aws DEV DAY ONLINE JAPAN

2020.10.20-22

A - 6

＼第2回！／

AWS Fargate かんたんデプロイ選手権

Tori Hara / toricls

Sr. Product Developer Advocate
Containers Product, AWS

In Partnership with intel

本セッションは…

想定聴講者

- AWS Fargate (for Amazon ECS) の利用を検討している
- 細かい話は抜きにしてまずはコンテナを AWS 上で動かしたい
- デプロイや CI/CD パイプラインについて考えるのが好き

ゴール

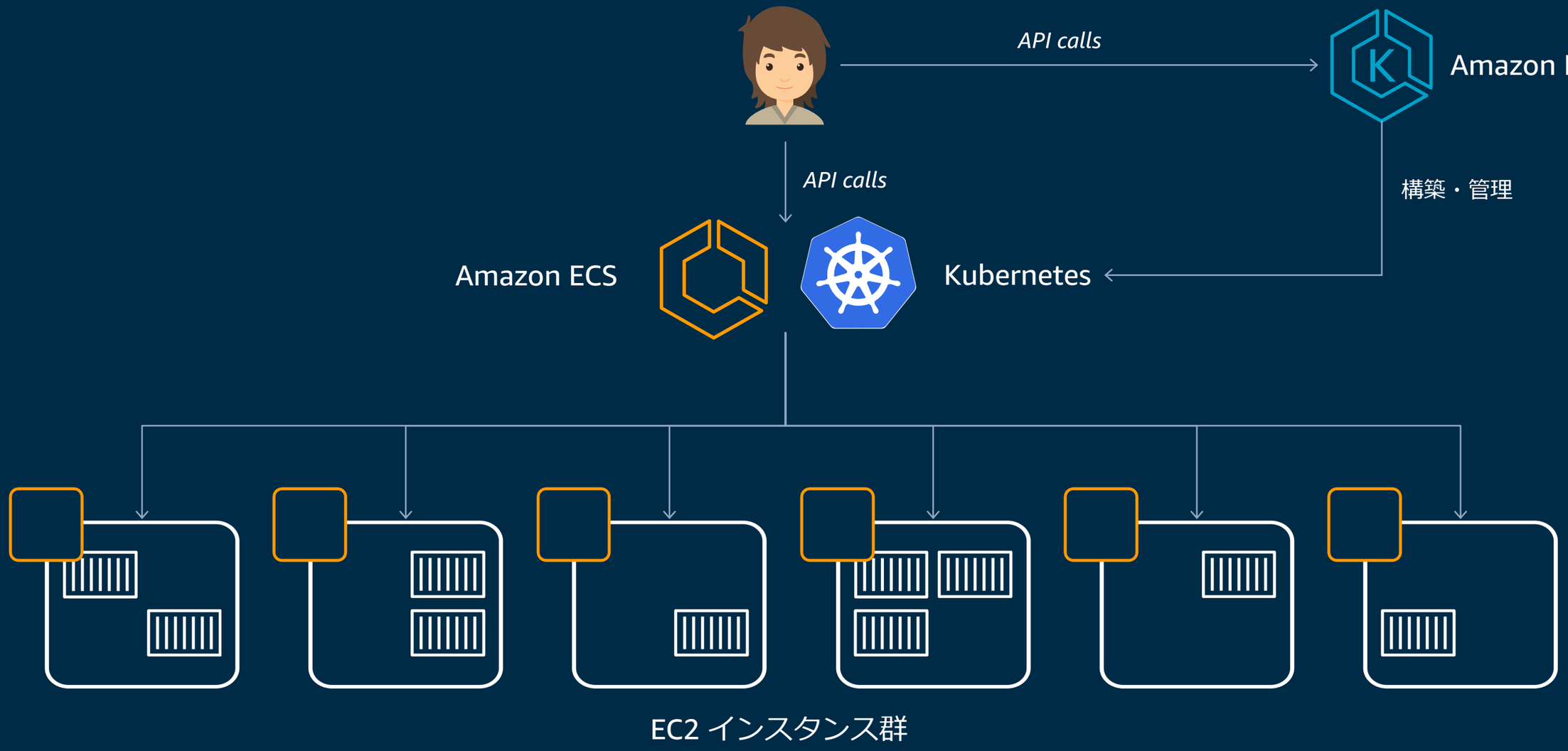
- AWS Fargate の特徴と、AWS Fargate を利用したコンテナ実行方法を知る
- 様々なデプロイツールの特徴を知り、デプロイへの探究心を高める
- 自社・自チームのデプロイ方法についてあらためて考えていただくキッカケを作る

aws DEV DAY
ONLINE JAPAN

AWS Fargate とは

In Partnership with intel

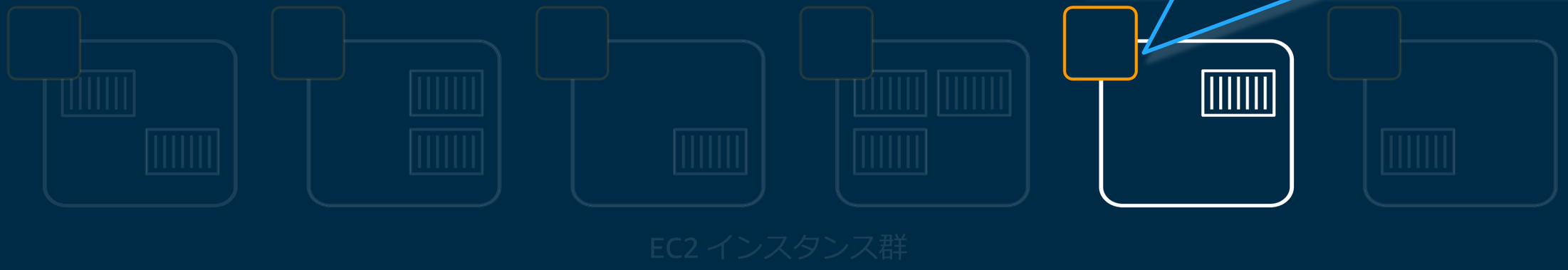
コンテナオーケストレータの概要



In Partnership with intel

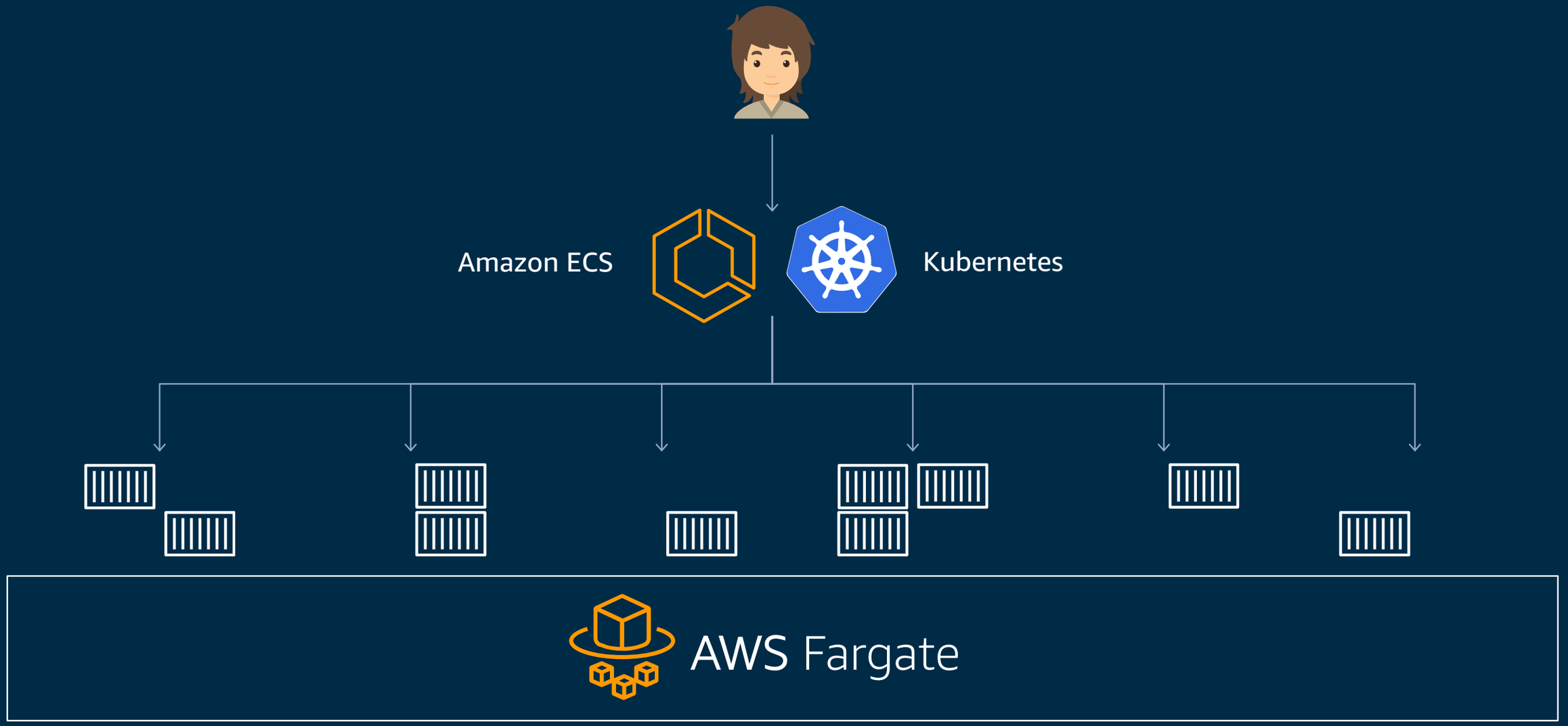
実行環境 EC2 インスタンスの運用業務

- OS やエージェント類へのパッチ当て・更新
- 実行中のコンテナ群に基づく最適なリソース使用率を保つための EC2 インスタンス数のプランニングやスケーリング



