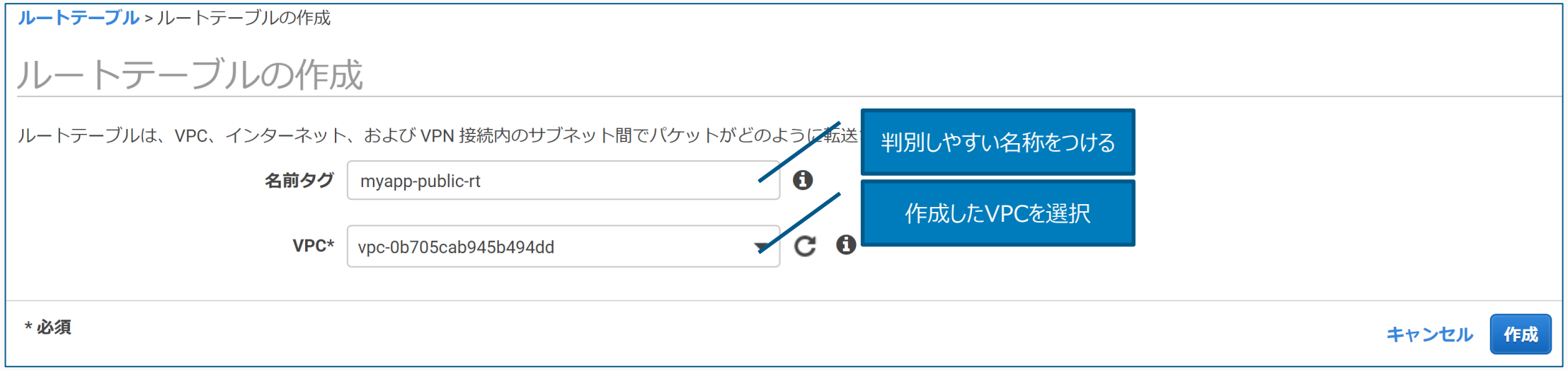
インターネットゲートウェイ作成



インターネットゲートウェイをVPCにアタッチ

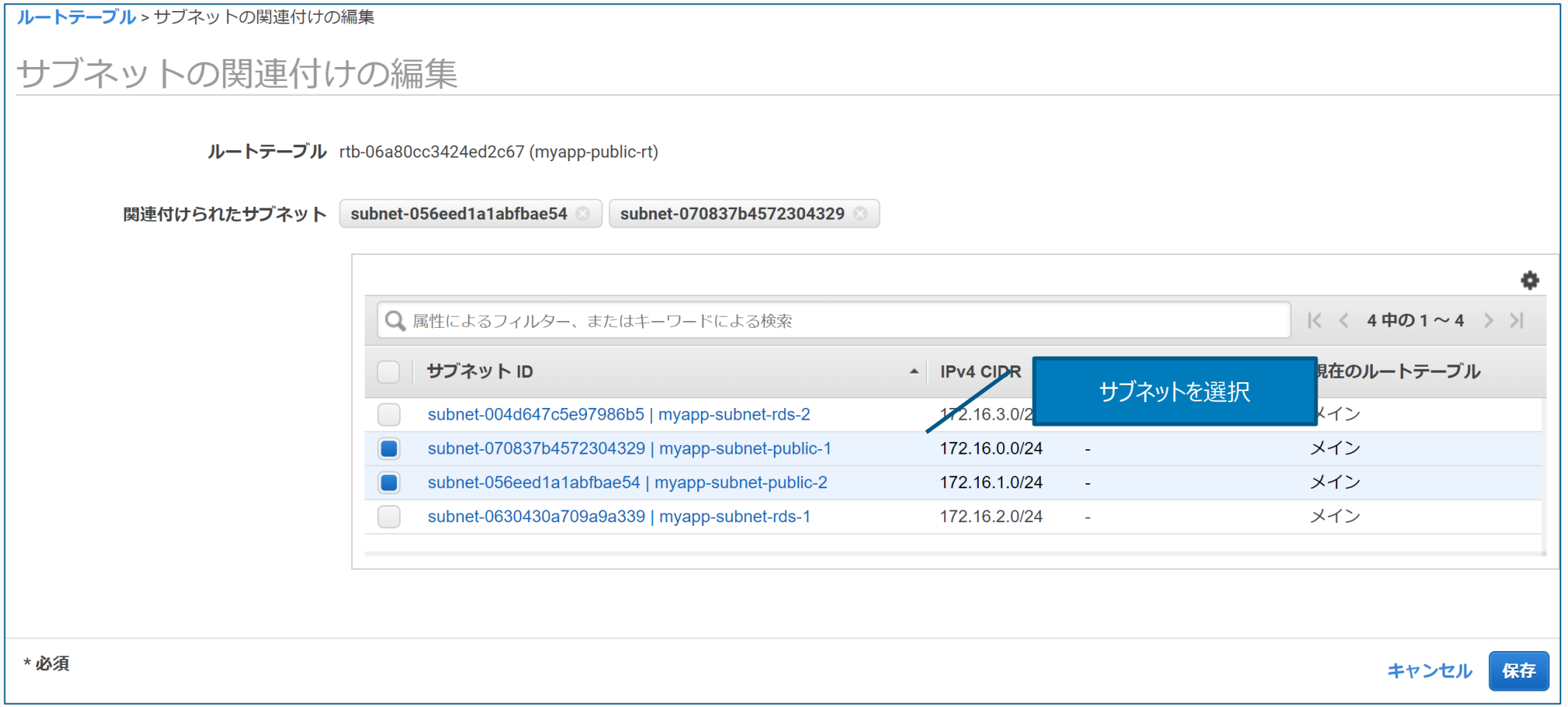
パブリックサブネット用のルートテーブル作成

1. VPC作成と同時に作られたメインルートテーブルとは別に、パブリックサブネット用のルートテーブルを作成する



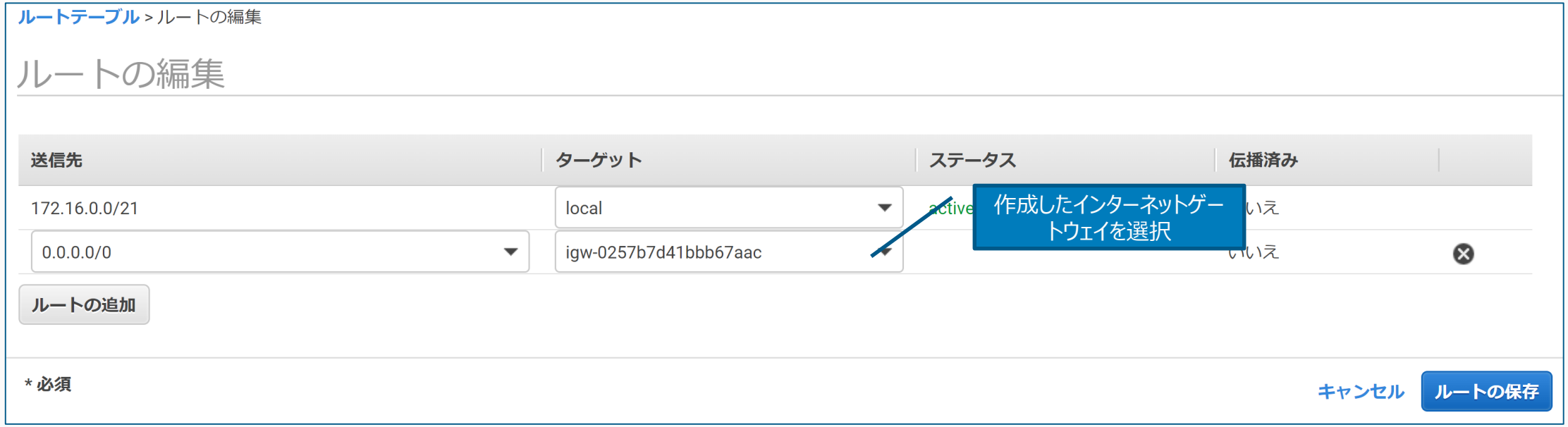
サブネット関連付け

1. パブリックサブネット用のルートテーブルに、サブネットの関連付けを行う
2. 外部公開するサブネットを選択する(下記の4つを外部公開向けサブネットとして使用する)
    1. myapp-subnet-public-1 / myapp-subnet-public-2
    2. myapp-subnet-ecs-1 / myapp-subnet-ecs-2



