

コードで学ぶAWS入門

真野 智之 (Tomoyuki Mano) <tomoyukimano@gmail.com>

Version 1.0, 2021-06-14

Table of Contents

1.はじめに	2
1.1. 本書の目的・内容	2
1.2. 本書のフィロソフィー	2
1.3. AWSアカウント	3
1.4. 必要な計算機環境	4
1.5. 前提知識	4
1.6. 講義に関連する資料	4
1.7. 本書で使用するノーテーションなど	5
2.クラウド概論	6
2.1. クラウドとは?	6
2.2. なぜクラウドを使うのか?	7
2.3. どうやってクラウドを使うのか?	8
3.AWS入門	10
3.1. AWSとは?	10
3.2. AWSの機能・サービス	10
3.3. AWSでクラウドを作るときの基本となる部品	11
3.4. Region と Availability Zone	11
3.5. AWSでのクラウド開発	13
3.6. CloudFormation と AWS CDK	17
4.Hands-on #1: 初めてのEC2インスタンスを起動する	20
4.1. 準備	20
4.2. SSH	21
4.3. アプリケーションの説明	21
4.4. プログラムを実行する	26
4.5. 小括	31
5.クラウドで行う科学計算・機械学習	33
5.1. なぜ機械学習をクラウドで行うのか?	33
5.2. GPU による機械学習の高速化	34
6.Hands-on #2: AWS でディープラーニングを実践	37
6.1. 準備	37
6.2. アプリケーションの説明	37
6.3. スタックのデプロイ	40
6.4. ログイン	41
6.5. Jupyter notebook の起動	42
6.6. PyTorchはじめの一歩	44
6.7. CPU vs GPU の簡易ベンチマーク	47
6.8. 実践ディープラーニング! MNIST手書き数字認識タスク	48
6.9. スタックの削除	52
7.Docker 入門	53
7.1. 機械学習の大規模化	53
7.2. Docker とは	54
7.3. Docker チュートリアル	56
7.4. Elastic Container Service (ECS)	59
8.Hands-on #3: AWSで自動質問回答ボットを走らせる	61
8.1. Fargate	61
8.2. 準備	62

8.3. Transformer を用いた question-answering プログラム	62
8.4. アプリケーションの説明	64
8.5. スタックのデプロイ	68
8.6. タスクの実行	69
8.7. タスクの同時実行	71
8.8. スタックの削除	72
8.9. 講義第二回目のまとめ	72
9. Hands-on #4: AWS Batch を使って機械学習のハイパーパラメータサーチを並列化する	74
9.1. Auto scaling groups (ASG)	74
9.2. AWS Batch	74
9.3. 準備	75
9.4. MNIST 手書き文字認識 (再訪)	75
9.5. ローカルで Docker を実行	76
9.6. アプリケーションの説明	78
9.7. スタックのデプロイ	81
9.8. Docker image を ECR に配置する	82
9.9. Job を実行する (まずはひとつだけ)	85
9.10. 並列にたくさんの Job を実行する	89
9.11. スタックの削除	93
10. Web サービスの作り方	95
10.1. ウェブサービスの仕組み – Twitter を例に	95
10.2. REST API	95
11. Serverless architecture	98
11.1. Serverful クラウド (従来型)	98
11.2. Serverless クラウドへ	99
11.3. Lambda	100
11.4. サーバーレスストレージ: S3	101
11.5. サーバーレスデータベース: DynamoDB	102
11.6. その他のサーバーレスクラウドの構成要素	103
12. Hands-on #4: サーバーレス入門	104
12.1. Lambda ハンズオン	104
12.2. DynamoDB ハンズオン	108
13. Hands-on #5: Bashoutter	112
13.1. 準備	112
13.2. アプリケーションの説明	113
13.3. アプリケーションのデプロイ	119
13.4. API を送信する	122
13.5. 大量の API リクエストをシミュレートする	124
13.6. Bashoutter GUI を動かしてみる	124
13.7. アプリケーションの削除	125
13.8. 講義第三回目のまとめ	126
14. まとめ	127
15. Appendix	128
15.1. AWS のシークレットキーの作成	128
15.2. AWS CLI のインストール	129
15.3. AWS CDK のインストール	131
15.4. Python <code>venv</code> クイックガイド	131
15.5. ハンズオン実行用の Docker image の使い方	132

16. 謝辞	134
17. 著者紹介	135
18. ライセンス	136

ハンズオンで使うプログラムや教科書のソースコードは以下のウェブページで公開している。

<https://github.com/tomomano/learn-aws-by-coding>

Chapter 1. はじめに

1.1. 本書の目的・内容

本書は、東京大学計数工学科で2021年度S1/S2タームに開講されている"システム情報工学特論"の講義資料として作成された。

本書の目的は、クラウドの初心者を対象とし、クラウドの基礎的な知識・概念を解説する。また、Amazon Web Service (AWS) の提供するクラウド環境を実例として、具体的なクラウドの利用方法をハンズオンを通して学ぶ。

特に、科学・エンジニアリングの学生を対象として、研究などの目的でクラウドを利用するための実践的な手順を紹介する。知識・理論の説明は最小限に留め、実践を行う中で必要な概念の解説を行う予定である。受講生が今後、研究などでクラウドを利用する際の、足がかりとなることができればこの講義の目的は十分達成されることになる。

本書は以下のような三部構成になっている。

Table 1. 本書の構成

	テーマ	ハンズオン
第一部 (1章-4章)	クラウドの基礎	<ul style="list-style-type: none">AWSに自分のサーバーを立ち上げる
第二部 (5章-9章)	クラウドを活用した機械学習	<ul style="list-style-type: none">AWSとJupyterを使って始めるディープラーニングスケーラブルな自動質問回答ボットを作る並列化されたハイパーパラメータサーチの実装
第三部 (10章-13章)	Serverless Architecture 入門	<ul style="list-style-type: none">俳句を投稿するSNS "Bashoutter" を作る

第一部は、クラウドの基礎となる概念・知識を解説する。セキュリティやネットワークなど、クラウドを利用する上で最低限おさえなければいけないポイントを説明する。ハンズオンでは、初めての仮想サーバーをAWSに立ち上げる演習を行う。

第二部では、クラウド上で科学計算(特に機械学習)を走らせるための入門となる知識・技術を解説する。併せて、Dockerとよばれる仮想計算環境の使用方法を紹介する。ハンズオンでは、AWSのクラウドでJupyter notebookを起動し簡単な機械学習の計算を走らせる課題を実践する。さらに、深層学習を用いた自然言語処理により、質間に自動で回答を生成するボットを作成する。また、同時に複数のGPUインスタンスを起動し、深層学習のハイパーパラメータサーチを行う方法を紹介する。

第三部では、Serverless architectureと呼ばれる最新のクラウドのアーキテクチャを紹介する。これは、サーバーの処理能力を負荷に応じてより柔軟に拡大・縮小するための概念であり、それ以前(Serverfullとしばしば呼ばれる)と質的に異なる設計思想をクラウドに導入するものである。これの実践として、ハンズオンでは簡単なSNSをクラウド上に作成する。

1.2. 本書のフィロソフィー

本書のフィロソフィーを一言で表すなら、"口ケットで宇宙まで飛んでいって一度地球を眺めてみよう!" である。

どういうことか?

ここでいう"地球"とは、クラウドコンピューティングの全体像のことである。言うまでもなく、クラウドという技術は非

常に広範かつ複雑な概念で、幾多の情報技術・ハードウェア・アルゴリズムが精緻に組み合わさせてできた総体である。そして、今日では科学研究から日常のインフラ設備に至るまで、我々の社会の多くの部分がクラウド技術によって支えられている。

ここでいう"ロケット"とはこの講義のことである。この講義では、ロケットに乗って宇宙まで飛び立ち、地球(クラウド)の全体を自身の目で眺めてもらう。その際、ロケットの成り立ちや仕組み(背後にある要素技術やプログラムのソースコード)は深くは問わない。将来、自分が研究などの目的でクラウドを利用することになった時に、改めて学んでもらえれば良い。本書の目的はむしろ、クラウドの最先端に実際に触れ、そこからどんな景色が見えるか(どんな応用が可能か)を実感してもらうことである。

そのような理由で、本書はクラウドの基礎から応用まで幅広いテーマを取り扱う。第一部はクラウドの基礎から始め、第二部では一気にレベルアップし機械学習(深層学習)をクラウドで実行する手法を解説する。さらに第三部では、サーバーレス・アーキテクチャというここ数年のうちに確立した全く新しいクラウドの設計について解説する。それぞれで一本一冊分以上の内容に相当するものであるが、本書はあえてこれらを一冊にまとめ連続的に俯瞰するという野心的な意図を持って執筆された。

決して楽な搭乗体験ではないかもしれないが、このロケットにしがみついてきてもらえば、とてもエキサイティングな景色が見られることを約束したい。

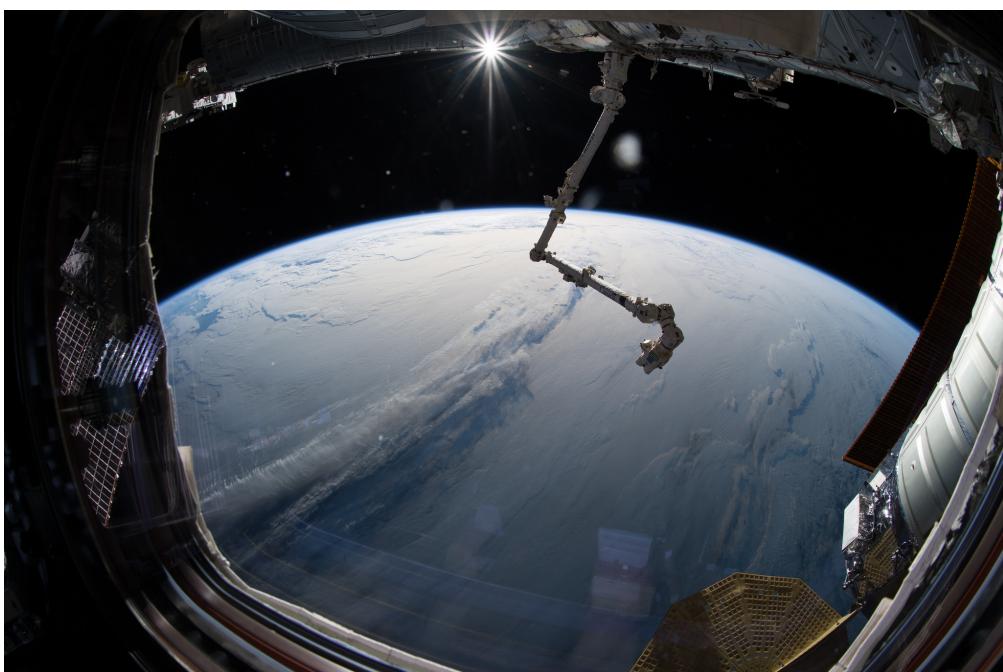


Figure 1. 宇宙からみた地球 (Image from NASA <https://www.nasa.gov/image-feature/planet-of-clouds>)

1.3. AWSアカウント

本書では、ハンズオン形式で AWS のクラウドを実際に動かす演習を提供する。自分でハンズオンを実行してみたい読者は、各自で AWS のアカウントの作成をしていただく。AWS のアカウントの作成の仕方は [公式のドキュメンテーション](#) を参照のこと。

AWS にはいくつかの機能に対して無料利用枠が設定されており、いくつかのハンズオンは無料の範囲内で実行できる。一方、他のハンズオン(特に機械学習を扱うもの)では数ドル程度のコストが発生する。ハンズオンごとに発生するおおよそのコストについて記述があるので、注意をしながらハンズオンに取り組んでいただきたい。

また、大学などの教育機関における講義で AWS を使用する際は、[AWS Educate](#) というプログラムを利用することも可能である。これは、講義の担当者が申請を行うことで、受講する学生に対し AWS クレジットが提供されるというプログラムである。これを利用することで金銭的な負担なしに AWS のクラウドを実際に使用することができる。また、講義を経由せず個人で AWS Educate に参加することも可能である。AWS Educate からは様々な学習教材が提供されているので、ぜひ活用してもらいたい。

1.4. 必要な計算機環境

講義では,AWS上にクラウドを展開するハンズオンを実施する.そこで紹介するプログラムを実行するため,以下の計算機環境が必要である.

- **UNIX 系コンソール:** ハンズオンで紹介するコマンドを実行したり, SSH でサーバーにアクセスするため, UNIX 系のコンソール環境が必要である. Mac または Linux のユーザーは, OS に標準搭載のコンソール(ターミナルとも呼ばれる)を使用すればよい. Windows のユーザーは, [Windows Subsystem for Linux \(WSL\)](#) を使って Linux の仮想環境をインストールすることを推奨する. WSL のインストールについては, [公式ドキュメンテーションを参照](#)のこと.
- **Docker:** 講義では Docker の使い方を解説する. 予め自身の計算機に Docker のインストールをしておくこと. Linux/Mac/Windows のインストール法については [公式ドキュメンテーションを参照](#). 執筆時点において, [Windows 10 Home](#) へのインストールには WSL2 バックエンドの設定が必要である. 詳細は [こちらのドキュメンテーションを参照](#)のこと.
- **Python** (Version 3.6以上) (venv の使い方は [Section 15.4 参照](#))
- **Node.js** (version 12.0以上)
- **AWS CLI** (インストールについては [Section 15.2 参照](#))
- **AWS CDK** (インストールについては [Section 15.3 参照](#))

1.4.1. ハンズオン実行用の Docker Image

Python, Node.js, AWS CDK など, ハンズオンのプログラムを実行するために必要なプログラム/ライブラリがインストール済みの Docker image を用意した. また, ハンズオンのソースコードもクローン済みである. Docker の使い方を知っている読者は, これを使えば, 諸々のインストールをする必要なく, すぐにハンズオンのプログラムを実行できる.

次のコマンドで起動する.

```
$ docker run -it tomomano/labc
```

この Docker image の使い方や詳細は [Section 15.5](#) に記載している.

1.5. 前提知識

本書を読むにあたり,一般教養レベル以上の前提知識は特に仮定しない. が,以下の事前知識があるとよりスムーズに理解をることができるだろう.

- **Python の基本的な理解:** 本書ではPythonを使ってプログラムの作成を行う. 使用するライブラリは十分抽象化されており, 関数の名前を見ただけで意味が明瞭なものがほとんどであるので, Python に詳しくなくても心配する必要はない.
- **Linux コマンドラインの基礎的な理解:** クラウドを利用する際, クラウド上に立ち上がるサーバーは基本的に Linux である. Linux のコマンドラインについて知識があると, トラブルシュートなどが容易になる. 筆者のおすすめの参考書は [The Linux Command Line by William Shotts](#) である. ウェブで無料で読むことができるので, 読んだことのない人はぜひ一読を.

1.6. 講義に関連する資料

ハンズオンで使うプログラムや教科書のソースコードは以下のウェブページで公開している.

<https://github.com/tomomano/learn-aws-by-coding>

1.7. 本書で使用するノーテーションなど

- ・コードやシェルのコマンドは `monospace letter` で記述する.
- ・シェルに入力するコマンドは,それがシェルコマンドであると明示する目的で,先頭に `$` がつけてある. `$` はコマンドをコピー&ペーストするときは除かなければならない. 逆に,コマンドの出力には `$` はついていない点に留意する.

また,以下のような形式で注意やチップスを提供する.



追加のコメントなどを記す.



発展的な議論やアイディアなどを紹介する.



陥りやすいミスなどの注意事項を述べる.



絶対に犯してはならないミスを指摘する.

Chapter 2. クラウド概論

2.1. クラウドとは？



クラウドとはなにか？クラウドという言葉は、それ自身がとても広い意味を持つので、厳密な定義付けを行うことは難しい。

学術的な意味でのクラウドの定義づけをするとしたら、NIST(米国・国立標準技術研究所)による [The NIST Definition of Cloud Computing](#) が引用されることが多い。これによると、クラウドとは以下の要件が満たされたハードウェア/ソフトウェアの総体のことをいう。

- **On-demand self-service** 利用者のリクエストに応じて計算資源が自動的に割り当てられる。
 - **Broad network access** 利用者はネットワークを通じてクラウドにアクセスできる。
 - **Resource pooling** クラウドプロバイダーは、所有する計算資源を分割することで複数の利用者に計算資源を割り当てる。
 - **Rapid elasticity** 利用者のリクエストに応じて、迅速に計算資源の拡大あるいは縮小を行うことができる。
 - **Measured service** 計算資源の利用量を計測・監視することができる。

…と、いわれても抽象的でよくわからないかもしれない。もう少し具体的な話をする。

個人が所有する計算機で、CPUをアップグレードしようと思ったら、物理的に筐体を開け、CPUソケットを露出させ、新しいCPUに交換する必要があるだろう。あるいは、ストレージがいっぱいになってしまったら、古いディスクを抜き取り、新しいディスクを挿入する必要がある。計算機の場所を移動させたときには、新しい部屋のLANケーブルを差し込まないとネットワークには接続できない。

クラウドでは、これらの操作がプログラムからのコマンドによって実行できる。CPUが1000個欲しいと思ったらならば、そのようにクラウドプロバイダーにリクエストを送れば良い。すると、数分もしないうちに1000 CPUの計算資源が割り当てられる。ストレージを1TBから10TBに拡張しようと思ったならば、そのようにコマンドを送ればよい（これは、Google Drive や Dropbox などのサービスなどで馴染みのある人も多いだろう）。計算資源を使い終わったら、そのことをプロバイダーに伝えれば、割り当て分はすぐさま削除される。クラウドプロバイダーは、使った計算資源の量を正確にモニタリングしており、その量をもとに利用料金の計算が行われる。

このように、クラウドの本質は物理的なハードウェアの仮想化・抽象化であり、利用者はコマンドを通じて、まるでソフトウェアの一部かのように、物理的なハードウェアの管理・運用を行うことができる。もちろん、背後では、データセンターに置かれた膨大な数の計算機が大量の電力を消費しながら稼働している。クラウドプロバイダーはデータセンターの計算資源を上手にやりくりし、ソフトウェアとしてのインターフェースをユーザーに提供することで、このような仮想化・抽象化を達成しているわけである。クラウドプロバイダーの視点からすると、大勢のユーザーに計算機を貸し出し、データセンターの稼働率を常に100%に近づけることで、利益率の最大化を図っているのである。

著者の言葉で、クラウドの重要な特性を定義するならば、以下のようなになる。

クラウドとは計算機ハードウェアの抽象化である。つまり、物理的なハードウェアをソフトウェアの一部かのように自在に操作・拡大・接続することを可能にする技術である。

2.2. なぜクラウドを使うのか？

上述のように、クラウドとはプログラムを通じて自由に計算資源を操作することができる計算環境である。ここでは、リアルなローカル計算環境と比べて、なぜクラウドを使うと良いことがあるのかについて述べたい。

1. 自由にサーバーのサイズをスケールできる

なにか新しいプロジェクトを始めるとき、あらかじめ必要なサーバーのスペックを知るのは難しい。いきなり大きなサーバーを買うのはリスクが高い。一方で、小さすぎるサーバーでは、後のアップグレードが面倒である。クラウドを利用すれば、プロジェクトを進めながら、必要な分だけの計算資源を確保することができる。

2. 自分でサーバーをメンテナンスする必要がない

悲しいことに、コンピュータとは古くなるものである。最近の技術の進歩の速度からすると、5年も経てば、もはや当時の最新コンピューターも化石と同じである。5年ごとにサーバーを入れ替えるのは相当な手間である。またサーバーの停電や故障など不意の障害への対応も必要である。クラウドでは、そのようなインフラの整備やメンテナンスはプロバイダーが自動でやってくれるので、ユーザーが心配する必要がない。

3. 初期コスト0

自前の計算環境とクラウドの、経済的なコストのイメージを示したのが Figure 2 である。クラウドを利用する場合の初期コストは基本的に0である。その後、使った利用量に応じてコストが増大していく。一方、自前の計算環境では、大きな初期コストが生じる。その分、初期投資後のコストの増加は、電気利用料やサーバー維持費などに留まるため、クラウドを利用した場合よりも傾きは小さくなる。自前の計算機では、ある一定期間後、サーバーのアップグレードなどによる支出が生じることがある。一方、クラウドを利用する場合は、そのような非連続なコストの増大は基本的に生じない。クラウドのコストのカーブが、自前計算環境のコストのカーブの下にある範囲においては、クラウドを使うことは経済的なコスト削減につながる。



Figure 2. クラウドと自前計算機環境の経済的コストの比較

特に、1.の点は研究の場面では重要であると筆者は感じる。研究をやっていて、四六時中計算を走らせ続けるという場合は少ない。むしろ、新しいアルゴリズムが完成したとき・新しいデータが届いたとき、集中的・突発的に計算タスクが増大することが多いだろう。そういうときに、フレキシブルに計算力を増強させることができるのは、クラウドを使う大きなメリットである。

ここまでクラウドを使うメリットを述べたが,逆に,デメリットというのも当然存在する.

1. クラウドは賢く使わないといけない

[Figure 2](#) で示したコストのカーブにあるとおり,使い方によっては自前の計算環境のほうがコスト的に有利な場面は存在しうる. クラウドを利用する際は,使い終わった計算資源はすぐに削除するなど,利用者が賢く管理を行う必要があり,これを怠ると思もしない額の請求が届く可能性がある.

2. セキュリティ

クラウドは,インターネットを通じて世界のどこからでもアクセスできる状態にあり,セキュリティ管理を怠ると簡単にハッキングの対象となりうる. ハッキングを受けると,情報流出だけでなく,経済的な損失を被る可能性がある.

3. ラーニングカーブ

上記のように,コスト・セキュリティなど,クラウドを利用する際に留意しなければならない点は多い. 賢くクラウドを使うには,十分なクラウドの理解が必要であり,そのラーニングカーブを乗り越える必要がある.

2.3. どうやってクラウドを使うのか?

大学や研究機関では,その機関の構成員向けの大規模計算機サーバーが運用されていることが多い. このような,特定の組織・団体の内部のみで使用されるクラウドを,プライベートクラウド (private cloud) と呼ぶ.

一方,商用のサービスとしてのクラウドも,現在は多くの企業から提供されている. このような,一般の顧客に向けたクラウドサービスのことを,パブリッククラウド (public cloud) と呼ぶ. 有名なクラウドプラットフォームの例を挙げると,Google社が提供する [Google Cloud Platform \(GCP\)](#), Microsoft 社が提供する [Azure](#), Amazon 社が提供する [Amazon Web Service \(AWS\)](#) などがある.

プライベートクラウドは,組織の構成員ならば無料もしくは極めて割安のコストで計算を実行できる. しかし,使用できる計算資源の量は,研究提案の申請により決定される場合が多く,柔軟性に欠ける場合もある. パブリッククラウドを利用する場合は,プロバイダーの設定した利用料金を支払うことになるが,計算リソースはほとんど上限なく使用することが可能である.

コラム: Terminal の語源

Mac/Linuxなどでコマンドを入力するときに使用する、あの黒い画面のことを Terminal と呼んだりする。この言葉の語源をご存知だろうか？



Macintosh HD — top — 80x24

```
Processes: 210 total, 2 running, 9 stuck, 199 sleeping, 901 threads 23:30:03
Load Avg: 1.48, 1.75, 1.00 CPU usage: 4.15% user, 4.40% sys, 91.44% idle
SharedLibs: 1648K resident, 0B data, 0B linkedit.
MemRegions: 31278 total, 1892H resident, 117H private, 564M shared.
PhysMem: 5893M used (1191M wired), 18G unused.
VM: 523G vsize, 1026M framework vsize, 0(0) swapins, 0(0) swapouts.
Networks: packets: 12105/8925K in, 11907/1964K out.
Disks: 80156/2205M read, 21235/425M written.
```

PID	COMMAND	%CPU	TIME	#TH	#WQ	#PORT	MEM	PURG	CMPR	PGRP	PPID
592	screenCaptur	0.0	00:00:02	7	5	55+	1952K+ 20K+	0B	262	262	
590	mdworker	0.0	00:00:01	3	0	44	2032K	0B	0B	590	1
589	mdworker	0.0	00:00:01	3	0	44	1572K	0B	0B	589	1
588	top	1.7	00:00:51	1/1	0	22+	2860K	0B	0B	588	584
584	bash	0.0	00:00:00	1	0	15	588K	0B	0B	584	583
583	login	0.0	00:00:01	3	1	28	1228K	0B	0B	583	482
574	audited	0.0	00:00:00	2	0	25	560K	0B	0B	574	1
567	System Prefe	0.0	00:03:23	3	0	270	39M	8364K	0B	567	1
561	systemstatsd	0.0	00:00:01	2	1	19	1048K	0B	0B	561	1
560	com.apple.We	0.0	00:01:42	9	0	229	25M	0B	0B	560	1
558	com.apple.We	0.0	00:05:07	15	3	224	151M	1716K	0B	558	1
555	bash	0.0	00:00:00	1	0	15	604K	0B	0B	555	554
554	login	0.0	00:00:01	3	1	28	1176K	0B	0B	554	482
550	bash	0.0	00:00:00	1	0	15	608K	0B	0B	550	549

この言葉の語源は、コンピュータが誕生して間もない頃の時代に遡る。その頃のコンピュータというと、何千何万のという数の真空管が接続された、会議室一個分くらいのサイズのマシンであった。そのような高価でメンテが大変な機材であるから、当然みんなでシェアして使うことが前提となる。ユーザーがコンピュータにアクセスするため、マシンからは何本かのケーブルが伸び、それぞれにキーボードとスクリーンが接続されていた…これを **Terminal** と呼んでいたのである。人々は、代わる代わる Terminal の前に座って、計算機との対話をっていた。

時代は流れ、WindowsやMacなどのいわゆるパーソナルコンピュータの出現により、コンピュータはみんなで共有するものではなく、個人が所有するものになった。

最近のクラウドの台頭は、みんなで大きなコンピュータをシェアするという、最初のコンピュータの使われ方に原点回帰していると捉えることもできる。一方で、スマートフォンやウェアラブルなどのエッジデバイスの普及も盛んであり、個人が複数の"小さな"コンピュータを所有する、という流れも同時に進行しているのである。

Chapter 3. AWS入門

3.1. AWSとは？

本書では、クラウドの実践を行うプラットフォームとして、AWSを用いる。実践にあたって、最低限必要なAWSの知識を本章では解説しよう。

AWS (Amazon Web Service) はAmazon社が提供する総合的なクラウドプラットフォームである。AWSはAmazon社が持つ膨大な計算リソースを貸し出すクラウドサービスとして、2006年に誕生した。2018年では、クラウドプロバイダーとして最大のマーケットシェア(約33%)を保持している(参照)。NetflixやSlackをはじめとした多くのウェブ関連のサービスで、一部または全てのサーバーリソースがAWSから提供されているとのことである。よって、知らないうちにAWSの恩恵にあずかっている人も少なくないはずだ。

最大のシェアをもつだけに、とても幅広い機能・サービスが提供されており、科学・エンジニアリングの研究用途としても頻繁に用いられるようになってきている。

3.2. AWSの機能・サービス

Figure 3は、執筆時点においてAWSで提供されている主要な機能・サービスの一覧である。

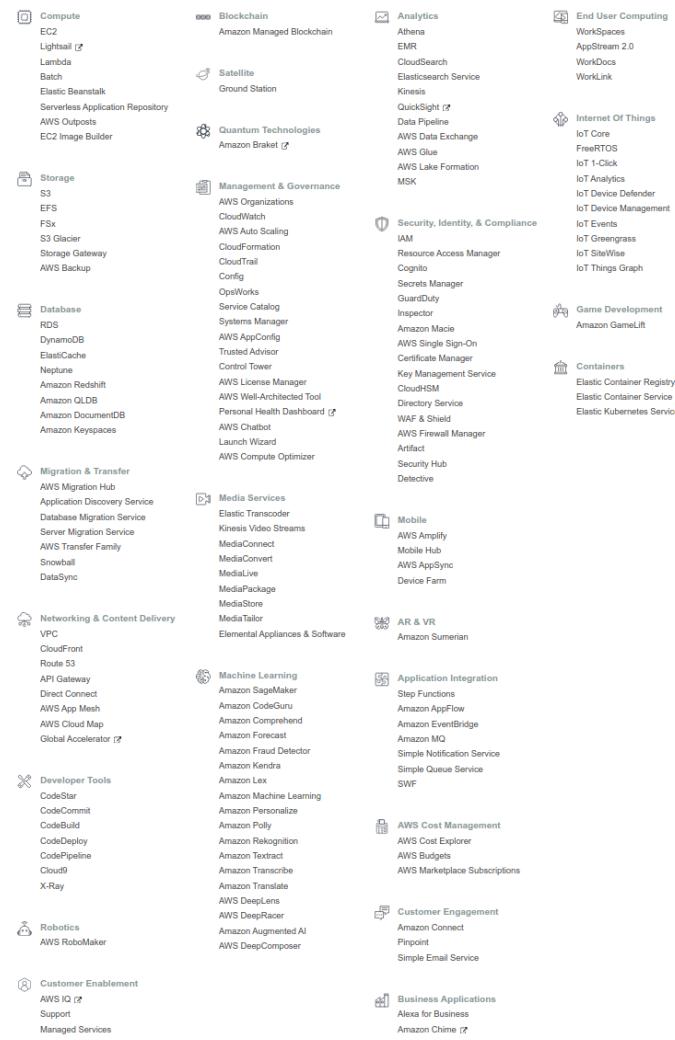


Figure 3. AWSで提供されている主要なサービス一覧

計算、ストレージ、データベース、ネットワーク、セキュリティなど、クラウドの構築に必要な様々な要素が独立したコンポーネントとして提供されている。基本的に、これらを組み合わせることでひとつのクラウドシステムができる。

また,機械学習・音声認識・AR/VRなど,特定のアプリケーションにパッケージ済みのサービスも提供されている.これらを合計すると全部で170個以上のサービスが提供されているとのことである([参照](#)).

AWS の初心者は,この**大量のサービスの数に圧倒され,どこから手をつけたらよいのかわからなくなる**,という状況に陥りがちである. だが実のところ,基本的な構成要素はそのうちの数個のみに限られる. 他の機能の多くは,基本の要素を組み合わせ特定のアプリケーションに特化したパッケージとして AWS が用意したものである. したがって,基本要素となる機能の使い方を知れば,AWSのおおよそのリソースを使いこなすことが可能になる.

3.3. AWSでクラウドを作るときの基本となる部品

以下では AWS 上で計算システムを構築するときの基本となる構成要素を列挙する. これらは後のハンズオンで実際にプログラムを書きながら体験する. 現時点では,名前だけでも頭の片隅に記憶してもらえばよい.

3.3.1. 計算



EC2 (Elastic Compute Cloud) 様々なスペックの仮想マシンを作成し,計算を実行することができる. クラウドの最も基本となる構成要素である. [Chapter 4](#), [Chapter 6](#), [Chapter 9](#) で詳しく触れる.



Lambda Function as a Service (FaaS) と呼ばれる,小さな計算をサーバーなしで実行するためのサービス. Serverless architecutre の章 ([Chapter 11](#)) で詳しく解説する.

3.3.2. ストレージ



EBS (Elastic Block Store) EC2に付与することのできる仮想データドライブ. いわゆる"普通の"(一般的なOSで使われている)ファイルシステムを思い浮かべてくれたらしい.



S3 (Simple Storage Service) Object Storage と呼ばれる,APIを使ってデータの読み書きを行う,いうなれば"クラウド・ネイティブ"なデータの格納システムである. Serverless architecutre の章 ([Chapter 11](#)) で詳しく解説する.

3.3.3. データベース



DynamoDB NoSQL 型のデータベースサービス(知っている人は [mongoDB](#)などを思い浮かべたらよい). Serverless architecutre の章 ([Chapter 11](#)) で詳しく解説する.

3.3.4. ネットワーク



VPC(Virtual Private Cloud) AWS 上に仮想ネットワーク環境を作成し,仮想サーバー間の接続を定義したり,外部からのアクセスなどを管理する. EC2 は VPC の内部に配置されなければならない.

3.4. Region と Availability Zone

AWS を使用する際に知っておかなければならぬ重要な概念として, **Region** と **Availability Zone (AZ)** がある([Figure 4](#)). 以下ではこの概念について簡単に記述する.