In Partnership with intel

Meet AWS Fargate



In Partnership with intel

AWS Fargate

AWS マネージド
EC2 インスタンスのプロビジョン、スケール、管理不要

コンテナネイティブ
仮想マシンを意識しないシームレスなスケーリング
コンテナの起動時間・使用リソースに応じた料金設定

セキュリティ
タスク (Pod) はそれぞれが独立した分離単位を持ち、実行環境の
カーネル、CPU、メモリ、ENI を他のタスク (Pod) と共有しない

AWS サービスとの連携
VPC ネットワーキング、Elastic Load Balancing、IAM、CloudWatch、
etc.

In Partnership with intel

aws DEV DAY
ONLINE JAPAN

Quick start with “AWS Copilot CLI”

In Partnership with intel

Quick start with “AWS Copilot CLI”

aws/copilot-cli

やりたいこと

アプリと Dockerfile は手元にある、あとはこれを Fargate で動かしたい

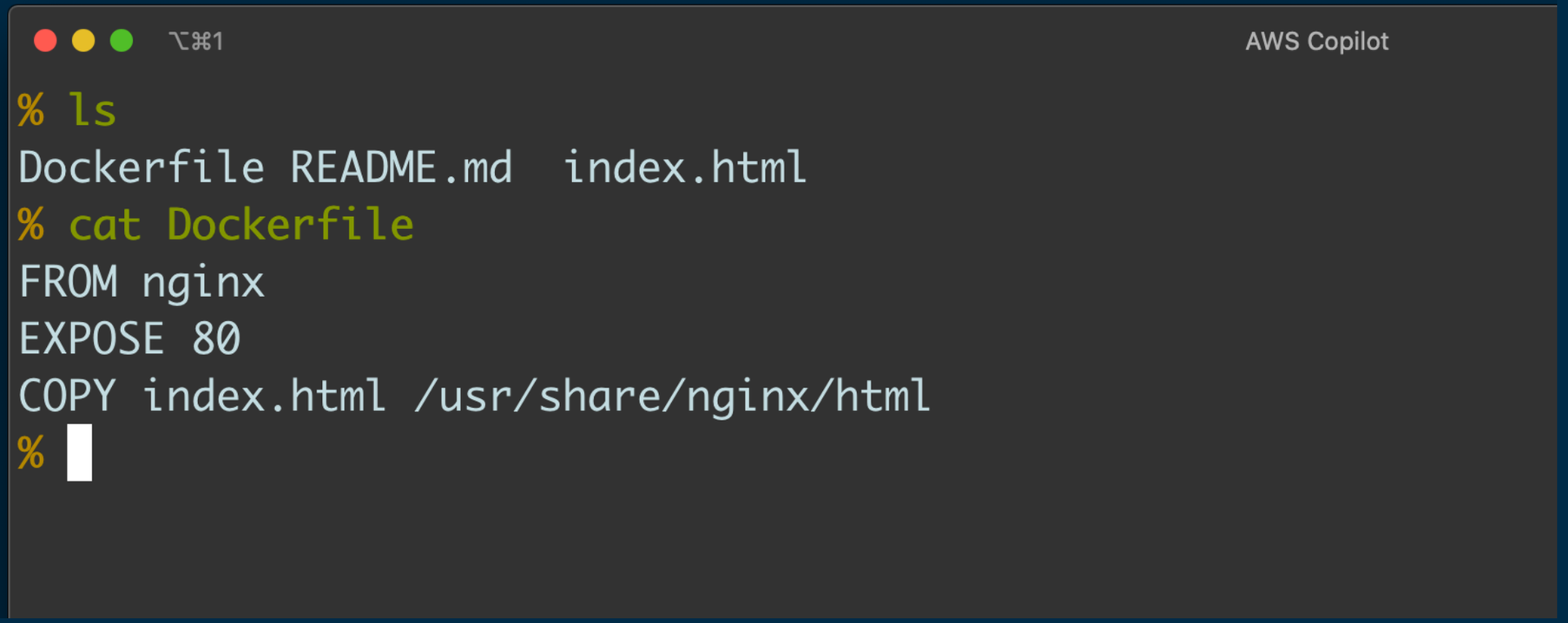
How To

```
$ copilot init
```


In Partnership with intel

Quick start with “AWS Copilot CLI”

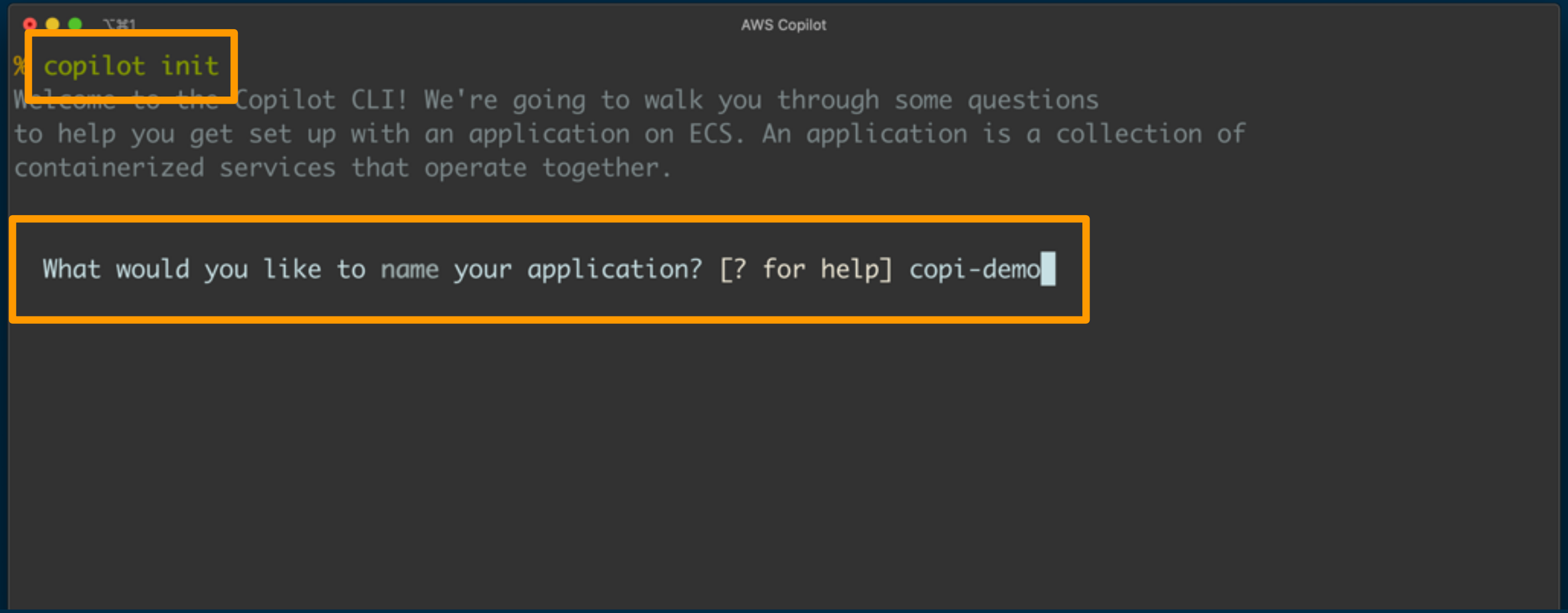
aws/copilot-cli



In Partnership with intel

Quick start with “AWS Copilot CLI”