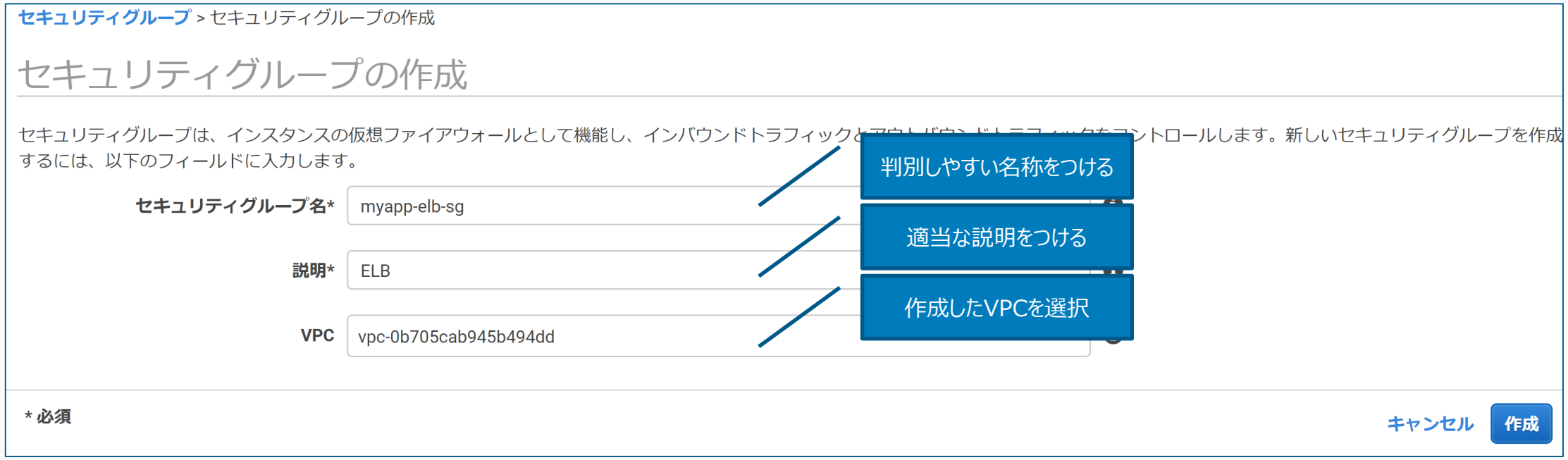
ルート編集

1. パブリックサブネット用のルートテーブルを外部公開するために、ルートにインターネットゲートウェイを追加する



セキュリティグループ作成

1. セキュリティグループは、ELB用、コンテナ用、RDS用という具合に細分化したほうが、アクセス制限に対応しやすい
2. 今回は、下記の3つを作成する
    1. myapp-elb-sg
    2. myapp-web-sg
    3. myapp-rds-sg



セキュリティグループのインバウンドルール編集

セキュリティグループのインバウンドルール編集(設定内容)

1. 外部からの接続を制限したい場合は、myapp-elb-sgのソースに接続元IPアドレスを指定する

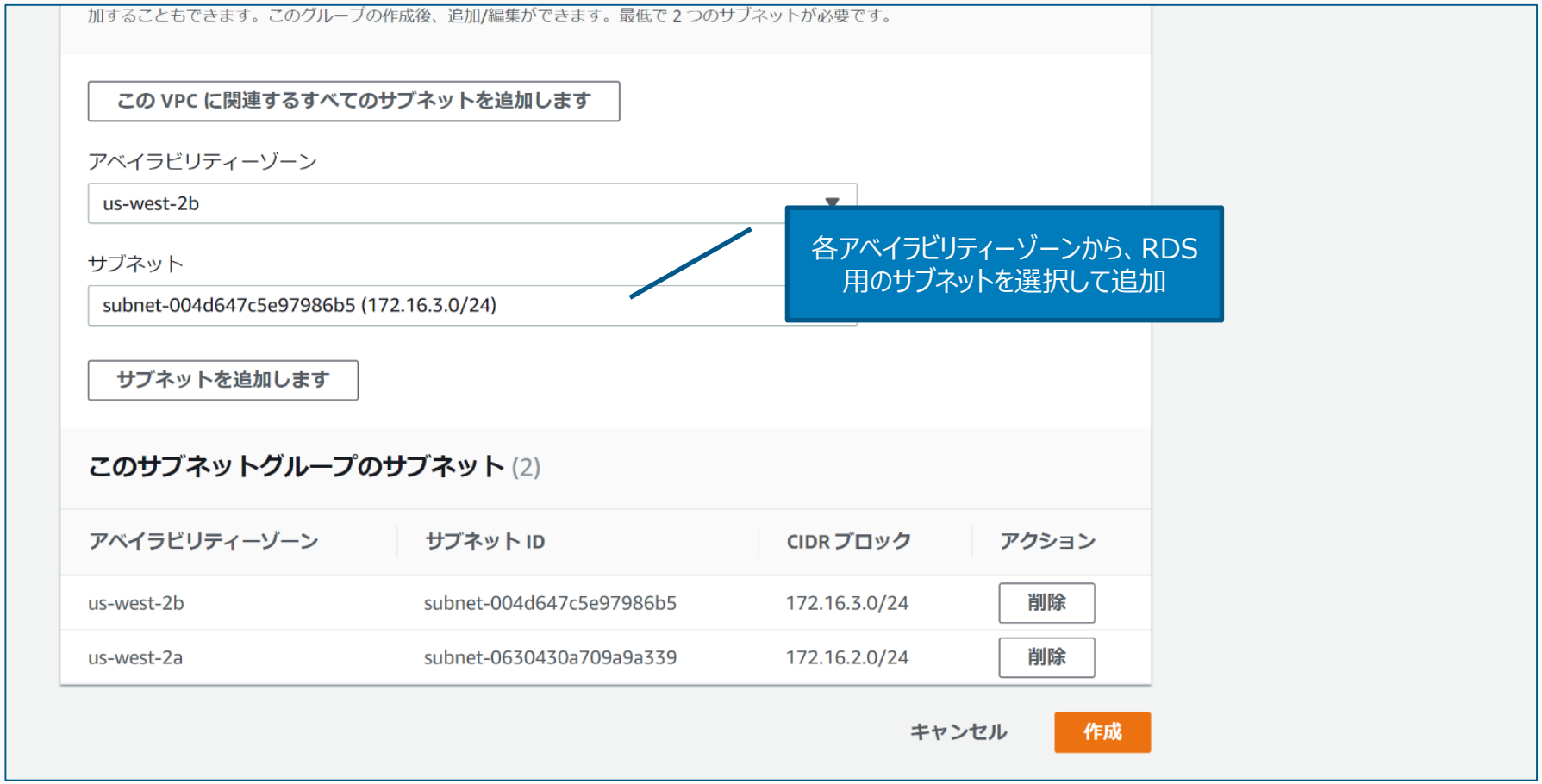
セキュリティグループ	プロトコル	ポート	ソース	説明
myapp-elb-sg	TCP	80	0.0.0.0/0, ::/0	HTTP
myapp-elb-sg	TCP	443	0.0.0.0/0, ::/0	HTTPS
myapp-elb-sg	TCP	8888	0.0.0.0/0, ::/0	HTTPのテストポート(CodeDeployで必要)
myapp-elb-sg	TCP	8443	0.0.0.0/0, ::/0	HTTPSのテストポート(CodeDeployで必要)
myapp-web-sg	TCP	80	グループID	myapp-elb-sgのグループID
myapp-web-sg	TCP	443	グループID	myapp-elb-sgのグループID
myapp-web-sg	TCP	8888	グループID	myapp-elb-sgのグループID
myapp-web-sg	TCP	8443	グループID	myapp-elb-sgのグループID
myapp-rds-sg	TCP	3306	グループID	myapp-web-sgのグループID

RDS編

サブネットグループ作成(サブネットグループの詳細)

1. RDSインスタンスに設定するサブネットグループを作成する(myapp-rds-subnet)
2. 画面が長かったので2ページに渡って説明

サブネットグループ作成(サブネットの追加)



データベース作成

1. RDSはアプリケーションの要件に合わせて適当なものを選択して作成する
2. 今回はAurora(MySQL)を選択して作成する
3. 設定内容はアプリケーションの要件にあわせて設定する
    1. セキュリティグループはRDS用に作成したものを使用する(ここではmyapp-rds-sgを使用する)

SSL証明書発行編

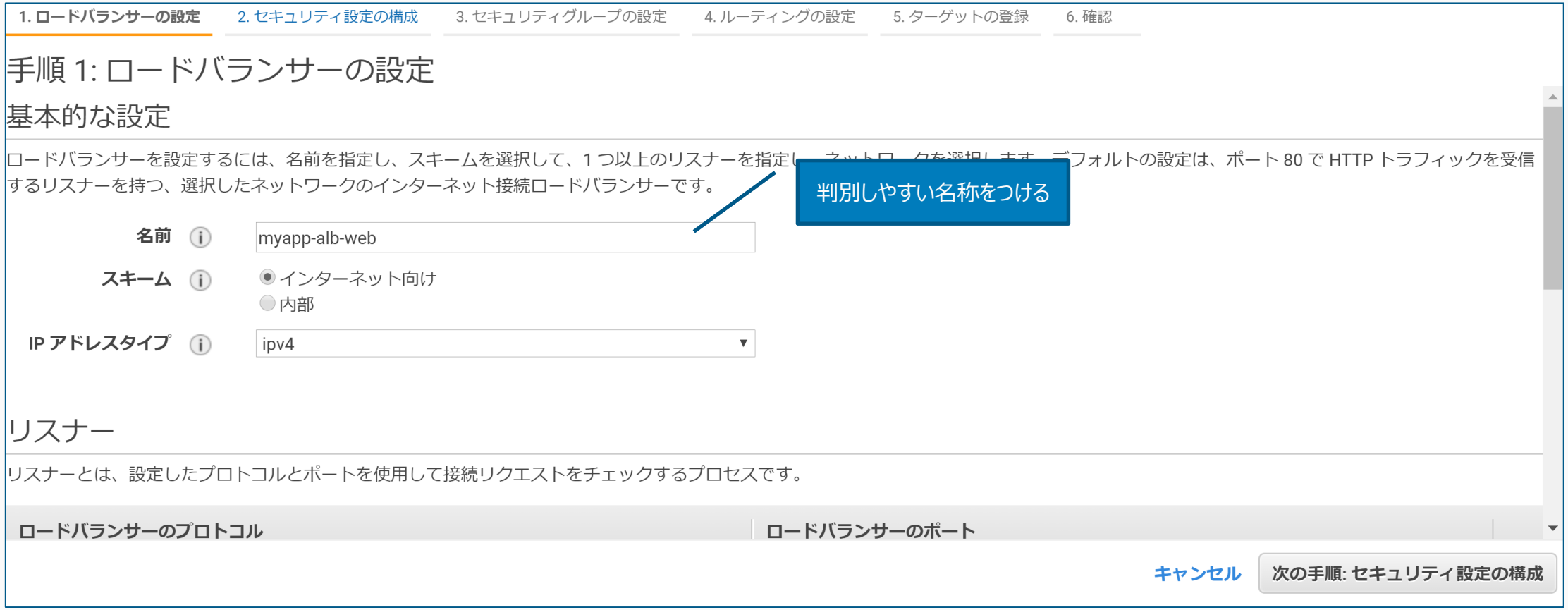
証明書リクエスト

1. パブリック証明書のリクエストを行う
2. ドメインは事前に用意する
3. 検証方法はDNS検証を利用する
    1. 確定するとDNSレコードに設定する値が表示(またはCSVダウンロード)されるので、ドメインを登録しているDNSサーバーに設定する
4. DNSレコード登録後、早ければ10分ぐらいで証明書が発行される
5. なお、DNSサーバーはRoute53でなくてもよい(弊社はPointDNSを使用)