Figure 4. AWSにおける Region と Availability Zones

Regionとは、おおまかに言うとデータセンターの所在地のことである。執筆時点において、AWSは世界の25の国と地域でデータセンターを所有している。Figure 5は執筆時点で利用できるRegionの世界地図を示している。日本では東京と大阪にデータセンターがある。各Regionには固有のIDがついており、例えば東京は **ap-northeast-1**、米国オハイオ州は **us-east-2**、などと定義されている。



Figure 5. Regions in AWS(出典: <https://aws.amazon.com/about-aws/global-infrastructure/>)

AWSコンソールにログインすると、画面右上のメニューバーでリージョンを選択することができる(Figure 6、赤丸で囲った箇所)。EC2、S3などのAWSのリソースは、リージョンごとに完全に独立である。したがって、リソースを新たにデプロイする時、あるいはデプロイ済みのリソースを閲覧するときは、コンソールのリージョンが正しく設定されているか、確認する必要がある。ウェブビジネスを展開する場合は、世界の各地にクラウドを展開する必要があるが、個人的な研究用途として用いる場合は、最寄りのリージョン(i.e. 東京)を使えば基本的に問題ない。

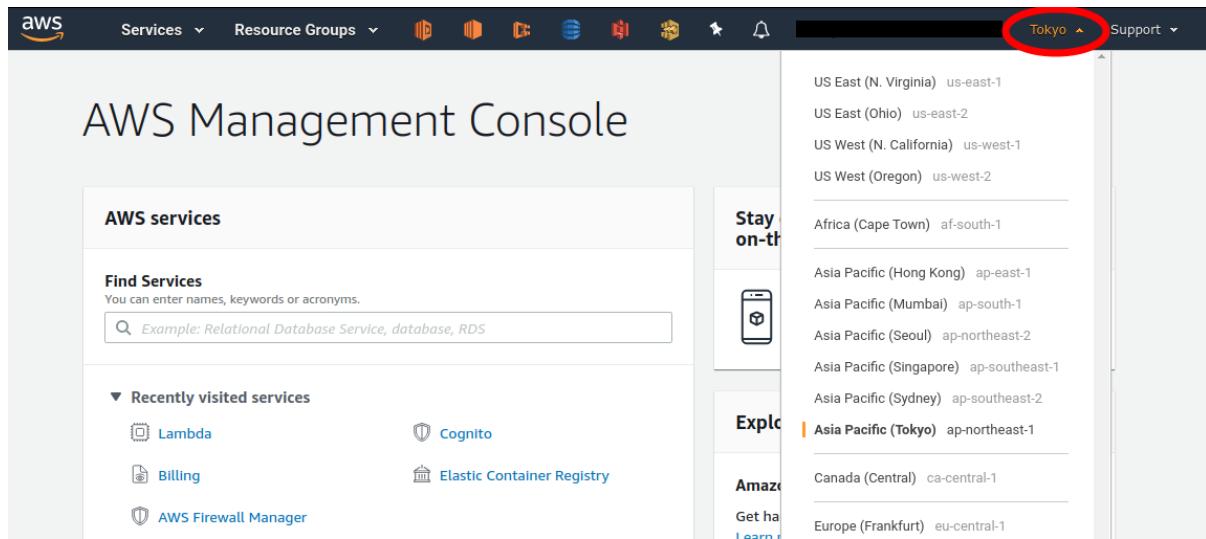


Figure 6. AWSコンソールでリージョンを選択

Availability Zone (AZ) とは、Region 内で地理的に隔離されたデータセンターのことである。それぞれのリージョンは2個以上のAZを有しており、もしひつAZで火災や停電などが起きた場合でも、他のAZがその障害をカバーすることができる。また、AZ 間は高速な AWS 専用ネットワーク回線で結ばれているため、AZ 間のデータ転送は極めて早い。AZ は、ビジネスなどでサーバーダウンが許容されない場合などに注意すべき概念であり、個人的な用途で使う限りにおいてはあまり深く考慮する必要はない。言葉の意味だけ知っておけば十分である。



AWS を使用するとき、どこの region を指定するのがよいのだろうか？インターネットの接続速度などの観点からは、地理的に一番近い region を使用するのが一般的によいだろう。一方、EC2 の利用料などは region ごとに価格設定が若干（10-20%程度）異なる。従って、自分が最も頻繁に利用するサービスの価格が最も安く設定されているリージョンを選択する、というのも重要な視点である。また、いくつかのサービスは、特定の region で利用できない場合もある。これらのポイントから総合的に判断して使用する region を決めるのが良い。

AWS Educate を利用している読者へ



執筆時点において、AWS Educate による Starter Account を使用している場合は **us-east-1** region のみ利用できる（[参照](#)）。

Further reading

- [AWS documentation "Regions, Availability Zones, and Local Zones"](#)

3.5. AWSでのクラウド開発

AWS のクラウドの全体像がわかつたところで、次のトピックとして、どのようにして AWS 上にクラウドの開発を行い、展開していくかについての概略を解説をしよう。

AWS のリソースを追加・編集・削除などの操作を実行するには、**コンソールを用いる方法**と、**API を用いる方法**の、二つの経路がある。

3.5.1. コンソール画面からリソースを操作する

AWS のアカウントにログインすると、まず最初に表示されるのが **AWS コンソール** である（Figure 7）。



Figure 7. AWSマネージメントコンソール画面

コンソールをすることで、EC2 のインスタンスを立ち上げたり、S3のデータを追加・削除したり、ログを閲覧したりなど、AWS上のあらゆるリソースの操作を GUI (Graphical User Interface) を通して実行することができる。初めて触る機能をポチポチと試したり、デバッグを行うときなどにとても便利である。

コンソールはさらっと機能を試したり、開発中のクラウドのデバッグをするときには便利なのであるが、実際にクラウドの開発をする場面でこれを直接いじることはあまりない。むしろ、次に紹介する API を使用して、プログラムとしてクラウドのリソースを記述することで開発を行うのが一般的である。そのような理由で、本書では AWS コンソールを使った AWS の使い方はあまり触れない。AWS のドキュメンテーションには、たくさんの [チュートリアル](#) が用意されており、コンソール画面から様々な操作を行う方法が記述されているので、興味がある読者はそちらを参照されたい。

3.5.2. APIからリソースを操作する

API (Application Programming Interface) を使うことで、コマンドを AWS に送信し、クラウドのリソースの操作をすることができる。API とは、端的に言えば AWS が公開しているコマンドの一覧であり、[GET](#), [POST](#), [DELETE](#) などの REST API から構成されている (REST API については [Section 10.2](#) で簡単に解説する)。が、直接 REST API を入力するのは面倒であるので、その手間を解消するための様々なツールが提供されている。

[AWS CLI](#) は、UNIX コンソールから AWS API を実行するための CLI (Command Line Interface) である。

CLIに加えて、いろいろなプログラミング言語での SDK (Software Development Kit) が提供されている。以下に一例を挙げる。

- Python ⇒ [boto3](#)
- Ruby ⇒ [AWS SDK for Ruby](#)
- node.js ⇒ [AWS SDK for Node.js](#)

具体的な API の使用例を見てみよう。

S3に新しい保存領域 (**Bucket (バケット)** と呼ばれる) を追加したいとしよう。AWS CLI を使った場合は、以下の ようなコマンドを打てばよい。

```
$ aws s3 mb s3://my-bucket --region ap-northeast-1
```

上記のコマンドは、`my-bucket` という名前のバケットを、`ap-northeast-1` のregionに作成する。

Pythonから上記と同じ操作を実行するには、`boto3` ライブラリを使って、以下のようなスクリプトを実行する。

```
1 import boto3
2
3 s3_client = boto3.client("s3", region_name="ap-northeast-1")
4 s3_client.create_bucket(Bucket="my-bucket")
```

もう一つ例をあげよう。

新しいEC2のインスタンス(インスタンスとは、起動状態にある仮想サーバーの意味である)を起動するには、以下のようないコマンドを打てば良い。

```
$ aws ec2 run-instances --image-id ami-xxxxxxxx --count 1 --instance-type t2.micro --key-name MyKeyPair
--security-group-ids sg-903004f8 --subnet-id subnet-6e7f829e
```

上記のコマンドにより、`t2.micro` というタイプ (1 vCPU, 1.0 GB RAM) のインスタンスが起動する。ここではその他のパラメータの詳細の説明は省略する (ハンズオン ([Chapter 4](#)) で詳しく解説する)。

Pythonから上記と同じ操作を実行するには、以下のようなスクリプトを使う。

```
1 import boto3
2
3 ec2_client = boto3.client("ec2")
4 ec2_client.run_instances(
5     ImageId="ami-xxxxxxxx",
6     MinCount=1,
7     MaxCount=1,
8     KeyName="MyKeyPair",
9     InstanceType="t2.micro",
10    SecurityGroupIds=["sg-903004f8"],
11    SubnetId="subnet-6e7f829e",
12 )
```

以上の例を通じて、APIによるクラウドのリソースの操作のイメージがつかめてきただろうか？コマンド一つで、新しい仮想サーバーを起動したり、データの保存領域を追加したり、任意の操作を実行することができるわけである。基本的に、このようなコマンドを複数組み合わせていくことで、自分の望むCPU・RAM・ネットワーク・ストレージが備わった計算環境を構築することができる。もちろん、逆の操作 (リソースの削除) も API を使って実行できる。

3.5.3. ミニ・ハンズオン: AWS CLI を使ってみよう

ここでは、ミニ・ハンズオンとして、AWS CLI を実際に使ってみる。AWS CLI は先述の通り、AWS 上の任意のリソースの操作が可能であるが、ここでは一番シンプルな、[S3 を使ったファイルの読み書きを実践する](#) (EC2の操作は少し複雑なので、第一回ハンズオンで行う)。aws s3 コマンドの詳しい使い方は [公式ドキュメンテーション](#) を参照。



AWS CLI のインストールについては、[Section 15.2](#) を参照。



以下に紹介するハンズオンは、基本的に [S3 の無料枠](#) の範囲内で実行することができる。



以下のコマンドを実行する前に,AWSの認証情報が正しく設定されていることを確認する. これには`~/.aws/credentials`のファイルに設定が書き込まれているか,環境変数(`AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, `AWS_DEFAULT_REGION`)が定義されている必要がある. 詳しくは[Section 15.2](#)を参照.

まず最初に,S3にデータの格納領域(Bucketと呼ばれる.一般的なOSでの"ドライブ"に相当する)を作成するところから始めよう.

```
$ bucketName=$(openssl rand -hex 12)"  
$ echo $bucketName  
$ aws s3 mb "s3://${bucketName}"
```

S3のバケットの名前は,AWS全体で一意的でなければならないことから,上ではランダムな文字列を含んだバケットの名前を生成し,`bucketName`という変数に格納している. そして,`aws s3 mb`(`mb`はmake bucketの略)によって,新しいバケットを作成する.

次に,バケットの一覧を取得してみよう.

```
$ aws s3 ls  
  
2020-06-07 23:45:44 mybucket-c6f93855550a72b5b66f5efe
```

先ほど作成したバケットがリストにあることを確認できる.



本書のノーテーションとして,コマンドラインに入力するコマンドは,それがコマンドであると明示する目的で先頭に`$`がついている. `$`はコマンドをコピー&ペーストするときは除かなければならない. 逆に,コマンドの出力は`$`なしで表示されている.

次に,バケットにファイルをアップロードする.

```
$ echo "Hello world!" > hello_world.txt  
$ aws s3 cp hello_world.txt "s3://${bucketName}/hello_world.txt"
```

上では`hello_world.txt`というダミーのファイルを作成して,それをアップロードした.

それでは,バケットの中にあるファイルの一覧を取得してみる.

```
$ aws s3 ls "s3://${bucketName}" --human-readable  
  
2020-06-07 23:54:19    13 Bytes hello_world.txt
```

先ほどアップロードしたファイルがたしかに存在することがわかる.

最後に,使い終わったバケットを削除する.

```
$ aws s3 rb "s3://${bucketName}" --force
```

`rb`はremove bucketの略である. デフォルトでは,バケットの中にファイルが存在すると削除できない. 空でないバケットを強制的に削除するには`--force`のオプションを付ける.

以上のように,AWS CLIを使ってS3バケットに対しての一連の操作を実行することができた. EC2やLambda,DynamoDBなどについても同様にAWS CLIを使ってあらゆる操作を実行することができる.

3.6. CloudFormation と AWS CDK

3.6.1. CloudFormation による Infrastructure as Code (IaC)

前節で述べたように,AWS API を使うことでクラウドのあらゆるリソースの作成・管理が可能である. よって,原理上は, API のコマンドを組み合わせていくことで,自分の作りたいクラウドを設計することができる.

しかし,ここで実用上考慮しなければならない点がひとつある. AWS API には大きく分けて,**リソースを操作するコマンド**と,**タスクを実行するコマンド**があることである (Figure 8).

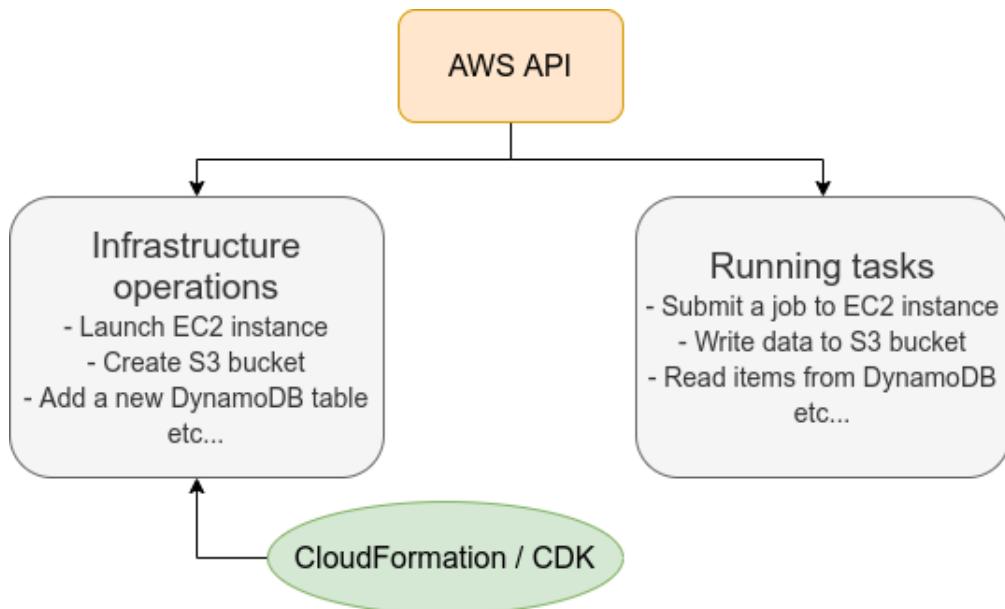


Figure 8. AWS APIはリソースを操作するコマンドとタスクを実行するコマンドに大きく分けられる.リソースを記述・管理するのに使われるのが、CloudFormation と CDK である。

リソースを操作するとは,EC2のインスタンスを起動したり,S3のバケットを作成したり,データベースに新たなテーブルを追加する,などの**静的なリソースを準備する**操作を指す. "ハコ"を作る操作と呼んでもよいだろう. このようなコマンドは,**クラウドのデプロイ時にのみ,一度だけ実行されればよい**.

タスクを実行するコマンドとは, EC2 のインスタンスにジョブを投入したり, S3 のバケットにデータを読み書きするなどの操作を指す. これは, EC2 や S3 などのリソース ("ハコ") を前提として,その内部で実行されるべき計算を記述するものである. 前者に比べてこちらは**動的な操作**を担当する,と捉えることもできる.

そのような観点から, **インフラを記述するプログラム**と**タスクを実行するプログラム**はある程度分けて管理されるべきである. クラウドの開発は, クラウドの(静的な)リソースを記述するプログラムを作成するステップと, インフラ上で動く動的な操作を行うプログラムを作成するステップの,二段階に分けて考えることができる.

AWSでの静的リソースを管理するための仕組みが, [CloudFormation](#) である. CloudFormation とは, CloudFormation の文法に従ったテキストファイルを使って,AWSのインフラを記述する仕組みである. CloudFormation を使って, 例えば, EC2のインスタンスをどれくらいのスペックで,何個起動するか,インスタンス間はどのようなネットワークで結び,どのようなアクセス権限を付与するか,などのリソースの要件を逐次的に記述することができる. 一度CloudFormation ファイルが出来上がれば,それにしたがったクラウドシステムをコマンド一つで AWS 上に展開することができる. また, CloudFormation ファイルを交換することで,全く同一のクラウド環境を他者が簡単に再現することも可能になる. このように,本来は物理的な実体のあるハードウェアを,プログラムによって記述し,管理するという考え方を, **Infrastructure as Code (IaC)**と呼ぶ.

CloudFormation を記述するには,基本的に **JSON** (JavaScript Object Notation) と呼ばれるフォーマットを使う.以下は,JSONで記述された CloudFormation ファイルの一例 (抜粋) である.

```

1 "Resources" : {
2   ...
3   "WebServer": {
4     "Type" : "AWS::EC2::Instance",
5     "Properties": {
6       "ImageId" : { "Fn::FindInMap" : [ "AWSRegionArch2AMI", { "Ref" : "AWS::Region" },
7                               { "Fn::FindInMap" : [ "AWSInstanceType2Arch", { "Ref" : "InstanceType" },
8                               "Arch" ] } ] },
9       "InstanceType" : { "Ref" : "InstanceType" },
10      "SecurityGroups" : [ {"Ref" : "WebServerSecurityGroup"} ],
11      "KeyName" : { "Ref" : "KeyName" },
12      "UserData" : { "Fn::Base64" : { "Fn::Join" : [ "", [
13        "#!/bin/bash -xe\n",
14        "yum update -y aws-cfn-bootstrap\n",
15        "/opt/aws/bin/cfn-init -v ",
16        "    --stack ", { "Ref" : "AWS::StackName" },
17        "    --resource WebServer ",
18        "    --configsets wordpress_install ",
19        "    --region ", { "Ref" : "AWS::Region" }, "\n",
20        "/opt/aws/bin/cfn-signal -e $? ",
21        "    --stack ", { "Ref" : "AWS::StackName" },
22        "    --resource WebServer ",
23        "    --region ", { "Ref" : "AWS::Region" }, "\n"
24      ]]]} }
25   },
26   ...
27 },
28 },
29 ...
30 },

```

ここでは、"WebServer" という名前のつけられた EC2 インスタンスを定義している。かなり長大で複雑な記述であるが、これによって所望のスペック・OSをもつEC2インスタンスを自動的に生成することが可能になる。

3.6.2. AWS CDK

前節で紹介した CloudFormation は、見てわかるとおり大変記述が複雑であり、またそれのどれか一つにでも誤りがあつてはいけない。また、基本的に"テキスト"を書いていくことになるので、プログラミング言語で使うような変数やクラスといった便利な概念が使えない（厳密には、CloudFormation にも変数に相当するような機能は存在する）。また、記述の多くの部分は繰り返しが多く、自動化できる部分も多い。

そのような悩みを解決してくれるのが、[AWS Cloud Development Kit \(CDK\)](#) である。CDKは Python などのプログラミング言語を使って CloudFormation を自動的に生成してくれるツールである。CDKは2019年にリリースされたばかりの比較的新しいツールで、日々改良が進められている ([GitHub レポジトリ](#) のリリースを見ればその開発のスピードの速さがわかるだろう)。CDKは TypeScript (JavaScript), Python, Java など複数の言語でサポートされている。

CDKを使うことで、CloudFormation に相当するクラウドリソースの記述を、より親しみのあるプログラミング言語を使って行うことができる。かつ、典型的なリソース操作に関してはパラメータの多くの部分を自動で決定してくれる所以、記述しなければならない量もかなり削減される。

以下に Python を使った CDK のコードの一例（抜粋）を示す。

```

1 from aws_cdk import (
2     core,
3     aws_ec2 as ec2,
4 )
5
6 class MyFirstEc2(core.Stack):
7
8     def __init__(self, scope, name, **kwargs):
9         super().__init__(scope, name, **kwargs)
10
11     vpc = ec2.Vpc(
12         ... # some parameters
13     )
14
15     sg = ec2.SecurityGroup(
16         ... # some parameters
17     )
18
19     host = ec2.Instance(
20         self, "MyGreatEc2",
21         instance_type=ec2.InstanceType("t2.micro"),
22         machine_image=ec2.MachineImage.latest_amazon_linux(),
23         vpc=vpc,
24         ...
25     )

```

上記のコードは、一つ前に示した JSON を使った CloudFormation と実質的に同じことを記述している。とても煩雑だった CloudFormation ファイルに比べて、CDK と Python を使うことで格段に短く、わかりやすく記述できることができるのがわかるだろう。

本書の主題は、**CDK を使って、コードを書きながら AWS の概念や開発方法を学んでいくことである**。後の章では CDK を使って様々なハンズオンを実施していく。

早速、最初のハンズオンでは、CDK を使って EC2 インスタンスを作成する方法を学んでいこう。

Further reading

- [AWS CDK Examples](#): CDKを使ったプロジェクトの例が多数紹介されている。ここにある例をテンプレートに自分のアプリケーションの開発を進めるとよい。

Chapter 4. Hands-on #1: 初めてのEC2インスタンスを起動する

ハンズオンの第一回では、CDK を使って EC2 のインスタンス(仮想サーバー)を作成し、SSHでサーバーにログインする、という演習を行う。このハンズオンを終えれば、あなたは自分だけのサーバーをAWS上に立ち上げ、自由に計算を走らせることができるようになるのである！

ハンズオンのソースコードは [こちらのリンク](#) に置いてある。



ハンズオン1は、基本的に AWS の無料枠 の範囲内で実行することができる。

4.1. 準備

まずは、ハンズオンを実行するための環境を整える。これらの環境整備は、後のハンズオンでも前提となるものなので確実にミスなく行っていただきたい。

4.1.1. AWS Account

ハンズオンを実行するには個人のAWSアカウントが必要である。AWSアカウントの取得については [Section 1.3](#) 参照。

4.1.2. Python と Node.js

本ハンズオンを実行するには、Python (3.6 以上), Node.js (10.3.0 以上) がインストールされていなければならぬ。

4.1.3. AWS CLI

AWS CLI のインストールについては、[Section 15.2](#) を参照。

4.1.4. AWS CDK

AWS CDK のインストールについては、[Section 15.3](#) を参照。

4.1.5. ソースコードのダウンロード

本ハンズオンで使用するプログラムのソースコードを、以下のコマンドを使って GitLab からダウンロードする。

```
$ git clone https://github.com/tomomano/learn-aws-by-coding.git
```

あるいは、<https://github.com/tomomano/learn-aws-by-coding> のページに行って、右上のダウンロードボタンからダウンロードすることもできる。

4.1.6. Docker を使用する場合

Python, Node.js, AWS CDK など、ハンズオンのプログラムを実行するために必要なプログラム/ライブラリがインストール済みの Docker image を用意した。また、ハンズオンのソースコードもパッケージ済みである。Docker の使い方を知っている読者は、これを使えば、諸々のインストールをする必要なく、すぐにハンズオンのプログラムを実行できる。

使用方法については [Section 15.5](#) を参照のこと。

4.2. SSH

SSH ([secure shell](#)) は Unix 系のリモートサーバーに安全にアクセスするためのツールである。本ハンズオンでは、SSH を使って仮想サーバーにアクセスする。SSH に慣れていない読者のため、簡単な説明をここで行う。

SSH による通信はすべて暗号化されているので、機密情報をインターネットを介して安全に送受信することができる。本ハンズオンで、リモートのサーバーにアクセスするための SSH クライアントがローカルマシンにインストールされている必要がある。SSH クライアントは Linux/Mac には標準搭載されている。Windows の場合は WSL をインストールすることで SSH クライアントを利用することを推奨する ([Section 1.4](#) を参照)。

SSH コマンドの基本的な使い方は

```
$ ssh <user name>@<host name>
```

である。`<host name>` はアクセスする先のサーバーの IP アドレスや DNS によるホストネームが入る。`<user name>` は接続する先のユーザー名である。

SSH は平文のパスワードによる認証を行ふこともできるが、より強固なセキュリティを施すため、**公開鍵暗号方式 (Public Key Cryptography)**による認証を行うことが強く推奨されており、EC2 はこの方法でしかアクセスを許していない。公開鍵暗号方式の仕組みについては各自勉強してほしい。本ハンズオンにおいて大事なことは、**EC2 インスタンスが公開鍵(Public key)を保持し、クライアントとなるコンピューター(あなたの自身のコンピュータ)が秘密鍵(Private key)を保持する**、という点である。EC2 のインスタンスには秘密鍵を持ったコンピュータのみがアクセスすることができる。逆に言うと、秘密鍵が漏洩すると第三者もサーバーにアクセスできることになるので、**秘密鍵は絶対に漏洩することのないよう注意して管理する**。

SSH コマンドでは、ログインのために使用する秘密鍵ファイルを `-i` もしくは `--identity_file` のオプションで指定することができる。例えば、

```
$ ssh -i Ec2SecretKey.pem <user name>@<host name>
```

のように使う。

4.3. アプリケーションの説明

このハンズオンで作成するアプリケーションの概要を [Figure 9](#) に示す。

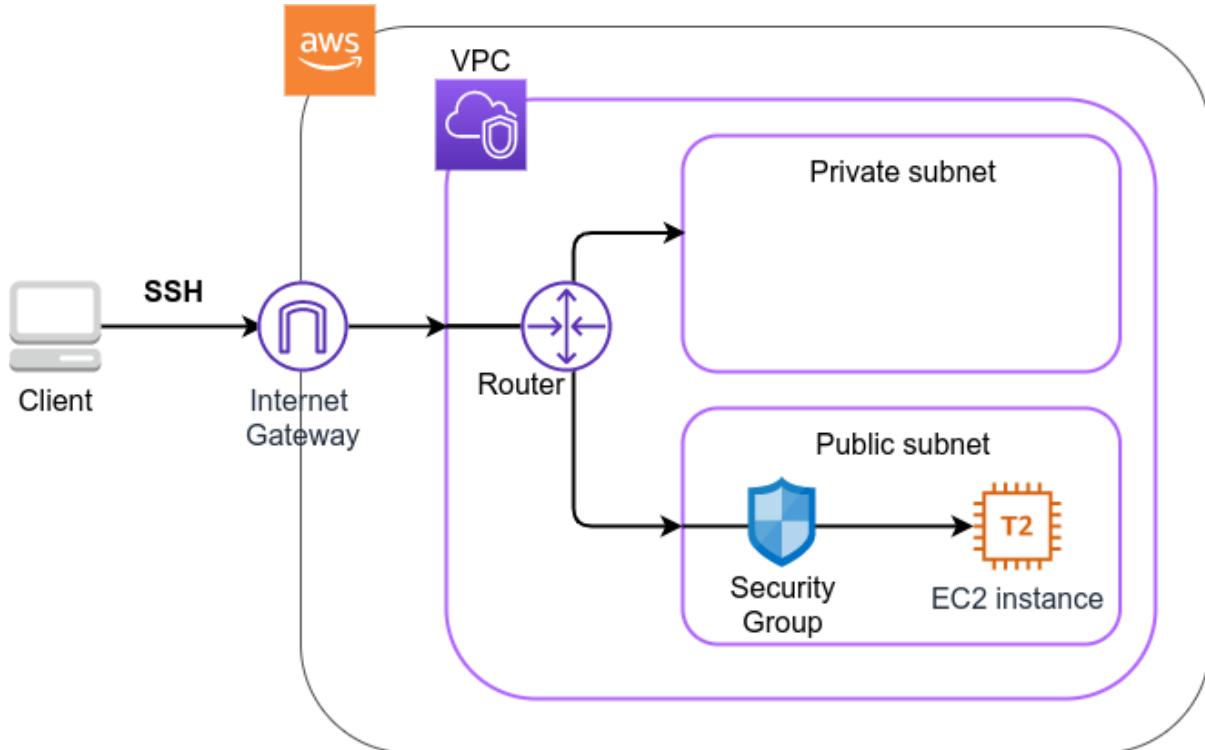


Figure 9. ハンズオン#1で作製するアプリケーションのアーキテクチャ

このアプリケーションではまず、**VPC (Virtual Private Cloud)** を使ってプライベートな仮想ネットワーク環境を立ち上げている。そのVPCの public subnet の内側に、**EC2 (Elastic Compute Cloud)** の仮想サーバーを配置する。さらに、セキュリティのため、**Security Group** によるEC2インスタンスへのアクセス制限を設定している。このようにして作成された仮想サーバーに、SSHを使ってアクセスし、簡単な計算を行う。

上記のようなアプリケーションを、CDKを使って構築する。

早速ではあるが、今回のハンズオンで使用するプログラムを見てみよう ([handson/ec2-get-started/app.py](#))。

```

1 class MyFirstEc2(core.Stack):
2
3     def __init__(self, scope: core.App, name: str, key_name: str, **kwargs) -> None:
4         super().__init__(scope, name, **kwargs)
5
6         ①
7         vpc = ec2.Vpc(
8             self, "MyFirstEc2-Vpc",
9             max_azs=1,
10            cidr="10.10.0.0/23",
11            subnet_configuration=[
12                ec2.SubnetConfiguration(
13                    name="public",
14                    subnet_type=ec2.SubnetType.PUBLIC,
15                )
16            ],
17            nat_gateways=0,
18        )
19
20         ②
21         sg = ec2.SecurityGroup(
22             self, "MyFirstEc2Vpc-Sg",
23             vpc=vpc,
24             allow_all_outbound=True,
25         )
26         sg.add_ingress_rule(
27             peer=ec2.Peer.any_ipv4(),
28             connection=ec2.Port.tcp(22),
29         )
30
31         ③
32         host = ec2.Instance(
33             self, "MyFirstEc2Instance",
34             instance_type=ec2.InstanceType("t2.micro"),
35             machine_image=ec2.MachineImage.latest_amazon_linux(),
36             vpc=vpc,
37             vpc_subnets=ec2.SubnetSelection(subnet_type=ec2.SubnetType.PUBLIC),
38             security_group=sg,
39             key_name=key_name
40         )

```

① まず最初に、VPCを定義する。

② 次に、security group (SG) を定義している。ここでは、任意のIPv4のアドレスからの、ポート22 (SSHの接続に使用される)への接続を許可している。それ以外の接続は拒絶される。

③ 最後に、上記で作った VPC と SG が付与された EC2 インスタンスを作成している。インスタンスタイプは **t2.micro** を選択し、**Amazon Linux** をOSとして設定している。

それについて、もう少し詳しく説明しよう。

4.3.1. VPC (Virtual Private Cloud)



VPC は AWS 上にプライベートな仮想ネットワーク環境を構築するツールである。高度な計算システムを構築する

には、複数のサーバーを連動させて計算を行う必要があるが、そのような場合に互いのアドレスなどを管理する必要があり、そのような場合にVPCは有用である。

本ハンズオンでは、サーバーは一つしか起動しないので、VPCの恩恵はよく分からないかもしれない。しかし、EC2インスタンスは必ずVPCの中に配置されなければならない、という制約があるので、このハンズオンでもミニマルなVPCを構成している。

興味のある読者のために、VPCのコードについてもう少し詳しく説明しよう。

```
1 vpc = ec2.Vpc(
2     self, "MyFirstEc2-Vpc",
3     max_azs=1,
4     cidr="10.10.0.0/23",
5     subnet_configuration=[
6         ec2.SubnetConfiguration(
7             name="public",
8             subnet_type=ec2.SubnetType.PUBLIC,
9         )
10    ],
11    nat_gateways=0,
12 )
```



- **max_azs=1** : このパラメータは、前章で説明した availability zone (AZ) を設定している。このハンズオンでは、特にデータセンターの障害などを気にする必要はないので **1** にしている。
- **cidr="10.10.0.0/23"** : このパラメーターは、VPC内のIPv4のレンジを指定している。CIDR記法については、[Wikipedia](#)などを参照。**10.10.0.0/23** は **10.10.0.0** から **10.10.1.255** までの512個の連続したアドレス範囲を指している。つまり、このVPCでは最大で512個のユニークなIPv4アドレスが使えることになる。今回はサーバーは一つなので512個は明らかに多すぎるが、VPCはアドレスの数はどれだけ作成しても無料なので、多めに作成した。
- **subnet_configuration=…** : このパラメータは、VPCにどのようなサブネットを作るか、を決めている。サブネットの種類には **private subnet** と **public subnet** の二種類がある。**private subnet** は基本的にインターネットとは遮断されたサブネット環境である。インターネットと繋がっていないので、セキュリティは極めて高く、VPC内のサーバーとのみ通信を行えばよい EC2 インスタンスはここに配置する。**Public subnet** とはインターネットに繋がったサブネットである。本ハンズオンで作成するサーバーは、外からSSHでログインを行いたいので、**Public subnet** 内に配置する。より詳細な記述は [公式ドキュメンテーション](#) を参照。
- **natgateways=0** : これは少し高度な内容なので省略する（興味のある読者は [公式ドキュメンテーション](#) を参照）。が、これを0にしておかないと、**NAT Gateway** の利用料金が発生してしまうので、注意！

4.3.2. Security Group

Security group (SG) は、EC2 インスタンスに付与することのできる仮想ファイアウォールである。例えば、特定の IP アドレスから来た接続を許可・拒絶したり（インバウンド・トラフィックの制限）、逆に特定の IP アドレスへのアクセスを禁止したり（アウトバウンド・トラフィックの制限）することができる。

コードの該当部分を見てみよう。

```

1 sg = ec2.SecurityGroup(
2     self, "MyFirstEc2Vpc-Sg",
3     vpc=vpc,
4     allow_all_outbound=True,
5 )
6 sg.add_ingress_rule(
7     peer=ec2.Peer.any_ipv4(),
8     connection=ec2.Port.tcp(22),
9 )

```

本ハンズオンでは、SSHによる外部からの接続を許容するため、`sg.add_ingress_rule` (`peer=ec2.Peer.any_ipv4()`, `connection=ec2.Port.tcp(22)`)により、すべてのIPv4アドレスからのポート22番へのアクセスを許容している。また、SSHでEC2インスタンスにログインしたのち、インターネットからプログラムなどをダウンロードできるよう、`allow_all_outbound=True`のパラメータを設定している。



SSHはデフォルトでは22番ポートを使用するのが慣例である。



セキュリティ上の観点からは、SSHの接続は自宅や大学・職場など特定の地点からの接続のみを許す方が望ましい。

4.3.3. EC2 (Elastic Compute Cloud)



EC2はAWS上に仮想サーバーを立ち上げるサービスである。個々の起動状態にある仮想サーバーのことをインスタンス(instance)と呼ぶ(しかし、口語的なコミュニケーションにおいては、サーバーとインスタンスという言葉は相互互換的に用いられることが多い)。

EC2では用途に応じて様々なインスタンスタイプが提供されている。以下に、代表的なインスタンスタイプの例を挙げる(執筆時点での情報)。EC2のインスタンスタイプのすべてのリストは[公式ドキュメンテーション](#)で見ることができる。

Table 2. EC2 instance types

Instance	vCPU	Memory (GiB)	Network bandwidth (Gbps)	Price per hour (\$)
t2.micro	1	1	-	0.0116
t2.small	1	2	-	0.023
t2.medium	2	4	-	0.0464
c5.24xlarge	96	192	25	4.08
c5n.18xlarge	72	192	100	3.888
x1e.16xlarge	64	1952	10	13.344

このようにCPUの数は1コアから96コアまで、メモリーは1GBから3000GB以上まで、ネットワークは最大で100Gbpsまで、幅広く選択することができる。また、時間あたりの料金は、CPU・メモリーの占有数にほぼ比例する形で増加する。EC2はサーバーの起動時間を秒単位で記録しており、**利用料金は使用時間に比例する形で決定される**。例えば、`t2.medium`のインスタンスを10時間起動した場合、 $0.0464 * 10 = 0.464$ ドルの料金が発生する。



AWSには [無料利用枠](#) というものがあり, [t2.micro](#) であれば月に750時間までは無料で利用することができる.