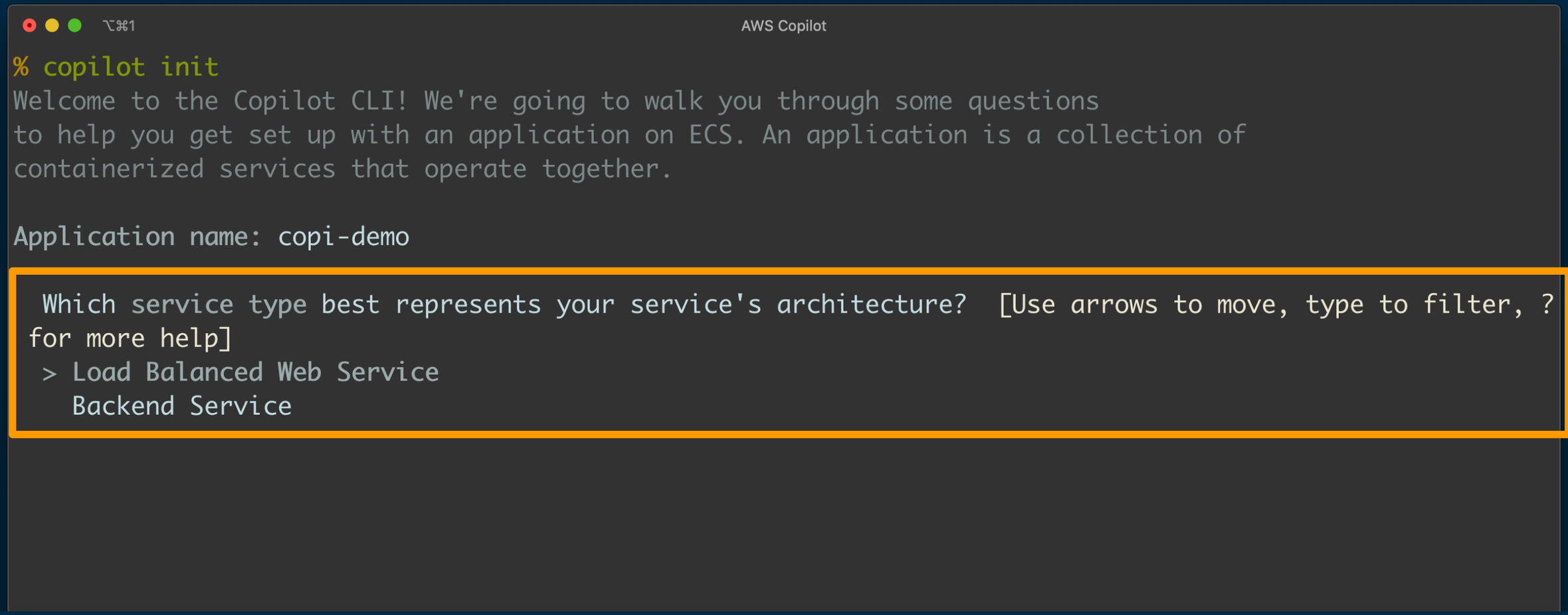
aws/copilot-cli



In Partnership with intel

Quick start with “AWS Copilot CLI”

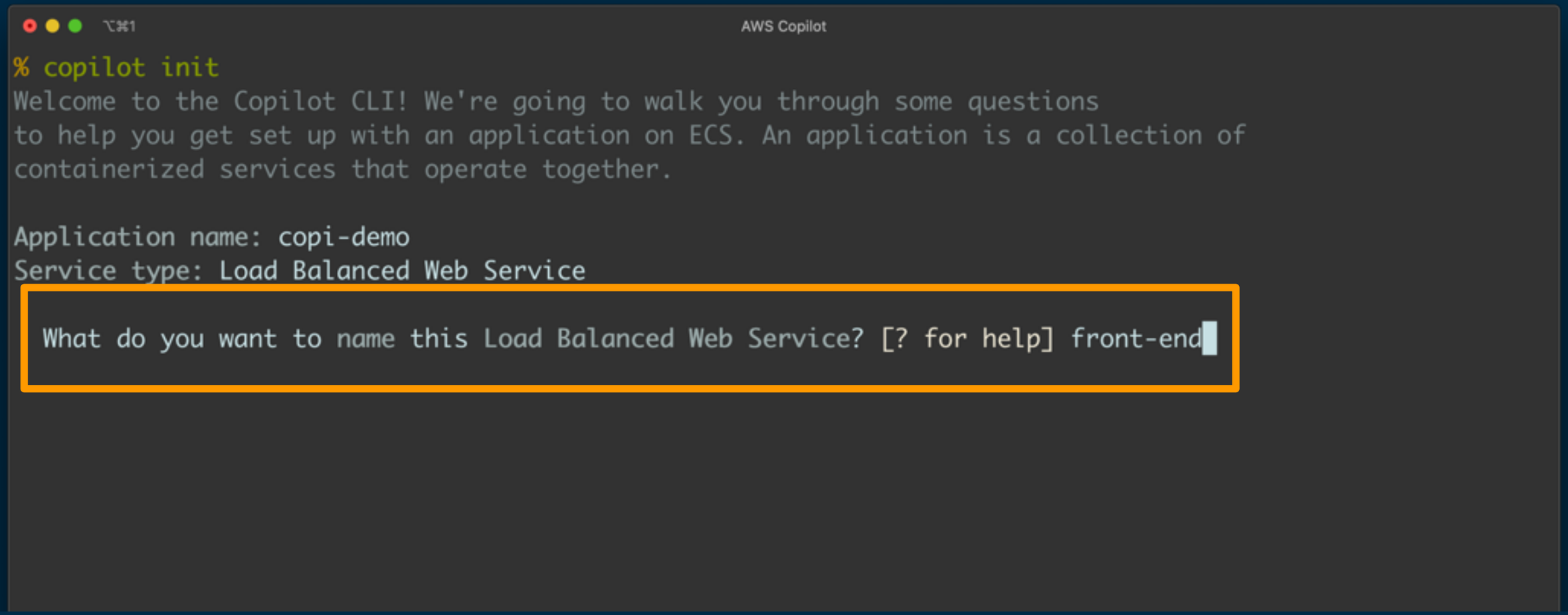
aws/copilot-cli



In Partnership with intel

Quick start with “AWS Copilot CLI”

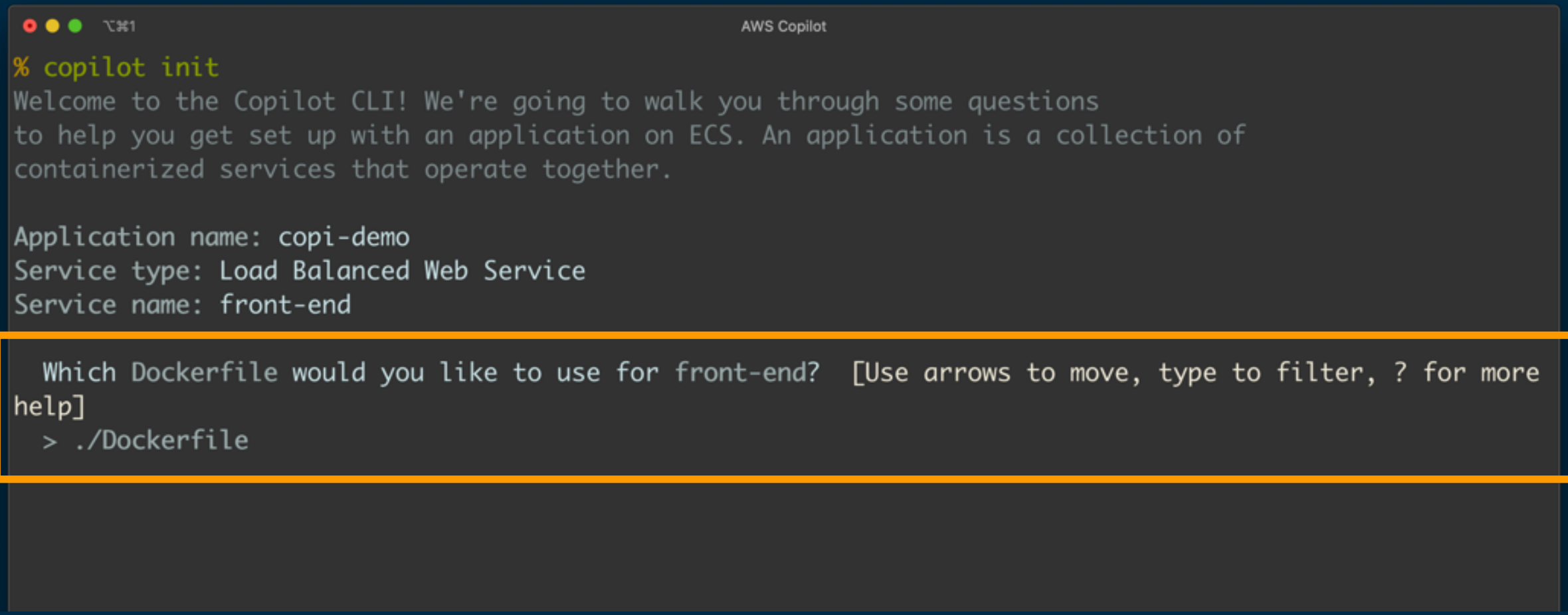
aws/copilot-cli



In Partnership with intel

Quick start with “AWS Copilot CLI”