ELB(ALB)編

ロードバランサー作成(基本的な設定)

1. Application Load Balancer(ALB)を作成する

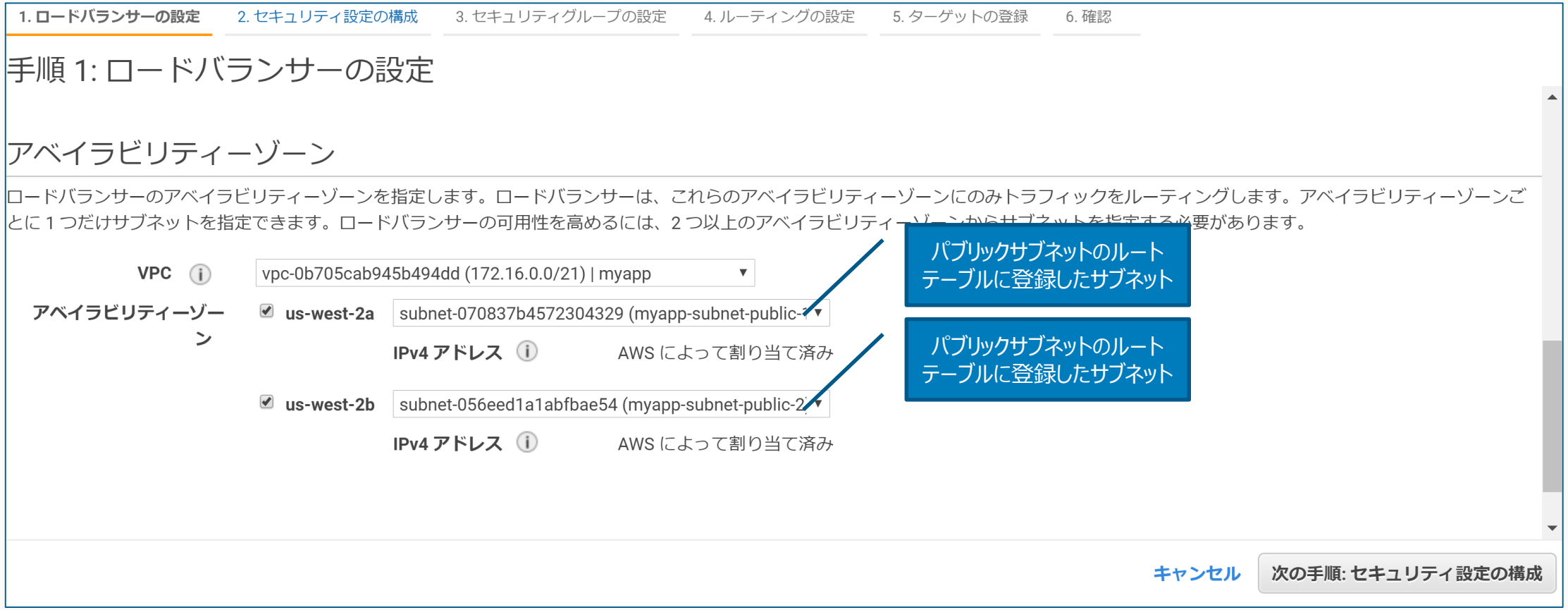


ロードバランサー作成(リスナー)

1. 通常のHTTP/HTTPSの他に、テスト用のポートを追加する(CodeDeployでデプロイする際のテストで利用する)

ロードバランサー作成(アベイラビリティーゾーン)

1. パブリックサブネット用のルートテーブルに登録したサブネットを指定する



ロードバランサー作成(デフォルトの証明書の選択)

1. 事前にACMで作成した証明書を選択する

ロードバランサー作成(セキュリティグループの設定)

1. ELB用のセキュリティグループを選択する

ロードバランサー作成(ルーティングの設定、ターゲットの登録)

1. CodeDeployの設定を行うときにターゲットグループは改めて作成するため、ここでは適当な設定でターゲットグループを作成する
    1. ターゲットグループの名前だけ適当に入力して、あとはデフォルトのままでも問題ない(インスタンス追加などしなくてもよい)
2. このターゲットグループは不要で削除するため、説明は割愛する

リスナー編集

1. 作成したロードバランサーのリスナーを、下表の設定になるよう編集する
2. HTTPは全部対応するHTTPSへ転送し、HTTPSは固定レスポンスを返すようにしておく
    1. HTTPSは、CodeDeployのターゲット作成を行うときにルールが追加される予定
    2. 固定レスポンスの「レスポンス本文」は空白でも問題ない(アプリケーションの要件で決める)
3. **終わったら、ロードバランサー作成時にできたターゲットグループは削除しておく**
4. **SSL証明書を発行した際に指定したドメインのDNSレコードに、ロードバランサーのDNS名(Aレコード)を登録しておく**
    1. **myapp.duxi.co.jp で発行した場合、myapp.duxi.co.jp のCNAMEレコードを作り、そこへDNS名を設定する**

リスナーID	ルール
HTTP : 80	デフォルト: 次の宛先にリダイレクト中: HTTPS://#{host}:443/#{path}?#{query}
HTTPS : 443	デフォルト: 固定レスポンスを返しています 503
HTTPS : 8443	デフォルト: 固定レスポンスを返しています 503
HTTP : 8888	デフォルト: 次の宛先にリダイレクト中: HTTPS://#{host}:8443/#{path}?#{query}

コンテナ編

リポジトリ作成

1. ECRでアプリケーションのコンテナイメージを格納するリポジトリを作成する
2. 今回はCakePHP 3.8.10で構築することを前提としているため、下記の2リポジトリを作成する
    1. nginx
    2. php-fpm

リポジトリにプッシュ

1. 作成したリポジトリにイメージをプッシュする
2. 事前にECRへプッシュするための、IAMユーザーを用意しておく
    1. ポリシー「AmazonEC2ContainerRegistryFullAccess」があればよい
    2. ユーザーを作成する際、アクセスキーIDとシークレットアクセスキーを取得しておく
3. 作成したリポジトリを選択するとプッシュコマンドを見ることができる(詳細は次ページ)

リポジトリにプッシュ(プッシュコマンド)

1. 「プッシュコマンドの表示」をクリックして表示された内容をもとにすすめる
    1. WSLの利用を前提とするため、コマンドは「macOS / Linux」を使用する
    2. AWS CLIが必要になるので、別途インストールを済ませておく(執筆時点のバージョンは「1.18.26」)
    3. Dockerビルドを行うので、Docker Desktopなどをインストールしてdockerコマンドを使用できるようにしておく
2. CakePHP 3.8.10で作成したプロジェクトのカレントディレクトリに移動する
    1. プロジェクトの構成についてはAPPENDIXを参照
3. aws ecr get-login-password --region [リポジトリを作成したリージョン] --profile [事前作成したIAMユーザーのプロファイル] | docker login --username AWS --password-stdin [アカウントID].dkr.ecr.[リポジトリを作成したリージョン].amazonaws.com/nginx
4. nginxのビルドとプッシュ
    1. docker build -t nginx -f docker/nginx/Dockerfile .
    2. docker tag nginx:latest [アカウントID].dkr.ecr.[リポジトリを作成したリージョン].amazonaws.com/nginx:latest
    3. docker push [アカウントID].dkr.ecr.[リポジトリを作成したリージョン].amazonaws.com/nginx:latest
5. php-fpmのビルドとプッシュ
    1. docker build -t php-fpm -f docker/php-fpm/Dockerfile .
    2. docker tag nginx:latest [アカウントID].dkr.ecr.[リポジトリを作成したリージョン].amazonaws.com/nginx:latest
    3. docker push [アカウントID].dkr.ecr.[リポジトリを作成したリージョン].amazonaws.com/nginx:latest

タスク定義

1. 起動タイプの互換性は「FARGATE」を選択する

タスク定義

1. タスク実行ロールは、はじめてタスク定義を行うときは「新しいロールの作成」が選択されているが、これはそのままにしておく(タスク作成と同時に実行ロールも作成される)
2. 2回目以降にタスク定義を行うときは、自動作成した「ecsTaskExecutionRole」が選択されている

タスク定義

1. タスクサイズのタスクメモリとタスク CPU(vCPU)は、アプリケーションの要件にあわせて設定する(今回は最小の構成を選択した)

タスク定義

1. コンテナは、nginxとphp-fpmの2つを追加する
2. nginx：コンテナ名「nginx」、イメージ「[アカウントID].dkr.ecr.[リポジトリを作成したリージョン].amazonaws.com/nginx:latest」、ポートマッピング：80
3. php-fpm：コンテナ名「php-fpm」、イメージ「[アカウントID].dkr.ecr.[リポジトリを作成したリージョン].amazonaws.com/php-fpm:latest」、ポートマッピング：9000