[Table 2](#) の価格は [us-east-1](#) のものである. リージョンによって多少価格設定が異なる.



上記で [t2.micro](#) の \$0.0116 / hour という金額は, On-demand インスタンスというタイプを選択した場合の価格である. EC2 では他に, [Spot instance](#) と呼ばれるインスタンスも存在する. Spot instance は, AWSのデータセンターの負荷が増えた場合, ユーザーのプログラムが実行中であってもAWSの判断により強制シャットダウンされる, という不便さを抱えているのだが, その分大幅に安い料金設定になっている. AWS で一時的に生じた余剰な空きCPUをユーザーに割安で貸し出す, という発想である. 科学計算で, コストを削減する目的で, この Spot Instance を活用する事例も報告されている ([Wu+, 2019](#)).

EC2 インスタンスを定義しているコードの該当部分を見てみよう.

```
1 host = ec2.Instance(
2     self, "MyFirstEc2Instance",
3     instance_type=ec2.InstanceType("t2.micro"),
4     machine_image=ec2.MachineImage.latest_amazon_linux(),
5     vpc=vpc,
6     vpc_subnets=ec2.SubnetSelection(subnet_type=ec2.SubnetType.PUBLIC),
7     security_group=sg,
8     key_name=key_name
9 )
```

ここでは, [t2.micro](#) というインスタンスタイプを選択している. さらに, [machine_image](#) (OSと考えてよい) として, [Amazon Linux](#) を選択している (Machine image については, 第二回ハンズオンでより詳しく触れる). さらに, 上で定義した VPC, SG をこのインスタンスに付与している.

以上が, 今回使用するプログラムの簡単な解説であった. ミニマルな形のプログラムではあるが, 仮想サーバーを作成するのに必要なステップがおわかりいただけただろうか?

4.4. プログラムを実行する

さて, ハンズオンのコードの理解ができたところで, プログラムを実際に実行してみよう. 繰り返しになるが, [Section 4.1](#) での準備ができていることが前提である.

4.4.1. Python の依存ライブラリのインストール

まずは, Python の依存ライブラリをインストールする. 以下では, Python のライブラリを管理するツールとして, [venv](#) を使用する.

まずは, [handson/ec2-get-started](#) のディレクトリに移動しよう.

```
$ cd handson/ec2-get-started
```

ディレクトリを移動したら, [venv](#) で新しい仮想環境を作成し, インストールを実行する.

```
$ python3 -m venv .env
$ source .env/bin/activate
$ pip install -r requirements.txt
```

これで Python の環境構築は完了だ.



`venv` の簡単な説明は [Section 15.4](#) に記述してある。



環境によっては `pip` ではなく `pip3` あるいは `python3 -m pip` に置き換える必要がある。

4.4.2. AWS のシークレットキーをセットする

AWS CLI および AWS CDK を使うには、AWS のシークレットキーが設定されている必要がある。シークレットキーの発行については [Section 15.1](#) を参照のこと。シークレットキーを発行したら、[Section 15.2](#) を参照し、コマンドラインの設定を行う。

繰り返しになるがここに簡単に記すと、一つ目の方法は `AWS_ACCESS_KEY_ID` などの環境変数を設定するやり方である。もう一つの方法は、`~/.aws/credentials` に認証情報を保存しておく方式である。シークレットキーの設定は AWS CLI/CDK を使用するうえで共通のステップになるので、しっかりと理解しておくように。

4.4.3. SSH鍵を生成

EC2 インスタンスには SSH を使ってログインする。EC2 インスタンスを作成するのに先行して、今回のハンズオンで専用に使う SSH の公開鍵・秘密鍵のペアを準備する必要がある。

以下の aws-cli コマンドにより、`HirakeGoma` という名前のついた鍵を生成する。

```
$ export KEY_NAME="HirakeGoma"  
$ aws ec2 create-key-pair --key-name ${KEY_NAME} --query 'KeyMaterial' --output text > ${KEY_NAME}.pem
```

上のコマンドを実行すると、現在のディレクトリに `HirakeGoma.pem` というファイルが作成される。これが、サーバーにアクセスするための秘密鍵である。SSH でこの鍵を使うため、`~/.ssh/` のディレクトリに鍵を移動する。さらに、秘密鍵が書き換えられたり第三者に閲覧されないように、ファイルのアクセス権限を `400` に設定する。

```
$ mv HirakeGoma.pem ~/.ssh/  
$ chmod 400 ~/.ssh/HirakeGoma.pem
```

4.4.4. デプロイを実行

これまでのステップで準備は整った！

早速、アプリケーションを AWS にデプロイしてみよう。

```
$ cdk deploy -c key_name="HirakeGoma"
```

`-c key_name="HirakeGoma"` というオプションで、先程生成した `HirakeGoma` という名前の鍵を使うよう指定している。

上記のコマンドを実行すると、VPC、EC2 などが AWS 上に展開される。そして、コマンドの出力の最後に [Figure 10](#) のような出力が得られるはずである。出力の中で `InstancePublicIp` に続く数字が、起動したインスタンスのパブリック IP アドレスである。IP アドレスはデプロイごとにランダムなアドレスが割り当てられる。

```

✓ MyFirstEc2

Outputs:
MyFirstEc2.InstancePublicIp = 54.238.112.5
MyFirstEc2.InstancePublicDnsName = ec2-54-238-112-5.ap-northeast-1.compute.amazonaws.com

Stack ARN:
arn:aws:cloudformation:ap-northeast-1:606887060834:stack/MyFirstEc2/46ed0490-aa2d-

```

Figure 10. CDKデプロイ実行後の出力

4.4.5. SSH でログイン

早速,SSH で接続してみよう.

```
$ ssh -i ~/.ssh/HirakeGoma.pem ec2-user@<IP address>
```

-i オプションで,先程生成した秘密鍵を指定している. EC2 インスタンスにはデフォルトで **ec2-user** という名前のユーザーが作られているので,それを使用する. 最後に, <IP address> の部分は自分が作成したEC2インスタンスのIPアドレスで置き換える (12.345.678.9 など).

ログインに成功すると, Figure 11 のような画面が表示される. リモートのサーバーにログインしているので,プロンプトが [ec2-user@ip-10-10-1-217 ~]\$ のようになっていることを確認しよう.

```

(.env) tomoyuki@eiffel:01-ec2$ ssh -i ~/.ssh/HirakeGoma.pem ec2-user@54.238.112.5
Last login: Tue Jun  9 09:18:09 2020 from 157.82.122.171

      _\   _ /  Amazon Linux AMI
     __\_\_|_|

https://aws.amazon.com/amazon-linux-ami/2018.03-release-notes/
5 package(s) needed for security, out of 7 available
Run "sudo yum update" to apply all updates.
[ec2-user@ip-10-10-1-217 ~]$

```

Figure 11. SSH で EC2 インスタンスにログイン

おめでとう!これで,めでたくAWS上にEC2仮想サーバーを起動し,リモートからアクセスすることができるようになった!

4.4.6. 起動した EC2 インスタンスで遊んでみる

せっかく新しいインスタンスを起動したので,少し遊んでみよう.

ログインした EC2 インスタンスで,次のコマンドを実行してみよう. CPU の情報を取得することができる.

```

$ cat /proc/cpuinfo

processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 63
model name : Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz
stepping : 2
microcode : 0x43
cpu MHz : 2400.096
cache size : 30720 KB

```

次に,実行中のプロセスやメモリの消費を見てみよう.

```
$ top -n 1

top - 09:29:19 up 43 min, 1 user, load average: 0.00, 0.00, 0.00
Tasks: 76 total, 1 running, 51 sleeping, 0 stopped, 0 zombie
Cpu(s): 0.3%us, 0.3%sy, 0.1%hi, 98.9%id, 0.2%wa, 0.0%hi, 0.0%si, 0.2%st
Mem: 1009140k total, 270760k used, 738380k free, 14340k buffers
Swap: 0k total, 0k used, 0k free, 185856k cached

PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM     TIME+ COMMAND
 1 root      20   0 19696 2596 2268 S  0.0  0.3    0:01.21 init
 2 root      20   0     0     0   0 S  0.0  0.0    0:00.00 kthreadd
 3 root      20   0     0     0   0 I  0.0  0.0    0:00.00 kworker/0:0
```

t2.micro インスタンスなので、1009140k = 1GB のメモリーがあることがわかる。

今回起動したインスタンスには Python 2 はインストール済みだが、Python 3 は入っていない。Python 3.6 のインストールを行ってみよう。インストールは簡単である。

```
$ sudo yum update -y
$ sudo yum install -y python36
```

インストールした Python を起動してみよう。

```
$ python3
Python 3.6.10 (default, Feb 10 2020, 19:55:14)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-28)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Python のインターフェリタが起動した! **Ctrl + D** あるいは **exit()** と入力することで、インターフェリタを閉じることができる。

さて、サーバーでのお遊びはこんなところにしておこう（興味があれば各自いろいろと試してみると良い）。次のコマンドでログアウトする。

```
$ exit
```

4.4.7. AWS コンソールから確認

これまで、すべてコマンドラインから EC2 に関連する操作を行ってきた。EC2インスタンスの状態を確認したり、サーバーをシャットダウンなどの操作は、AWS コンソールから実行することもできる。軽くこれを紹介しよう。

まず、ウェブブラウザを開いて AWS コンソールにログインする。

ログインしたら、**Services** から **EC2** を検索(選択)する。次に、左のサイドバーの **Instances** とページを辿る。すると、[Figure 12](#) のような画面が得られるはずである。この画面で、自分のアカウントの管理下にあるインスタンスを確認することができる。



Figure 12. EC2 コンソール画面



コンソール右上で、正しいリージョン（今回の場合は ap-northeast-1, Tokyo）が選択されているか、注意する！

同様に、VPC・SGについてもコンソールから確認することができる。

前章で CloudFormation について触れたが、今回デプロイしたアプリケーションも、CloudFormation のスタックとして管理されている。スタック (stack) とは、AWS リソースの集合のことを指す。今回の場合は、VPC/EC2/SG などがスタックの中に含まれている。

コンソールで CloudFormation のページに行ってみよう (Figure 13)。



Figure 13. CloudFormation コンソール画面

"MyFirstEc2" という名前のスタックがあることが確認できる。クリックをして中身を見てみると、EC2、VPN などのリソースがこのスタックに紐付いていることがわかる。

4.4.8. スタックを削除

これにて、第一回のハンズオンで説明すべき事柄はすべて完了した。最後に、使わなくなったスタックを削除しよう。

スタックの削除には、2つの方法がある。

1つめの方法は、前節の CloudFormation のコンソール画面で、"Delete" ボタンを押すことである (Figure 14)。

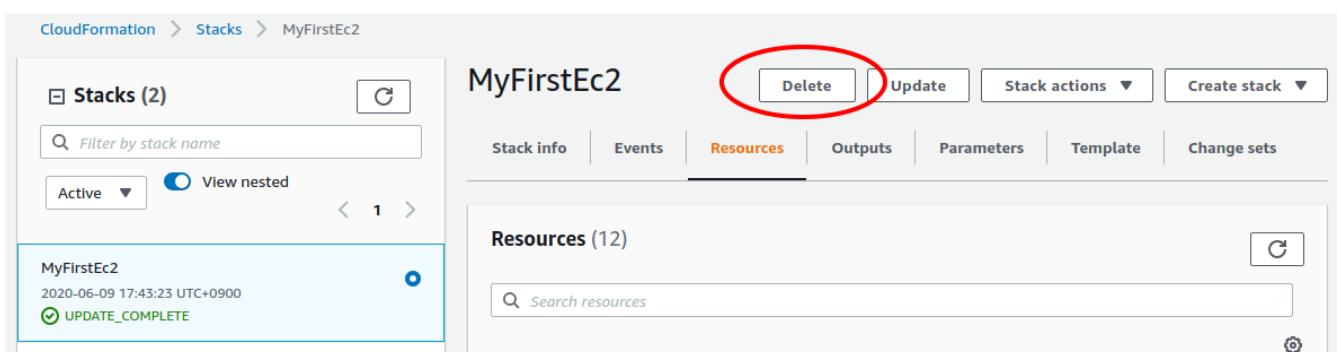


Figure 14. CloudFormation コンソール画面から、スタックを削除

2つめの方法は、コマンドラインから行う方法である。

先ほど、デプロイを行ったコマンドラインに戻ろう。そうしたら、

```
$ cdk destroy
```

と実行する。すると、スタックの削除が始まる。

削除した後は、VPC、EC2など、すべて跡形もなく消え去っている。

このように、自分の使いたいときにだけ、サーバーを立ち上げ、使い終わったら直ちに削除する、というのが現代のクラウドの正しい使い方である。



■ **スタックの削除は各自で必ず行うこと!** 行わなかった場合、EC2 インスタンスの料金が発生し続けることになる!

また、本ハンズオンのために作成した SSH 鍵ペアも不要なので、削除しておく。

まず、EC2 側に登録してある公開鍵を削除する。これも、コンソールおよびコマンドラインの2つの方法で実行できる。

コンソールから実行するには、EC2 の画面に行き、左のサイドバーの **Key Pairs** を選択。鍵の一覧が表示されるので、**HirakeGoma** となる鍵にチェックを入れ、画面右上の **Actions** から、**Delete** を実行 (Figure 15)。

The screenshot shows the AWS EC2 Key Pairs page. On the left sidebar, under the 'INSTANCES' section, 'Key Pairs' is selected. In the main content area, the title 'Key pairs (1/1)' is displayed above a table. The table has columns: Name, Fingerprint, and ID. A single row is present, showing 'HirakeGoma' in the Name column. To the right of the table, there is an 'Actions' button with a dropdown menu, and an orange 'Delete' button.

Figure 15. EC2でSSH鍵ペアを削除

コマンドラインから実行するには、以下のコマンドを使う。

```
$ aws ec2 delete-key-pair --key-name "HirakeGoma"
```

最後に、ローカルのコンピュータから鍵を削除する。

```
$ rm -f ~/.ssh/HirakeGoma.pem
```

これで、クラウドの片付けもすべて終了だ。



なお、頻繁に EC2 インスタンスを起動したりする場合は、いちいち SSH 鍵を削除する必要はない。

4.5. 小括

ここまでが、本書の第一部の内容である。盛りだくさんの内容であったが、ついてこれたであろうか？

Chapter 2 では、クラウドの概要と、なぜクラウドを使うのか、という点を議論した。続いて Chapter 3 では、クラウド

を学ぶ具体的な題材として AWS を取り上げ、AWS の概要説明を行った。さらに、Chapter 4 のハンズオンでは AWS CLI/CDK を使って、自分のプライベートなサーバーを AWS 上に立ち上げる演習を行った。

これらを通じて、いかに簡単に（たった数行のコマンドで！）仮想サーバーを立ち上げたり、削除したりすることができるか、体験することができただろう。このように、**ダイナミックに計算リソースを拡大・縮小をできることが、クラウドの最も本質的な側面であると、筆者は考えている。**

次章からは、今回学んだクラウドの技術を基に、より現実に即した問題を解くことを体験してもらう。お楽しみに！

Chapter 5. クラウドで行う科学計算・機械学習

計算機が発達した現代では、計算機によるシミュレーションやビッグデータの解析は、科学・エンジニアリングの研究の主要な柱である。これらの大規模な計算を実行するには、クラウドは最適である。本章から始まる第二部では、どのようにしてクラウド上で科学計算を実行するのかを、ハンズオンとともに体験してもらう。科学計算の具体的な題材として、今回は機械学習（深層学習）を取り上げる。

なお、本書では [PyTorch](#) ライブラリを使って深層学習のアルゴリズムを実装するが、深層学習および PyTorch の知識は不要である。講義ではなぜ・どうやって深層学習をクラウドで実行するかに主眼を置いているので、実行するプログラムの詳細には立ち入らない。将来自分で深層学習を使う機会が来たときに、詳しく学んでもらいたい。

5.1. なぜ機械学習をクラウドで行うのか？

2010年代に始まった AI ブームのおかげで、研究だけでなく社会・ビジネスの文脈でも機械学習に高い関心が寄せられている。特に、深層学習（ディープラーニング）と呼ばれる多層のレイヤーからなるニューラルネットワークを用いたアルゴリズムは、画像認識や自然言語処理などの分野で圧倒的に高い性能を実現し、革命をもたらしている。

ディープラーニングの特徴は、なんといってもそのパラメータの多さである。層が深くなるほど、層間のニューロンを結ぶ重みパラメータの数が増大していく。例えば、最新の言語モデルである [GPT-3](#) には **1750億個** のパラメータが含まれている！このような膨大なパラメータを有することで、ディープラーニングは高い表現力と汎化性能を実現しているのである。

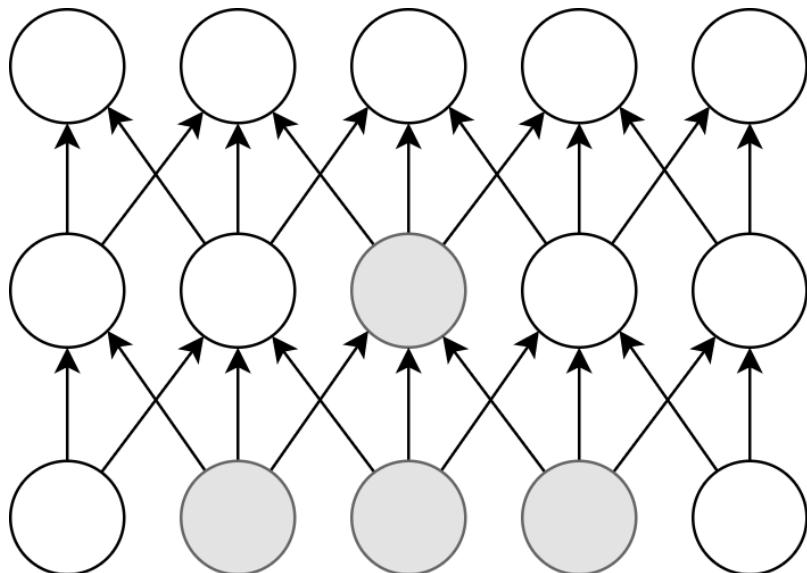


Figure 16. ニューラルネットワークにおける畳み込み演算。

GPT-3に限らず、最近の SOTA (State-of-the-art) の性能を達成するニューラルネットでは、百万から億のオーダーのパラメータを有することは頻繁になってきている。そのような巨大なニューラルネットを訓練（最適化）させるのは、当然のことながら膨大な計算コストがかかる。そんな巨大な計算に最適なのが、クラウドである！事実、GPT-3の学習も、詳細は明かされていないが、Microsoft 社のクラウドを使って行われたと報告されている。

GPT-3 で使われた計算リソースの詳細は論文でも明かされていないのだが、[Lambda 社のブログ](#)で興味深い考察が行われている（Lambda 社は機械学習に特化したクラウドサービスを提供している）。



記事によると、1750億のパラメータを訓練するには、一台の GPU (NVIDIA V100) を用いた場合、342年の月日と460万ドルのクラウド利用料が必要となる、とのことである。GPT-3 のチームは、複数の GPU に処理を分散することで現実的な時間のうちに訓練を完了させたのであろうが、このレベルのモデルになるとクラウド技術の限界を攻めないと達成できることは確かである。

5.2. GPUによる機械学習の高速化

ディープラーニングの計算で欠かすことのできない技術として, **GPU (Graphics Processing Unit)**について少し説明する。

GPUは,その名のとおり,元々はコンピュータグラフィックスを出力するための専用計算チップである. CPU (Central Processing Unit)に対し, グラフィックスの演算に特化した設計がなされている. 身近なところでは, XBoxやPS5などのゲームコンソールなどに搭載されているし,ハイエンドなノート型・デスクトップ型計算機にも搭載されていることがある. コンピュータグラフィックスでは,スクリーンにアレイ状に並んだ数百万個の画素をビデオレート(30fps)以上で処理する必要がある. そのため, GPUはコアあたりの演算能力は比較的小さいかわりに, チップあたり数百から数千のコアを搭載しており(Figure 17),スクリーンの画素を並列的に処理することで,リアルタイムでの描画を実現している.

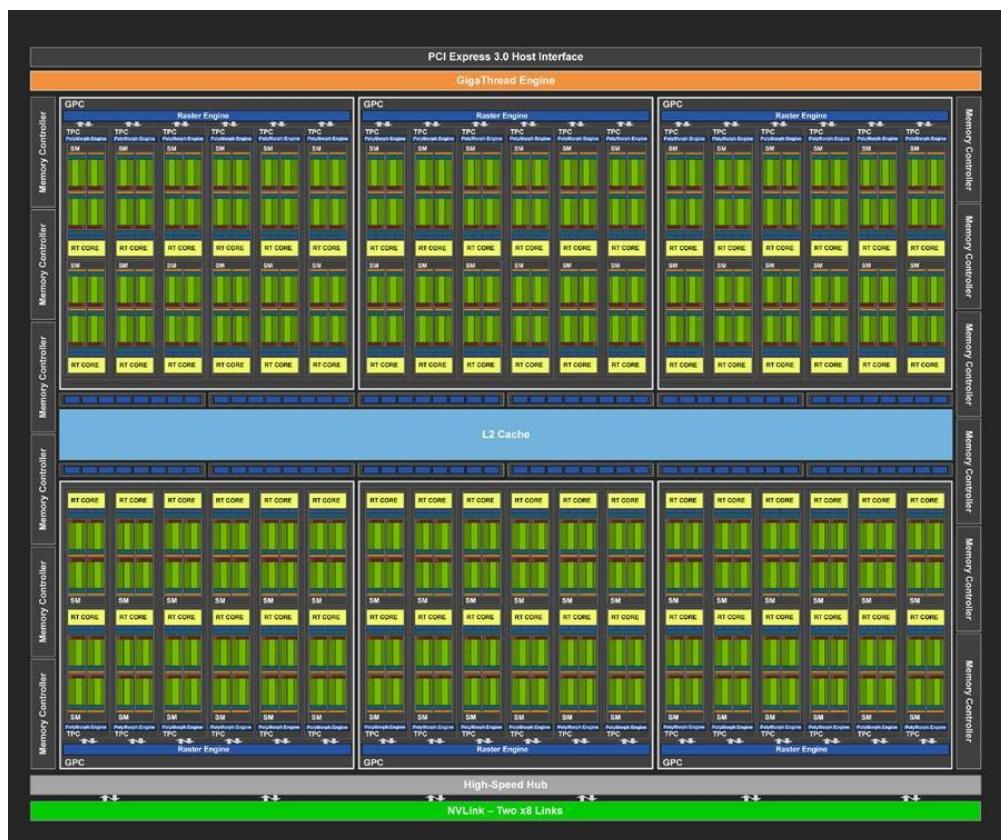


Figure 17. GPUのアーキテクチャ. GPUには数百から数千の独立した計算コアが搭載されている. (画像出典: <https://devblogs.nvidia.com/nvidia-turing-architecture-in-depth/>)

このように,コンピュータグラフィックスの目的で生まれたGPUだが,2010年前後から,その高い並列計算能力をグラフィックス以外の計算(科学計算など)に用いるという流れ(**General-purpose computing on GPU; GPGPU**)が生まれた. GPUのコアは,その設計から,行列の計算など,単純かつ規則的な演算が得意であり,そのような演算に対しては数個程度のコアしか持たないCPUに比べて圧倒的に高い計算速度を実現することができる. 現在ではGPGPUは分子動力学や気象シミュレーション,そして機械学習など多くの分野で使われている.

ディープラーニングで最も頻繁に起こる演算が,ニューロンの出力を次の層のニューロンに伝える**畳み込み(Convolution)**演算である(Figure 16). 畳み込み演算は,まさにGPUが得意とする演算であり,CPUではなくGPUを用いることで学習を飛躍的に(最大で数百倍程度)加速させることができる.

このようにGPUは機械学習の計算で欠かせないものであるが,なかなか高価である. 例えば,科学計算・機械学習に専用設計されたNVIDIA社のTesla V100というチップは,一台で約百万円の価格が設定されている. 機械学習を始めるのに,いきなり百万円の投資はなかなか大きい. だが,クラウドを使えば,初期コスト0でGPUを使用することができる.

 機械学習を行うのに、V100 が必ずしも必要というわけではない。むしろ、研究者などではしばしば行われるのは、コンピュータゲームに使われるグラフィックス用の GPU を買ってきて (NVIDIA GeForce シリーズなど)、開発のときはそれを用いる、というアプローチである。グラフィックス用のいわゆる"コンシューマ GPU"は、市場の需要が大きいおかげで、10万円前後の価格で購入することができる。V100 と比べると、コンシューマ GPU はコアの数が少なかったり、メモリーが小さかったりなどで劣る点があるが、それらを除いては計算能力に特に制限があるわけではなく、開発の段階では十分な性能である。

プログラムができあがって、ビッグデータの解析や、モデルをさらに大きくしたいときなどに、クラウドは有効だろう。

クラウドで GPU を使うには、GPU が搭載された EC2 インスタンスタイプ ([P3](#), [P2](#), [G3](#), [G4](#) など) を選択しなければならない。[Table 3](#) に、代表的な GPU 搭載のインスタンスタイプを挙げる (執筆時点での情報)。

Table 3. GPU を搭載した EC2 インスタンスタイプ

Instance	GPUs	GPU model	GPU Mem (GiB)	vCPU	Mem (GiB)	Price per hour (\$)
p3.2xlarge	1	NVIDIA V100	16	8	61	3.06
p3n.16xlarge	8	NVIDIA V100	128	64	488	24.48
p2.xlarge	1	NVIDIA K80	12	4	61	0.9
g4dn.xlarge	1	NVIDIA T4	16	4	16	0.526

[Table 3](#) からわかるとおり、CPU のみのインスタンスと比べると少し高い価格設定になっている。また、古い世代の GPU (V100 に対しての K80) はより安価な価格で提供されている。1 インスタンスあたりの GPU の搭載数は 1 台から最大で 8 台まで選択することが可能である。

GPU を搭載した一番安いインスタンスタイプは、[g4dn.xlarge](#) であり、これには廉価かつ省エネルギー設計の NVIDIA T4 が搭載されている。後のハンズオンでは、このインスタンスを使用して、ディープラーニングの計算を行ってみる。



[Table 3](#) の価格は [us-east-1](#) のものである。リージョンによって多少価格設定が異なる。

 V100 を一台搭載した [p3.2xlarge](#) の利用料金は一時間あたり \$3.06 である。V100 が約百万円で売られていることを考えると、約 3000 時間 (= 124 日間)、通算で計算を行った場合に、クラウドを使うよりも V100 を自分で買ったほうがお得になる、という計算になる。(実際には、自分で V100 を用意する場合は、V100 だけでなく、CPU やネットワーク機器、電気使用料も必要なので、百万円よりもさらにコストがかかる。)

Further reading

深層学習を詳しく勉強したい人には以下の参考書を推薦したい。深層学習の基礎的な概念や理論は普遍的であるが、この分野は日進月歩なので、常に最新の情報を取り入れることを忘れずに。

- [Deep Learning \(Ian Goodfellow, Yoshua Bengio and Aaron Courville\)](#) 出版されてから数年が経つが、深層学習の理論的な側面を学びたいならばおすすめの入門書。ウェブで無料で読むことができる。日本語版も出版されている。実装についてはほとんど触れられていないので、理論家向けの本。
- [ゼロから作る Deep Learning \(斎藤 康毅\)](#) 合計三冊からなるシリーズ。理論と実装がバランスよく説明されていて、深層学習の入門書の決定版。
- [Dive into Deep Learning \(Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola\)](#) 深層学習の基礎から最新のアルゴリズムまでを、実装を通して学んでいくスタイルの本。ウェブで無料で読むことができる、1000ページ越えの超大作。これを読破すれば、深層学習の実装で困ることはないだろう。

Chapter 6. Hands-on #2: AWS でディープラーニングを実践

ハンズオン第二回では、GPU を搭載したEC2インスタンスを起動し、ディープラーニングの学習と推論を実行する演習を行う。

ハンズオンのソースコードは [こちらのリンク](#) に置いてある。



このハンズオンは、AWS の無料枠内では実行できない。g4dn.xlarge タイプの EC2 インスタンスを使うので、us-east-1 リージョンでは 0.526 \$/hour のコストが発生する。東京 (ap-northeast-1) を選択した場合は 0.71 \$/hour のコストが発生する。

6.1. 準備

本ハンズオンの実行には、第一回ハンズオンで説明した準備 (Section 4.1) が整っていることを前提とする。それ以外に必要な準備はない。

6.2. アプリケーションの説明

このハンズオンで作成するアプリケーションの概要を Figure 18 に示す。

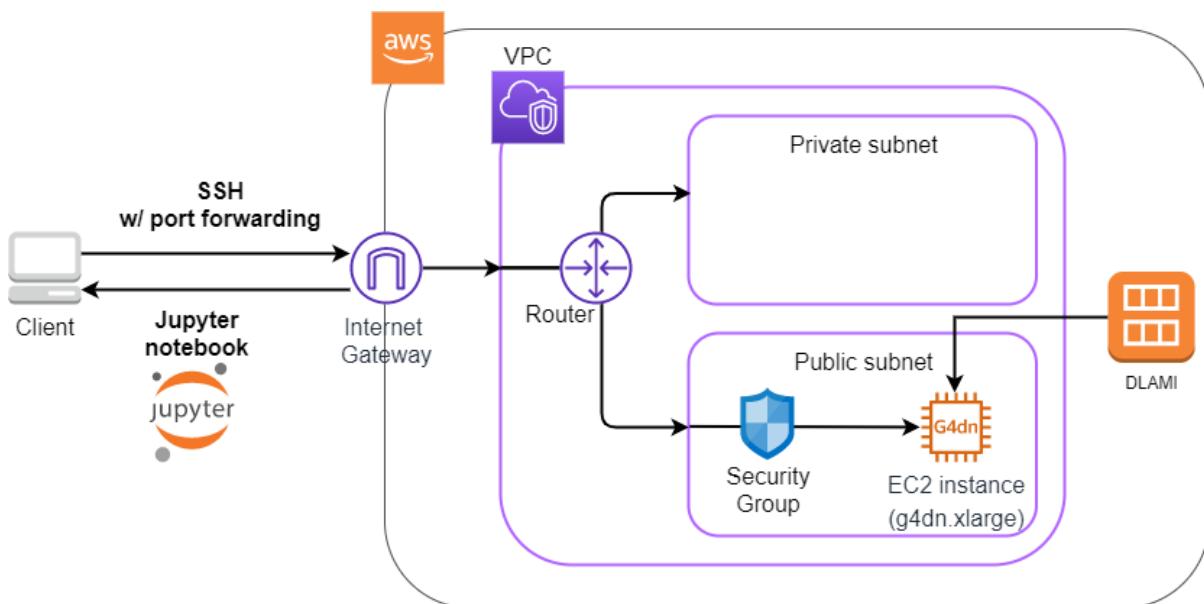


Figure 18. ハンズオン#2で作製するアプリケーションのアーキテクチャ

図の多くの部分が、第一回ハンズオンで作成したアプリケーションと共に通していることに気がつくだろう。少しの変更で、簡単にディープラーニングを走らせる環境を構築することができる! 主な変更点は次の3点である。

- GPUを搭載した g4dn.xlarge インスタンスタイプを使用。
- ディープラーニングに使うプログラムが予めインストールされた DLAMI (後述) を使用。
- SSHにポートフォワーディングのオプションつけてサーバーに接続し、サーバーで起動しているJupyter notebook (後述) を使ってプログラムを書いたり実行したりする。

ハンズオンで使用するプログラムのコードをみてみよう ([/handson/mnist/app.py](#))。コードは第一回目とほとんど共通である。変更点のみ解説を行う。