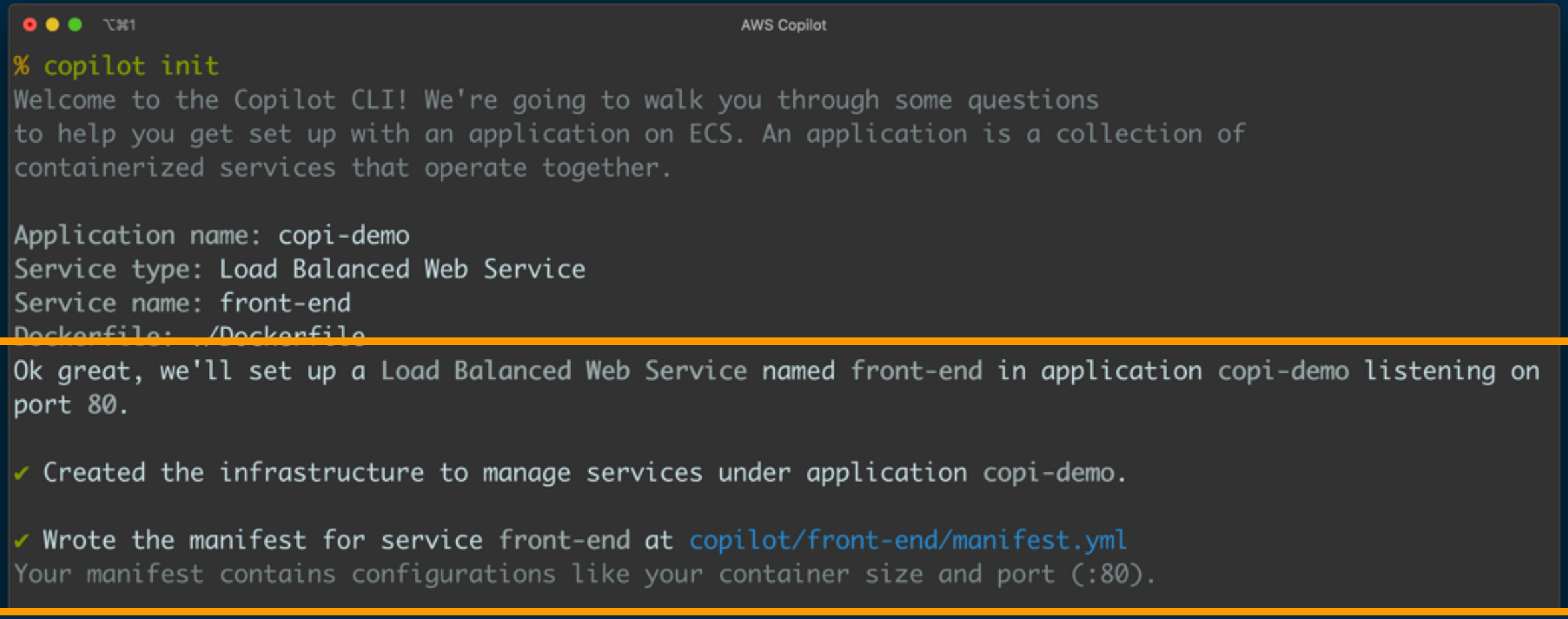
aws/copilot-cli



In Partnership with intel

Quick start with “AWS Copilot CLI”

aws/copilot-cli



In Partnership with intel

Quick start with “AWS Copilot CLI”

aws/copilot-cli

```
Application name: copi-demo
Service type: Load Balanced Web Service
Service name: front-end
Dockerfile: ./Dockerfile
Ok great, we'll set up a Load Balanced Web Service named front-end in application copi-demo listening on
port 80.

✔ Created the infrastructure to manage services under application copi-demo.

✔ Wrote the manifest for service front-end at copilot/front-end/manifest.yml
Your manifest contains configurations like your container size and port (:80).

✔ Created ECR repositories for service front-end.

All right, you're all set for local development.

  Would you like to deploy a test environment? [? for help] (y/N) y
```


In Partnership with intel

Quick start with “AWS Copilot CLI”

aws/copilot-cli

```
✔ Created the infrastructure to manage services under application copi-demo.

✔ Wrote the manifest for service front-end at copilot/front-end/manifest.yml
Your manifest contains configurations like your container size and port (:80).

✔ Created ECR repositories for service front-end.

All right, you're all set for local development.
Deploy: Yes

⠧ Creating the infrastructure for the test environment.
- Virtual private cloud on 2 availability zones to hold your services     [In Progress]
  - Internet gateway to connect the network to the internet               [In Progress]
  - Public subnets for internet facing services                           [In Progress]
  - Private subnets for services that can't be reached from the internet  [In Progress]
  - Routing tables for services to talk with each other                   [In Progress]
- ECS Cluster to hold your services                                       [In Progress]
- Application load balancer to distribute traffic                        [In Progress]
```


In Partnership with intel

Quick start with “AWS Copilot CLI”

aws/copilot-cli

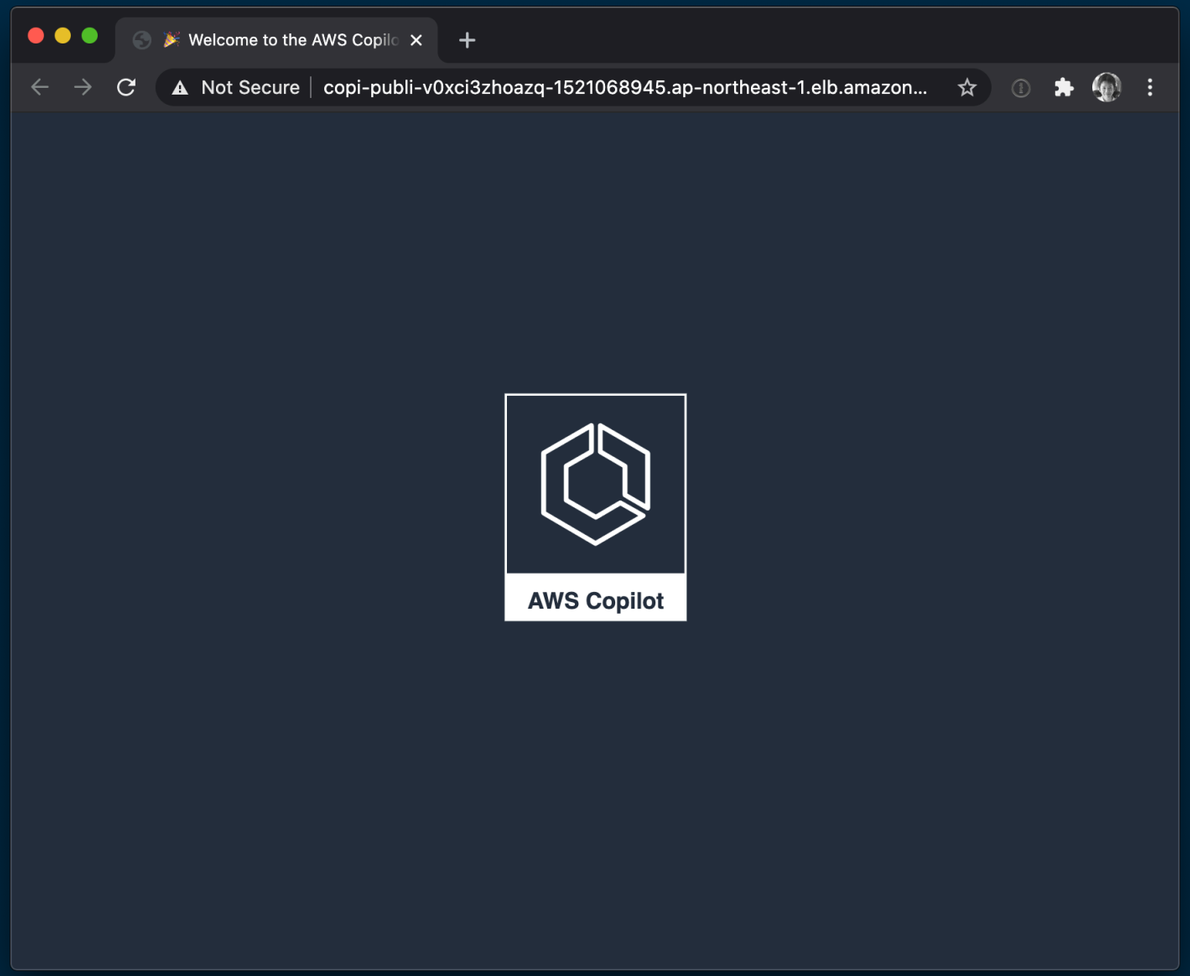
```
 ---> Using cache
 ---> 2a2bccb8a34f
Successfully built 2a2bccb8a34f
Successfully tagged 184834139833.dkr.ecr.ap-northeast-1.amazonaws.com/copi-demo/front-end:975886a
Login Succeeded
The push refers to repository [184834139833.dkr.ecr.ap-northeast-1.amazonaws.com/copi-demo/front-end]
14933325658e: Pushed
550333325e31: Pushed
22ea89b1a816: Pushed
a4d893caa5c9: Pushed
0338db614b95: Pushed
d0f104dc0a1f: Pushed
975886a: digest: sha256:f81af7aee1662c4d4f90347fed81d43282970b193544cbc7bc20952e682d8e88 size: 1570

✔ Deployed front-end, you can access it at http://copi-Publi-V0XCI3ZHOAZQ-1521068945.ap-northeast-1.elb.amazonaws.com.
%
```