タスク定義

1. その他の触れていない項目についてはデフォルトの状態でよい

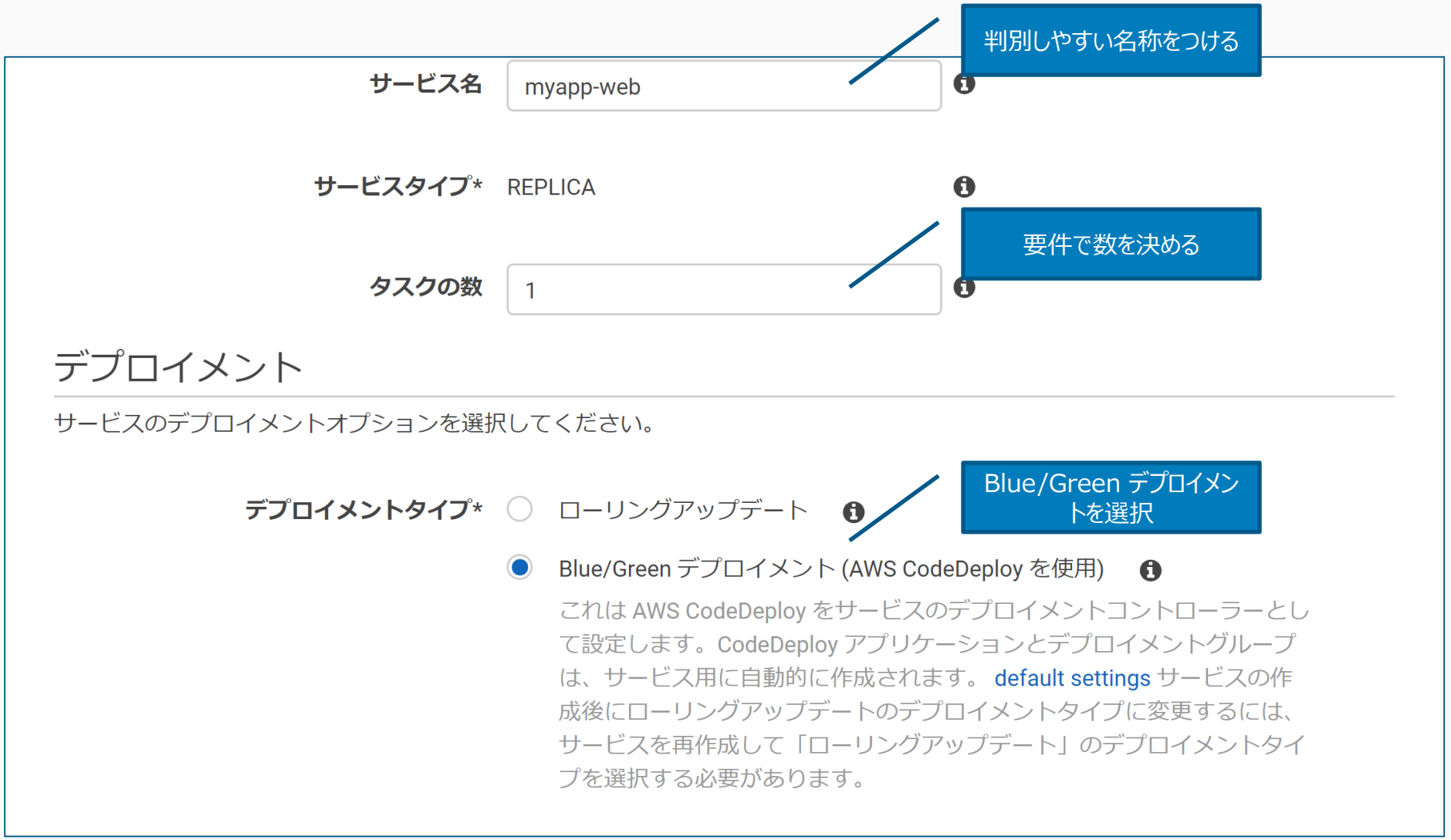
クラスター作成

1. テンプレートは「AWS Fargate を使用」を選択する

サービス作成(設定)

1. CodeDeploy用のロールが必要になるため、事前にユースケース「CodeDeploy - ECS」から作成したロールを準備しておく

サービス作成(設定)


判別しやすい名称をつける
サービス名
myapp-web
サービスタイプ*
REPLICA
要件で数を決める
タスクの数
1
デプロイメント
サービスのデプロイメントオプションを選択してください。
Blue/Green デプロイメントを選択
デプロイメントタイプ*
ローリングアップデート
Blue/Green デプロイメント (AWS CodeDeploy を使用)
これは AWS CodeDeploy をサービスのデプロイメントコントローラーとして設定します。CodeDeploy アプリケーションとデプロイメントグループは、サービス用に自動的に作成されます。 default settings サービスの作成後にローリングアップデートのデプロイメントタイプに変更するには、サービスを再作成して「ローリングアップデート」のデプロイメントタイプを選択する必要があります。

サービス作成(設定)

1. ロールを事前に作成していない状態について確認することができなかったので、事前にロールを作成していなかった場合は、空白かまたはAWSが自動作成したロールが選択されるかもしれない

サービス作成(ネットワーク構成)

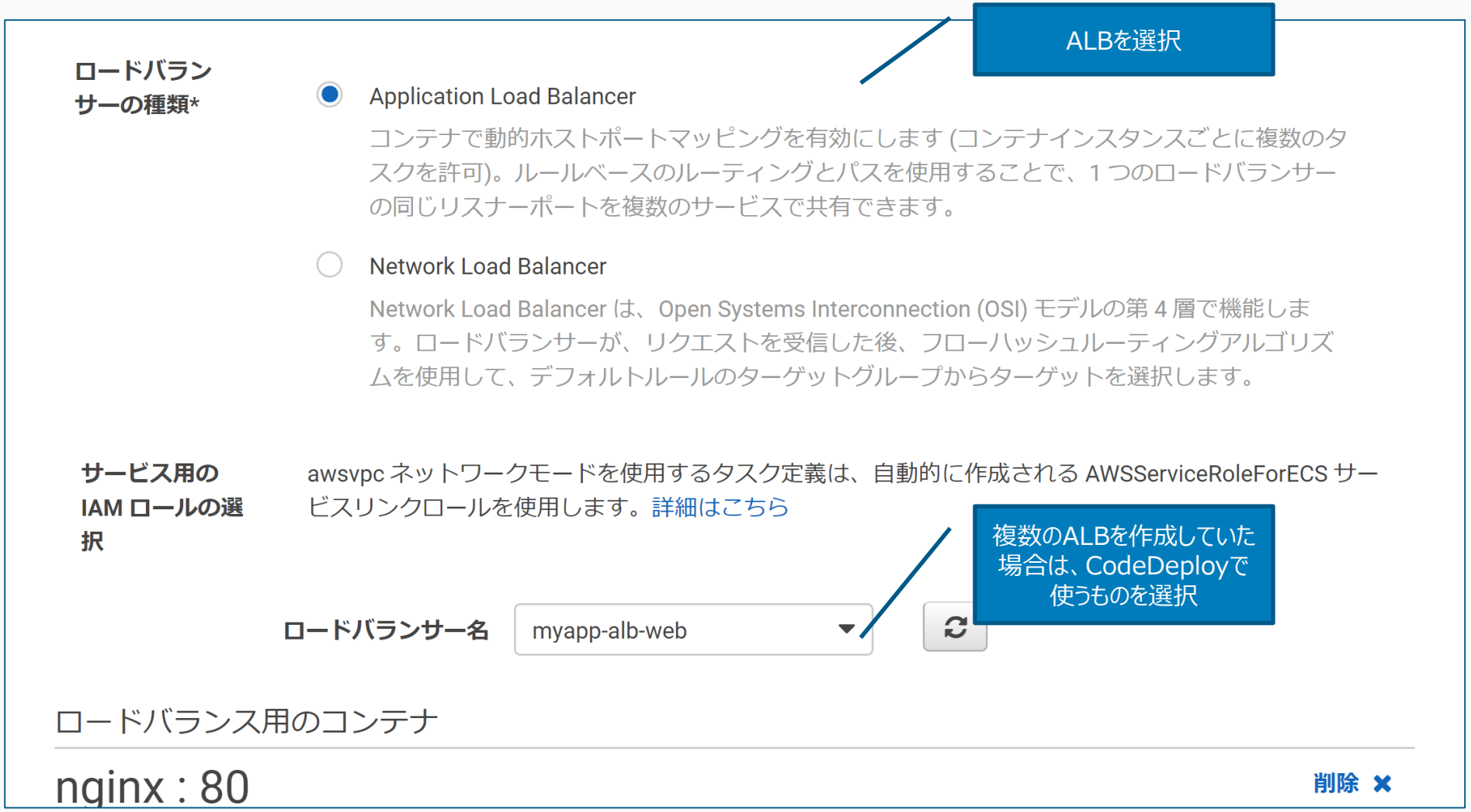
1. サブネットはECS用に作成したものを選択する

サービス作成(ネットワーク構成)

1. WEB用に作成したセキュリティグループを選択する(本資料では「myapp-web-sg」が該当する)

サービス作成(ネットワーク構成)

1. 作成したロードバランサーを選択する



サービス作成(ネットワーク構成)

1. ロードバランス用のコンテナとして「nginx」を選択し、ロードバランサーに追加する
2. 追加後は、下記の図のとおりとなる
3. プロダクションリスナーポートは、アプリケーションが通常のリクエストを受け付けるポートになるため、443ポートを指定する

サービス作成(ネットワーク構成)

1. ターゲットグループを2つ設定する
2. プロトコルはHTTPのままでよい(コンテナが80で待ち受けているため)
3. パスパターンは、ここではひとつしか設定できないが、あとからロードバランサーのリスナー設定で追加することができる

サービス作成(ネットワーク構成)