```

1 class Ec2ForDl(core.Stack):
2
3     def __init__(self, scope: core.App, name: str, key_name: str, **kwargs) -> None:
4         super().__init__(scope, name, **kwargs)
5
6         vpc = ec2.Vpc(
7             self, "Ec2ForDl-Vpc",
8             max_azs=1,
9             cidr="10.10.0.0/23",
10            subnet_configuration=[
11                ec2.SubnetConfiguration(
12                    name="public",
13                    subnet_type=ec2.SubnetType.PUBLIC,
14                )
15            ],
16            nat_gateways=0,
17        )
18
19        sg = ec2.SecurityGroup(
20            self, "Ec2ForDl-Sg",
21            vpc=vpc,
22            allow_all_outbound=True,
23        )
24        sg.add_ingress_rule(
25            peer=ec2.Peer.any_ipv4(),
26            connection=ec2.Port.tcp(22),
27        )
28
29        host = ec2.Instance(
30            self, "Ec2ForDl-Instance",
31            instance_type=ec2.InstanceType("g4dn.xlarge"), ①
32            machine_image=ec2.MachineImage.generic_linux({
33                "ap-northeast-1": "ami-09c0c16fc46a29ed9"
34            }), ②
35            vpc=vpc,
36            vpc_subnets=ec2.SubnetSelection(subnet_type=ec2.SubnetType.PUBLIC),
37            security_group=sg,
38            key_name=key_name
39        )

```

① ここで、GPU を搭載した `g4dn.xlarge` インスタンスタイプを選択している (第一回では、CPU のみの `t2.micro` だった).

② ここでは、Deep Learning 用の諸々のソフトウェアがプリントールされたAMI ([Deep Learning Amazon Machine Image; DLAMI](#)) を選択している. (第一回では、Amazon Linux というAMIを使用していた). 使用するAMIのIDは `ami-09c0c16fc46a29ed9` である.

`g4dn.xlarge` のインスタンスタイプについては、[Chapter 5](#) すでに触れた. DLAMIについて、少し説明しよう.

6.2.1. DLAMI (Deep Learning Amazon Machine Image)

AMI (Amazon Machine Image) とは、大まかには OS (Operating System) に相当する概念である. 当然のことながら、OS がなければコンピュータはなにもできないので、EC2 インスタンスを起動する時には必ずなにかの OS を "インストール" する必要がある. EC2 が起動したときにロードされる OS に相当するものが、AMI である. AMI には、例えば [Ubuntu](#) などの Linux 系 OS に加えて、Windows Server を選択することもできる. また、EC2 での使用に最適化された [Amazon Linux](#) という AMI も提供されている.

しかしながら、AMI を単なる OS と理解するのは過剰な単純化である. AMI には、ベースとなる (空っぽの) OS を選択できることもできるが、それに加えて、各種のプログラムがインストール済みの AMI も定義することができる. 必要なプログラムがインストールされている AMI を見つけることができれば、自分でインストールを行ったり環境設

定をする手間が大幅に省ける。具体例を挙げると、ハンズオン第一回では EC2 インスタンスに Python 3.6 をインストールする例を示したが、そのような操作をインスタンスを起動するたびに行うのは手間である！

AMI は、AWS 公式のものに加えて、サードパーティーから提供されているものもある。また、自分自身の AMI を作って登録することも可能である（[参考](#)）。AMI は EC2 のコンソールから検索することが可能である。あるいは、AWS CLI を使って、次のコマンドでリストを取得することができる（[参考](#)）。

```
$ aws ec2 describe-images --owners amazon
```

ディープラーニングで頻繁に使われるプログラムが予めインストールしてあるAMIが、[DLAMI \(Deep Learning AMI\)](#) である。DLAMI には [TensorFlow](#), [PyTorch](#) などの人気の高いディープラーニングのフレームワーク・ライブラリが既にインストールされているため、EC2 インスタンスを起動してすぐさまディープラーニングの計算を実行できる。

本ハンズオンでは、Amazon Linux 2 をベースにした DLAMI を使用する（AMI ID = ami-09c0c16fc46a29ed9）。AWS CLI を使って、このAMIの詳細情報を取得してみよう。

```
$ aws ec2 describe-images --owners amazon --image-ids "ami-09c0c16fc46a29ed9"
```

```
tomoyuki@balthasar:02-ec2-dnn$ aws ec2 describe-images --owners amazon --image-ids "ami-09c0c16fc46a29ed9"
{
    "Images": [
        {
            "Architecture": "x86_64",
            "CreationDate": "2020-05-20T14:47:04.000Z",
            "ImageId": "ami-09c0c16fc46a29ed9",
            "ImageLocation": "amazon/Deep Learning AMI (Amazon Linux 2) Version 29.0",
            "ImageType": "machine",
            "Public": true,
            "OwnerId": "898082745236",
            "PlatformDetails": "Linux/UNIX",
            "UsageOperation": "RunInstances",
            "State": "available",
            "BlockDeviceMappings": [
                {
                    "DeviceName": "/dev/xvda",
                    "Ebs": {
                        "DeleteOnTermination": true,
                        "SnapshotId": "snap-0bd381ab76e5a6146",
                        "VolumeSize": 90,
                        "VolumeType": "gp2",
                        "Encrypted": false
                    }
                }
            ],
            "Description": "MXNet-1.6.0, Tensorflow-2.1.0 & 1.15.2, PyTorch-1.4.0 & 1.5.0, Neuron, & other frameworks, NVIDIA CUDA, cuDNN, NCCL, Intel MKL-DNN, Docker, NVIDIA-Docker & EFA support. For fully managed experience, check: <https://aws.amazon.com/sagemaker>.",
            "EnaSupport": true,
            "Hypervisor": "xen",
            "ImageOwnerAlias": "amazon",
            "Name": "Deep Learning AMI (Amazon Linux 2) Version 29.0",
            "RootDeviceName": "/dev/xvda",
            "RootDeviceType": "ebs",
            "SriovNetSupport": "simple",
            "VirtualizationType": "hvm"
        }
    ]
}
```

Figure 19. AMI ID = ami-09c0c16fc46a29ed9 の詳細情報

Figure 19 のような出力が得られるはずである。得られた出力から、例えばこの DLAMI には PyTorch のバージョン 1.4.0 と 1.5.0 がインストールされていることがわかる。この DLAMI を使って、早速ディープラーニングの計算を実行してみよう。

DLAMIには具体的には何がインストールされているのだろうか？興味のある読者のために、簡単な解説をしよう（参考：[公式ドキュメンテーション](#)）。

最も low-level なレイヤーとしては、GPU ドライバーがインストールされている。GPU ドライバーなしには OS は GPU とコマンドのやり取りをすることができない。次のレイヤーが [CUDA](#) と [cuDNN](#) である。CUDA は、NVIDIA 社が開発した、GPU 上で汎用コンピューティングを行うための言語であり、C++ 言語を拡張したシンタックスを備える。cuDNN は CUDA で書かれたディープラーニングのライブラリであり、n 次元の畳み込みなどの演算が実装されている。



以上までが、"Base" と呼ばれるタイプの DLAMI の中身である。

これに加えて、"Conda" と呼ばれるタイプには、これらのプログラム基盤の上に、[TensorFlow](#) や [PyTorch](#) などのライブラリがインストールされている。さらに、[Anaconda](#) による仮想環境を使うことによって、[TensorFlow](#) の環境、[PyTorch](#) の環境、を簡単に切り替えることができる（これについては、ハンズオンで触れる）。また、Jupyter notebook もインストール済みである。

6.3. スタックのデプロイ

スタックの中身が理解できたところで、早速スタックをデプロイしてみよう。

デプロイの手順は、ハンズオン1とほとんど共通である。ここでは、コマンドのみ列挙する（# で始まる行はコメントである）。それぞれのコマンドの意味を忘れてしまった場合は、ハンズオン1に戻って復習していただきたい。シークレットキーの設定も忘れずに（[Section 15.2](#)）。

```
# プロジェクトのディレクトリに移動
$ cd handson/mnist

# venv を作成し、依存ライブラリのインストールを行う
$ python3 -m venv .env
$ source .env/bin/activate
$ pip install -r requirements.txt

# SSH鍵を生成
$ export KEY_NAME="HirakeGoma"
$ aws ec2 create-key-pair --key-name ${KEY_NAME} --query 'KeyMaterial' --output text > ${KEY_NAME}.pem
$ mv HirakeGoma.pem ~/ssh/
$ chmod 400 ~/ssh/HirakeGoma.pem

# デプロイを実行
$ cdk deploy -c key_name="HirakeGoma"
```



ハンズオン1で作成した SSH 鍵の削除を行わなかった場合は、SSH 鍵を改めて作成する必要はない。逆に言うと、同じ名前の SSH が既に存在する場合は、鍵生成のコマンドはエラーを出力する。

デプロイのコマンドが無事に実行されれば、[Figure 20](#) のような出力が得られるはずである。AWS により割り振られた IP アドレス（[InstancePublicIp](#) に続く文字列）をメモしておこう。

```
✓ Ec2ForDl

Outputs:
Ec2ForDl.InstancePublicIp = 52.192.211.12
Ec2ForDl.InstancePublicDnsName = ec2-52-192-211-12.ap-northeast-1.compute.amazonaws.com

Stack ARN:
arn:aws:cloudformation:ap-northeast-1:606887060834:stack/Ec2ForDl/dd8361d0-
```

Figure 20. CDK デプロイ実行後の出力

6.4. ログイン

早速, デプロイしたインスタンスにSSHでログインしてみよう.

ここでは, この後で使う Jupyter notebook に接続するため, ポートフォワーディング のオプション (-L) をつけてログインする.

```
$ ssh -i ~/.ssh/HirakeGoma.pem -L localhost:8931:localhost:8888 ec2-user@<IP address>
```

ポートフォワーディングとは, クライアントマシンの特定のアドレスへの接続を, SSH の暗号化された通信を介して, リモートマシンの特定のアドレスへ転送する, という意味である. 上のコマンドの -L **localhost:8931:localhost:8888** は, 自分のローカルマシンの **localhost:8931** へのアクセスを, リモートサーバーの **localhost:8888** のアドレスに転送せよ, という意味である (: につづく数字はポート番号を意味している). リモートサーバーのポート8888には, 後述する Jupyter notebook が起動している. したがって, ローカルマシンの **localhost:8931** にアクセスすることで, リモートサーバーの Jupyter notebook にアクセスすることができる所以である (このようなSSHによる接続方式を **トンネル接続**と呼ぶ).

ポートフォワーディングについて混乱した読者は, より詳しい解説が [このブログ記事](#) にある.



ポートフォワーディングのオプションで, ポートの番号 (:8931, :8888など) には1から65535までの任意の整数を指定できる. しかし, 例えば ポート22(SSH)やポート80(HTTP)など, いくつか既に使われているポート番号もあることに注意する. また, Jupyter notebook はデフォルトではポート8888番を使用する. したがって, リモート側のポート番号は, 8888を使うのがよい.



SSHログインコマンドの <IP address> 部分は自分のインスタンスのIPアドレスを代入することを忘れずに.

本書の提供している Docker を使ってデプロイを実行した人へ

SSHによるログインは, **Dockerの外** (すなわちクライアントマシン本体) から行わなければならない. なぜなら, Jupyterを開くウェブブラウザは Docker の外にあるからである.



その際, 秘密鍵を Docker の外に持ってこなければならない. 手っ取り早い方法は, **cat ~/.ssh/HirakeGoma** と打って, 出力結果をコピー&ペーストで Docker の外に持ってくる方法である.

あるいは **-v** オプションをつけて, ファイルシステムをマウントしてもよい (詳しくは [こちらを参照](#)).

SSHによるログインができたら, 早速, GPUの状態を確認してみよう. 以下のコマンドを実行する.

```
$ nvidia-smi
```

[Figure 21](#) のような出力が得られるはずである. 出力を見ると, Tesla T4型のGPUが1台搭載されていることが確認できる. その他, GPU Driver や CUDA のバージョンを確認することができる.

```
[ec2-user@ip-10-10-1-172 ~]$ nvidia-smi
Sat Jun 13 04:55:22 2020
+-----+
| NVIDIA-SMI 440.33.01    Driver Version: 440.33.01    CUDA Version: 10.2 |
+-----+
| GPU  Name      Persistence-M| Bus-Id     Disp.A  | Volatile Uncorr. ECC | | | | | | | | | |
| Fan  Temp  Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M. |
|====|====|====|====|====|====|====|====|====|====|====|====|
| 0  Tesla T4           On | 00000000:00:1E.0 Off |          0 |
| N/A   31C   P8    11W /  70W |      0MiB / 15109MiB |      0%       Default |
+-----+
+-----+
| Processes:                               GPU Memory |
| GPU      PID  Type  Process name        Usage      |
|====|====|====|====|
| No running processes found               |
+-----+
```

Figure 21. nvidia-smi の出力

6.5. Jupyter notebook の起動

Jupyter notebook とは、インタラクティブに Python のプログラムを書いたり実行したりするためのツールである。Jupyter は GUI としてウェブブラウザを介してアクセスする形式をとっており、まるでノートを書くように、プロットやテーブルのデータも美しく表示することができる (Figure 22)。Python に慣れている読者は、きっと一度は使ったことがあるだろう。

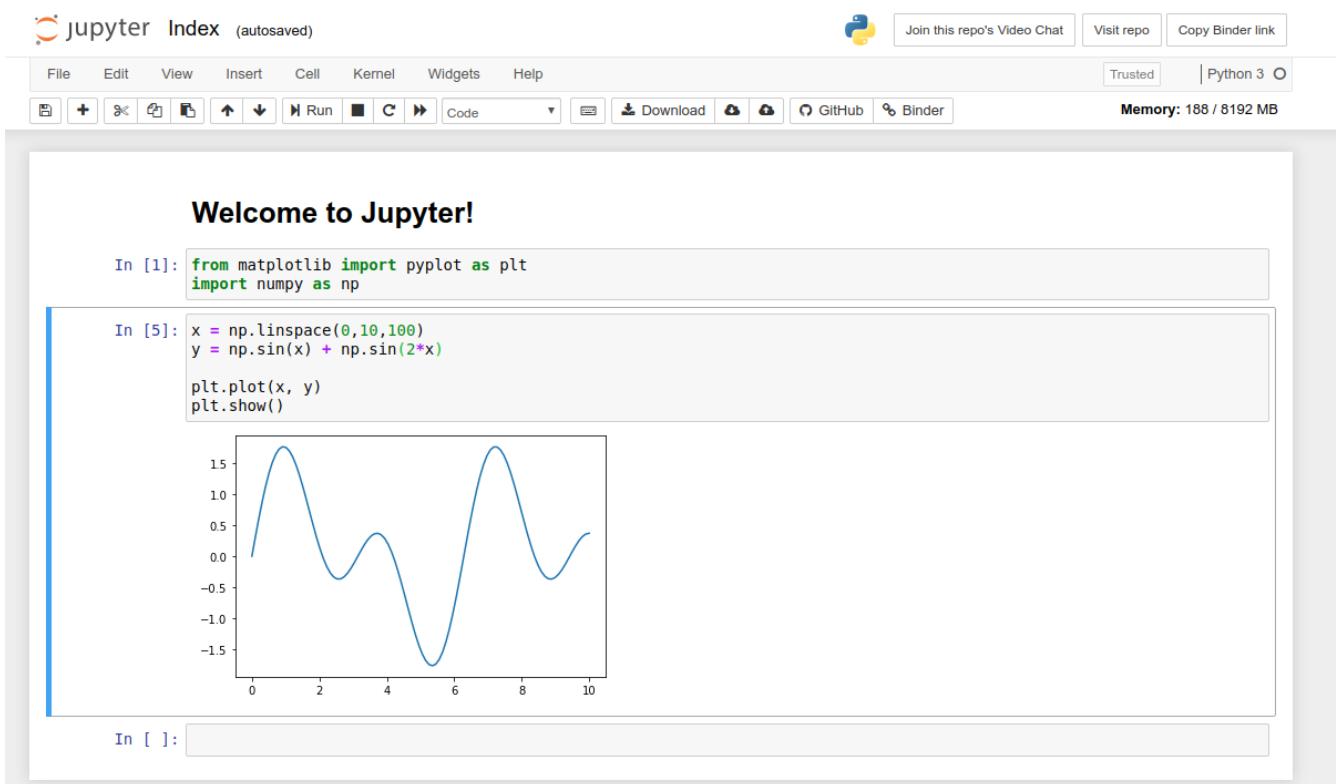


Figure 22. Jupyter notebook の画面

このハンズオンでは、Jupyter notebook を使ってディープラーニングのプログラムを書いたり実行していく。DLAMI には既に Jupyter がインストールされているので、特段の設定なしに使い始めることができる。

早速、Jupyter を起動しよう。SSHでログインした先の EC2 インスタンスで、次のコマンドを実行すればよい。

```
$ cd ~ # go to home directory
$ jupyter notebook
```

このコマンドを実行すると, Figure 23 のような出力が確認できるだろう. この出力から, Jupyter のサーバーが EC2 インスタンスの `localhost:8888` というアドレスに起動していることがわかる. また, `localhost:8888` に続く `?token=XXXXXXX` は, アクセスに使うための一時的なトークンである.

```
[I 06:01:10.466 NotebookApp] The Jupyter Notebook is running at:
[I 06:01:10.466 NotebookApp] http://localhost:8888/?token=537fee42934a7db9d0540024260d305b08bb0bf9fc41c5a7
[I 06:01:10.466 NotebookApp] or http://127.0.0.1:8888/?token=537fee42934a7db9d0540024260d305b08bb0bf9fc41c5a7
[I 06:01:10.466 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[W 06:01:10.469 NotebookApp] No web browser found: could not locate runnable browser.
[C 06:01:10.469 NotebookApp]

To access the notebook, open this file in a browser:
  file:///home/ec2-user/.local/share/jupyter/runtime/nbsrvr-11720-open.html
Or copy and paste one of these URLs:
  http://localhost:8888/?token=537fee42934a7db9d0540024260d305b08bb0bf9fc41c5a7
  or http://127.0.0.1:8888/?token=537fee42934a7db9d0540024260d305b08bb0bf9fc41c5a7
```

Figure 23. Jupyter notebook サーバーを起動



Jupyter notebook を初回に起動するときは, 起動に1,2分程度の時間がかかることがある. ほかの動作も起動直後はなぜか遅く, いくつかプログラムを走らせていくうちに俊敏に反応するようになってくる. AWS の GPU 搭載の仮想マシンの運用方法に起因する現象だと考えられる.

先ほど, ポートフォワーディングのオプションをつけて SSH 接続をしているので, Jupyter の起動している `localhost:8888` には, ローカルマシンの `localhost:8931` からアクセスすることができる.

したがって, ローカルマシンから Jupyter にアクセスするには, ウェブブラウザ (Chrome, FireFox など) から次のアドレスにアクセスすれば良い.

```
http://localhost:8931/?token=XXXXXXXXXX
```

`?token=XXXXXXX` の部分は, 上で Jupyter を起動したときに発行されたトークンの値に置き換える.

上のアドレスにアクセスすると, Jupyter のホーム画面が起動するはずである (Figure 24). これで, Jupyter の準備が整った!

The screenshot shows the Jupyter Notebook home page. At the top, there is a navigation bar with tabs for 'Files' (selected), 'Running', 'IPython Clusters', and 'Conda'. To the right of the tabs are 'Quit' and 'Logout' buttons. Below the navigation bar, there is a search bar with placeholder text 'Select items to perform actions on them.' and buttons for 'Upload', 'New', and a refresh icon. The main area displays a list of files and folders in the current directory ('/'). The list includes:

	Name	Last Modified	File size
0	/		
anaconda3		25 days ago	
data		16 hours ago	
examples		20 hours ago	
examples_		25 days ago	
src		25 days ago	
tools		a month ago	
tutorials		25 days ago	

Figure 24. Jupyter ホーム画面

Jupyter notebook の使い方 (超簡易版)



- **Shift + Enter**: セルを実行
- **Esc**: **Command mode** に遷移
- メニューバーの "+" ボタン または Command mode で **A** ⇒ セルを追加
- メニューバーの "ハサミ" ボタン または Command mode で **X** ⇒ セルを削除

ショートカットの一覧などは [このブログ](#) が参考になる。

6.6. PyTorchはじめの一歩

PyTorch は Facebook AI Research LAB (FAIR) が中心となって開発を進めている、オープンソースのディープラーニングのライブラリである。PyTorch は有名な例で言えば Tesla 社の自動運転プロジェクトなどで使用されており、執筆時点において最も人気の高いディープラーニングライブラリの一つである。本ハンズオンでは、PyTorch を使ってディープラーニングの実践を行う。



PyTorch の歴史のお話

Facebook は PyTorch の他に Caffe2 と呼ばれるディープラーニングのフレームワークを開発していた (初代Caffe は UC Berkley の博士課程学生だった Yangqing Jia によって創られた)。Caffe2 は 2018年に PyTorch プロジェクトに合併された。

また、2019年12月、日本の Preferred Networks 社が開発していた **Chainer** も、開発を終了し、PyTorchの開発チームと協業していくことが発表された ([プレスリリース](#))。PyTorch には、開発統合前から Chainer からインスパイアされた API がいくつもあり、Chainer の DNA は今も PyTorch に引き継がれているのである…！

本格的なディープラーニングの計算に移る前に、PyTorch ライブラリを使って、GPU で計算を行うはどういうものか、その入り口に触れてみよう。

まずは、新しいノートブックを作成する。Jupyterのホーム画面の右上の "New" を押し、"conda_pytorch_p36" という環境を選択した上で、新規ノートブックを作成する ([Figure 25](#))。"conda_pytorch_p36" の仮想環境には、PyTorch がインストール済みである (他にある TensorFlow なども同様)。

Files Running IPython Clusters Conda

Select items to perform actions on them.

Upload New ▾



0	/
	anaconda3
	examples
	src
	tools
	tutorials
	LICENSE
	Nvidia_Cloud_EULA.pdf
	README

Notebook:
Environment (conda_anaconda3)
Environment (conda_aws_neuron_mxnet_p36)
Environment (conda_aws_neuron_pytorch_p36)
Environment (conda_aws_neuron_tensorflow_p36)
Environment (conda_chainer_p27)
Environment (conda_chainer_p36)
Environment (conda_mxnet_p27)
Environment (conda_mxnet_p36)
Environment (conda_python2)
Environment (conda_python3)
Environment (conda_pytorch_latest_p36)
Environment (conda_pytorch_p27)
Environment (conda_pytorch_p36)
Environment (conda_tensorflow2_p27)

Figure 25. 新規ノートブックの作成. "conda_pytorch_p36" の環境を選択する.

次のようなプログラムを書いてみよう (Figure 26).

```
In [1]: import torch
print("Is CUDA ready?", torch.cuda.is_available())
Is CUDA ready? True

In [3]: # create a random array in CPU
x = torch.rand(3,3)
print(x)

tensor([[0.6896, 0.2428, 0.3269],
       [0.0533, 0.3594, 0.9499],
       [0.9764, 0.5881, 0.0203]])

In [4]: # create another array in GPU device
y = torch.ones_like(x, device="cuda")
# move 'x' from CPU to GPU
x = x.to("cuda")

In [5]: # run addition operation in GPU
z = x + y
print(z)

tensor([[1.6896, 1.2428, 1.3269],
       [1.0533, 1.3594, 1.9499],
       [1.9764, 1.5881, 1.0203]], device='cuda:0')

In [6]: # move z from GPU to CPU
z = z.to("cpu")
print(z)

tensor([[1.6896, 1.2428, 1.3269],
       [1.0533, 1.3594, 1.9499],
       [1.9764, 1.5881, 1.0203]])
```

Figure 26. PyTorch始めの一歩

まずは, PyTorch をインポートする. さらに, GPUが使える環境にあるか, 確認する.

```
1 import torch
2 print("Is CUDA ready?", torch.cuda.is_available())
```

出力:

```
Is CUDA ready? True
```

次に, 3x3 のランダムな行列を **CPU** 上に作ってみよう.

```
1 x = torch.rand(3,3)
2 print(x)
```

出力:

```
tensor([[0.6896, 0.2428, 0.3269],
       [0.0533, 0.3594, 0.9499],
       [0.9764, 0.5881, 0.0203]])
```

次に, 行列を **GPU** 上に作成する.

```
1 y = torch.ones_like(x, device="cuda")
2 x = x.to("cuda")
```

そして, 行列 **x** と **y** の加算を, **GPU**上で実行する.

```
1 z = x + y
2 print(z)
```

出力:

```
tensor([[1.6896, 1.2428, 1.3269],
       [1.0533, 1.3594, 1.9499],
       [1.9764, 1.5881, 1.0203]], device='cuda:0')
```

最後に, GPU 上にある行列を, CPU に戻す.

```
1 z = z.to("cpu")
2 print(z)
```

出力:

```
tensor([[1.6896, 1.2428, 1.3269],
       [1.0533, 1.3594, 1.9499],
       [1.9764, 1.5881, 1.0203]])
```

以上の例は, GPU を使った計算の初步であるが, 雰囲気はつかめただろうか? CPU と GPU で明示的にデータを交換するのが肝である. この例は, たった 3x3 の行列の足し算なので, GPU を使う意味はまったくないが, これが数千, 数万のサイズの行列になった時, GPU は格段の威力を発揮する.

完成した Jupyter notebook は [/handson/mnist/pytorch/pytorch_get_started.ipynb](#) にある。Jupyter の画面右上の "Upload" から、このファイルをアップロードして、コードを走らせることが可能である。



しなしながら、勉強の時にはコードはすべて自分の手で打つことが、記憶に残りやすくより効果的である、というのが筆者の意見である。

6.7. CPU vs GPU の簡易ベンチマーク

実際に、ベンチマークを取ることで GPU と CPU の速度を比較をしてみよう。実行時間を計測するツールとして、Jupyter の提供する `%time` マジックコマンドを利用する。

まずは、CPU を使用して、 10000×10000 の行列の行列積を計算した場合の速度を測ってみよう。先ほどのノートブックの続きに、次のコードを実行する。

```
1 s = 10000
2 device = "cpu"
3 x = torch.rand(s, s, device=device, dtype=torch.float32)
4 y = torch.rand(s, s, device=device, dtype=torch.float32)
5
6 %time z = torch.matmul(x,y)
```

出力は以下のようなものが得られるだろう。これは、行列積の計算に実時間で 5.8 秒かかったことを意味する（実行のたびに計測される時間はばらつくことに留意）。

```
CPU times: user 11.5 s, sys: 140 ms, total: 11.6 s
Wall time: 5.8 s
```

次に、GPU を使用して、同じ演算を行った場合の速度を計測しよう。

```
1 s = 10000
2 device = "cuda"
3 x = torch.rand(s, s, device=device, dtype=torch.float32)
4 y = torch.rand(s, s, device=device, dtype=torch.float32)
5 torch.cuda.synchronize()
6
7 %time z = torch.matmul(x,y); torch.cuda.synchronize()
```

出力は以下のようなものになるだろう。GPU では 553 ミリ秒で計算を終えることができた！

```
CPU times: user 334 ms, sys: 220 ms, total: 554 ms
Wall time: 553 ms
```



PyTorchにおいて、GPUでの演算は asynchronous（非同期）で実行される。その理由で、上のベンチマークコードでは、`torch.cuda.synchronize()` というステートメントを埋め込んである。



このベンチマークでは、`dtype=torch.float32` と指定することで、32bitの浮動小数点型を用いている。ディープラーニングの学習および推論の計算には、32bit型、場合によっては16bit型が使われるのが一般的である。これの主な理由として、教師データやミニバッチに起因するノイズが、浮動小数点の精度よりも大きいことがあげられる。32bit/16bitを採用することで、メモリー消費を抑えたり、計算速度の向上が達成できる。

上記のベンチマークから, GPUを用いることで, 約10倍のスピードアップを実現することができた. スピードアップの性能は, 演算の種類や行列のサイズに依存する. 行列積は, そのなかでも最も速度向上が見込まれる演算の一つである.

6.8. 実践ディープラーニング! MNIST手書き数字認識タスク

ここまで,AWS上でディープラーニングの計算をするための概念や前提知識をながながと説明してきたが,ついにここからディープラーニングの計算を実際に走らせてみる.

ここでは, 機械学習のタスクで最も初歩的かつ有名な **MNIST データセットを使った数字認識**を扱う (Figure 27). 0から9までの手書きの数字の画像が与えられ, その数字が何の数字なのかを当てる, というシンプルなタスクである.



Figure 27. MNIST 手書き数字データセット

今回は, MNIST 文字認識タスクを, **畳み込みニューラルネットワーク (Convolutional Neural Network; CNN)** を使って解く. ソースコードは [/handson/minist/pytorch/](#) にある `mnist.ipynb` と `simple_mnist.py` である. なお, このプログラムは, [PyTorch の公式 Example Project 集](#) を参考に, 少少の改変を行ったものである.

まず最初に, カスタムのクラスや関数が定義された `simple_mnist.py` をアップロードしよう (Figure 28). 画面右上の "Upload" ボタンをクリックし, ファイルを選択すればよい. この Python プログラムの中に, CNN のモデルや, 学習の各イテレーションにおけるパラメータの更新などが記述されている. 今回は, この中身を説明することはしないが, 興味のある読者は, 自分でソースコードを読んでみるとよい.



Figure 28. `simple_mnist.py` をアップロード

`simple_mnist.py` をアップロードできたら、次に新しい notebook を作成しよう。"conda_pytorch_p36" の環境を選択することを忘れずに。

新しいノートブックが起動したら、まず最初に、必要なライブラリをインポートしよう。

```

1 import torch
2 import torch.optim as optim
3 import torchvision
4 from torchvision import datasets, transforms
5 from matplotlib import pyplot as plt
6
7 # custom functions and classes
8 from simple_mnist import Model, train, test

```

`torchvision` パッケージには、MNIST データセットをロードするなどの便利な関数が含まれている。また、今回のハンズオンで使うカスタムのクラス・関数 (`Model`, `train`, `test`) のインポートを行っている。

次に、MNIST テストデータをダウンロードしよう。同時に、画像データの輝度の正規化も行っている。

```

1 transf = transforms.Compose([transforms.ToTensor(),
2                             transforms.Normalize((0.1307,), (0.3081,))])
3
4 trainset = datasets.MNIST(root='./data', train=True, download=True, transform=transf)
5 trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)
6
7 testset = datasets.MNIST(root='./data', train=False, download=True, transform=transf)
8 testloader = torch.utils.data.DataLoader(testset, batch_size=1000, shuffle=True)

```

今回扱う MNIST データは 28x28 ピクセルの正方形の画像(モノクロ)と、それぞれのラベル(0 - 9 の数字)の組で構成されている。いくつかのデータを抽出して、可視化してみよう。Figure 29 のような出力が得られるはずである。

```

1 examples = iter(testloader)
2 example_data, example_targets = examples.next()
3
4 print("Example data size:", example_data.shape)
5
6 fig = plt.figure(figsize=(10,4))
7 for i in range(10):
8     plt.subplot(2,5,i+1)
9     plt.tight_layout()
10    plt.imshow(example_data[i][0], cmap='gray', interpolation='none')
11    plt.title("Ground Truth: {}".format(example_targets[i]))
12    plt.xticks([])
13    plt.yticks([])
14 plt.show()

```

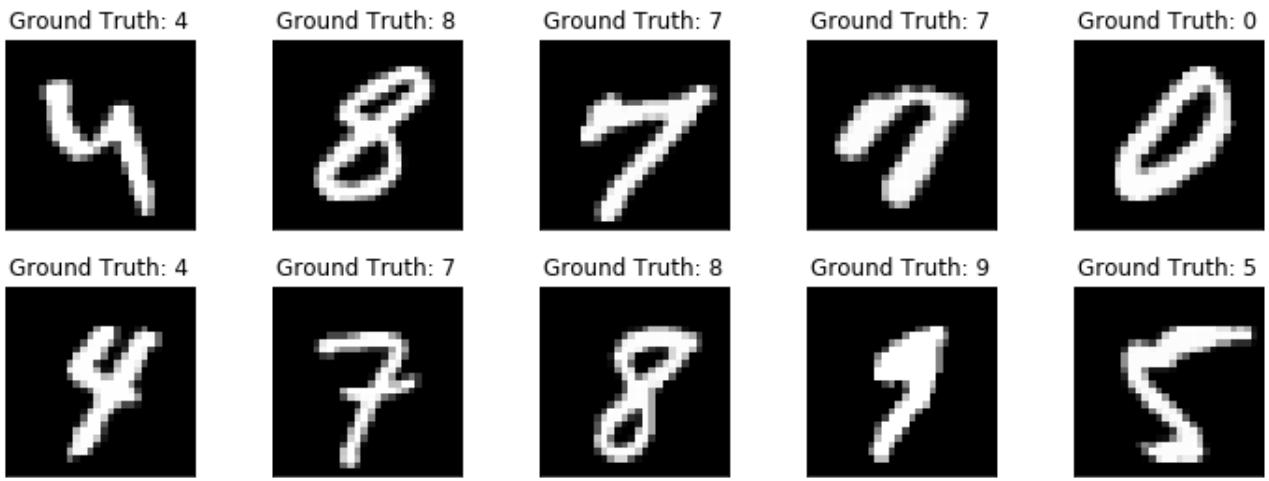


Figure 29. MNIST の手書き数字画像とその教師ラベル

次に, CNN のモデルを定義する.