In Partnership with intel

Quick start with “AWS Copilot CLI”

aws/copilot-cli



In Partnership with intel

aws DEV DAY
ONLINE JAPAN

AWS Fargate に対応したデプロイツール

(for Amazon ECS)

In Partnership with intel

Awesome-ecs からの抜粋

nathanpeck/awesome-ecs

- AWS CLI
- AWS CloudFormation
- Terraform
- AWS Copilot
- AWS CDK
- ecs-deploy
- ecspresso
- Docker Compose ECS integration

etc.

In Partnership with intel

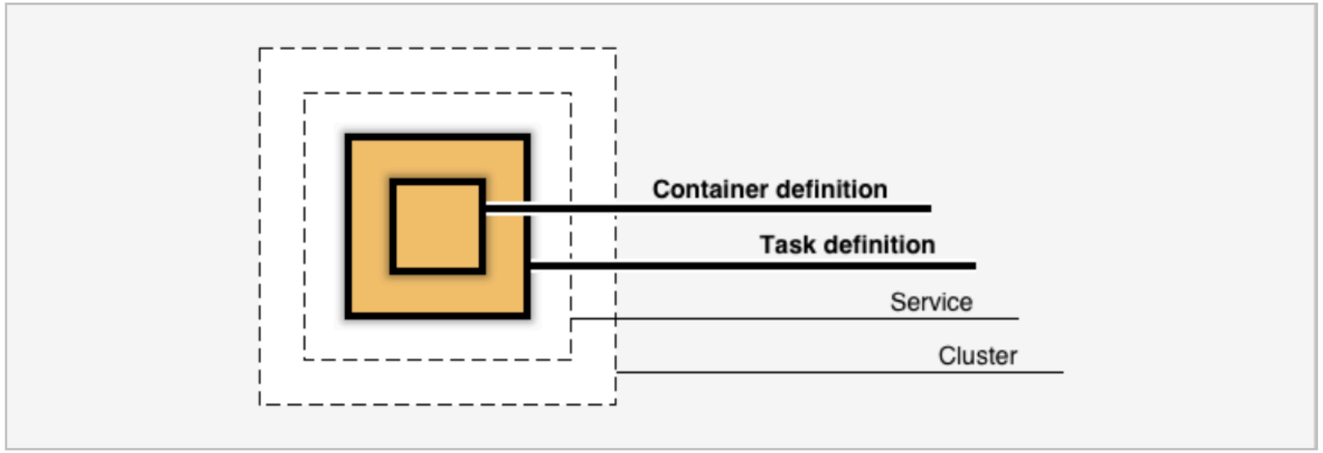
aws DEV DAY
ONLINE JAPAN

各デプロイツールの守備範囲や背景を読み解く

In Partnership with intel

AWS Management Console

ECS オブジェクトとそれらの関連を示す図



コンテナの定義

編集

以下のコンテナのイメージを選択し、すばやく使用を開始するか、使用するコンテナイメージを定義します。

sample-app

イメージ : httpd:2.4

メモリ : 0.5GB (512)

cpu : 0.25 vCPU (256)

nginx

イメージ : nginx:latest

メモリ : 0.5GB (512)

cpu : 0.25 vCPU (256)

In Partnership with intel

AWS Management Console

Pros

GUI で Fargate タスクを作成できる
ECS 以外の AWS サービスもまとめて設定可 e.g. Application Auto Scaling, CodeDeploy
実行中のタスクやサービスなどの情報閲覧が容易

Cons

デプロイが手作業になり再現性がない
ミスオペレーションのリスク
CI/CD パイプラインとの親和性はあまりない

👉 継続的なデプロイに使うのは難しそう

AWS CLI

aws/aws-cli

Pros

- AWS の新サービスリリース時やサービスアップデートのタイミングで概ねそれらに対応している
- AWS API の呼び出しパラメーターをほぼ全て利用可能

Cons

- CI/CD パイプラインがシェル芸になりがち
- リソース依存関係まで考えたデプロイには高度なスクリプト技術が必要
- ロールバックを想定したスクリプトは難しい

In Partnership with intel

ecs-deploy

silinternational/ecs-deploy

```
Examples:
  Simple deployment of a service (Using env vars for AWS settings):

    ecs-deploy -c production1 -n doorman-service -i docker.repo.com/doorman:latest

  All options:

    ecs-deploy -k ABC123 -s SECRETKEY -r us-east-1 -c production1 -n doorman-service -i docker.repo.com

  Updating a task definition with a new image:

    ecs-deploy -d open-door-task -i docker.repo.com/doorman:17

  Using profiles (for STS delegated credentials, for instance):

    ecs-deploy -p PROFILE -c production1 -n doorman-service -i docker.repo.com/doorman -t 240 -e CI_TIM
```


In Partnership with intel

ecs-deploy

silinternational/ecs-deploy

Pros

- 1つのシェルスクリプトの中で AWS の API を呼んでいるだけなので、中身を読んで理解しやすい
- コミュニティで揉まれているので、AWS CLI を使った自前シェルスクリプトを書くより堅牢
- ロールバックなども考慮されている

Cons

- ネットワークやロードバランサー、Fargate サービスなどのリソースは他のツールで作成済みであることを前提にしている
- シェルスクリプトであるため、つい魔改造したくなるし、しやすい

AWS CloudFormation / Terraform

Infrastructure as code

YAML ファイルなどに必要な AWS リソースについて宣言し、リソースの宣言的デプロイメントを可能にする

```
Resources:

  # The task definition. This is a simple metadata description of what
  # container to run, and what resource requirements it has.
  TaskDefinition:
    Type: AWS::ECS::TaskDefinition
    Properties:
      Family: !Ref 'ServiceName'
      Cpu: !Ref 'ContainerCpu'
      Memory: !Ref 'ContainerMemory'
      NetworkMode: awsvpc
      RequiresCompatibilities:
        - FARGATE
      ExecutionRoleArn:
        Fn::ImportValue:
          !Join [':', [!Ref 'StackName', 'ECSTaskExecutionRole']]
      TaskRoleArn:
        Fn::If:
          - 'HasCustomRole'
          - !Ref 'Role'
          - !Ref "AWS::NoValue"
      ContainerDefinitions:
```

```
resource "aws_ecs_service" "mongo" {
  name            = "mongodb"
  cluster         = "${aws_ecs_cluster.foo.id}"
  task_definition = "${aws_ecs_task_definition.mongo.arn}"
  desired_count   = 3
  iam_role        = "${aws_iam_role.foo.arn}"
  depends_on      = ["aws_iam_role_policy.foo"]

  ordered_placement_strategy {
    type  = "binpack"
    field = "cpu"
  }

  load_balancer {
    target_group_arn = "${aws_lb_target_group.foo.arn}"
    container_name   = "mongo"
    container_port   = 8080
  }

  placement_constraints {
    type       = "memberOf"
```

AWS CloudFormation / Terraform

Pros

- 単体で全ての AWS リソースを用意できるため、1つのツールで済む
- Fargate タスクだけでなく、関連する AWS リソースがどのような状態にあるのかを定義ファイルから確認しやすい
- 定義ファイルをバージョン管理システム管理下に置くことで変更履歴や変更理由を追いやすい
- 予定される変更差分を確認してから実行できる
- AWS リソース間の依存関係も定義できる
- 更新できないリソースの場合は新規作成してから古いリソースを消すような挙動を期待できる
- CI/CD パイプラインとの相性が良い