1. ヘルスチェックパスは、通常時にHTTPステータスコード200を返すパスであればよい
2. 例えば特定のHTMLファイルをリクエストして200を返すことがテストになるようであれば、そのHTMLファイルのパスを指定する(/healthy.htmlなど)

位の値を後で追加できます。

評価順: ルールは優先順位の低いものから高いものへと評価されます。パスパターンルールに一致したら、他のすべてのルールは無視されます。サービスごとにパスを作成することで、このリスナーからのトラフィックを複数のサービスにルーティングできます。

このリスナーで使用されている既存のパス

パスには、アプリケーションで使用する可能性があるすべてのパスを含める必要があります。たとえば、パス /webapp1* が含まれたサービスは、このリスナーで /webapp1 and /webapp1/page.html に送信されたトラフィックを受信します。一意のパスを選択することをお勧めします。また、評価の順序を低くすると、複数の競合するパス間でトラフィックをルーティングできます。

評価順	ルールパス	ターゲットグループ
default	/	

専用のパスがあればそれを指定する

ヘルスチェックパス* /

追加のヘルスチェックオプションは、サービスの作成後、ELB コンソールで設定できます。

サービス作成(ネットワーク構成)

サービスの検出 (オプション)

サービスの検出では、Amazon Route 53 を使用してサービスの名前空間を作成します。これにより、サービスは DNS を介して検出可能になります。

サービスの検出の統合の有効化 ☑

名前空間* local | プライベート

サービスの検出サービスの設定

- ◉ 新しいサービスの検出サービスの作成
- ○ 既存のサービスの検出サービスの選択

サービスの検出名* myapp-web

英数字とアンダースコアの文字列 (中にピリオドを含む) は有効です。詳細については、Route 53 Auto Naming のドキュメントを参照してください。

サービス作成(ネットワーク構成)

サービスの検出名* myapp-web

英数字とアンダースコアの文字列 (中にピリオドを含む) は有効です。詳細については、Route 53 Auto Naming のドキュメントを参照してください。

ECS タスク状態の伝達の有効化

☑

このフィールドを有効にすると、ECS はタスク状態を Route 53 に伝え、異常なタスクを DNS から削除するためにかかる時間を減らします。

Docker ヘルスチェック

Docker ヘルスチェックは、タスク定義で定義されます。これにより、必要不可欠な集約コンテナの複数のヘルスチェックで、サービスの状態を確認できます。

パブリック DNS ヘルスチェックの有効化

プライベート名前空間の DNS ヘルスチェックは、現在サポートされていません。

サービス作成(ネットワーク構成)

パブリック DNS ヘルスチェックの有効化 プライベート名前空間の DNS ヘルスチェックは、現在サポートされていません。

サービスの検出の DNS レコード

DNS レコード型* A

TTL* 60 秒

DNS レコードの追加

*必須

キャンセル 戻る 次のステップ

サービス作成(Auto Scaling)

1. 必要であれば設定するが、今回は使用しない

サービス作成(確認)

1. 最終的な設定内容は下図の通り

サービス作成(確認)

VPC ID	vpc-0b705cab945b494dd
サブネット	subnet-0d06c760edb534b96, subnet-0da04f1ba029b9ad0
選択したセキュリティグループ	sg-0fb0e2eaba673d5cd
IP の自動割り当て	ENABLED

ロードバランシング

ロードバランサーの種類	Application Load Balancer (ALB)
ロードバランサー名	myapp-alb-web
ロードバランス用のコンテナ	nginx
コンテナポート	80

サービス作成(確認)

ターゲットグループ 1 の名前	myapp-web-1
プロダクションリスナーのパスパターン	/*
プロダクションリスナーのヘルスチェックパス	/
テストリスナーポート	8443
ターゲットグループ 2 の名前	myapp-web-2
テストリスナーのパスパターン	/*
テストリスナーのヘルスチェックパス	/

サービスの検出の設定

編集

サービス作成(確認)

サービスの検出の設定 編集

名前空間	ns-ctzxpymvn3bt3eav
サービスの検出名	myapp-web
ECS タスク状態の伝達の有効化	true
DNS レコード型および TTL	A 60

Auto Scaling (オプション) 編集

not configured

キャンセル 戻る サービスの作成

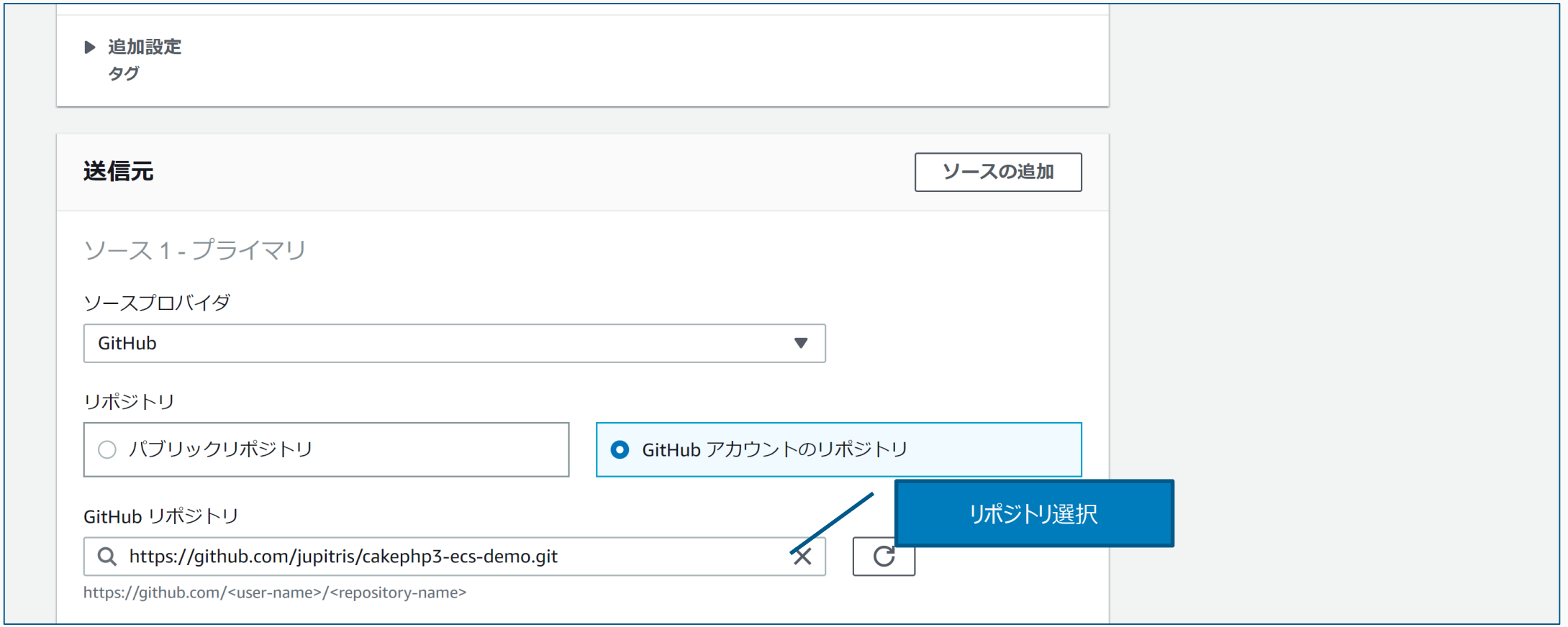
デプロイ編

ビルドプロジェクト作成

1. .envを利用している場合は、s3のバケットに事前に.envをアップロードしておき、buildspec.ymlのpre_buildでコピーすると良い
    1. aws s3 cp s3://bucket_name/.env path/to/.env
    2. ビルドプロジェクトに設定するロールに、「bucket_name」へのアクセス権限を付与しておく

ビルドプロジェクト作成

1. ソースコードはGitHubのリポジトリを使用する
    1. 自分のアカウントから参照できるリポジトリを利用する場合は、事前にGitHubのパーソナルアクセストークンを取得しておく