```

1 model = Model()
2 model.to("cuda") # load to GPU

```

今回使う **Model** は [simple_mnist.py](#) の中で定義されている. このモデルは, [Figure 30](#) に示したような, 2層の畳み込み層と2層の全結合層からなるネットワークである. 出力層 (output layer) には Softmax 関数を使用し, 損失関数 (Loss function) には 負の対数尤度関数 (Negative log likelihood; NLL) を使用している.



Figure 30. 本ハンズオンで使用するニューラルネットの構造.

続いて, CNN のパラメータを更新する最適化アルゴリズムを定義する. ここでは, **Stochastic Gradient Descent (SGD)** を使用している.

```
1 optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
```

これで、準備が整った。CNNの学習ループを開始しよう！

```
1 train_losses = []
2 for epoch in range(5):
3     losses = train(model, trainloader, optimizer, epoch)
4     train_losses += losses
5     test(model, testloader)
6
7 plt.figure(figsize=(7,5))
8 plt.plot(train_losses)
9 plt.xlabel("Iterations")
10 plt.ylabel("Train loss")
```

ここでは5エポック分学習を行っている。GPUを使えば、これくらいの計算であれば1分程度で完了するだろう。

出力として、Figure 31のようなプロットが得られるはずである。イテレーションを重ねるにつれて、損失関数 (Loss function) の値が減少している（=精度が向上している）ことがわかる。

出力には各エポック終了後のテストデータに対する精度も表示されている。最終的には98%以上の極めて高い精度を実現できていることが確認できるだろう（Figure 32）。

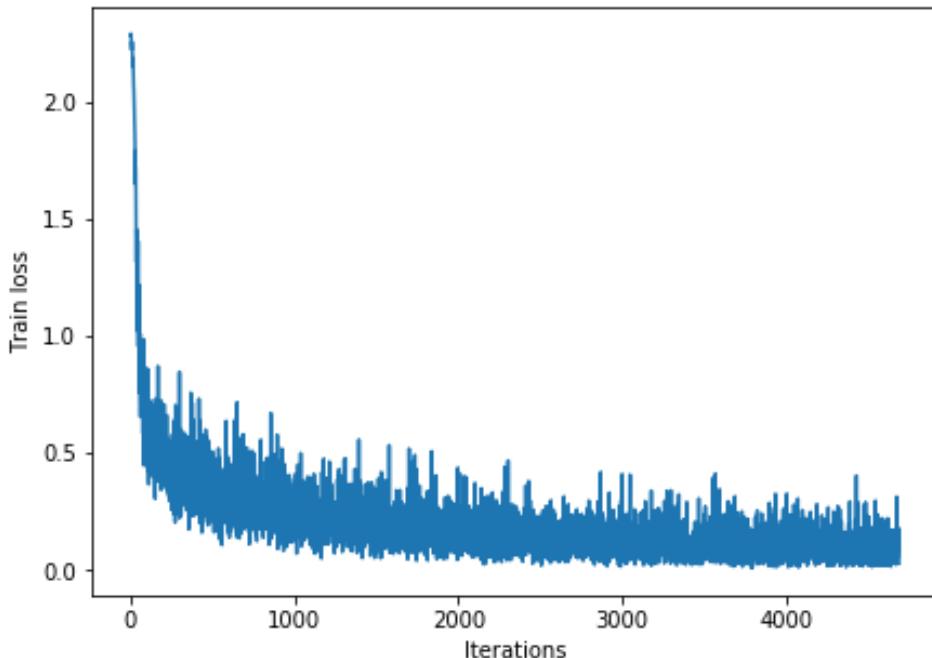


Figure 31. 学習の進行に対する Train loss の変化

Test set: Average loss: 0.0471, Accuracy: 59159/60000 (99%)

Figure 32. 学習したCNNのテストデータに対するスコア（5エポック後）

最後に、学習したCNNの推論結果を可視化してみよう。次のコードを実行することで、Figure 33のような出力が得られるだろう。この図で、下段右下などは、「1」に近い見た目をしているが、きちんと「9」と推論できている。なかなか賢いCNNを作り出すことができたようだ！

```

1 model.eval()
2
3 with torch.no_grad():
4     output = model(example_data.to("cuda"))
5
6 fig = plt.figure(figsize=(10,4))
7 for i in range(10):
8     plt.subplot(2,5,i+1)
9     plt.tight_layout()
10    plt.imshow(example_data[i][0], cmap='gray', interpolation='none')
11    plt.title("Prediction: {}".format(output.data.max(1, keepdim=True)[1][i].item()))
12    plt.xticks([])
13    plt.yticks([])
14 plt.show()

```

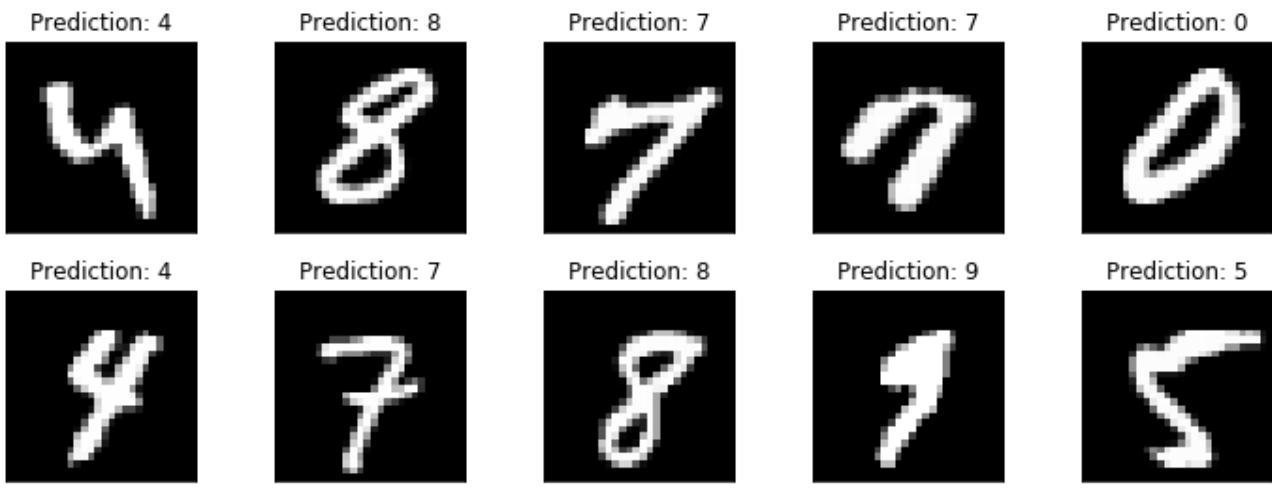


Figure 33. 学習した CNN による, MNIST 画像の推論結果

以上が, AWS クラウドの仮想サーバーを立ち上げ, 最初のディープラーニングの計算を行う一連の流れである. MNIST 文字認識のタスクを行うニューラルネットを, クラウド上の GPU を使って高速に学習させ, 現実的な問題を一つ解くことができたのである. 興味のある読者は, 今回のハンズオンを雛形に, 自分の所望の計算を走らせてみるとよいだろう.

6.9. スタックの削除

これにて, ハンズオン第二回の内容はすべて説明した. クラウドの利用料金を最小化するため, 使い終わったEC2インスタンスはすぐさま削除しよう.

ハンズオン第一回と同様に, AWS の CloudFormation コンソールか, AWS CLI により削除を実行する (詳細は [Section 4.4.8 参照](#)).

```
$ cdk destroy
```



スタックの削除は各自で必ず行うこと! 行わなかった場合, EC2インスタンスの料金が発生し続けることになる! `g4dn.xlarge` は \$0.526 / hour の料金設定なので, 一日起動しつづけると約\$12の請求が発生することになる!

Chapter 7. Docker 入門

Chapter 5, Chapter 6 にわたって, AWS 上に仮想サーバーを立ち上げ, 深層学習のプログラムを走らせる方法を紹介してきた. ここまでは, 単一のサーバーを立ち上げ, それに SSH でログインをして, コマンドを叩くことで計算を行ってきた. いわば, パーソナルコンピュータの延長のような形でクラウドを使ってきたわけである.

このような, インターネットのどこからでもアクセスできるパーソナルコンピュータとしてのクラウドという使い方も,もちろん便利であるし, いろいろな応用の可能性がある. しかし, これだけではクラウドの本当の価値は十分に發揮されていないと言うべきだろう. Chapter 2 で述べたように, 現代的なクラウドの一番の強みは自由に計算機の規模を拡大できることにある. すなわち, 多数のサーバーを同時に起動し, 複数のジョブを分散並列的に実行することで大量のデータを処理してこそ, クラウドの本領が発揮されるのである.

本章からはじまる3章を使って, クラウドを使うことでどのように大規模な計算システムを構築しビッグデータの解析に立ち向かうのか, その片鱗をお見せしたい. 特に, 前章で扱った深層学習を, どのようにビッグデータに適用していくかという点に焦点を絞って議論していきたい. そのための前準備として, 本章では Docker と呼ばれる計算機環境の仮想化ソフトウェアを紹介する. 現代のクラウドは Docker なしには成り立たないといって過言ではないだろう. クラウドに限らず, ローカルで行う計算処理にも Docker は大変便利である. クラウドからは少し話が離れるが, しっかりと理解して前に進んでもらいたい.

7.1. 機械学習の大規模化

先ほどから"計算システムの大規模化"と繰り返し唱えているが, それは具体的にはどのようなものを指しているのか? ここでは大規模データを処理するための計算機システムを, 機械学習を例にとって見てみよう.

Chapter 5 で紹介した GPT-3 のような, 超巨大な数のパラメータを有する深層学習モデルを学習させたいとしよう. そのような計算を行いたい場合, 一つのサーバーでは計算力が到底足りない. したがって, 典型的には Figure 34 に示すような計算システムの設計がなされる. すなわち, 大量の教師データを, 小さなチャンクとして複数のマシンに分散し, 並列的にニューラルネットのパラメータを最適化していくという構造である.



Figure 34. 複数の計算機を使った大規模な深層学習モデルの訓練

あるいは, 学習済みのモデルを大量のデータに適用し, 解析を行いたいとしよう. 例えば, SNS のプラットフォームで大量の画像が与えられて, それぞれの写真に何が写っているのかをラベルづけする, などのアプリケーションを想定できる. そのような場合は, Figure 35 のようなアーキテクチャが考えられるだろう. すなわち, 大量のデータ

を複数のマシンで分割し、それぞれのマシンで推論の計算を行うという構造である。

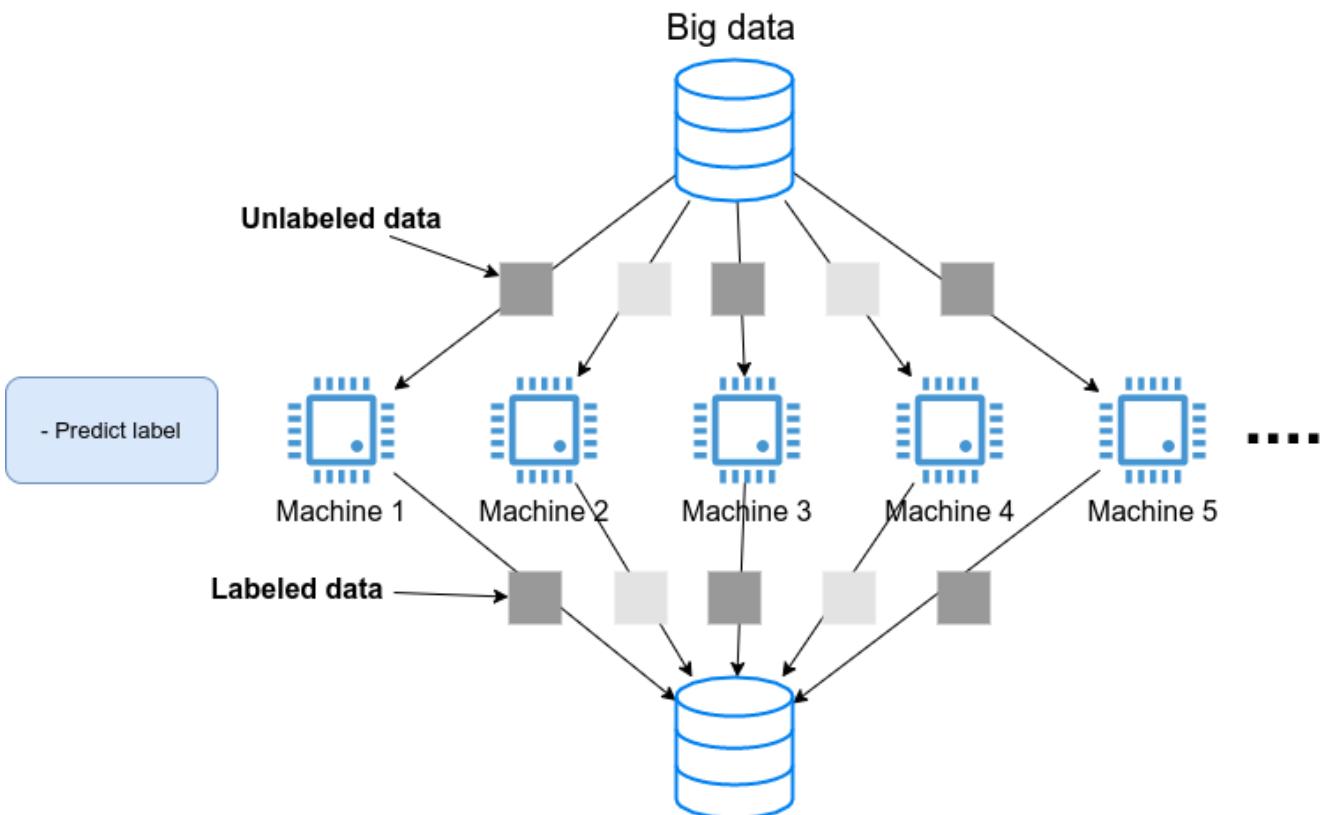


Figure 35. 複数の計算機による深層学習モデルを使った推論計算

このような複数の計算機を同時に走らせるようなアプリケーションをクラウド上で実現するには、どのようにすればよいのだろうか？

ひとつ重要なポイントとして、Figure 34 や Figure 35 で起動している複数のマシンは、**基本的に全く同一のOS・計算環境を有している**点である。ここで、個人のコンピュータでやるようなインストールの操作を、各マシンで行うこともできるが、それは大変な手間であるし、メンテナンスも面倒だろう。すなわち、大規模な計算システムを構築するには、**簡単に計算環境を複製できるような仕組み**が必要であるということがわかる。

そのような目的を実現するために使われるのが、Dockerと呼ばれるソフトウェアである。

7.2. Docker とは



Figure 36. Docker のロゴ

Docker とは、**コンテナ (Container)** と呼ばれる仮想環境下で、ホストOSとは独立した別の計算環境を走らせるためのソフトウェアである。Docker を使うことで、OS を含めた全てのプログラムをコンパクトにパッケージングすることが可能になる（パッケージされたひとつの計算環境のことを **イメージ (Image)** と呼ぶ）。Docker を使うことで、クラウドのサーバー上に瞬時に計算環境を複製することが可能になり、上で見たような複数の計算機を同時に走らせるためのシステムが実現できる。

Docker は2013年に Solomon Hykes らを中心開発され、以降爆発的に普及し、クラウドコンピューティングだけでなく、機械学習・科学計算の文脈などで、欠かすことのできないソフトウェアとなった。概念としては、Docker は仮想マシン (Virtual machine; VM) にとても近い。ここでは、VMとの対比をしながら、Docker とはなにかを簡

単に説明しよう。

仮想マシン(VM)とは、ホストとなるマシンの上に、仮想化されたOSを走らせる技術である(Figure 37)。VMにはハイパーバイザ(Hypervisor)と呼ばれるレイヤーが存在する。Hypervisorはまず、物理的な計算機リソース(CPU, RAM, networkなど)を分割し、仮想化する。例えば、ホストマシンに物理的なCPUが4コアあるとして、ハイパーバイザはそれを(2,2)個の組に仮想的に分割することができる。VM上で起動するOSには、ハイパーバイザによって仮想化されたハードウェアが割り当てられる。VM上で起動するOSは基本的に完全に独立であり、例えばOS-AはOS-Bに割り当てられたCPUやメモリー領域にアクセスすることはできない。VMを作成するための有名なソフトウェアとしては、[VMware](#), [VirtualBox](#), [Xen](#)などがある。また、これまで触ってきたEC2も、基本的にVM技術をすることで所望のスペックを持った仮想マシンがユーザーに提示される。

Dockerも、VMと同様に、仮想化されたOSをホストのOS上に走らせるための技術である。VMに対し、Dockerではハードウェアレベルの仮想化は行われておらず、すべての仮想化はソフトウェアレベルで実現されている(Figure 37)。Dockerで走る仮想OSは、多くの部分をホストのOSに依存しており、結果として非常にコンパクトである。その結果、Dockerで仮想OSを起動するために要する時間は、VMに比べて圧倒的に早い。また、パッケージ化された環境(=image)のサイズも完全なOSに比べ圧倒的に小さくなるので、ネットワークを通じたやり取りが非常に高速化される点も重要である。加えて、VMのいくつかの実装では、メタル(仮想化マシンに対して、物理的なハードウェア上で直接起動する場合のこと)と比べ、ハイパーバイザーレイヤでのオーバーヘッドなどにより性能が低下することが知られているが、Dockerではメタルとほぼ同様の性能を引き出すことができるとされている。

その他、VMとの相違点などはたくさんあるのだが、ここではこれ以上詳細には立ち入らない。大事なのは、Dockerとはとてもコンパクトかつハイパフォーマンスな仮想計算環境を作るツールである、という点である。その手軽さゆえに、2013年の登場以降、クラウドシステムでの利用が急速に増加し、現代のクラウドでは欠くことのできない中心的な技術になっている。

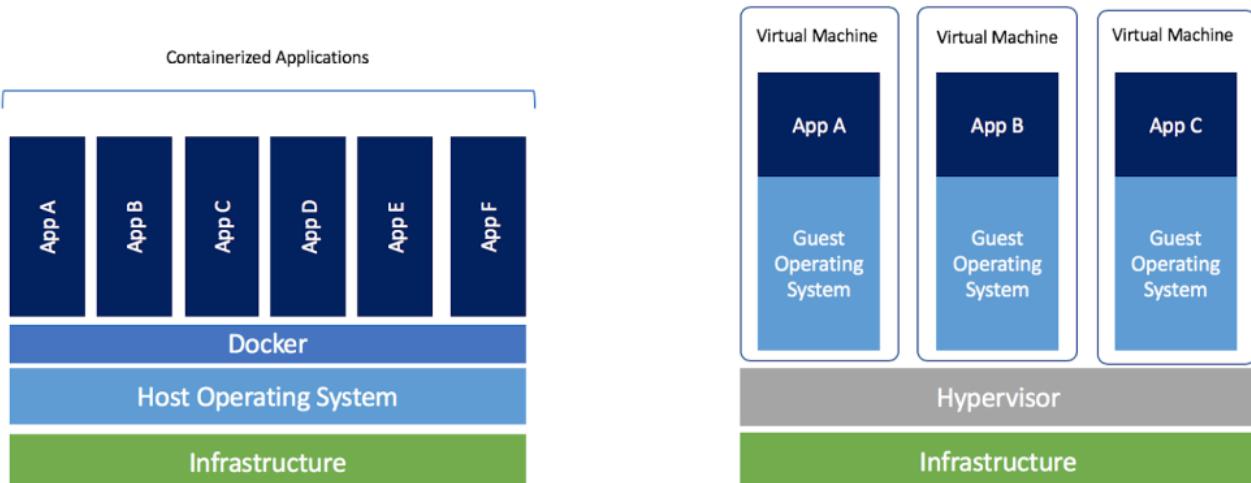


Figure 37. Docker と VM の比較 (画像出典: <https://www.docker.com/blog/containers-replacing-virtual-machines/>)

コラム: プログラマー三種の神器?

職業的プログラマーにとっての"三種の神器"とはなんだろうか? 多様な意見があると思うが,筆者は **Git**, **Vim** そして **Docker** を挙げたい.

Git は多くの読者がご存じの通り, コードの変更を追跡するためのシステムである. Linux の作成者である Linus Torvalds によって2005年に誕生した. チームでの開発を進める際には欠かせないツールだ.

Vim は1991年から30年以上の間プログラマーたちに愛されてきたテキストエディターである.

[Stackoverflow が行った2019年のアンケート](#)によると, 開発環境の部門で5位の人気を獲得している. たくさんのショートカットと様々なカスタム設定が提供されているので, 初見の人にはなかなかハードルが高いが, 一度マスターすれば他のモダンなエディターや統合開発環境に負けない, あるいはそれ以上の開発体験を実現することができる.

これらの十年以上の歴史あるツールに並んで, 第三番目の三種の神器として挙げたいのが Docker だ. Docker はプログラマーの開発のワークフローを一変させた. 例えば, プロジェクトごとに Docker image を作成することで, どの OS でも全く同じ計算環境で開発・テストを実行することができるようになった. また, [DevOps](#) や [CI / CD](#) といった最近の開発ワークフローも Docker のようなコンテナ技術の存在に立脚している. さらには Serverless computing ([Chapter 11](#)) といった概念も, コンテナ技術の生んだ大きな技術革新といえる.

あなたにとっての三種の神器はなんだろうか? また, 今後の未来ではどんな新しいツールが三種の神器としてプログラマーのワークフローを革新していくだろうか?

7.3. Docker チュートリアル

Docker とはなにかを理解するためには, 実際に触って動かしてみるのが一番有効な手立てである. ここでは, Docker の簡単なチュートリアルを行っていく.

Docker のインストールについては, [Section 1.4](#) および [公式のドキュメンテーション](#) を参照してもらいたい. Docker のインストールが完了している前提で, 以下は話を進めるものとする.

7.3.1. Image をダウンロード

パッケージ化された Docker の仮想環境 (**Image** と呼ぶ) は, [Docker Hub](#) からダウンロードできる. Docker Hub には, 個人や企業・団体が作成した Docker Image が集められており, GitHub などと同じ感覚で, オープンな形で公開されている.

例えば, Ubuntu の Image は [このリンク](#) で公開されており, **pull** コマンドを使うことでローカルにダウンロードすることができる.

```
$ docker pull ubuntu:18.04
```

ここで, イメージ名の : (コロン) 以降に続く文字列を **タグ (tag)** と呼び, 主にバージョンを指定するなどの目的で使われる.



pull コマンドはデフォルトでは Docker Hub でイメージを検索し, ダウンロードを行う. Docker image を公開するためのデータベース (registry と呼ぶ) は Docker Hub だけではなく, 例えば GitLab や GitHub は独自の registry 機能を提供しているし, 個人のサーバーで registry を立ち上げることも可能である. Docker Hub 以外の registry から pull するには, `myregistry.local:5000/testing/test-image` のように, イメージ名の先頭につける形で registry のアドレス (とオプションとしてポート番号) を指定すればよい.

7.3.2. Image を起動

Pullしてきた Image を起動するには, **run** コマンドを使う.

```
$ docker run -it ubuntu:18.04
```

ここで, **-it** とは, インタラクティブな shell のセッションを開始するために必要なオプションである.

上のコマンドを実行すると, 仮想化された Ubuntu が起動され, コマンドラインからコマンドが打ち込めるようになる (Figure 38). なお, このように起動状態にある計算環境 (ランタイム) のことを **Container (コンテナ)** と呼ぶ.

```
tomoyuki@eiffel:~$ docker run -it ubuntu:18.04
root@3d7e5903b640:/# █
```

Figure 38. Docker を使って ubuntu:18.04 イメージを起動

上で使った **ubuntu:18.04** のイメージは, 空の Ubuntu OS だが, 既にプログラムがインストール済みのものもある. これは, Chapter 6 でみた DLAMI と概念として似ている. たとえば, pytorch がインストール済みの Image は [こちら](#) で公開されている.

これを起動してみよう.

```
$ docker run -it pytorch/pytorch
```



docker run を実行したとき, ローカルに該当する Image がない場合は, 自動的に Docker Hub からダウンロードされる

pytorch の container が起動したら, Python のシェルを立ち上げて, pytorch をインポートしてみよう.

```
$ python3
Python 3.7.7 (default, May  7 2020, 21:25:33)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> torch.cuda.is_available()
False
```

このように, Docker を使うことで簡単に特定のOS・プログラムの入った計算環境を再現することが可能になる.

7.3.3. 自分だけの Image を作る

自分の使うソフトウェア・ライブラリがインストールされた, 自分だけの Image を作ることも可能である.

例えば, [本書で提供している docker image](#) には, Python, Node.js, AWS CLI, AWS CDK などのソフトウェアがインストール済みであり, ダウンロードしてくるだけですぐにハンズオンのプログラムが実行できるようになっている.

カスタムの docker image を作るには, **Dockerfile** という名前のついたファイルを用意し, その中にどんなプログラムをインストールするなどを記述していく.

具体例として, 本書で提供している Docker image のレシピを見てみよう ([/docker/Dockerfile](#)).

```

FROM node:12
LABEL maintainer="Tomoyuki Mano"

RUN apt-get update \
    && apt-get install nano

①
RUN cd /opt \
    && curl -q "https://www.python.org/ftp/python/3.7.6/Python-3.7.6.tgz" -o Python-3.7.6.tgz \
    && tar -xzf Python-3.7.6.tgz \
    && cd Python-3.7.6 \
    && ./configure --enable-optimizations \
    && make install

RUN cd /opt \
    && curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip" \
    && unzip awscliv2.zip \
    && ./aws/install