AWS CloudFormation / Terraform

Cons

- 使いこなすためには定義する AWS リソースそれぞれについて十分に知っておく必要がある
- 最新の API パラメーターのサポートまでに時間がかかることも
- VPC やデータベースと AWS Fargate タスクのようなライフサイクルが異なるリソースの管理方法を考えられる一定のスキルが必要
- 記述量が多くなりがち
- 記述量が多くなりがち
- 記述量が多くなりがち
  - Terraform Modules の利用で書く量は減らせる
  - が、ECS 向けのモジュールが(デプロイツール同様)たくさんある

AWS CDK

aws/aws-cdk

```
import ec2 = require('@aws-cdk/aws-ec2');
import ecs = require('@aws-cdk/aws-ecs');
import ecs_patterns = require('@aws-cdk/aws-ecs-patterns');
import cdk = require('@aws-cdk/core');

class BonjourFargate extends cdk.Stack {
  constructor(scope: cdk.App, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Create VPC and Fargate Cluster
    const vpc = new ec2.Vpc(this, 'MyVpc', { maxAzs: 2 });
    const cluster = new ecs.Cluster(this, 'Cluster', { vpc });

    const fargateService = new ecs_patterns.NetworkLoadBalancedFargateService(this, "FargateService", {
      cluster,
      image: ecs.ContainerImage.fromRegistry("amazon/amazon-ecs-sample"),
    });

    new cdk.CfnOutput(this, 'LoadBalancerDNS', { value: fargateService.loadBalancer.loadBalancerDnsName });
  }
}

const app = new cdk.App();
new BonjourFargate(app, 'Bonjour');
app.synth();
```

https://github.com/aws-samples/aws-cdk-examples/tree/master/typescript/ecs/fargate-application-load-balanced-service

In Partnership with intel

AWS CDK

aws/aws-cdk

Pros

- コード(e.g. TypeScript, Python…)でインフラを記述できるためデベロッパとの親和性が高く、テストも書きやすい
- 他のツールを併用しなくても Fargate を含めた AWS リソースを一通り作成・管理可能
- 抽象度が高く多くのパラメーターを省略でき、記述量を減らせる
- バックエンドが CloudFormation

Cons

- 抽象度が高いため記述量が少ないが、意識していないとなぜそういうリソースが作られるのか説明できなくなる可能性も

ECS extensions for CDK (Developer Preview)

New!

aws/aws-cdk

```
const svcDesc = new ServiceDescription();
svcDesc.add(new Container({
   cpu: 1024,
   memoryMiB: 2048,
   trafficPort: 80,
   image: ContainerImage.fromRegistry('ecs-sample'),
}));
svcDesc.add(new HttpLoadBalancerExtension());
svcDesc.add(new AppMeshExtension({ mesh }));
svcDesc.add(new FireLensExtension());
svcDesc.add(new XRayExtension());
svcDesc.add(new CloudwatchAgentExtension());
svcDesc.add(new ScaleOnCpuUtilization({
   initialTaskCount: 2,
   minTaskCount: 2,
}));

const mySvc = new Service(stack, 'my-service', {
   environment: environment,
   serviceDescription: svcDesc,
});
```

“Extensions” による拡張

- Elastic Load Balancing
- AWS App Mesh
- AWS FireLens
- AWS X-Ray
- AWS CloudWatch Agent
- CPU ベースのアプリケーション自動スケーリング

etc.

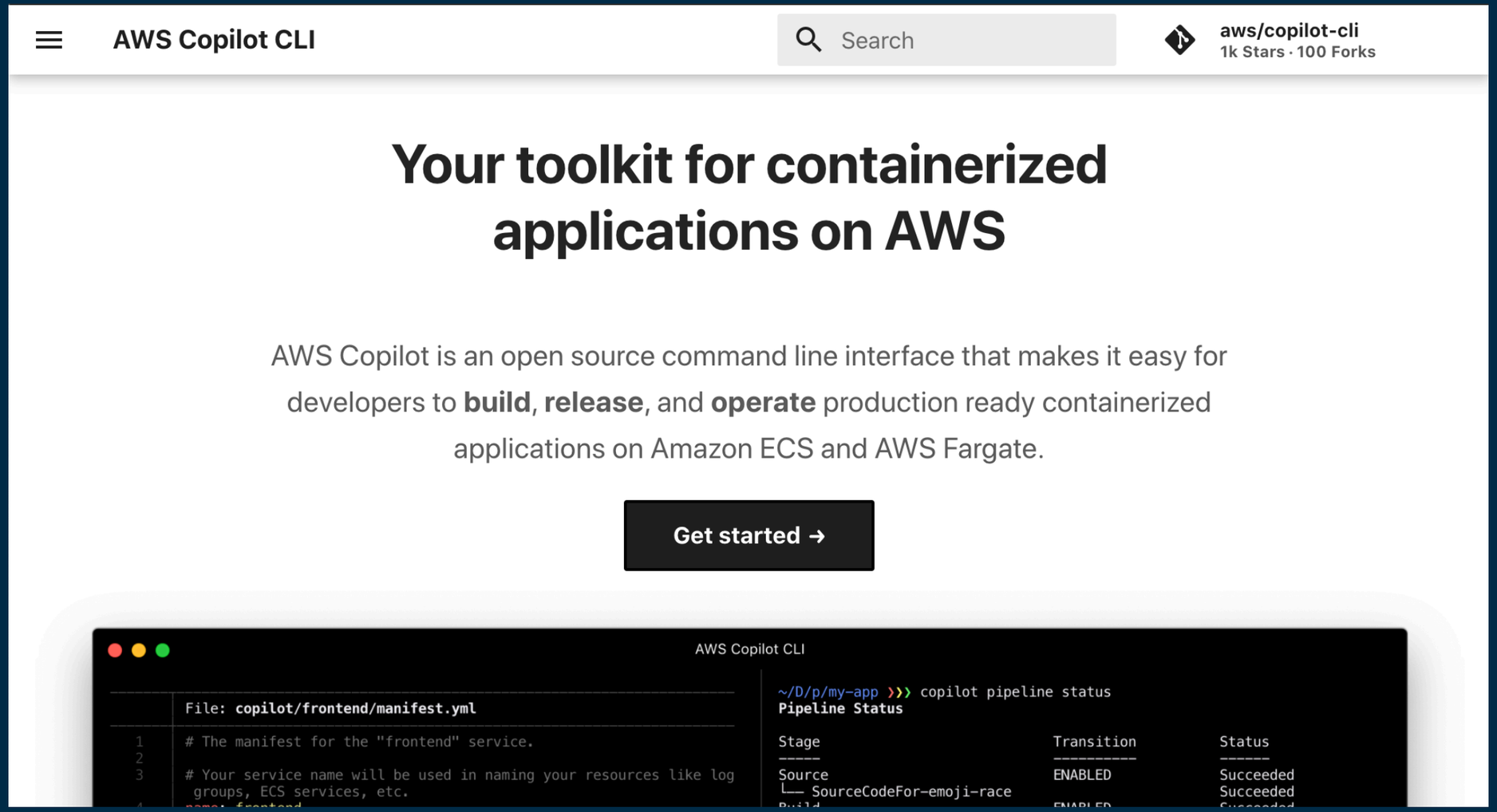
https://github.com/aws/aws-cdk/tree/master/packages/%40aws-cdk-containers/ecs-service-extensions

In Partnership with intel

AWS Copilot (再掲, Developer Preview)

New!