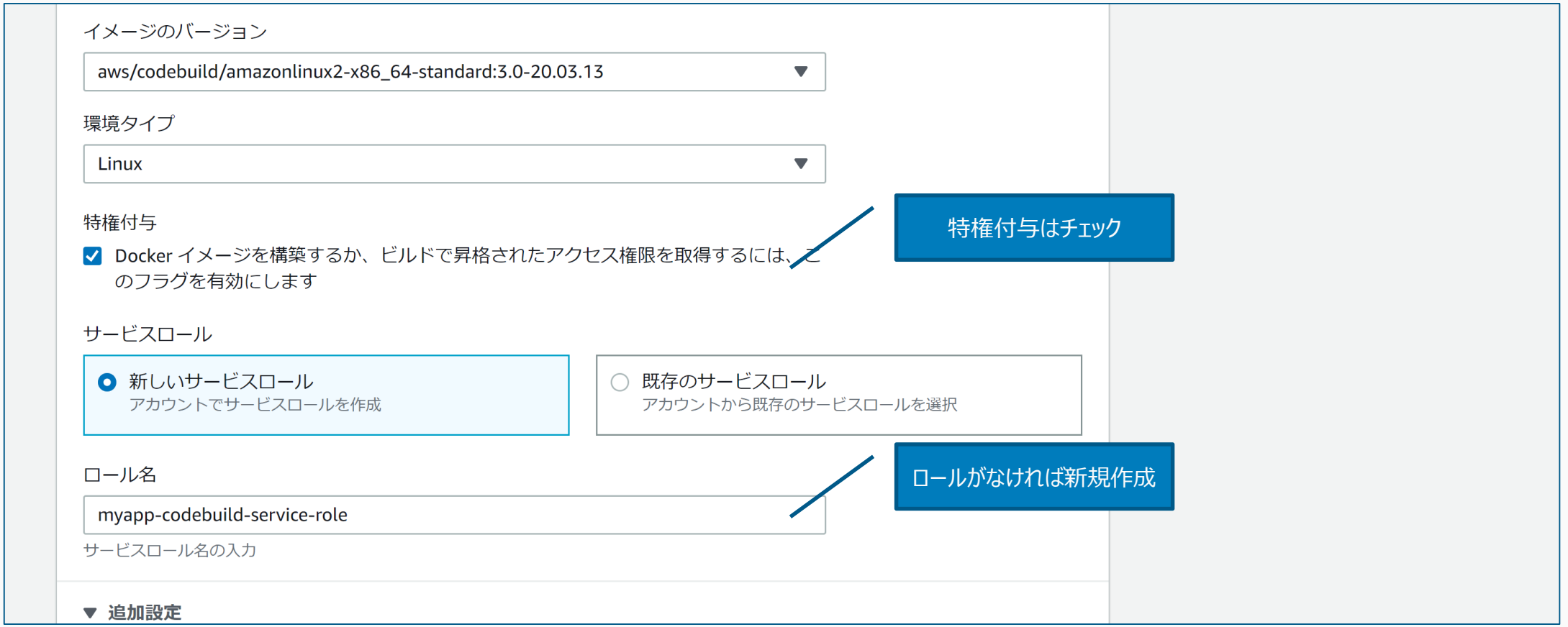


ビルドプロジェクト作成

1. オペレーティングシステムは「Amazon Linux 2」を選択し、あとは図の通りに選択する

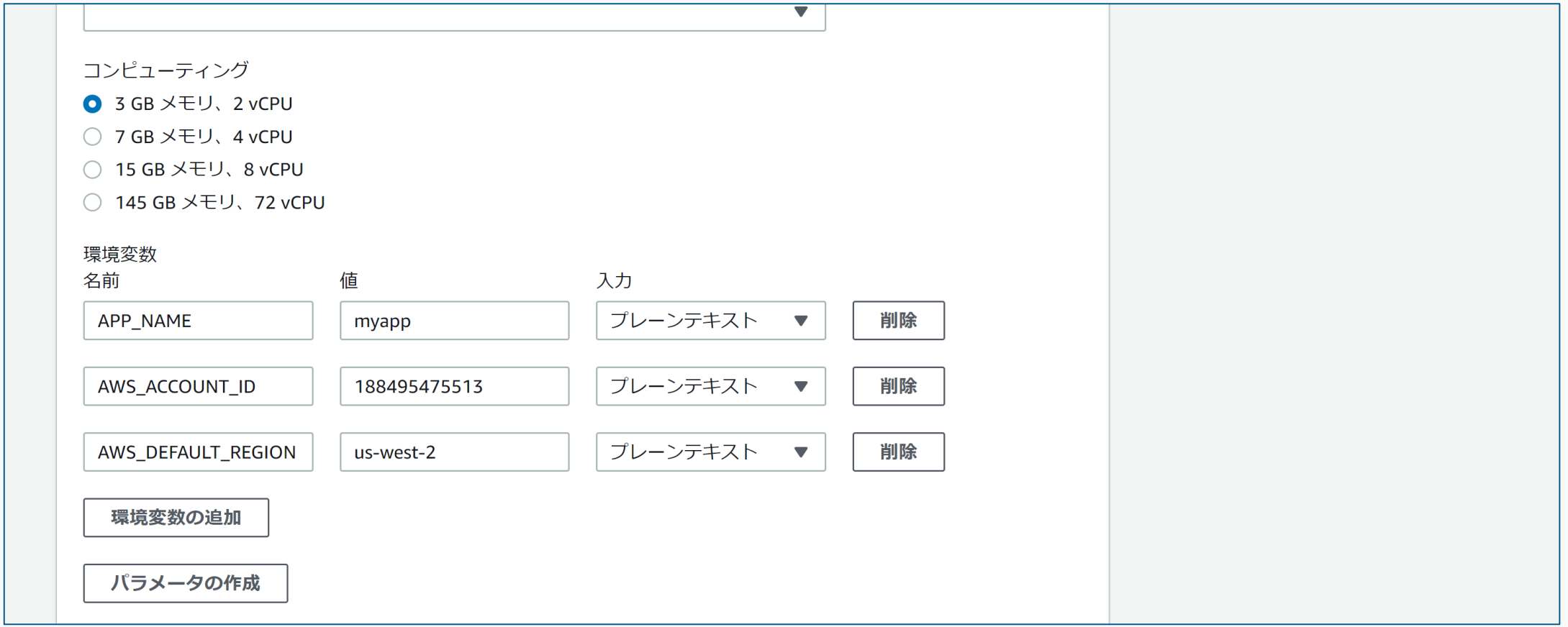
ビルドプロジェクト作成

1. サービスロールについては、CodeBuild用のロールを作っていない場合は新規に作成する
2. 利用するロールに「AmazonEC2ContainerRegistryFullAccess」を付与しておく(ビルド実行時に権限エラーが発生するため)



ビルドプロジェクト作成

1. buildspec.ymlで使用する環境変数を定義する
2. リポジトリに登録できないような定義は、環境変数を使うと良い



ビルドプロジェクト作成

1. リポジトリに登録したbuildspec.ymlを指定する(別名の場合は、そのファイル名を指定する)

ビルドプロジェクト作成

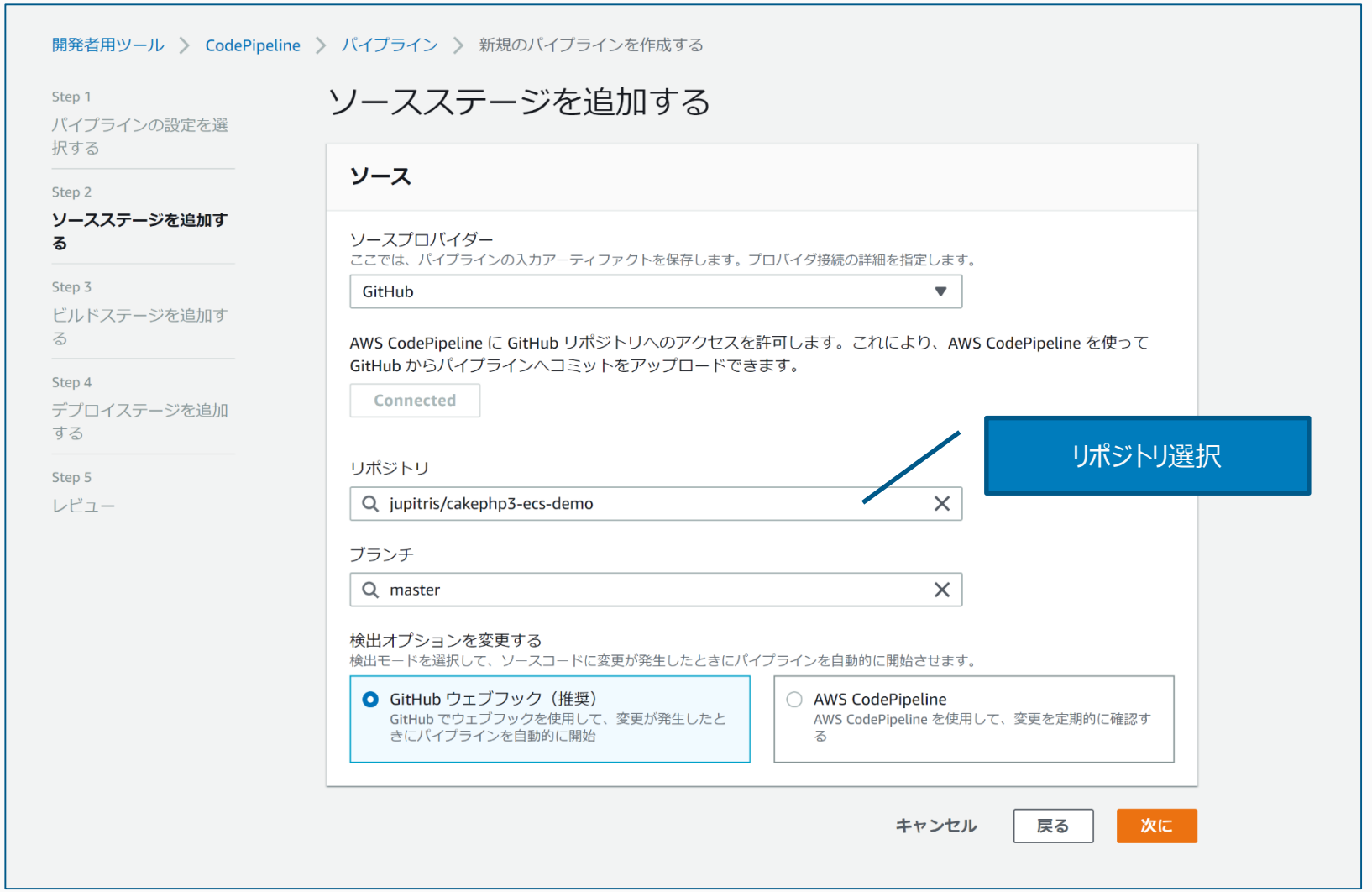
1. ビルドプロジェクト作成後、一度ビルドを実行してすべて正常に完了するか確認しておくと良い

パイプライン作成


開発者用ツール > CodePipeline > パイプライン > 新規のパイプラインを作成する
Step 1
パイプラインの設定を選択する
Step 2
ソースステージを追加する
Step 3
ビルドステージを追加する
Step 4
デプロイステージを追加する
Step 5
レビュー
パイプラインの設定を選択する
パイプラインの設定
パイプライン名
パイプライン名を入力します。作成後にパイプライン名を編集することはできません。
myapp-pipeline
100 文字を超えることはできません
判別しやすい名称をつける
サービスロール
新しいサービスロール
アカウントでサービスロールを作成
既存のサービスロール
アカウントから既存のサービスロールを選択
ロール名
myapp-pipeline-service-role
サービスロール名の入力
判別しやすい名称をつける
AWS CodePipeline がサービスロールを作成できるようになるため、この新しいパイプラインでの使用が可能になります。
高度な設定
キャンセル
次に