②
RUN npm install -g aws-cdk@1.100

# clean up unnecessary files
RUN rm -rf /opt/*

# copy hands-on source code in /root/
COPY handson/ /root/handson

```

Dockerfile の中身の説明は特に行わないが、例えば上のコードで <1> で示したところは、Python 3.7 のインストールを実行している。また、<2> で示したところは、AWS CDK のインストールを行っていることがわかるだろう。このように、リアルな OS で行うのと同じ流れでインストールのコマンドを逐一記述していくことで、自分だけの Docker image を作成することができる。一度 image を作成すれば、それを他人に渡すことで、他者も同一の計算環境を簡単に再構成することができる。

"ぼくの環境ではそのプログラム走ったのにな…" というのは、プログラミング初心者ではよく耳にする会話だが、docker を使いこなせばそのような心配とは無縁である。そのような意味で、クラウド以外の場面でも、Docker の有用性・汎用性は極めて高い。

コラム: Is Docker alone?

コンテナを用いた仮想計算環境ツールとして Docker を紹介したが、他に選択肢はないのか？よくぞ聞いてくれた！Docker の登場以降、複数のコンテナベースの仮想環境ツールが開発されてきた。いずれのツールも、概念や API については Docker と共通するものが多いが、Docker にはない独自の特徴を提供している。ここではその中でも有名ないくつかを紹介しよう。

[Singularity](#) は科学計算や HPC (High Performance Computing) の分野で人気の高いコンテナプラットフォームである。Singularity では大学・研究機関の HPC クラスターでの運用に適したような設計が施されている。例えば、Docker は基本的には root 権限で実行されるのに対し、Singularity はユーザー権限（コマンドを実行したユーザー自身）でプログラムが実行される。root 権限での実行は Web サーバーのように個人・企業がある特定のサービスのために運用するサーバーでは問題ないが、多数のユーザーが多様な目的で計算を実行する HPC クラスターでは問題となる。また、Singularity は独自のイメージの作成方法・エコシステムを持っているが、Docker イメージを Singularity のイメージに変換し実行する機能も有している。

[podman](#) は Red Hat 社によって開発されたもう一つのコンテナプラットフォームである。podman は基本的に Docker と同一のコマンドを採用しているが、実装は Red Hat によってスクラッチから行われた。podman では、Singularity と同様にユーザー権限でのプログラムの実行を可能であり、クラウドおよび HPC の両方の環境に対応するコンテナプラットフォームを目指して作られた。また、その名前にある通り pod と呼ばれる独自の概念が導入されている。

著者の個人的な意見としては、現時点では Docker をマスターしておけば当面は困ることはないと考えるが、興味のある読者はぜひこれらのツールも試してみてはいかがだろうか？

7.4. Elastic Container Service (ECS)



Figure 39. ECS のロゴ

以上で説明したように、Docker を使うことで仮想計算環境を簡単に複製・起動することが可能になる。本章の最後の話題として、AWS 上で Docker を使った計算システムを構築する方法を解説しよう。

Elastic Container Service (ECS) とは、Docker を使った計算機クラスターを AWS 上に作成するためのツールである。ECS の概要を示したのが [Figure 40](#) である。

ECS は、タスク (Task) と呼ばれる単位で管理された計算ジョブを受け付ける。システムにタスクが投入されると、ECS はまず最初にタスクで指定された Docker イメージを外部レジストリからダウンロードしてくれる。外部レジストリとしては、DockerHub や AWS 独自の Docker レジストリである **ECR (Elastic Container Registry)** を指定することができる。

ECS の次の重要な役割はタスクの配置である。予め定義されたクラスター内で、計算負荷が小さい仮想インスタンスを選び出し、そこに Docker イメージを配置することで指定された計算タスクが開始される。“計算負荷が小さい仮想インスタンスを選び出す”と言ったが、具体的にどのような戦略・ポリシーでこの選択を行うかは、ユーザーの指定したパラメータに従う。

またクラスターのスケーリングも ECS における重要な概念である。スケーリングとは、クラスター内のインスタンスの計算負荷をモニタリングし、計算負荷に応じてインスタンスの起動・停止を行う操作を指す。クラスター全体の計算負荷が指定された閾値（例えば 80% の稼働率）を超えていた場合、新たな仮想インスタンスをクラスター内に立ち上げる操作を **scale-out**（スケールアウト）と呼び、負荷が減った場合に不要なインスタンスを停止する操作を **scale-in**（スケールイン）と呼ぶ。クラスターのスケーリングは、ECS が他の AWS のサービスと連携すること

で実現される。具体的には、EC2 の **Auto scaling group (ASG)** や **Fargate** の2つの選択肢が多くの場合選択される。ASGについては[Chapter 9](#), Fargateについては[Chapter 8](#) でより詳細に解説する。

これら一連のタスクの管理を、ECS は自動でやってくれる。クラスターのスケーリングやタスクの配置に関してのパラメータを一度指定てしまえば、ユーザーは(ほとんどなにも考えずに) 大量のジョブを投入することができる。クラスターのスケーリングによってジョブの量にちょうど十分なだけのインスタンスが起動し、ジョブが完了した後は不要なインスタンスはインスタンスすべて停止される。

さて、ここまで説明的な話が続いてしまったが、次章からは早速 Docker と AWS を使って大規模な並列計算システムを構築していこう!

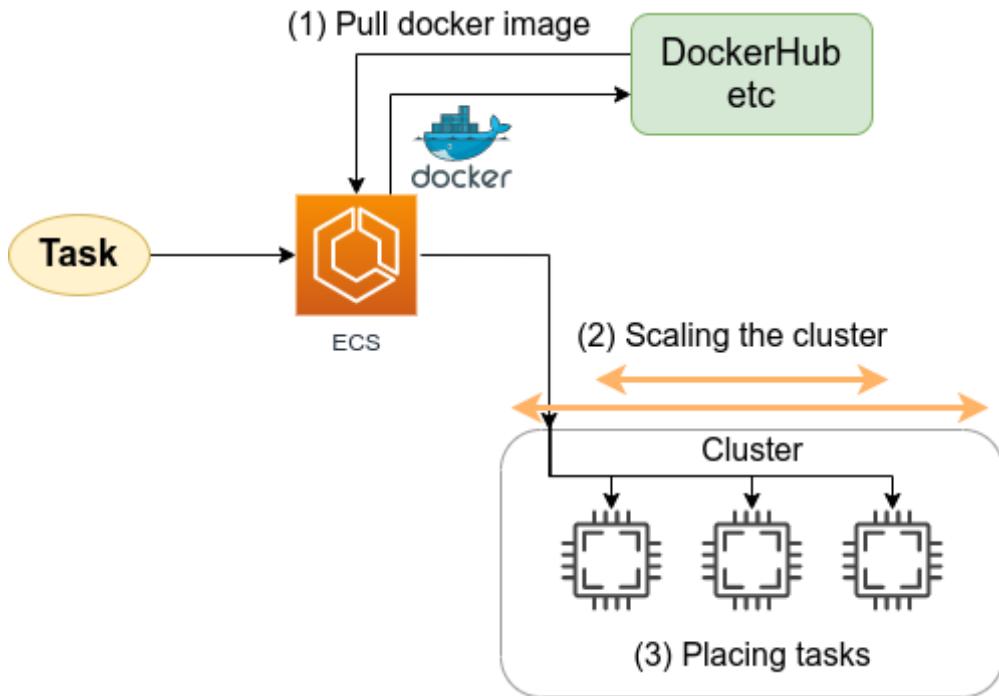


Figure 40. ECS の概要

Chapter 8. Hands-on #3: AWSで自動質問回答ボットを走らせる

ハンズオン第三回では、前章で学んだ Docker と ECS を使うことで、大規模化可能な機械学習システムの最もシンプルなものを実装する。

具体的には、Transformer と呼ばれるディープラーニングのモデルを使った自然言語処理を利用してすることで、英語で与えられた質問への回答を自動で生成するボットを作成してみる。特に、何百何千もの質問に同時にに対応できるように、単一のサーバーに展開するのではなく、リクエストに応じて複数のサーバーを自動的に起動し、並列でジョブを実行させるシステムを設計する。まさに、初步的ながら、Siri・Alexa・Google assistant のようなシステムを作り上げるのである！

ハンズオンのソースコードは [こちらのリンク](#) にある。



通常の機械学習のワークフローでは、モデルの訓練 ⇒ 推論（データへの適用）が基本的な流れである。しかしながら、GPU を使ったモデルの訓練はやや難易度が高いため、次章（Chapter 9）で取り扱う。本章は、クラウド上でのクラスターの構築・タスクの管理などの概念に慣れるため、よりシンプルな実装で実現できる推論計算の並列化を紹介する。



このハンズオンでは 1CPU/4GB RAM の Fargate インスタンスを使用する。計算の実行には 0.025 \$/hour のコストが発生することに注意。

8.1. Fargate



Figure 41. Fargate のロゴ

ハンズオンに入っていく前に、**Fargate** という AWS の機能を知っておく必要がある。

ECS の概要を示した Figure 40 をもう一度見てみよう。この図で、ECS の管理下にあるクラスターが示されているが、このクラスターの中で計算を行う実体としては二つの選択肢がある。EC2 あるいは Fargate のいずれかである。EC2 を用いた場合は、先の章（Chapter 4, Chapter 6）で説明したような流れでインスタンスが起動し、計算が実行される。しかし、EC2 クラスターの作成・管理は技術的な難易度がやや高いので、次章（Chapter 9）で説明することにする。

Fargate とは、ECS での利用に特化して設計された、コンテナを使用した計算タスクを走らせるための仕組みである。計算を走らせるという点では EC2 と役割は似ているが、Fargate は EC2 インスタンスのような物理的実体は持たない。ECS の利用に専用設計がされているがために、ECS のクラスターとして Fargate を選択すると、とても簡単な設定・プログラムで、非常にパワフルかつスケーラブルな計算システムを構築することができる。

Fargate では、EC2 と同様に CPU とメモリーのサイズを必要な分だけ指定できる。執筆時点では、CPU は 0.25 - 4 コア、RAM は 0.5 - 30 GB の間で選択することができる（[公式ドキュメンテーション参照](#)）。クラスターのスケーリングが容易な分、Fargate では EC2 ほど大きな CPU コア・RAM 容量を单一インスタンスに付与することができず、また GPU を利用することもできない。

以上が Fargate の概要であったが、くどくど言葉で説明してもなかなかピンとこないだろう。ここからは実際に手を動かしながら、ECS と Fargate を使った並列タスクの処理の仕方を学んでいこう。



厳密には、ECS に付与するクラスターには EC2 と Fargate のハイブリッドを使用することも可能である。



"Fargate は EC2 インスタンスのような物理的実体は持たない", と上に書いたが, 少し語弊があるかもしれない。Fargate も, データセンターのどこかの CPU 上で起動するので, 物理的な実体はどこかには必ず存在する。しかし, EC2 インスタンスと違って, 例えば SSH でログインすることは基本的に想定されていないし, なにかのソフトウェアをインストールしたりなどの概念も存在しない(基本的に Docker のコンテナを介してすべてのプログラムが実行される)。その意味で "物理的実体を持たない" と表現したのである。実は, Fargate は Chapter 11 で紹介する, serverless architecture の重要な構成要素の一つである。

8.2. 準備

本ハンズオンの実行には, 第一回ハンズオンで説明した準備 (Section 4.1) が整っていることを前提とする。それ以外に必要な準備はない。

8.3. Transformer を用いた question-answering プログラム

このハンズオンで開発する, 自動質問回答システムをより具体的に定義しよう。次のような文脈 (context) と質問 (question) が与えられた状況を想定する。

context: Albert Einstein (14 March 1879 – 18 April 1955) was a German-born theoretical physicist who developed the theory of relativity, one of the two pillars of modern physics (alongside quantum mechanics). His work is also known for its influence on the philosophy of science. He is best known to the general public for his mass – energy equivalence formula $E = mc^2$, which has been dubbed \"the world's most famous equation\". He received the 1921 Nobel Prize in Physics \"for his services to theoretical physics, and especially for his discovery of the law of the photoelectric effect\", a pivotal step in the development of quantum theory.

question: In what year did Einstein win the Nobel prize?

今回作成する自動回答システムは, このような問題に対して, context に含まれる文字列から正解となる言葉を見つけ出すものとする。上の問題では, 次のような回答を返すべきである。

answer: 1921

人間にとては, このような文章を理解することは容易であるが, コンピュータにそれをやらせることはなかなか難しいことは容易に想像ができるだろう。しかし, 近年の深層学習を使った自然言語処理の進歩は著しく, 上で示したような例題などは, 極めて高い正答率で回答できるモデルを作ることができる。

今回は, [huggingface/transformers](#) で公開されている学習済みの言語モデルを利用することで, 上で定義した問題を解く Q&A ボットを作る。この Q&A ボットは **Transformer** と呼ばれるモデルを使った自然言語処理に支えられている (Figure 42)。このプログラムを, Docker にパッケージしたものが <https://hub.docker.com/repository/docker/tomomano/qabot> に用意してある。クラウドの設計に入る前に, まずはこのプログラムを単体で動かしてみよう。

なお, 今回は学習済みのモデルを用いているので, 私達が行うのは, 与えられた入力をモデルに投入して予測を行う(推論)のみである。推論の演算は, CPU だけでも十分高速に行うことができるので, コストの削減と, よりシンプルにシステムを設計をする目的で, このハンズオンでは GPU は利用しない。一般的に, ニューラルネットは学習のほうが圧倒的に計算コストが大きく, そのような場合に GPU はより威力を発揮する。



Figure 42. Transformer モデルアーキテクチャ (画像出典: [Vaswani+ 2017](#))

次のコマンドで、今回使う Docker image をローカルにダウンロード (pull) してこよう。

```
$ docker pull tomomano/qabot:latest
```

pull できたら、早速この Docker に質問を投げかけてみよう。

```
$ context="Albert Einstein (14 March 1879 – 18 April 1955) was a German-born theoretical physicist who developed the theory of relativity, one of the two pillars of modern physics (alongside quantum mechanics). His work is also known for its influence on the philosophy of science. He is best known to the general public for his mass–energy equivalence formula E = mc2, which has been dubbed the world's most famous equation. He received the 1921 Nobel Prize in Physics for his services to theoretical physics, and especially for his discovery of the law of the photoelectric effect, a pivotal step in the development of quantum theory."
$ question="In what year did Einstein win the Nobel prize ?"
$ docker run tomomano/qabot "${context}" "${question}" foo --no_save
```

今回用意した Docker image は、第一引数に context となる文字列を、第二引数に question に相当する文字列を受けつける。第三引数、第四引数については、クラウドに展開するときの実装上の都合なので、今は気にしなくてよい。

上のコマンドを実行すると、以下のような出力が得られるはずである。

```
{'score': 0.9881729286683587, 'start': 437, 'end': 441, 'answer': '1921'}
```

"score" は正解の自信度を表す数字で, [0,1] の範囲で与えられる. "start", "end" は, context 中の何文字目が正解に相当するかを示しており, "answer" が正解と予測された文字列である.

1921 年という, 正しい答えが返ってきてることに注目してほしい.

もう少し難しい質問を投げかけてみよう.

```
$ question="Why did Einstein win the Nobel prize ?"  
$ docker run tomomano/qabot "${context}" "${question}" foo --no_save
```

出力:

```
{'score': 0.5235594527494207, 'start': 470, 'end': 506, 'answer': 'his services to theoretical physics,'}
```

今度は, score が 0.52 と, 少し自信がないようだが, それでも正しい答えにたどりつけていることがわかる.

このように, ディープラーニングに支えられた言語モデルを用いることで, なかなかに賢い Q-A ボットを実現できていることがわかる. 以降では, このプログラムをクラウドに展開することで, 大量の質問に自動で対応できるようなシステムを設計していく.



今回使用する Question & Answering システムには, DistilBERT という Transformer を基にした言語モデルが用いられている. 興味のある読者は, [原著論文](#) を参照してもらいたい. また, [huggingface/transformers](#) の DistilBert についてのドキュメンテーションは [こちら](#).



[huggingface/transformers](#) には, 様々な最新の言語モデルが実装されている. 解けるタスクも, question-answering だけではなく, 翻訳や要約など複数用意されている. 興味のある読者は, [ドキュメンテーション](#) を参照.



今回提供する Docker のソースコードは <https://github.com/tomomano/intro-aws-2021/tree/main/handson/qa-bot/docker> にある.

8.4. アプリケーションの説明

このハンズオンで作成するアプリケーションの概要を [Figure 43](#) に示す.

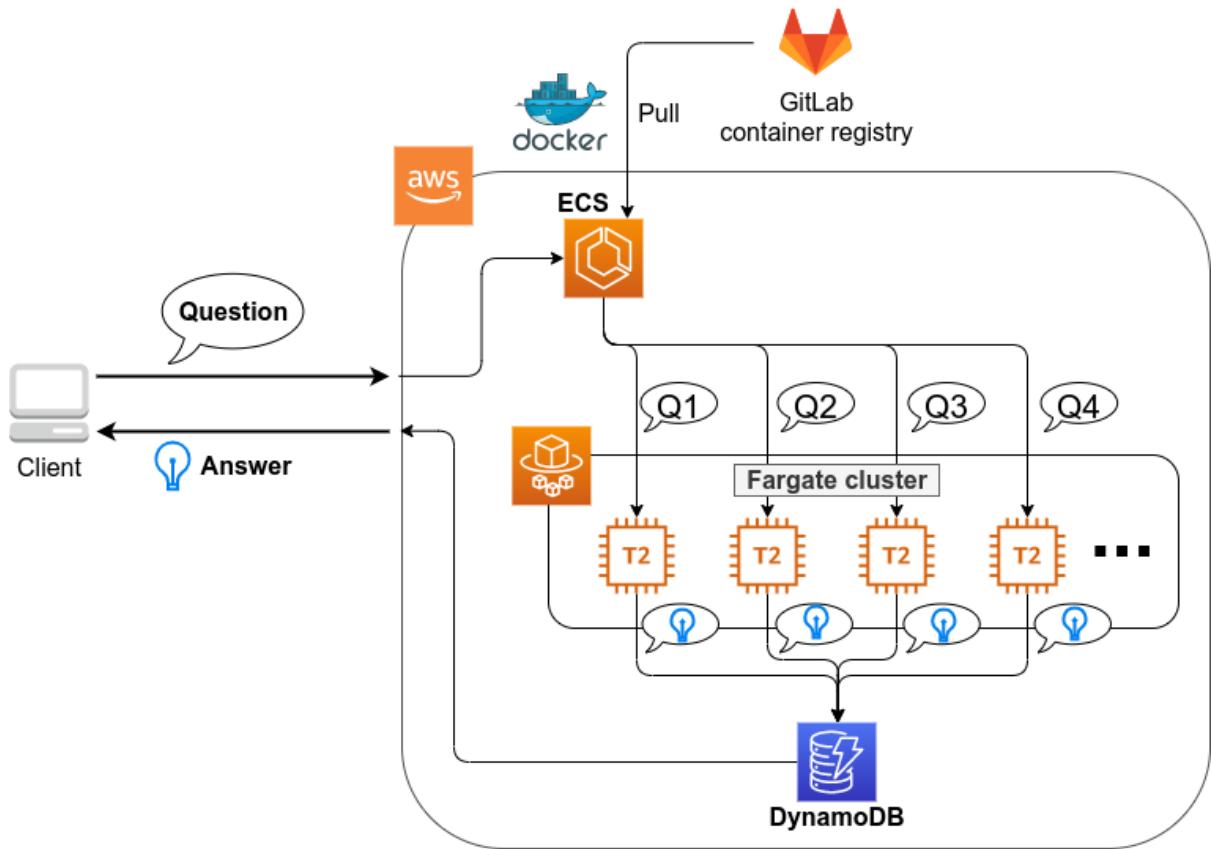


Figure 43. ハンズオン#3で作製するアプリケーションのアーキテクチャ

簡単にまとめると、以下のような設計である。

- ・ クライアントは、質問を AWS 上のアプリケーションに送信する。
- ・ 質問のタスクは ECS によって処理される。
- ・ ECS は、GitLab container registry から、Docker image をダウンロードする。
- ・ 次に、ECS はクラスター内に新たな仮想インスタンスを立ち上げ、ダウンロードされた Docker image をこの新規インスタンスに配置する。
 - このとき、ひとつの質問に対し一つの仮想インスタンスを立ち上げることで、複数の質問を並列的に処理できるようにする。
- ・ ジョブが実行される。
- ・ ジョブの実行結果(質問への回答)は、データベース(DynamoDB)に書き込まれる。
- ・ 最後に、クライアントは DynamoDB から質問への回答を読み取る。

それでは、プログラムのソースコードを見てみよう ([/handson/03-qa-bot/app.py](#)).

```

1 class EcsClusterQaBot(core.Stack):
2
3     def __init__(self, scope: core.App, name: str, **kwargs) -> None:
4         super().__init__(scope, name, **kwargs)
5
6         ①
7         # dynamoDB table to store questions and answers
8         table = dynamodb.Table(
9             self, "EcsClusterQaBot-Table",
10            partition_key=dynamodb.Attribute(
11                name="item_id", type=dynamodb.AttributeType.STRING
12            ),
13            billing_mode=dynamodb.BillingMode.PAY_PER_REQUEST,
14            removal_policy=core.RemovalPolicy.DESTROY
15        )
16
17         ②
18         vpc = ec2.Vpc(
19             self, "EcsClusterQaBot-Vpc",
20             max_azs=1,
21         )
22
23         ③
24         cluster = ecs.Cluster(
25             self, "EcsClusterQaBot-Cluster",
26             vpc=vpc,
27         )
28
29         ④
30         taskdef = ecs.FargateTaskDefinition(
31             self, "EcsClusterQaBot-TaskDef",
32             cpu=1024, # 1 CPU
33             memory_limit_mib=4096, # 4GB RAM
34         )
35
36         # grant permissions
37         table.grant_read_write_data(taskdef.task_role)
38         taskdef.add_to_task_role_policy(
39             iam.PolicyStatement(
40                 effect=iam.Effect.ALLOW,
41                 resources=["*"],
42                 actions=["ssm:GetParameter"]
43             )
44         )
45
46         ⑤
47         container = taskdef.add_container(
48             "EcsClusterQaBot-Container",
49             image=ecs.ContainerImage.from_registry(
50                 "registry.gitlab.com/tomomano/intro-aws/handson03:latest"
51             ),
52         )

```

- ① ここでは、回答の結果を書き込むためのデータベースを用意している。DynamoDBについては、Serverless architecture の章で扱うので、今は気にしなくてよい。
- ② ここでは、ハンズオン #1, #2 で行ったのと同様に、VPC を定義している。
- ③ ここで、ECS のクラスター (cluster) を定義している。クラスターとは、仮想サーバーのプールのことであり、クラスターの中に複数の仮想インスタンスを配置する。
- ④ ここで、実行するタスクを定義している (task definition)。

⑤ ここで、タスクの実行で使用する Docker image を定義している。

8.4.1. ECS と Fargate

ECS と Fargate の部分について、コードをくわしく見てみてみよう。

```
1 cluster = ecs.Cluster(  
2     self, "EcsClusterQaBot-Cluster",  
3     vpc=vpc,  
4 )  
5  
6 taskdef = ecs.FargateTaskDefinition(  
7     self, "EcsClusterQaBot-TaskDef",  
8     cpu=1024, # 1 CPU  
9     memory_limit_mib=4096, # 4GB RAM  
10 )  
11  
12 container = taskdef.add_container(  
13     "EcsClusterQaBot-Container",  
14     image=ecs.ContainerImage.from_registry(  
15         "registry.gitlab.com/tomomano/intro-aws/handson03:latest"  
16     ),  
17 )
```

`cluster =` の箇所で、空の ECS クラスターを定義している。

次に、`taskdef=ecs.FargateTaskDefinition` の箇所で、Fargate インスタンスを使ったタスクを定義しており、特にここでは 1 CPU, 4GB RAM というマシンスペックを指定している。また、このようにして定義されたタスクは、デフォルトで1タスクにつき1インスタンスが使用される。

最後に、`container =` の箇所で、タスクの実行で使用する Docker image を定義している。ここでは、GitLab container registry に置いてある image をダウンロードしてくるよう指定している。

このようにわずか数行のコードであるが、これだけで上で説明したような、タスクのスケジューリングなどが自動で実行される。

上のコードで `cpu=1024` と指定されているのに注目してほしい。これは CPU ユニットと呼ばれる数で、以下の換算表に従って仮想CPU (virtual CPU; vCPU) が割り当てられる。1024 が 1 CPU に相当する。0.25 や 0.5 vCPU などの数字は、それぞれ実効的に 1/4, 1/2 の CPU 時間が割り当てられることを意味する。また、CPU ユニットによって使用できるメモリー量も変わってくる。例えば、1024 CPU ユニットを選択した場合は、2 から 8 GB の範囲でのみメモリー量を指定することができる。最新の情報は [公式ドキュメンテーション](#) を参照のこと。

Table 4. CPU ユニットと指定可能なメモリー量の換算表



CPU ユニット	メモリーの値
256 (.25 vCPU)	0.5 GB, 1 GB, 2 GB
512 (.5 vCPU)	1 GB, 2 GB, 3 GB, 4 GB
1024 (1 vCPU)	2 GB, 3 GB, 4 GB, 5 GB, 6 GB, 7 GB, 8 GB
2048 (2 vCPU)	Between 4 GB and 16 GB in 1-GB increments
4096 (4 vCPU)	Between 8 GB and 30 GB in 1-GB increments

8.5. スタックのデプロイ

スタックの中身が理解できたところで、早速スタックをデプロイしてみよう。

デプロイの手順は、これまでのハンズオンとほとんど共通である。SSHによるログインの必要がないので、むしろ単純なくらいである。ここでは、コマンドのみ列挙する（#で始まる行はコメントである）。それぞれの意味を忘れてしまった場合は、ハンズオン1、2に戻って復習していただきたい。

```
# プロジェクトのディレクトリに移動
$ cd intro-aws/handson/03-qa-bot

# venv を作成し、依存ライブラリのインストールを行う
$ python3 -m venv .env
$ source .env/bin/activate
$ pip install -r requirements.txt

# AWS の認証情報をセットする
# 自分自身の認証情報に置き換えること！
export AWS_ACCESS_KEY_ID=XXXXXX
export AWS_SECRET_ACCESS_KEY=YYYYYY
export AWS_DEFAULT_REGION=ap-northeast-1

# デプロイを実行
$ cdk deploy
```

デプロイのコマンドが無事に実行されれば、Figure 44 のような出力が得られるはずである。



```
✓ EcsClusterQaBot

Outputs:
EcsClusterQaBot.ClusterName = EcsClusterQaBot-EcsClusterQaBotCluster6488E31F-[REDACTED]
EcsClusterQaBot.TaskDefinitionArn = arn:aws:ecs:ap-northeast-1:606887060834:task-definition/EcsClusterQaBotEcsClusterQaBotTaskDef|[REDACTED]

Stack ARN:
arn:aws:cloudformation:ap-northeast-1:606887060834:stack/EcsClusterQaBot/f89b5980-b42d-1ac70
(.env) tomoyuki@balthasar:03-qa-bot$ █
```

Figure 44. CDKデプロイ実行後の出力

AWS コンソールにログインして、デプロイされたスタックを確認してみよう。コンソールから、ECS のページに行くと Figure 45 のような画面が表示されるはずである。

Cluster というのが、先ほど説明したとおり、複数の仮想インスタンスを束ねる一つの単位である。この時点ではひとつもタスクが走っていないので、タスクの数字はすべて0になっている。この画面にはまたすぐ戻ってくるので、開いたままにしておこう。

Figure 45. ECS コンソール画面

8.6. タスクの実行

それでは、早速、質問を実行してみよう。

ECS にタスクを投入するのはやや複雑なので、タスクの投入を簡単にするプログラム (`run_task.py`) を用意した ([/handson/03-qa-bot/run_task.py](#))。

次のようなコマンドで、ECS クラスターに新しい質問を投入することができる。

```
$ python run_task.py ask "A giant peach was flowing in the river. She picked it up and brought it home.  
Later, a healthy baby was born from the peach. She named the baby Momotaro." "What is the name of the  
baby?"
```



`run_task.py` を実行するには、環境変数によって AWS の認証情報が設定されていることが前提である。

"ask" の引数に続き、文脈 (context) と質問を引数として渡している。

上のコマンドを実行すると、"Waiting for the task to finish..." と出力が表示され、回答を得るまでしばらく待たされることになる。この間、AWS では、ECS がタスクを受理し、新しい Fargate のインスタンスを起動し、Docker image をそのインスタンスに配置する、という一連の処理がなされている。AWS コンソールから、この一連の様子をモニタリングしてみよう。

先ほどの ECS コンソール画面にもどり、クラスターの名前をクリックすることで、クラスターの詳細画面を開く。次に、"Tasks" という名前のタブがあるので、それを開く (Figure 46)。すると、実行中のタスクの一覧が表示されるだろう。

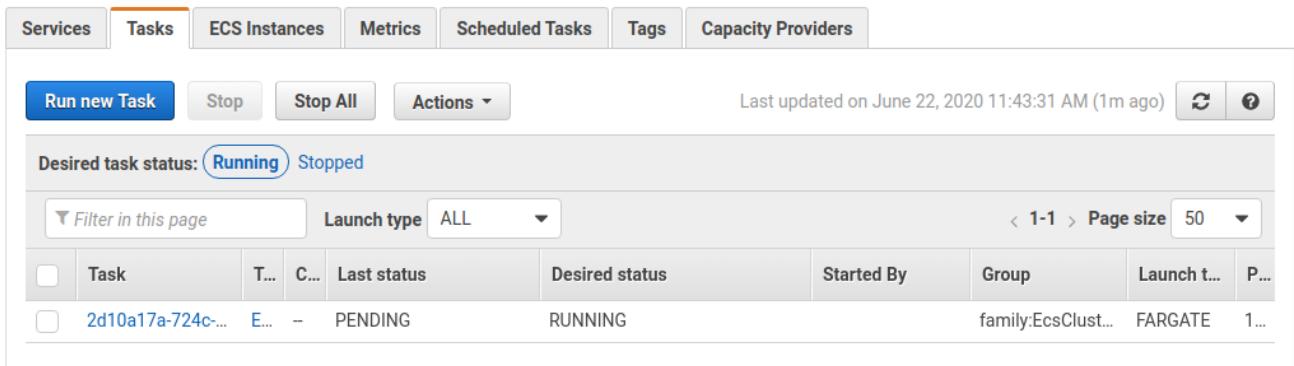


Figure 46. ECS のタスクの実行状況をモニタリング

[Figure 46](#) で見て取れるように, "Desired status = RUNNING", "Last status = PENDING" となっていることから, この時点では, タスクを実行するための準備している段階である, ということがわかる. Fargate のインスタンスを起動し, Docker image を配置するまでおよそ1-2分の時間がかかる.

しばらく待つうちに、Status が "RUNNING" に遷移し、計算が始まる。計算が終わると、Status は "STOPPED" に遷移し、ECS によって Fargate インスタンスは自動的にシャットダウンされる。

Figure 46 の画面から, "Task" の列にあるタスクIDクリックすることで, タスクの詳細画面を開いてみよう (Figure 47). "Launch type = FARGATE", "Last status = STOPPED" など, タスクの情報が表示されている. また, "Logs" のタブを開くことで, container の吐き出した実行ログを閲覧することができる.

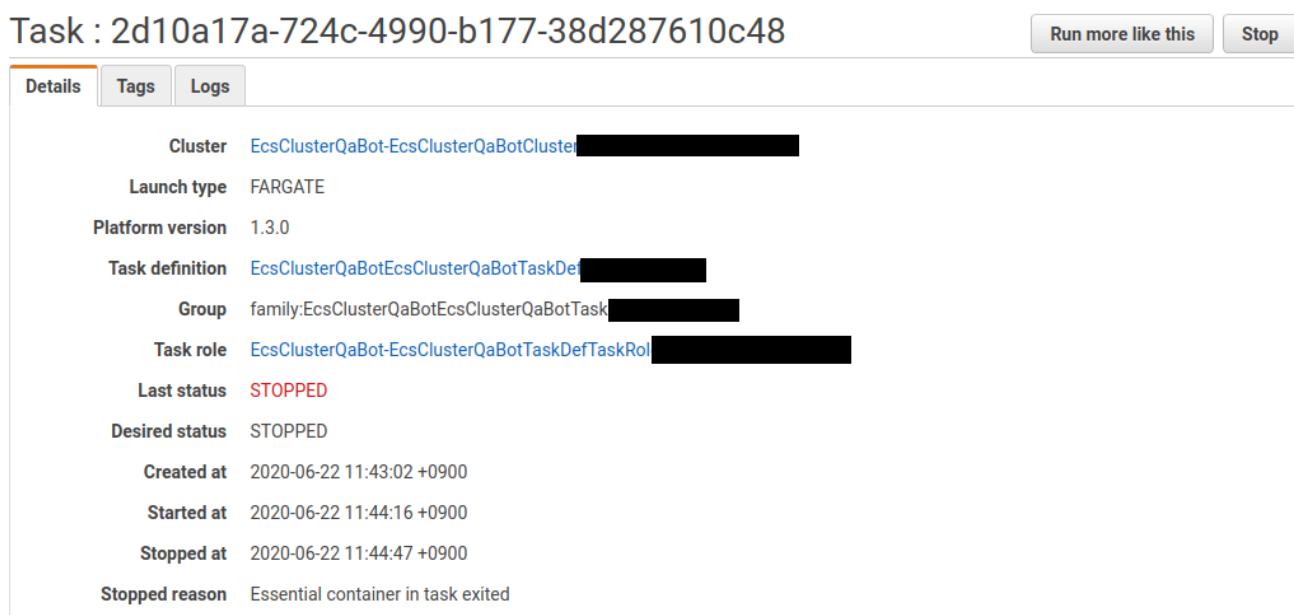


Figure 47. 質問タスクの実行結果

さて、`run_task.py` を実行したコマンドラインに戻ってきてみると、Figure 48 のような出力が得られているはずである。"Momotaro" という正しい回答が返ってきてている！

```
(.env) tomoyuki@balthasar:03-qa-bot$ python run_task.py ask "A giant peach was flowing in the river. She picked it up and brought it home. Later, a healthy baby was born from the peach. She named the baby Momotaro." "What is the name of the baby?"  
Submitting task...  
Task ARN: arn:aws:ecs:ap-northeast-1:606887060834:task/76ad5daf-12ae-400c-a4ef-4dcaf78420e0  
Waiting for the task to finish...  
.....  
Context: A giant peach was flowing in the river. She picked it up and brought it home. Later, a healthy baby was born from the peach. She named the baby Momotaro.  
Question: What is the name of the baby?  
Answer: Momotaro.  
Score: 0.9704681368063461
```

Figure 48. 質問タスクの実行結果

8.7. タスクの同時実行

さて,先ほどはたった一つの質問を投入したわけだが,今回設計したアプリケーションは, ECS と Fargate を使うことで同時にたくさんの質問を処理することができる. 実際に,たくさんの質問を一度に投入してみよう.

`run_task.py` に `ask_many` というオプションを付けることで,複数の質問を一度に送信できる. 質問の内容は [/handson/03-qa-bot/problems.json](#) に定義されている.

次のようなコマンドを実行しよう.

```
$ python run_task.py ask_many
```

このコマンドを実行した後で,先ほどの ECS コンソールに行き,タスクの一覧を見てみよう (Figure 49). 複数の Fargate インスタンスが起動され,タスクが並列に実行されているのがわかる.

Task	Task definit...	Container in...	Last status ...	Desired stat...	Started By ...	Group	Launch typ...	Platform ve...
06047e34-8...	EcsClusterQ...	-	RUNNING	RUNNING		family:EcsCl...	FARGATE	1.3.0
2dfd7f37-a4...	EcsClusterQ...	-	RUNNING	RUNNING		family:EcsCl...	FARGATE	1.3.0
7c0ae96f-ee...	EcsClusterQ...	-	RUNNING	RUNNING		family:EcsCl...	FARGATE	1.3.0
83612a14-7...	EcsClusterQ...	-	PENDING	RUNNING		family:EcsCl...	FARGATE	1.3.0
883f6fc6-0a...	EcsClusterQ...	-	RUNNING	RUNNING		family:EcsCl...	FARGATE	1.3.0
cad770f0-1d...	EcsClusterQ...	-	PENDING	RUNNING		family:EcsCl...	FARGATE	1.3.0
e3f0c1c2-3b...	EcsClusterQ...	-	RUNNING	RUNNING		family:EcsCl...	FARGATE	1.3.0
ee109b33-c...	EcsClusterQ...	-	RUNNING	RUNNING		family:EcsCl...	FARGATE	1.3.0
f56f566f-ef9...	EcsClusterQ...	-	RUNNING	RUNNING		family:EcsCl...	FARGATE	1.3.0

Figure 49. 複数の質問タスクを同時に投入する

すべてのタスクのステータスが "STOPPED" になったことを確認した上で,質問への回答を取得しよう. それに,次のコマンドを実行すれば良い.