aws/copilot-cli



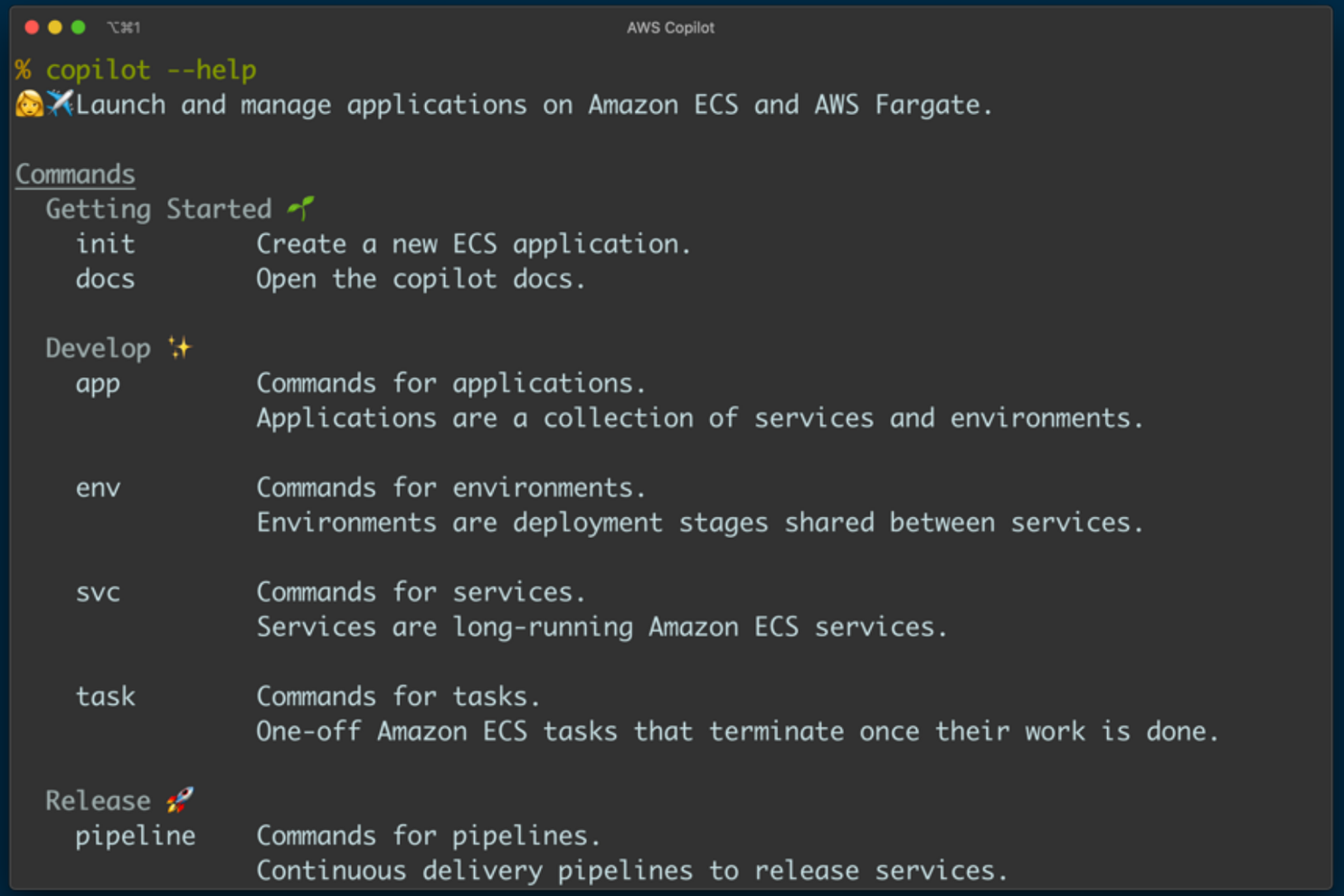
https://aws.github.io/copilot-cli/

In Partnership with intel

AWS Copilot (再掲, Developer Preview)

New!

aws/copilot-cli



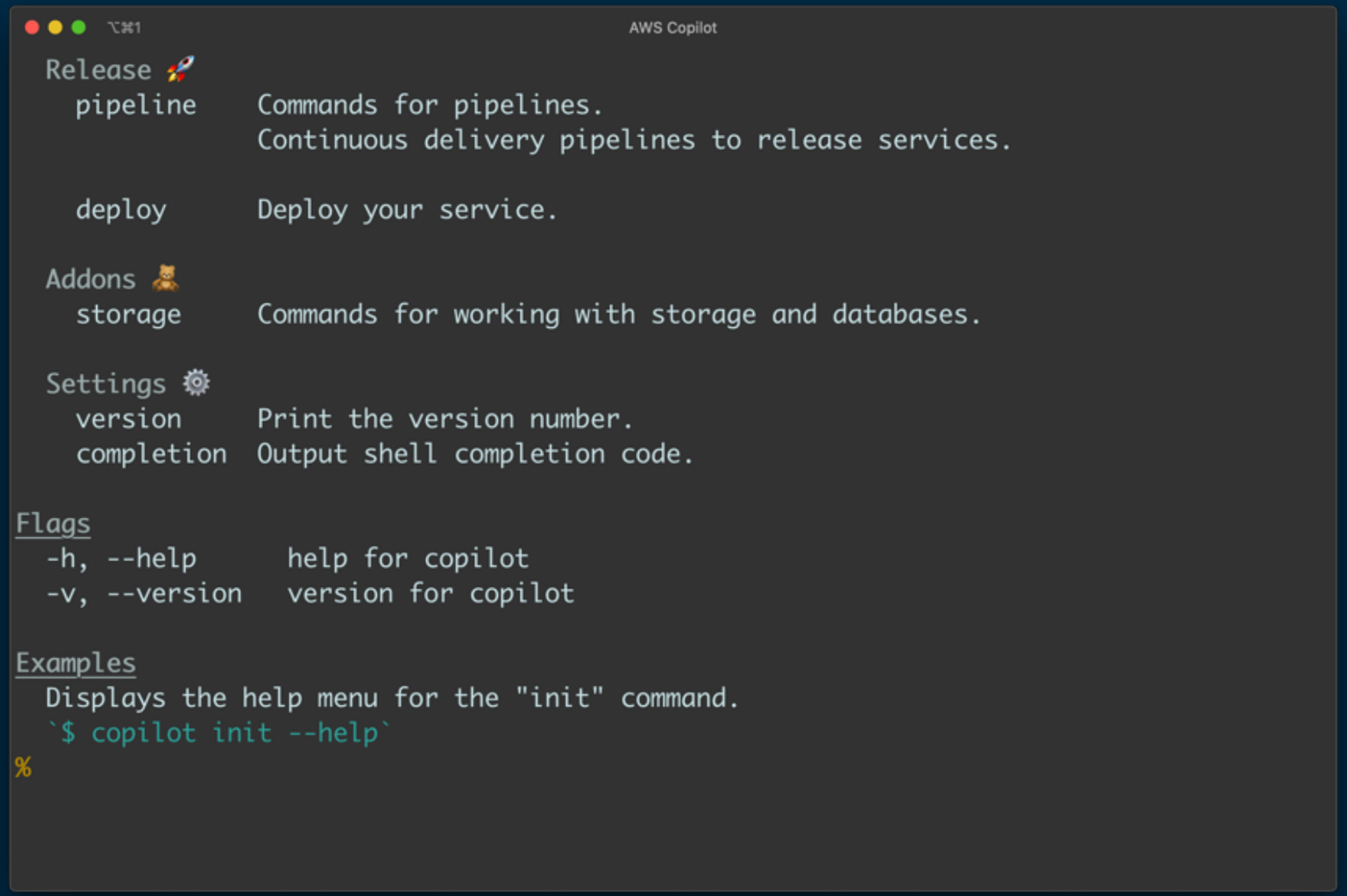
https://aws.github.io/copilot-cli/

In Partnership with intel

AWS Copilot (再掲, Developer Preview)

New!

aws/copilot-cli



https://aws.github.io/copilot-cli/

In Partnership with intel

AWS Copilot (再掲, Developer Preview)

New!

aws/copilot-cli

Pros

- 最低限必要なのはアプリケーションと Dockerfile のみ
- アーキテクチャが *“Load Balanced Web Service”, “Backend Service”* のようなユースケースの選択に抽象化されており、それらに適した AWS リソースをまとめて構築可能
- 抽象度が高く、 AWS CDK よりも更に少ない記述量
- 複数 AWS アカウントや複数環境へのデプロイをサポートし、それらへの継続的デリバリを実施する CI/CD パイプラインを作成可能
- CloudFormation による任意のリソース持ち込みを可能にする “Add-on”
  - この CloudFormation テンプレートを AWS CDK で書くことも当然可能
- バックエンドが CloudFormation
- プロダクションレディな環境を CloudFormation で定義する勉強材料に

In Partnership with intel

AWS Copilot (再掲, Developer Preview)

New!

aws/copilot-cli

Cons

- 現時点では Developer Preview のステージであり、細かい設定変更は未サポート e.g. 『セキュリティグループで特定 IP アドレスからのみアクセスを許可したい』

💡AWS Copilot については Appendix も合わせてご覧ください

In Partnership with intel

ecspresso

＼祝🎉 v1.0.0！／

kayac/ecspresso

```
usage: ecspresso --config=CONFIG [<flags>] <command> [<args>

Flags:
  --help           Show context-sensitive help (also try --he
  --config=CONFIG  config file
  --debug          enable debug log

Commands:
  help [<command>...]
    Show help.

  version
    show version

  deploy [<flags>]
    deploy service

  create [<flags>]
    create service

  status [<flags>]
    show status of service
```

```
  rollback [<flags>]
    rollback service

  delete [<flags>]
    delete service

  run [<flags>]
    run task

  register [<flags>]
    register task definition

  wait
    wait until service stable

  init --region=REGION --service=SERVICE [<flags>]
    create service/task definition files by existing ECS se

  diff [<flags>]
    display diff for task definition compared with latest d
```