パイプライン作成

1. CodeBuildのビルドプロジェクト作成のときに指定したリポジトリを選択する



パイプライン作成

1. CodeBuildで作成したビルドプロジェクトを選択する

パイプライン作成

1. デプロイプロバイダーに「Amazon ECS(ブルー/グリーン)」を選択する
2. CodeDeployのアプリケーション名とデプロイグループは、ECSサービス作成時に自動的に作成されたものがあるので、それを選択する

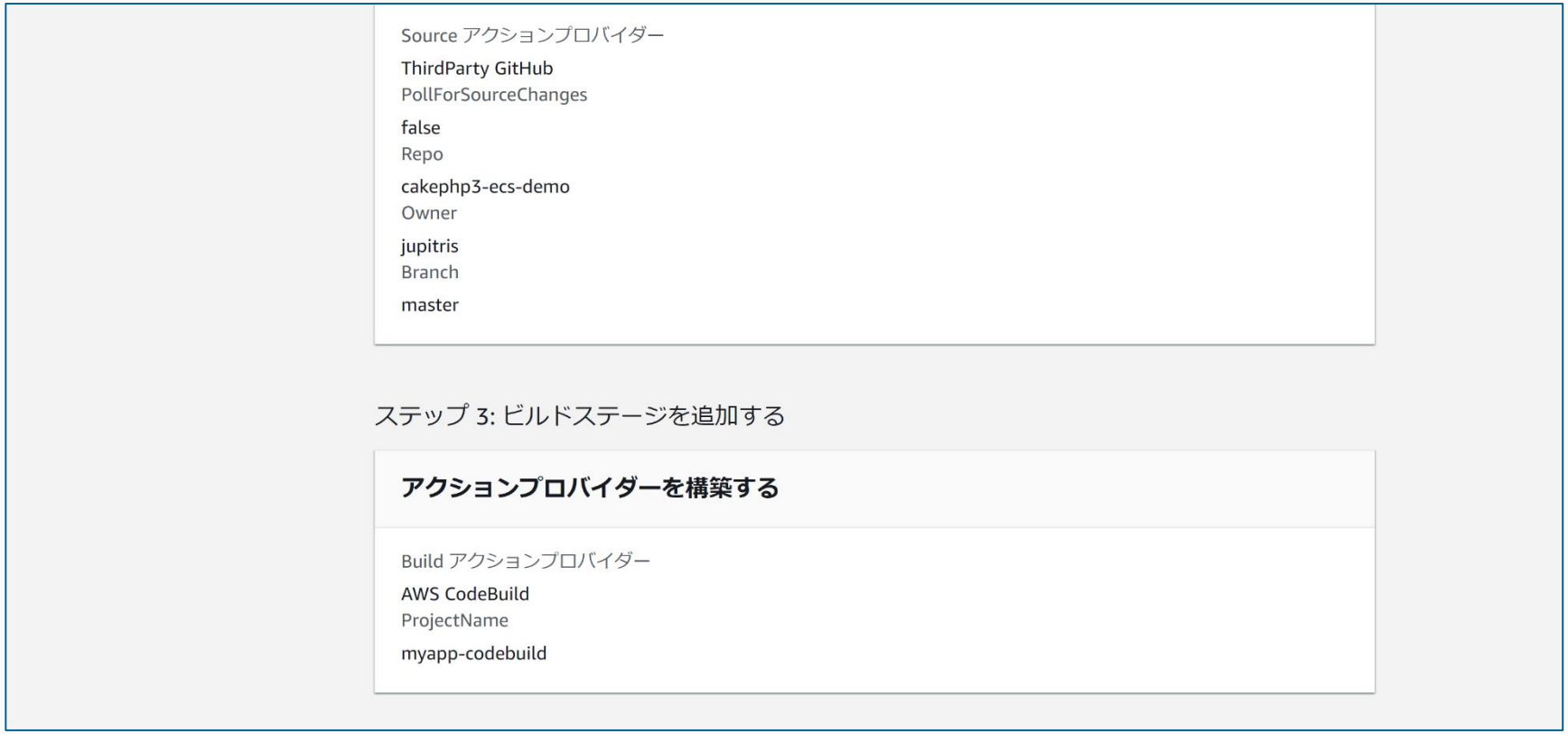
パイプライン作成

1. ECS タスク定義およびCodeDeploy AppSpecは、ともにリポジトリに入っているファイル名を入力する

パイプライン作成(レビュー)

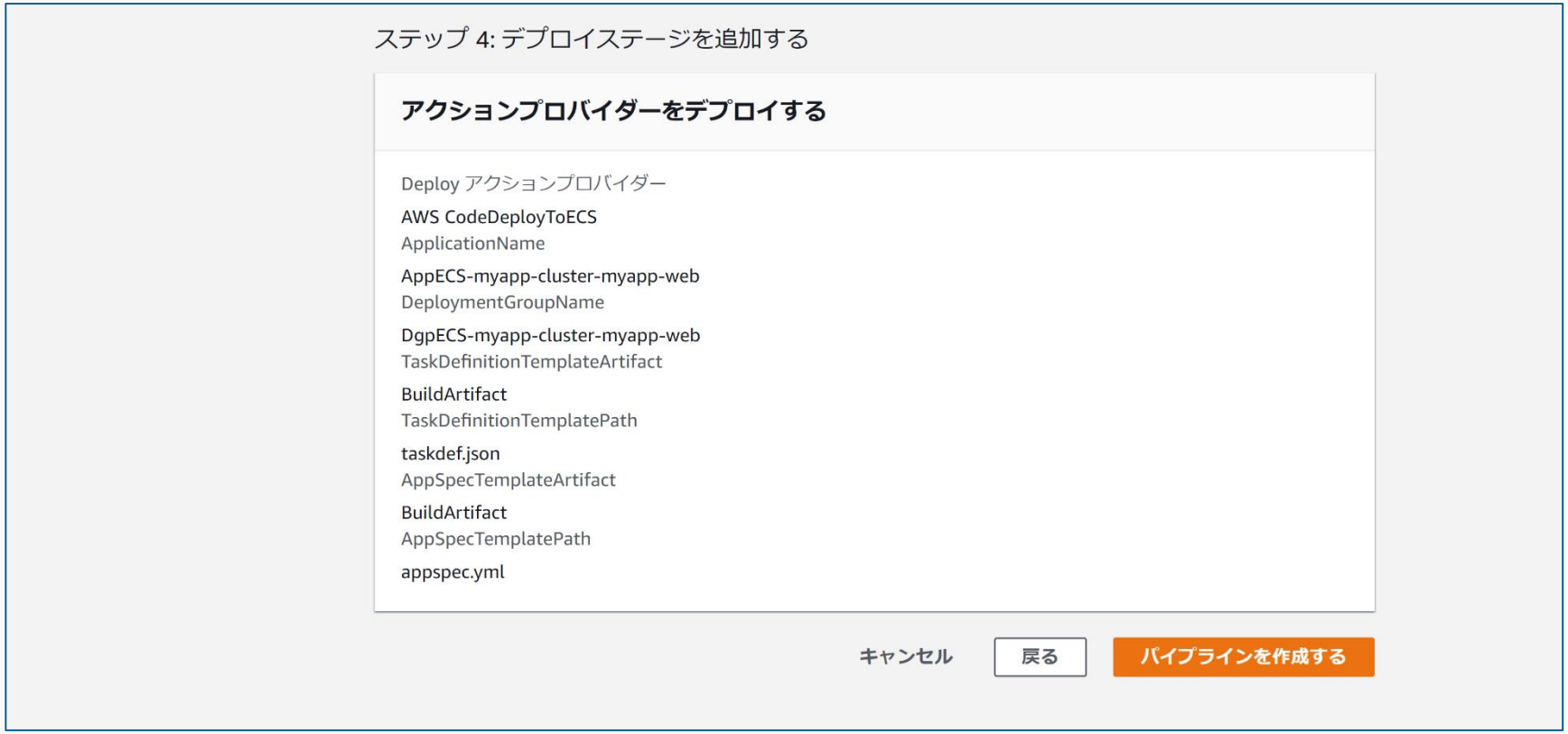
1. 最終的な設定内容は下図の通り

パイプライン作成(レビュー)