```
$ python run_task.py list_answers
```

結果として, Figure 50 のような出力が得られるだろう. それなりに複雑な文章問題に対し,高い正答率で回答できていることがわかるだろう.

Context: Nikola Tesla (Serbian Cyrillic: ??????????? ??????????; 10 July 1856 ??? 7 January 1943) was a Serbian American inventor, electrical engineer, mechanical engineer, physicist, and futurist best known for his contributions to the design of the modern alternating current (AC) electricity supply system.

Question: In what year did Tesla die?

Answer: 1943

Score: 0.47624243081909157

2

Context: The Normans (Norman: Nourmands; French: Normands; Latin: Normanni) were the people who in the 10th and 11th centuries gave their name to Normandy, a region in France. They were descended from Norse ("Norman" comes from "Norseman") raiders and pirates from Denmark, Iceland and Norway who, under their leader Rollo, agreed to swear fealty to King Charles III of West Francia. Through generations of assimilation and mixing with the native Frankish and Roman-Gaulish populations, their descendants would gradually merge with the Carolingian-based cultures of West Francia. The distinct cultural and ethnic identity of the Normans emerged initially in the first half of the 10th century, and it continued to evolve over the succeeding centuries.

Question: What century did the Normans first gain their separate identity?

Answer: 10th

Score: 0.7034500812467961

3

Context: The Normans (Norman: Nourmands; French: Normands; Latin: Normanni) were the people who in the 10th and 11th centuries gave their name to Normandy, a region in France. They were descended from Norse ("Norman" comes from "Norseman") raiders and pirates from Denmark, Iceland and Norway who, under their leader Rollo, agreed to swear fealty to King Charles III of West Francia. Through generations of assimilation and mixing with the native Frankish and Roman-Gaulish populations, their descendants would gradually merge with the Carolingian-based cultures of West Francia. The distinct cultural and ethnic identity of the Normans emerged initially in the first half of the 10th century, and it continued to evolve over the succeeding centuries.

Question: Who was the Norse leader?

Answer: Rollo,

Score: 0.9961329869148798

Figure 50. \$ python run_task.py list_answers の実行結果

おめでとう! ここまでついてこれた読者は、とても初歩的ながらも、ディープラーニングによる言語モデルを使って自動で質問への回答を生成するシステムを創り上げることができた! それも、数百の質問にも同時にに対応できるよう、とても高いスケーラビリティーを持ったシステムである!

run_task.py で質問を投入し続けると、回答を記録しているデータベースにどんどんエントリーが溜まっていく。これらのエントリーをすべて消去するには、次のコマンドを使う。



\$ python run_task.py clear

8.8. スタックの削除

これにて、第三回ハンズオンは終了である。最後にスタックを削除しよう。

スタックを削除するには、次のコマンドを実行すればよい。

\$ cdk destroy

8.9. 講義第二回目のまとめ

ここまでが、第二回目の講義の内容である。第一回に引き続き盛りだくさんの内容であったが、ついでこれたであろうか?

第二回では、ディープラーニングの計算をクラウドで実行するため、GPU 搭載型の EC2 インスタンスの起動について解説した。その際、CUDA や PyTorch などのディープラーニング使うソフトウェアのインストールの手間を省くため、DLAMI を利用した。さらに、ハンズオン第二回では、クラウドで起動した仮想サーバーを使って、MNIST 文字認識タスクを解くニューラルネットを学習させた。

また、より大規模な機械学習アプリケーションを作るための手段として、Docker と ECS による動的に計算リソースが管理されるクラスターの作り方の初步を説明した。その応用として、英語で与えられた文章問題への回答を自動で生成するボットをクラウドに展開した。

もちろん、この講義で紹介したプログラムはごく初歩的なものなので、現実的な問題を解くためにはプログラムのいろいろな側面を精緻化していく必要がある。しかしながら、このような技術を応用することでどのようにして現実世界の問題を解くのか、なんとなくイメージが伝わっただろうか?

第三回では、さらにレベルアップし、Serverless architecture という最新のクラウドの設計方法について解説する。

その応用として、簡単な SNS サービスをゼロから実装する予定である。お楽しみに！

Chapter 9. Hands-on #4: AWS Batch を使って機械学習のハイパープラメータサーチを並列化する

ハンズオン第三回では、ECS と Fargate を使って自動質問回答システムを構築した。シンプルながらも、複数の質問が送られた場合には並列にジョブが実行され、ユーザーに答えが返されるシステムを作ることができた。ここでは、既に学習済みの言語モデルを用いてアプリケーションを構築した。しかし、機械学習のワークフローでは、まずは自分で作ったモデルを訓練することが最初のステップにあるはずである。そこで、ハンズオン第四回では、クラウドを用いて機械学習の訓練を並列化・高速化することを考える。

具体的には、本ハンズオンでは深層学習におけるハイパープラメータ最適化を取り上げる。ハイパープラメータとは、勾配降下法によって最適化されるニューラルネットのパラメータの外にあるパラメータのことであり、具体的にはモデルの層の幅・深さなどネットワークのアーキテクチャに関わるもの、学習率やモメンタムなどパラメータの更新則に関わるものなどが含まれる。深層学習においてハイパープラメータの調整はとても重要なタスクである。しかしながら、ハイパープラメータを調整するには、少しづつ条件を変えながら何度もニューラルネットを学習させる必要があり、多くの計算時間がかかる。本ハンズオンでは、クラウドの強力な計算リソースを利用し並列的にニューラルネットの訓練を実行することで、この問題を高速に解く方法を学んでいこう。

このハンズオンは、本書で扱うハンズオンの中でも最も難易度が高いものだ（もう少し正確に言うなら、最もステップ数が多い）。内容が少し難しく感じる読者もいるかもしれないが、頑張ってついてきてほしい。

9.1. Auto scaling groups (ASG)

ハンズオンに入っていく前に、**Auto scaling groups (ASG)** と呼ばれる EC2 の概念を知っておく必要がある。

ECS の概要を示した [Figure 40](#) を振り返って見てほしい。前章 ([Chapter 8](#)) でも説明したが、ECS のクラスターで計算を担う実体としては EC2 と Fargate を指定することができる。Fargate については、前章で記述した。クラスターで EC2 が選択された場合について、以下に簡単に説明を行っていきたい。

EC2 クラスターには **Auto scaling groups (ASG)** と呼ばれるサービスが配置される。ASG はクラスターのスケーリングを担っており、ASG によって新しいインスタンスの起動や、不要になったインスタンスの停止が実行される。どのような基準でインスタンスの起動・停止を行うかのルールのことを、**スケーリングポリシー** と呼ぶ。ASG にはユーザーの指定したスケーリングポリシーを定義することが可能である。例えば、クラスター全体の稼働率（負荷）を 80% に維持する、かつ、起動インスタンスの上限は 100 台とする、などのポリシーを設定できる。あるいはユーザーの作成したカスタムのプログラムでスケーリングを制御することも可能である。ECS と ASG が協調することで、フレキシブルかつ効率的なタスクの配置とクラスターのスケーリングが可能になるのである。

9.2. AWS Batch

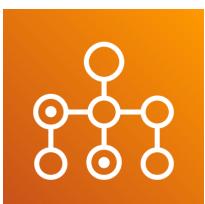


Figure 51. AWS Batch のロゴ

上記で説明したように、ECS と ASG を組み合わせることで、所望の計算クラスターを構築することが可能である。しかしながら、ECS と ASG にはかなり込み入った設定が必要であり、初心者にとっても経験者にとってもなかなか面倒なプログラミングが要求される。そこで、ECS と ASG によるクラスターの設計を自動化してくれるサービスが提供されている。それが **AWS Batch** である。

AWS Batch はその名の通りバッチ (Batch) 化されたジョブ（入力データだけが異なる独立した演算が繰り返し実行されること）を想定している。多くの科学計算や機械学習がバッチ計算に当てはまる。例えば、初期値のパラメータを変えてシミュレーションを走らせる、といったケースだ。AWS Batch を用いることの利点は、クラスターのスケーリングやジョブの割り振りはすべて自動で実行され、ユーザーはクラウドの背後で起こっている詳細を気に

することなく、大量のジョブを投入することができるシステムが手に入る点である。

AWS Batch では、ジョブの投入・管理をスムーズに行うため、次のような概念が定義されている (Figure 52)。まず、**Job** というのが、AWS Batch によって実行されるひとつひとつの計算の単位である。**Job Definitions** とは Job の内容を定義するものであり、これには実行されるべき Docker のイメージのアドレスや、割り当てる CPU・RAM の容量、環境変数などの設定が含まれる。ユーザーが Job の実行を実行すると、Job は **Job Queues** に入る。**Job queues** とは、実行されるのを待っているジョブの待ち列のことであり、時間的に最も先頭に投入されたジョブが最初に実行される。また、複数の queue を配置し、queue ごとに priority (優先度) を設定することが可能であり、priority の高い queue に溜まつた Job が優先的に実行される。**Compute environment** とは、先述したクラスターとほぼ同義の概念であり、計算が実際に実行される場所 (EC2 や Fargate からなるクラスター) を指す。Compute environment には、使用する EC2 のインスタンスタイプや同時に起動するインスタンス数の上限などの簡易なスケーリングポリシーが指定されている。**Job queues** は Compute environment の空き状況を監視しており、それに応じて Job を Compute environment に投下する。

以上が AWS Batch を使用する上で理解しておかなければならぬ概念であるが、くどくど言葉で説明してもなかなかピンとこないだろう。ここからは、実際に自分で手を動かしながら学んでいこう。

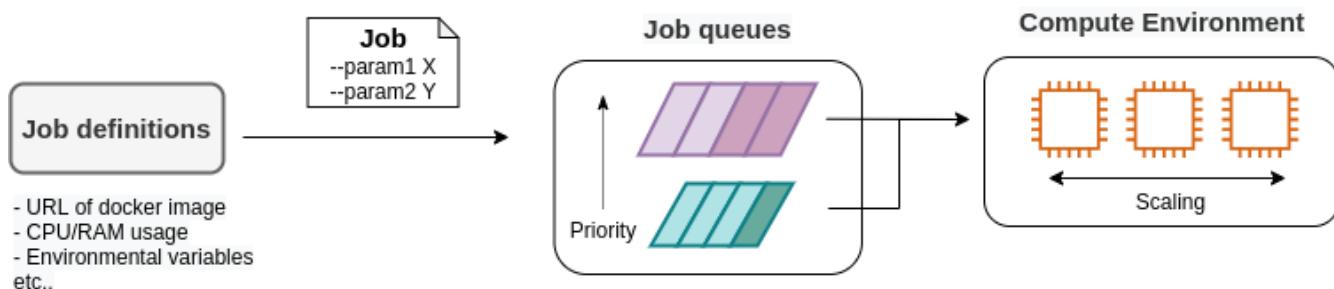


Figure 52. AWS Batch の主要な概念

9.3. 準備

本ハンズオンの実行には、第一回ハンズオンで説明した準備 (Section 4.1) が整っていることを前提とする。それ以外に必要な準備はない。

9.4. MNIST 手書き文字認識 (再訪)

今回のハンズオンでは、機械学習のハイパーパラメータ調整を取り上げると冒頭で述べた。その最もシンプルな例題として、Section 6.8 で扱った MNIST 手書き文字認識の問題を再度取り上げよう。Section 6.8 では、適当にチョイスしたハイパーパラメータを用いてモデルの訓練を行った。ここで使用したプログラムの中でのハイパーパラメータとしては、確率的勾配降下法 (SGD) における学習率やモメンタムが含まれる。コードでいうと、以下の行が該当する。

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
```

ここで使用された 学習率 (`lr=0.01`) や モメンタム (`momentum=0.5`) は恣意的に選択された値であり、これがベストな数値であるのかはわからない。たまたまこのチョイスが最適であるかもしれないし、もっと高い精度を出すハイパーパラメータの組が存在するかもしれない。この問題に答えるため、ハイパーパラメータサーチを行おう。今回は、最もシンプルなアプローチとして、**グリッドサーチ**によるハイパーパラメータサーチを行おう。

ハイパーパラメータの最適化について

機械学習のハイパーパラメータの最適化には大きく3つのアプローチが挙げられる。グリッドサーチ法、ランダムサーチ法、そしてベイズ最適化による方法である。

グリッドサーチ法とは、ハイパーパラメータの組をある範囲の中で可能な組み合わせをすべて計算し、最適なパラメータの組を見出す方法である。最もシンプルかつ確実な方法であるが、すべての組み合わせの可能性を愚直に計算するので計算コストが大きい。

ランダムサーチ法とは、ハイパーパラメータの組をある範囲の中でランダムに抽出し、大量に試行されたランダムな組の中から最適なパラメータの組を見出す方法である。すべての可能性を網羅的に探索できるわけではないが、調整すべきパラメータの数が多数ある場合に、グリッドサーチよりも効率的に広い探索空間をカバーすることができる。

ベイズ最適化を用いた方法では、過去の探索結果から次にどの組み合わせを探索すべきかという指標を計算し、次に探索するパラメータを決定する。これによりグリッドサーチやランダムサーチよりも少ない試行回数で最適なパラメータにたどり着くことができる。

並列化の観点でいうと、グリッドサーチとランダムサーチは各ハイパーパラメータの組の計算は独立に実行することができるため並列化が容易である。このように独立したジョブとして分割・並列化可能な問題を *Embarrassingly parallel* な問題と呼ぶ（直訳すると“恥ずかしいほど並列化可能な問題”，ということになる）。*Embarrassingly parallel* な問題はクラウドの強力な計算リソースを用いることで、非常にシンプルな実装で解くことができる。この章ではこのようなタイプの並列計算を取り上げる。

一方、ベイズ最適化による方法は、過去の結果をもとに次の探索が決定されるので、並列化はそれほど単純ではない。最近では [optuna](#) などのハイパーパラメータ探索のためのライブラリが発達しており、ベイズ最適化の数理的な処理を自動で実行してくれる所以便利である。これらのライブラリを使うと、もし一台のコンピュータ（ノード）の中に複数の GPU が存在する場合は、並列に計算を実行することができる。しかしながら、一台のノードにとどまらず、複数のノードをまたいだ並列化は、高度なプログラミングテクニックが必要とされるだけでなく、ノード間の接続様式などクラウドのアーキテクチャにも深く依存するものである。本書ではここまで高度なクラウドの使用方法には立ち入らない。

9.5. ローカルで Docker を実行

まずは、本ハンズオンで使用する Docker image をローカルで計算してみよう。

Docker image のソースコードは [handson/aws-batch/docker](#) にある。基本的に [Section 6.8](#) のハンズオンを元にし、本ハンズオン専用の軽微な変更が施してある。興味のある読者はソースコードも含めて読んでいただきたい。

次のコマンドで Docker image をローカルに pull してこよう。

```
$ docker pull XXXXXXXXXXXXXXXX
```

Pull できたら、次のコマンドでコンテナを起動し、実行する。

```
$ docker run -it mymnist python3 main.py --lr 0.1 --momentum 0.5 --epochs 100
```

上記のコマンドを実行すると、指定したハイパーパラメータ（学習率とモメンタム）を使ってニューラルネットの最適化が始まる。学習を行う最大のエポック数は `--epochs` パラメータで指定する。[Chapter 6](#) のハンズオンで見たような、Loss の低下がコマンドライン上に出力されるだろう（Figure 53）。

```

Train Epoch: 0 [0/48000 (0.0%)] Loss: 2.297341
Train Epoch: 0 [6400/48000 (13.3%)] Loss: 0.298299
Train Epoch: 0 [12800/48000 (26.7%)] Loss: 0.083849
Train Epoch: 0 [19200/48000 (40.0%)] Loss: 0.252932
Train Epoch: 0 [25600/48000 (53.3%)] Loss: 0.160509
Train Epoch: 0 [32000/48000 (66.7%)] Loss: 0.082315
Train Epoch: 0 [38400/48000 (80.0%)] Loss: 0.153711
Train Epoch: 0 [44800/48000 (93.3%)] Loss: 0.222485

Val set: Average loss: 0.0733, Accuracy: 97.8%

Train Epoch: 1 [0/48000 (0.0%)] Loss: 0.165711
Train Epoch: 1 [6400/48000 (13.3%)] Loss: 0.136394
Train Epoch: 1 [12800/48000 (26.7%)] Loss: 0.089186
Train Epoch: 1 [19200/48000 (40.0%)] Loss: 0.095106
Train Epoch: 1 [25600/48000 (53.3%)] Loss: 0.025505
Train Epoch: 1 [32000/48000 (66.7%)] Loss: 0.061345
Train Epoch: 1 [38400/48000 (80.0%)] Loss: 0.163712
Train Epoch: 1 [44800/48000 (93.3%)] Loss: 0.122928

Val set: Average loss: 0.0552, Accuracy: 98.4%

```

Figure 53. Docker を実行した際の出力

上に示したコマンドを使うと, 計算は CPU を使って実行される. もし, ローカルの計算機に GPU が備わっており, [nvidia-docker](#) の設定が済んでいるならば, 以下のコマンドにより GPU を使って計算を実行することができる.

```
$ docker run -it --gpus all mymnist python3 main.py --lr 0.1 --momentum 0.5 --epochs 100
```

上のコマンドでは, `--gpus all` というパラメータが加わった.

CPU/GPU どちらで実行した場合でも, エポックを重ねるにつれて訓練データ (Train データ) の Loss は単調に減少していくのが見て取れるだろう. 一方, 検証データ (Validation データ) の Loss および Accuracy はある程度まで減少した後, それ以上性能が向上しないことに気がつくだろう. これを実際にプロットしてみると Figure 54 のようになるはずである.

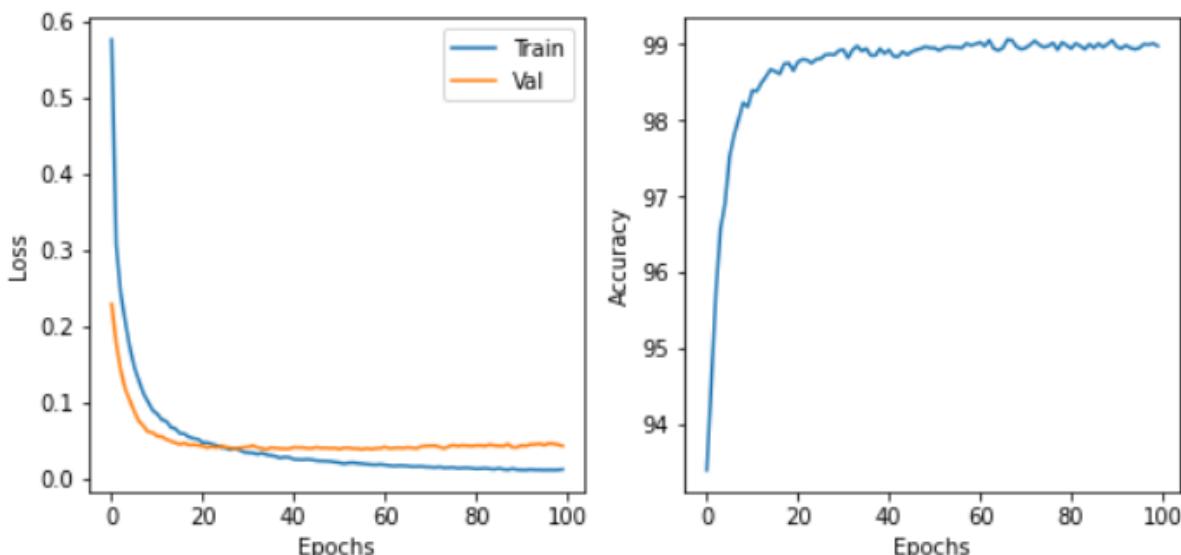


Figure 54. (左) Train/Validation データそれぞれの Loss のエポックごとの変化. (右) Validation データの Accuracy のエポックごとの変化

これはオーバーフィッティングと呼ばれる現象で, ニューラルネットが訓練データに過度に最適化され, 訓練データ

タの外のデータに対しての精度(汎化性能)が向上していないことを示している。このような場合の対処法として、**Early stopping**と呼ばれるテクニックが知られている。Early stoppingとは、訓練データのLossを追跡し、それが減少から増加に転じるエポックで学習をうち止め、そのエポックでのウェイトパラメータを採用する、というものである。本ハンズオンでも、Early stoppingによって訓練の終了を判断し、モデルの性能評価を行っていく。

機械学習では基本的な概念であるが、教師あり学習によってモデルを訓練する際に使うデータセットは**訓練(Train)**、**検証(Validation)**、**テスト(Test)**データに分割する。訓練データとは、モデルのウェイトパラメータの最適化に使用されるデータセットである。検証データとは、Early stoppingによって過学習のタイミングを判断したり、ハイパーパラメータの調整を行ったりするのに用いるデータセットである。最後に、テストデータとは、ハイパーパラメータの調整も済んで最終的に出来上がったモデルの性能を評価するために用いるデータセットである。論文などで報告するモデルのベンチマークの結果は、基本的にテストデータに対しての性能である。



しばしば見かける重大な間違いが、テストデータを使ってハイパーパラメータを調整する、という行為である。これは絶対に行ってはいけない。というのは、このやり方だとテストデータに最適にフィットするようなハイパーパラメータが意図的に選択されてしまっているからである。モデルの性能を正しく評価するには、訓練に一度も使われていないデータを用いなければならない。

MNIST 手書き文字データセットでは、訓練データとして 60,000 枚、テストデータとして 10,000 枚の画像が与えられている。本ハンズオンで使用するコードでは、訓練データのうち 80% の 48,000 枚を訓練データとして使用し、残り 20% の 12,000 枚を検証データとして用いている。詳しくは、ソースコードを参照のこと。

9.6. アプリケーションの説明

このハンズオンで作成するアプリケーションの概要を Figure 55 に示す。

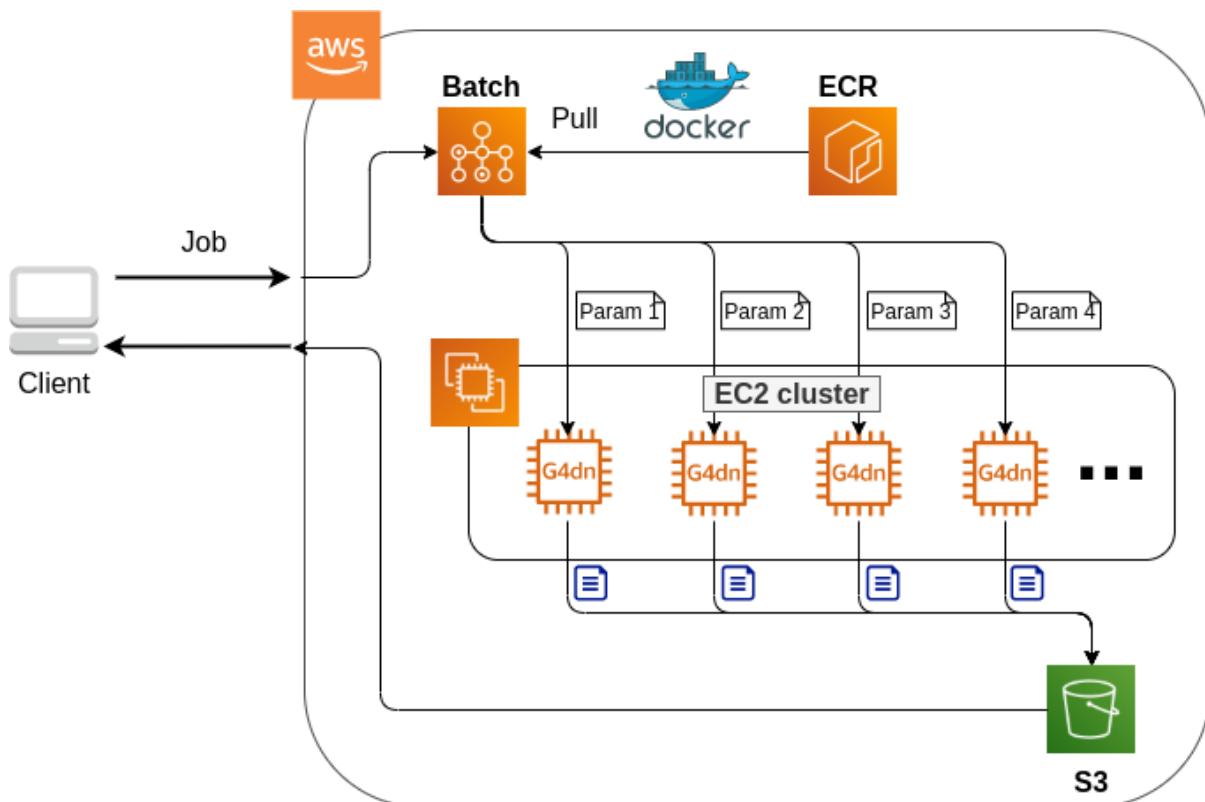


Figure 55. アプリケーションのアーキテクチャ

簡単にまとめると、以下のような設計である。

- クライアントは、あるハイパーパラメータの組を指定して Batch にジョブを提出する

- Batch はジョブを受け取ると、EC2 からなるクラスターで計算を実行する
- クラスター内では `g4dn.xlarge` インスタンスが起動する
- Docker image は、AWS 内に用意された ECR (Elastic Container Registry) から取得される
- 複数のジョブが投下された場合は、その数だけのインスタンスが起動し並列に実行される。
- 各ジョブによる計算の結果は S3 に保存される
- 最後にクライアントは S3 から結果をダウンロードし、最適なハイパーパラメータの組を決定する

それでは、プログラムのソースコードを見てみよう ([handson/aws-batch/app.py](#)).

```

1 class SimpleBatch(core.Stack):
2
3     def __init__(self, scope: core.App, name: str, **kwargs) -> None:
4         super().__init__(scope, name, **kwargs)
5
6         # S3 bucket to store data ①
7         bucket = s3.Bucket(
8             self, "bucket",
9             removal_policy=core.RemovalPolicy.DESTROY,
10            auto_delete_objects=True,
11        )
12
13         vpc = ec2.Vpc(
14             self, "vpc",
15             max_azs=1,
16             cidr="10.10.0.0/23",
17             subnet_configuration=[
18                 ec2.SubnetConfiguration(
19                     name="public",
20                     subnet_type=ec2.SubnetType.PUBLIC,
21                 )
22             ],
23             nat_gateways=0,
24         )
25
26         ②
27         managed_env = batch.ComputeEnvironment(
28             self, "managed-env",
29             compute_resources=batch.ComputeResources(
30                 vpc=vpc,
31                 allocation_strategy=batch.AllocationStrategy.BEST_FIT,
32                 desiredv_cpus=0,
33                 maxv_cpus=64,
34                 minv_cpus=0,
35                 instance_types=[
36                     ec2.InstanceType("g4dn.xlarge")
37                 ],
38             ),
39             managed=True,
40             compute_environment_name=self.stack_name + "compute-env"
41         )
42
43         ③
44         job_queue = batch.JobQueue(
45             self, "job-queue",
46             compute_environments=[
47                 batch.JobQueueComputeEnvironment(
48                     compute_environment=managed_env,
49                     order=100
50                 )

```

```

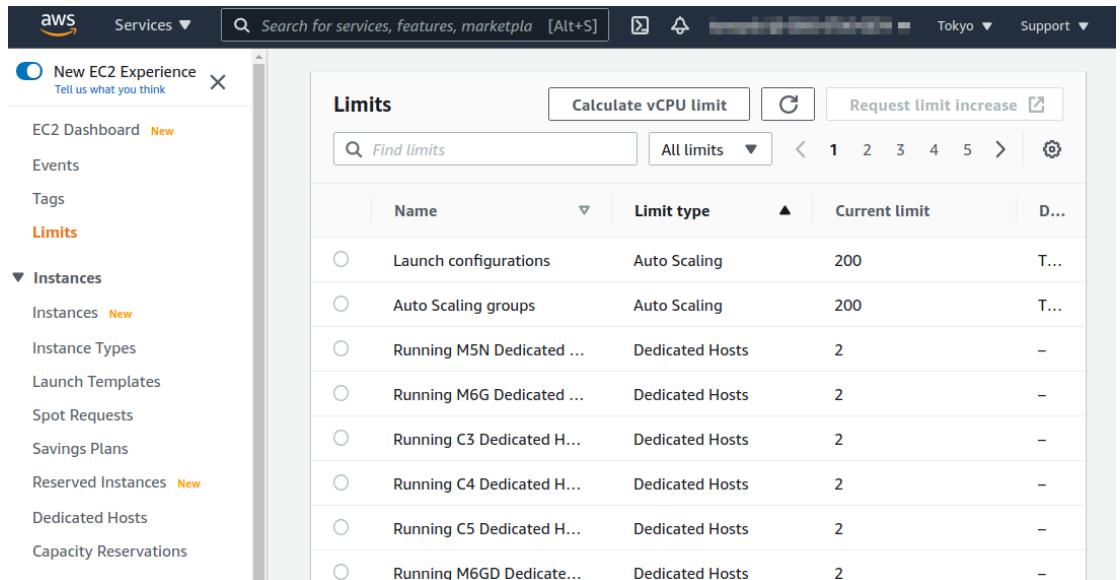
51     ],
52     job_queue_name=self.stack_name + "job-queue"
53 )
54
55 ④
56 job_role = iam.Role(
57     self, "job-role",
58     assumed_by=iam.CompositePrincipal(
59         iam.ServicePrincipal("ecs-tasks.amazonaws.com")
60     )
61 )
62 # allow read and write access to S3 bucket
63 bucket.grant_read_write(job_role)
64
65 # create a ECR repository to push docker image ⑤
66 repo = ecr.Repository(
67     self, "repository",
68     removal_policy=core.RemovalPolicy.DESTROY,
69 )
70
71 ⑥
72 job_def = batch.JobDefinition(
73     self, "job-definition",
74     container=batch.JobDefinitionContainer(
75         image=ecs.ContainerImage.from_ecr_repository(repo),
76         command=["python3", "main.py"],
77         vcpus=4,
78         gpu_count=1,
79         memory_limit_mib=12000,
80         job_role=job_role,
81         environment={
82             "BUCKET_NAME": bucket.bucket_name
83         }
84     ),
85     job_definition_name=self.stack_name + "job-definition",
86     timeout=core.Duration.hours(2),
87 )

```

- ① で、計算結果を保存するための S3 バケットを用意している
- ② で、Compute environment を定義している。ここでは `g4dn.xlarge` のインスタンスタイプを使用するとし、最大の vCPU 使用数は 64 と指定している。
- ③ で、<2> で作成した Compute environment と紐付いた Job queue を定義している。
- ④ で、Job が計算結果を S3 に書き込むことができるよう、IAM ロールを定義している。
- ⑤ では、Docker image を配置するための ECR を定義している。
- ⑥ で Job definition を作成している。ここでは、4 vCPU、12000 MB (=12GB) の RAM を使用するように指定している。また、今後必要となる環境変数 (`BUCKET_NAME`) を設定している。さらに、<4> で作った IAM を付与している。

g4dn.xlarge は 4 vCPU が割り当てられているので、上のプログラムで最大の vCPU 使用数が 64 だとすると、最大で 16 台のインスタンスが同時に起動することになる。

ここで注意が一点ある。AWS では各アカウントごとに EC2 で起動できるインスタンスの上限が設定されている。この上限は AWS コンソールにログインし、EC2 コンソールの左側メニューバーの **Limits** をクリックすることで確認できる (Figure 56)。**g4dn.xlarge** (EC2 の区分でいうと G ファミリーに属する) の制限を確認するには、**Running On-Demand All G instances** という名前の項目を見る。ここにある数字が、AWS によって課されたアカウントの上限であり、この上限を超えたインスタンスを起動することはできない。もし、自分の用途に対して上限が低すぎる場合は、上限の緩和申請を行うことができる。詳しくは [公式ドキュメンテーション](#) を参照のこと。



The screenshot shows the AWS EC2 Limits page. The left sidebar has a lightbulb icon and navigation links like EC2 Dashboard, Events, Tags, and Limits. Under Instances, there are links for Instances, Instance Types, Launch Templates, Spot Requests, Savings Plans, Reserved Instances, Dedicated Hosts, and Capacity Reservations. The main content area is titled 'Limits' with a search bar and a 'Find limits' button. It includes buttons for 'Calculate vCPU limit', 'Request limit increase', and a refresh icon. A table lists various limits with columns for Name, Limit type, Current limit, and a Details... link. Some entries have a small circular icon next to them.