In Partnership with intel

ecspresso

＼祝🎉v1.0.0！／
kayac/ecspresso

Pros

- 非常に薄いレイヤーで作られた AWS API ラッパー
- 画面から作ったタスクや、既存の ECS サービスを後から CI/CD に載せることが想定されていてとても好き

```
$ ecspresso init --region ap-northeast-1 --cluster default --service myservice --config config.yaml
2019/10/12 01:31:48 myservice/default save service definition to ecs-service-def.json
2019/10/12 01:31:48 myservice/default save task definition to ecs-task-def.json
2019/10/12 01:31:48 myservice/default save config to config.yaml
```

```
$ ecspresso deploy --config config.yaml
```

- テンプレーティング + 環境変数利用での定義ファイル出力が大好き
  e.g. *{{ env `FOO` `bar` }}* や *{{ must_env `FOO` }}* など

In Partnership with intel

ecspresso

＼祝🎉v1.0.0！／
kayac/ecspresso

Pros

- *tfstate* ファイル内の値を参照できて好き

config.yaml

```yaml
region: ap-northeast-1
cluster: default
service: test
service_definition: ecs-service-def.json
task_definition: ecs-task-def.json
plugins:
- name: tfstate
  config:
    path: terraform.tfstate    # path to
```

ecs-service-def.json

```
{
  "networkConfiguration": {
    "awsvpcConfiguration": {
      "subnets": [
        "{{ tfstate `aws_subnet.private-a.id` }}"
      ],
      "securityGroups": [
        "{{ tfstate `data.aws_security_group.default.id` }}"
      ]
    }
```


In Partnership with intel

ecspresso

＼祝🎉 v1.0.0！／

kayac/ecspresso

Cons

- VPC やロードバランサーなど、Fargate タスク・サービス以外のリソースは事前に作成済みであることを前提としており、AWS についてある程度の事前知識が必要
- また、それらリソースの作成・管理は別ツールの併用が前提

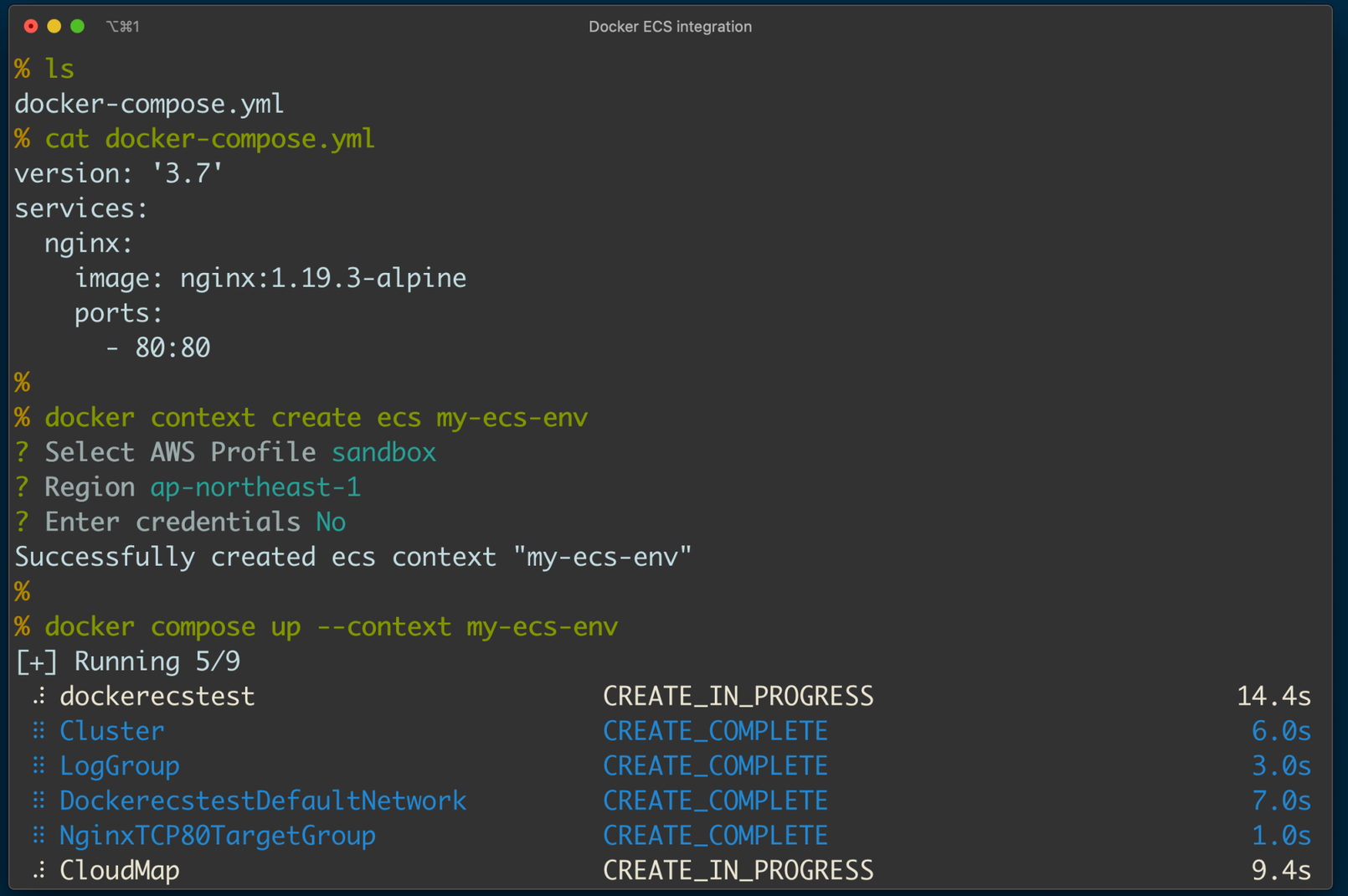
💡これらは ecspresso において意図的にそのようにデザイン・実装されていることに由来すると思われるため、実は Cons らしい Cons でもない

In Partnership with intel

Docker Compose ECS integration (Beta)

New!

docker/compose-cli



In Partnership with intel

Docker Compose ECS integration (Beta)

New!

docker/compose-cli

Pros

- Docker Compose を利用したローカル開発と非常に親和性が高い
- ロードバランサや CloudWatch ロググループなど Fargate タスクの実行に必要なものが一通り作成される
- バックエンドが CloudFormation

Cons

- 現時点では Beta であり、例えば ECS サービスの更新は未サポート
- VPC やサブネットなどのリソースは事前に作成済みであることを前提としており、未指定の場合は default リソースが利用される
- Docker Compose が管理しない AWS リソースの作成・管理には別ツールの併用が必要

aws

In Partnership with intel

aws DEV DAY
ONLINE JAPAN

マッチしそうなツールはありましたか？

In Partnership with intel

マッチしそうなツールはありましたか？

とりあえず動いているところを見たい / これから AWS をはじめる

AWS Copilot が一番簡単そう (講演者調べ)

運用も考えつつ、CI/CD を意識してツールを選びたい

1. 単一のツールで全てのリソースを記述したい

AWS CLI / CloudFormation / Terraform / AWS CDK

2. Fargate へのデプロイはより抽象化されたものでやりたい

- ecs-deploy / ecspresso / Docker Compose ECS integration のいずれかと上記ツールの組み合わせ
- AWS CDK + ECS extensions for CDK
- AWS Copilot

aws DEV DAY
ONLINE JAPAN

Thank you!

Tori Hara / toricls
AWS

In Partnership with intel

aws DEV DAY
ONLINE JAPAN

Appendix

In Partnership with intel

Appendix

💡 AWS Copilot - https://aws.github.io/copilot-cli/

- AWS Copilot のご紹介 https://aws.amazon.com/jp/blogs/news/introducing-aws-copilot/
- AWS Copilot によるコンテナアプリケーションの自動デプロイ https://aws.amazon.com/jp/blogs/news/automatically-deploying-your-container-application-with-aws-copilot/

💡 Docker Compose ECS integration - https://github.com/docker/compose-cli

- Deploying Docker containers on ECS https://docs.docker.com/engine/context/ecs-integration/

💡 第1回 AWS Fargate かんたんデプロイ選手権

https://speakerdeck.com/toricls/the-very-first-aws-fargate-easy-deployment-tooling-championship

In Partnership with intel