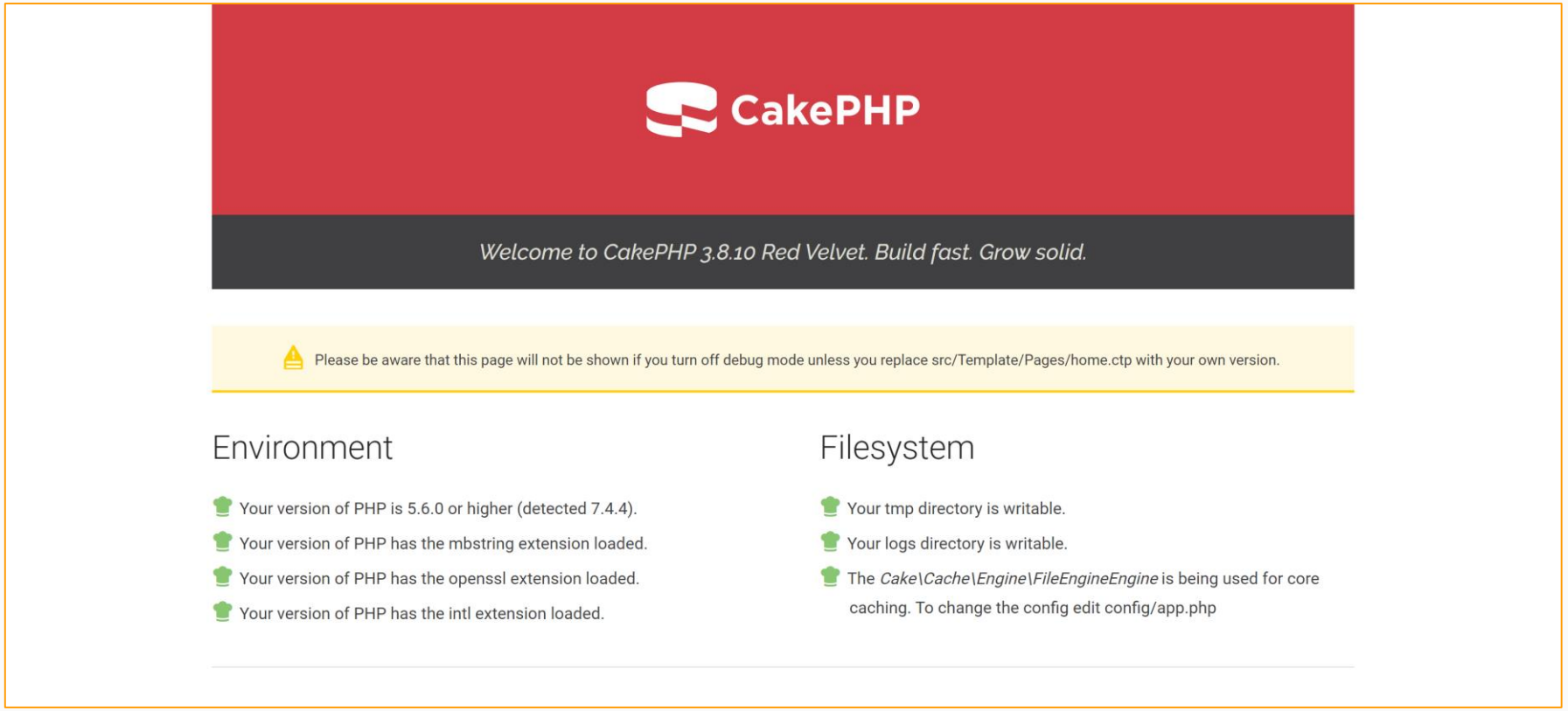
パイプライン作成(レビュー)

1. 作成したロールには「AmazonEC2ContainerRegistryFullAccess」を付与する
2. 作成後、パイプラインを実行してみて、正常に完了するか確認しておくと良い
3. ソースコードをリポジトリにpushすると、自動的にパイプラインを実行してデプロイする



動作確認

1. 最後に、アプリケーションにアクセスして動作を確認する



APPENDIX編

プロジェクト構成

1. プロジェクトの構成は、CakePHP 3.8.10とほぼ同様
    1. 公式ドキュメント内「クイックスタートガイド」で説明しているcomposerコマンドを利用し、「composer self-update && composer create-project --prefer-dist cakephp/app:^3.8 cakephp3-ecs-demo」でプロジェクトを作成している
    2. 詳細は公式ドキュメント内「CakePHP のフォルダー構成」を参照
2. ローカルのDockerやECS利用のために、いくつかのファイルやフォルダを追加している

CakePHP 3.8.10に標準で含まれるファイルやフォルダ

```
/cakephp3-ecs-demo
 /bin
 /config
 /logs
 /plugins
 /src
 /tests
 /tmp
 /vendor
 /webroot
 .editorconfig
 .gitignore
 .htaccess
 .travis.yml
 composer.json
 index.php
 phpunit.xml.dist
 README.md
```

追加したファイルやフォルダ

```
/cakephp3-ecs-demo
 /config
  app_local.php
 docker-compose.yml
 appspec.yml
 buildspec.yml
 taskdef.json
 /docker
  /php-fpm
   Dockerfile
   php.ini
  /nginx
   Dockerfile
   default.conf.production
   default.conf.local
```

appspec.ymlのサンプル

```
version: 0.0
Resources:
  - TargetService:
      Type: AWS::ECS::Service
      Properties:
        TaskDefinition: <TASK_DEFINITION>
        LoadBalancerInfo:  # ContainerNameはECRに登録したリポジトリ名を指定する
          ContainerName: "nginx"
          ContainerPort: 80
```

buildspec.ymlのサンプル

```
version: 0.2

phases:
  install:
    runtime-versions:
      docker: 18
  pre_build:
    commands:  # ${APP_NAME}や${AWS_DEFAULT_REGION}、${AWS_ACCOUNT_ID}はCodeBuild環境変数
      - aws --version
      - $(aws ecr get-login --no-include-email --region ${AWS_DEFAULT_REGION})
      - APP_ENV=production
      - BUCKET=${APP_NAME}.duxi.co.jp/${APP_ENV}
      - aws s3 cp s3://${BUCKET}/app.php config/app.php
      - IMAGE_NAME_PHP_FPM=php-fpm
      - IMAGE_NAME_NGINX=nginx
      - REPOSITORY_URI_PHP_FPM=${AWS_ACCOUNT_ID}.dkr.ecr.${AWS_DEFAULT_REGION}.amazonaws.com/${IMAGE_NAME_PHP_FPM}
      - REPOSITORY_URI_NGINX=${AWS_ACCOUNT_ID}.dkr.ecr.${AWS_DEFAULT_REGION}.amazonaws.com/${IMAGE_NAME_NGINX}
      - COMMIT_HASH=$(echo ${CODEBUILD_RESOLVED_SOURCE_VERSION} | cut -c 1-7)
      - IMAGE_TAG=${COMMIT_HASH:=latest}
  build:
    commands:  # --build-argで渡した値はDockerfile内で利用する
      - echo Build started on `date`
      - echo Building the Docker image...
      - docker build -t ${REPOSITORY_URI_PHP_FPM}:latest -f docker/php-fpm/Dockerfile .
      - docker build --build-arg APP_ENV=${APP_ENV} -t ${REPOSITORY_URI_NGINX}:latest -f docker/nginx/Dockerfile .
      - docker tag ${REPOSITORY_URI_PHP_FPM}:latest ${REPOSITORY_URI_PHP_FPM}:$IMAGE_TAG
      - docker tag ${REPOSITORY_URI_NGINX}:latest ${REPOSITORY_URI_NGINX}:$IMAGE_TAG
  post_build:
    commands:
      - echo Build completed on `date`
      - echo Pushing the Docker images...
      - docker push ${REPOSITORY_URI_PHP_FPM}:latest
      - docker push ${REPOSITORY_URI_PHP_FPM}:$IMAGE_TAG
      - docker push ${REPOSITORY_URI_NGINX}:latest
      - docker push ${REPOSITORY_URI_NGINX}:$IMAGE_TAG
      - echo Writing image definitions file...
      - cat taskdef.json | sed -e s@¥<IMAGE_NAME_PHP_FPM¥>@$REPOSITORY_URI_PHP_FPM:$IMAGE_TAG@ -e s@¥<IMAGE_NAME_NGINX¥>@$REPOSITORY_URI_NGINX:$IMAGE_TAG@ -e s@¥<AWS_ACCOUNT_ID¥>@${AWS_ACCOUNT_ID}@ -e s@¥<APP_NAME¥>@${APP_NAME}@ -e s@¥<APP_REGION¥>@${AWS_DEFAULT_REGION}@ > taskdef.json
      - cat appspec.yml | sed -e s@¥<APP_NAME¥>@${APP_NAME}@ > appspec.yml

artifacts:
  files:
    - appspec.yml
    - taskdef.json
```

taskdef.jsonのサンプル

```
{
 "executionRoleArn": "arn:aws:iam::<AWS_ACCOUNT_ID>:role/ecsTaskExecutionRole",
 "containerDefinitions": [
  {
   "name": "php-fpm",
   "image": "<IMAGE_NAME_PHP_FPM>",
   "logConfiguration": {
    "logDriver": "awslogs",
    "secretOptions": null,
    "options": {
     "awslogs-group": "/ecs/<APP_NAME>-web",
     "awslogs-region": "<APP_REGION>",
     "awslogs-stream-prefix": "ecs"
    }
   },
   "portMappings": [
    {
     "containerPort": 9000,
     "hostPort": 9000,
     "protocol": "tcp"
    }
   ],
   "essential": true
  },
```

```
  {
   "name": "nginx",
   "image": "<IMAGE_NAME_NGINX>",
   "logConfiguration": {
    "logDriver": "awslogs",
    "secretOptions": null,
    "options": {
     "awslogs-group": "/ecs/<APP_NAME>-web",
     "awslogs-region": "<APP_REGION>",
     "awslogs-stream-prefix": "ecs"
    }
   },
   "portMappings": [
    {
     "containerPort": 80,
     "hostPort": 80,
     "protocol": "tcp"
    }
   ],
   "essential": true
  }
 ],
 "requiresCompatibilities": ["FARGATE"],
 "networkMode": "awsvpc",
 "cpu": "256",
 "memory": "512",
 "family": "<APP_NAME>-web"
}
```

サンプルコード

1. サンプルコードは「https://github.com/jupitris/cakephp3-ecs-demo」からクローンできる
2. サンプルコードのリポジトリにconfig/app.phpは含まれていないので、別途自身で作成してS3にアップロードしておく
    1. 本資料でのアップロード先は「s3://myapp.duxi.co.jp/production」を想定しているため、適当な設定値へ変更する

サンプルコードをローカル環境で利用する

1. DockerとDocker Composeが必要
2. クローンしたリポジトリに移動し、「docker-compose up -d --build」を実行
3. ビルドが完了したら、「http://localhost:40080」にアクセスし、CakePHPのページが表示できれば完成