Name	Limit type	Current limit	Details...
Launch configurations	Auto Scaling	200	T...
Auto Scaling groups	Auto Scaling	200	T...
Running M5N Dedicated ...	Dedicated Hosts	2	-
Running M6G Dedicated ...	Dedicated Hosts	2	-
Running C3 Dedicated H...	Dedicated Hosts	2	-
Running C4 Dedicated H...	Dedicated Hosts	2	-
Running C5 Dedicated H...	Dedicated Hosts	2	-
Running M6GD Dedicated...	Dedicated Hosts	2	-

Figure 56. EC2 コンソールから各種の上限を確認する

9.7. スタックのデプロイ

スタックの中身が理解できたところで、早速スタックをデプロイしてみよう。

デプロイの手順は、これまでのハンズオンとほとんど共通である。ここでは、コマンドのみ列挙する (# で始まる行はコメントである)。それぞれの意味を忘れてしまった場合は、前のハンズオンに戻って復習していただきたい。

```
# プロジェクトのディレクトリに移動
$ cd handson/aws-batch

# venv を作成し、依存ライブラリのインストールを行う
$ python3 -m venv .env
$ source .env/bin/activate
$ pip install -r requirements.txt

# AWS の認証情報をセットする
# 自分自身の認証情報に置き換えること！
export AWS_ACCESS_KEY_ID=XXXXXX
export AWS_SECRET_ACCESS_KEY=YYYYYY
export AWS_DEFAULT_REGION=ap-northeast-1

# デプロイを実行
$ cdk deploy
```

デプロイのコマンドが無事に実行されたことが確認できたら、AWS コンソールにログインして、デプロイされたスタックを確認してみよう。コンソールの検索バーで **batch** と入力し、AWS Batch の管理画面を開く (Figure 57)。

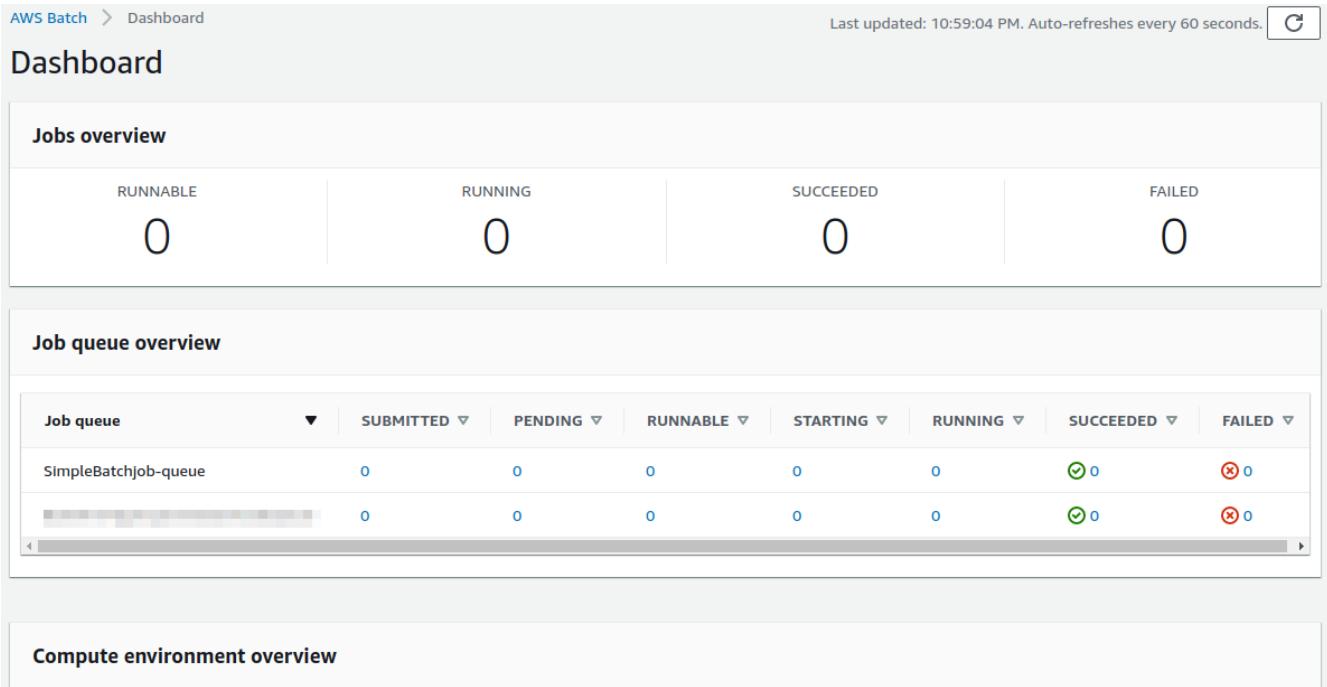


Figure 57. AWSBatch のコンソール画面 (ダッシュボード)

まず目を向けてほしいのが、画面の一番下にある Compute environment overview の中の **SimpleBatchcompute-env** という名前の項目だ。Compute environment とは、先ほど述べたとおり、計算が実行される環境（クラスターと読み替えるても良い）である。上のプログラムで指定したとおり、**g4dn.xlarge** が実際に使用されるインスタンスタイプとして表示されている。また、この時点ではひとつのジョブが走っていないので、**desired vCPUs** は 0 になっている。

次に、Job queue overview にある **SimpleBatch-queue** という項目に注目してほしい。ここでは実行待ちのジョブ・実行中のジョブ・実行が完了したジョブを一覧で確認することができる。

9.8. Docker image を ECR に配置する

さて、Batch がジョブを実行するには、どこか指定された場所から Docker image をダウンロード (pull) してくる必要がある。前回のハンズオン (Chapter 8) では、公開設定にしてある GitHub Container Registry から Docker image を pull してきた。今回のハンズオンでは、AWS から提供されているコンテナ置き場である **ECR (Elastic Container Registry)** に image を配置するという設計を採用する。Batch は ECR から image を pull していくことで、タスクを実行する (Figure 55)。ECR を利用する利点は、自分だけがアクセスすることのできるプライベートな image の置き場所を用意できる点である。

スタックのソースコードでいうと、以下の箇所が ECR を定義している。

```

①
repo = ecr.Repository(
    self, "repository",
    removal_policy=core.RemovalPolicy.DESTROY,
)

job_def = batch.JobDefinition(
    self, "job-definition",
    container=batch.JobDefinitionContainer(
        image=ecs.ContainerImage.from_ecr_repository(repo), ②
        ...
    ),
    ...
)

```

①で、新規の ECR を作成している。

②で Job definition を定義する中で、image を<1>で作った ECR から取得するように指定している。同時に、Job definition には ECR へのアクセス権が自動的に付与されることになる。

さて、スタックをデプロイした時点では、ECR は空っぽである。ここに自分のアプリケーションで使う Docker image を push してあげる必要がある。

そのために、まずは AWS コンソールから ECR の画面を開こう（検索バーに **Elastic Container Registry** と入力すると出てくる）。**Private** というタブを選択すると、**simplebatch-repositoryXXXXXX** という名前のレポジトリが見つかるだろう（Figure 58）。

Repository name	URI	Created at	Tag immutability	Scan on push	Encryption type
simplebatch-repository	[REDACTED]	[REDACTED]	[REDACTED]	[REDACTED]	[REDACTED]
simplebatch-repository9f1a3f0b-zfcfedjaz8q30	[REDACTED]	Jun 07, 2021 10:24:14 PM	Disabled	Disabled	AES-256

Figure 58. ECR のコンソール画面

次に、このレポジトリの名前をクリックするとレポジトリの詳細画面に遷移する。そうしたら、画面右上にある **View push commands** というボタンをクリックする。すると Figure 59 のようなポップアップ画面が立ち上がる。

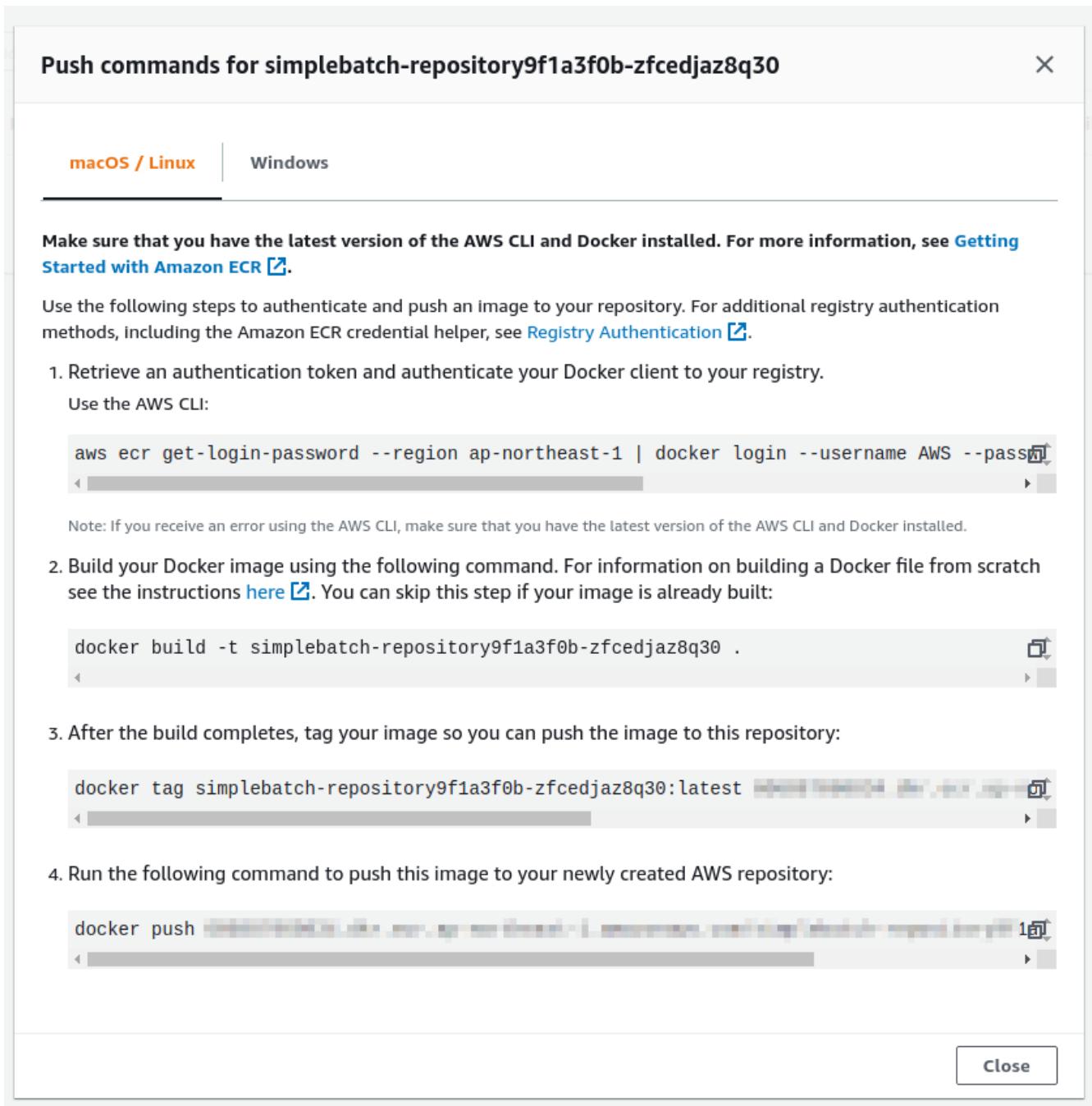


Figure 59. ECR への push コマンド

このポップアップ画面で表示されている4つのコマンドを順番に実行していくことで、手元の Docker image を ECR に push することができる。

push を実行する前に、AWS の認証情報が設定されていることを確認しよう。その上で、ハンズオンのソースコードの中にある `docker/` という名前のディレクトリに移動する。そうしたら、ポップアップ画面で表示されたコマンドを上から順に実行していく。



ポップアップで表示されるコマンドの2つめを見てみると `docker build -t XXXXX .` となって いる。最後の `.` が重要で、これは、現在のディレクトリにある Dockerfile を使って image をビ ルドせよという意味である。このような理由で、`Dockerfile` が置いてあるディレクトリに移動す る必要がある。

4つめのコマンドには少し時間がかかるかもしれないが、これが完了するとめでたく image が ECR に配置されたことになる。もう一度 ECR のコンソールを見てみると、確かに image が配置されていることが確認できる (Figure 60)。

これで,AWS Batch を使ってジョブを実行させるための最後の準備が完了した.

Image tag	Pushed at	Size (MB)	Image URI	Digest	Scan status	Vulnerabilities
latest	Jun 07, 2021 11:44:27 PM	4676.39	Copy URI	sha256:1f44b04...	-	-

Figure 60. ECR へ image の配置が完了した



今回のハンズオンで紹介するアプリケーションは, Docker image を置き換えることで, ユーザー自身の計算ジョブを実行することが可能である. 興味のある読者は, 自分自身の Docker image を ECR に配置し, ジョブを実行してみると良い.

9.9. Job を実行する(まずはひとつだけ)

さて,ここからは実際に AWS Batch にジョブを投入する方法を見ていこう.

ハンズオンのディレクトリの `notebook/` というディレクトリの中に, `run_single.ipynb` というファイルが見つかるはずである (`.ipynb` は Jupyter notebook のファイル形式). これを Jupyter notebook から開こう.

今回のハンズオンでは, `venv` による仮想環境の中に Jupyter notebook もインストール済みである. なので,以下のコマンドで Jupyter notebook を立ち上げる.

```
# .env の仮想環境にいることを確認
(.env) $ cd notebook
(.env) $ jupyter notebook
```

Jupyter notebook が起動したら, `run_single.ipynb` を開く.

最初の [1], [3] 番のセルは, ジョブをサブミットするための関数 (`submit_job()`) を定義している.

```
# [1]
import boto3
import argparse

# [2]
def submit_job(lr:float, momentum:float, epochs:int, profile_name="default"):
    # 省略...
```

`submit_job()` 関数について簡単に説明しよう. Section 9.5 で, MNIST を学習する Docker をローカルで実行したとき, 以下のようなコマンドを使用した.

```
$ docker run -it mymnist python3 main.py --lr 0.1 --momentum 0.5 --epochs 100
```

ここで, `python3 main.py --lr 0.1 --momentum 0.5 --epochs 100` の部分が, Docker に渡されるコマンドである.

AWS Batch でジョブを実行する際も, 同じようなコマンドを Docker に渡せば良い. `submit_job()` 関数は, このコマンドの文字列を生成し, ジョブに渡している. コードでは以下の部分が該当する.

```
containerOverrides={  
    "command": ["python3", "main.py",  
                "--lr", str(lr),  
                "--momentum", str(momentum),  
                "--epochs", str(epochs),  
                "--uploadS3", "true"]  
}
```

続いて, [4] 番のセルに移ろう. ここでは, 上記の `submit_job()` 関数を用いて, 学習率 = 0.01, モメンタム = 0.1 を指定したジョブを投入する.

```
submit_job(0.01, 0.1, 100)
```

AWS の認証情報は, Jupyter notebook の内部から再度定義する必要がある. これを手助けするため, notebook の [2] 番のセル (デフォルトではすべてコメントアウトされている) を用意した. これを使うにはコメントアウトを解除すればよい. このセルを実行すると, AWS の認証情報を入力する対話的なプロンプトが表示される. プロンプトに従って aws secret key などを入力することで, (Jupyter のセッションに固有な) 環境変数に AWS の認証情報が記録される.



もう一つの認証方法として, `sumit_job()` 関数に `profile_name` というパラメータを用意した. もし `~/.aws/credentials` に認証情報が書き込まれているのならば (詳しくは [Section 15.2](#)), `profile_name` に使用したいプロファイルの名前を渡すだけで, 認証を行うことができる.

慣れている読者は後者のほうが便利であると感じるだろう.

[4] 番のセルを実行したら, ジョブが実際に投入されたかどうかを AWS コンソールから確認してみよう. AWS Batch の管理コンソールを開くと, [Figure 61](#) のような画面が表示されるだろう.

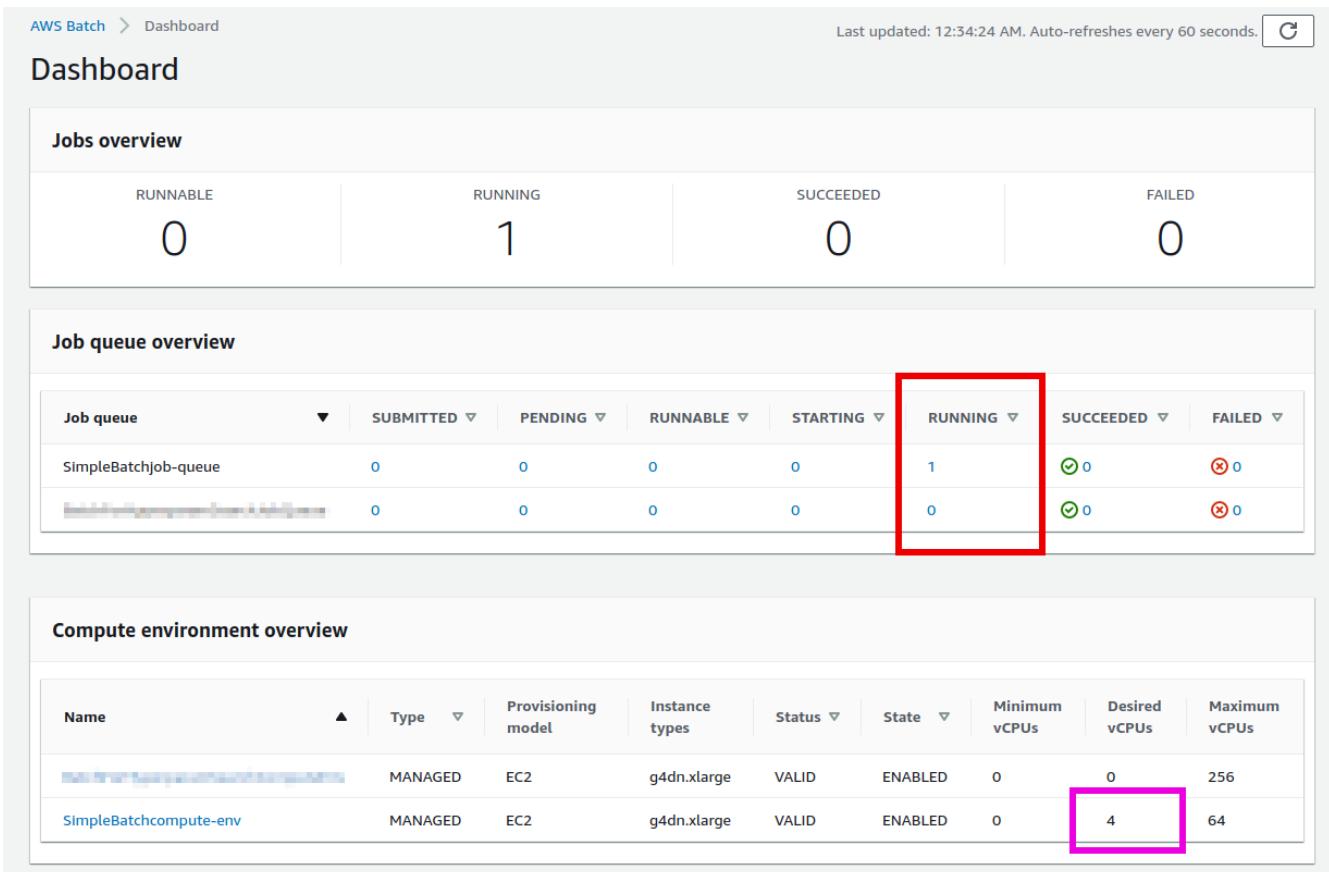


Figure 61. AWS Batch でジョブが実行されている様子

Figure 61 で赤で囲った箇所に注目してほしい。ひとつのジョブが投入されると、それは **SUBMITTED** という状態を経て **RUNNABLE** という状態に遷移する。**RUNNABLE** とは、ジョブを実行するためのインスタンスが Compute Environment に不足しているため、新たなインスタンスが起動されるのを待っている状態に相当する。インスタンスの準備が整うと、ジョブの状態は **STARTING** を経て **RUNNING** に至る。

次に、ジョブのステータスが **RUNNING** のときの Compute Environment の **Desired vCPU** を見てみよう (Figure 61 で紫で囲った箇所)。ここで 4 と表示されているのは、**g4dn.xlarge** インスタンス一つ分の vCPU の数である。ジョブの投入に応じて、それを実行するのに最低限必要な EC2 インスタンスが起動されたことが確認できる。(興味のある人は、EC2 コンソールも同時に覗いてみるとよい)。

しばらく経つと、ジョブの状態は **RUNNING** から **SUCCEEDED** (あるいは何らかの理由でエラーが発生したときには **FAILED**) に遷移する。今回のハンズオンで使っている MNIST の学習はだいたい 10 分くらいで完了するはずである。ジョブの状態が **SUCCEEDED** になるまで見届けよう。

ジョブが完了すると、学習の結果 (エポックごとの Loss と Accuracy を記録した CSV ファイル) は S3 に保存される。AWS コンソールからこれを確認しよう。

S3 のコンソールに行くと **simplebatch-bucketXXXXXX** (XXXX の部分はユーザーによって異なる) という名前のバケットが見つかるはずである。これをクリックして中身を見てみると、**metrics_lr0.0100_m0.1000.csv** という名前の CSV があることが確認できるだろう (Figure 62)。これが、学習率 = 0.01、モメンタム = 0.1 として学習を行ったときの結果である。

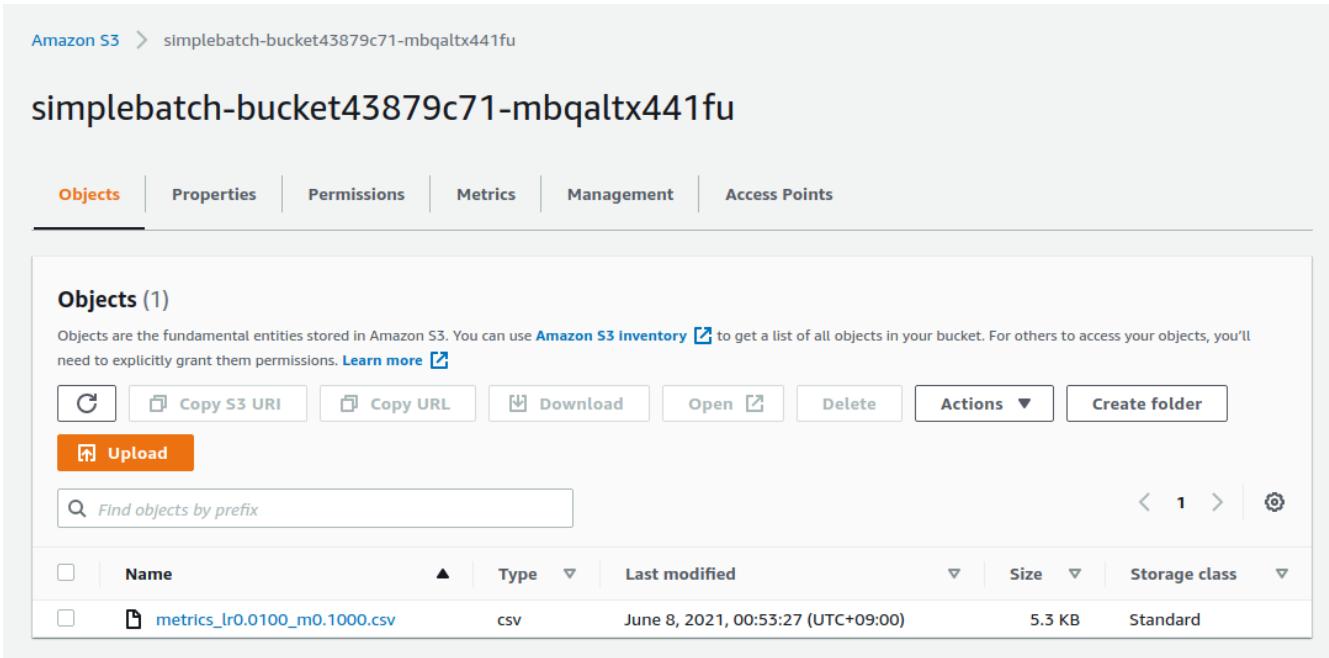


Figure 62. ジョブの実行結果は S3 に保存される

さて、ここで `run_single.ipynb` に戻ってこよう。[5] から [9] 番のセルでは、学習結果の CSV ファイルをダウンロードしてきて、結果の確認と可視化を行っている。

```

1 # [5]
2 import pandas as pd
3 import io
4 from matplotlib import pyplot as plt
5
6 # [6]
7 def read_table_from_s3(bucket_name, key, profile_name=None):
8     if profile_name is None:
9         session = boto3.Session()
10    else:
11        session = boto3.Session(profile_name=profile_name)
12    s3 = session.resource("s3")
13    bucket = s3.Bucket(bucket_name)
14
15    obj = bucket.Object(key).get().get("Body")
16    df = pd.read_csv(obj)
17
18    return df
19
20 # [7]
21 df = read_table_from_s3(
22     "simplebatch-bucket43879c71-mbqaltx441fu",
23     "metrics_lr0.0100_m0.1000.csv"
24 )

```

[7] を実行する際、最初の引数のバケットの名前を、**自分自身のバケットの名前に置き換えることに注意しよう。(先ほど S3 コンソールから確認した `simplebatch-bucketXXXX` のことである。)**

[9] 番のセルで、CSV のデータをプロットしている (Figure 63)。ローカルで実行したときと同じように、AWS Batch を用いて MNIST モデルを訓練することに成功した!

```

1 fig, (ax1, ax2) = plt.subplots(1,2, figsize=(9,4))
2 x = [i for i in range(df.shape[0])]
3 ax1.plot(x, df["train_loss"], label="Train")
4 ax1.plot(x, df["val_loss"], label="Val")
5 ax2.plot(x, df["val_accuracy"])
6
7 ax1.set_xlabel("Epochs")
8 ax1.set_ylabel("Loss")
9 ax1.legend()
10
11 ax2.set_xlabel("Epochs")
12 ax2.set_ylabel("Accuracy")

```

```

Best loss: 0.2320499013662338
Best loss epoch: 0
Best accuracy: 99.15833333333332
Best accuracy epoch: 67

```

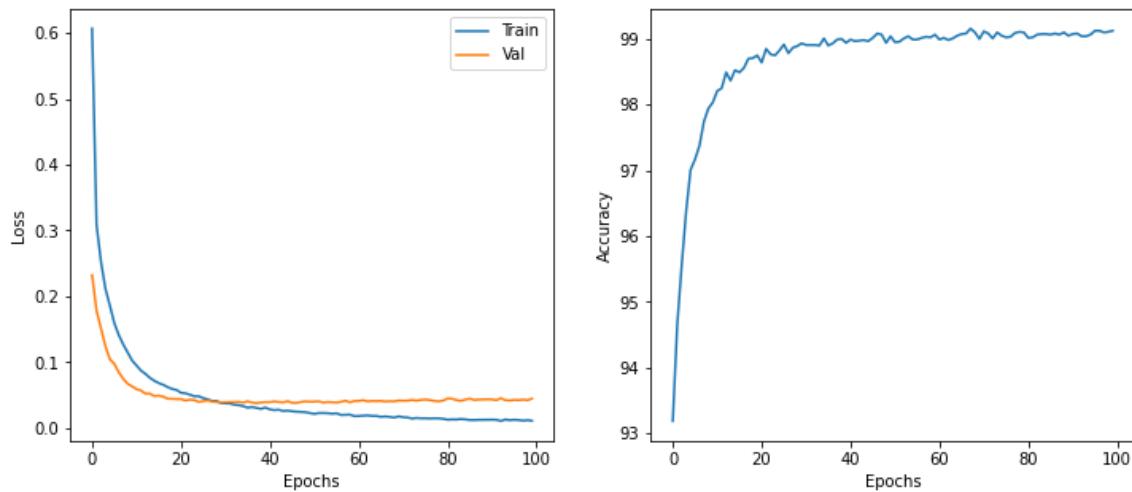


Figure 63. AWS Batch で行った MNIST モデルの学習の結果

9.10. 並列にたくさんの Job を実行する

さて、ここからが最後の仕上げである。ここまでハンズオンで構築した AWS Batch のシステムを使って、ハイパーパラメータサーチを実際に行おう。

先ほど実行した `run_single.ipynb` と同じディレクトリにある `run_sweep.ipynb` を開く。

セル [1], [2], [3] は `run_single.ipynb` と同一である。セル[4] の for ループを使って、グリッド状にハイパーパラメータの組み合わせを用意し、batch にジョブを投入している。

セル [4] を実行したら、Batch のコンソールを開こう。先ほどと同様に、ジョブのステータスは **SUBMITTED** > **RUNNABLE** > **STARTING** > **RUNNING** と移り変わっていくことがわかるだろう。最終的に 9 個のジョブがすべて **RUNNING** の状態になることを確認しよう (Figure 64)。また、このとき Compute environment の **Desired vCPUs** は $4 \times 9 = 36$ となっていることを確認しよう (Figure 64)。



Dashboard

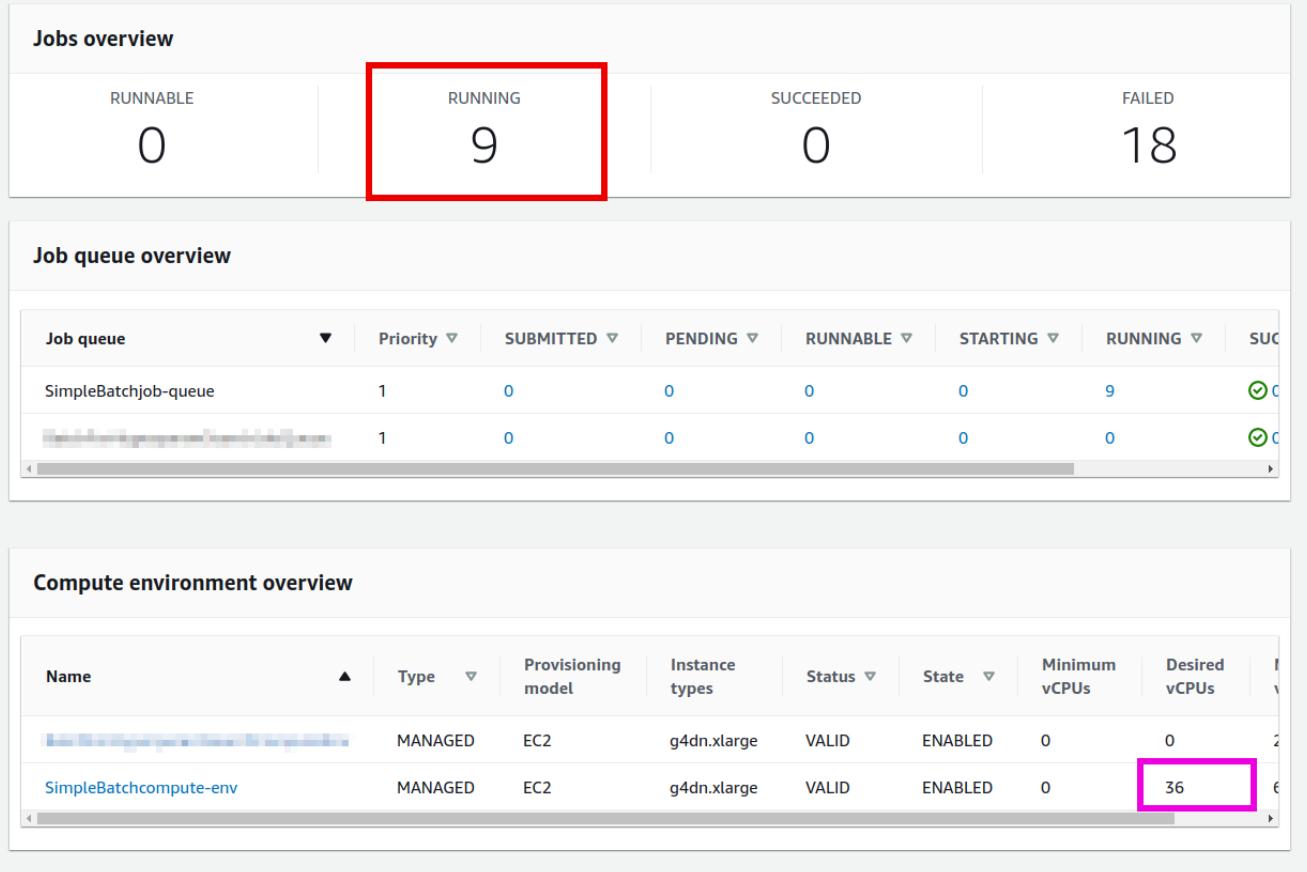


Figure 64. 複数のジョブを同時投入したときの Batch コンソール

次に, Batch のコンソールの左側のメニューから **Jobs** をクリックしてみよう. ここでは, 実行中の Job の一覧が確認することができる (Figure 65). Job のステータスでフィルタリングすることも可能である. 9個のジョブがどれも **RUNNING** 状態にあることが確認できるだろう.

Jobs (9)

Job queue	Status	Clone job	Cancel job	Terminate job	Submit new job	
SimpleBatchjob-queue	RUNNING	<	1	>		
Name	ID	Started at	Stopped at	Total run time	Status	
lr01_m05	d965ff7d-82f1-47d6-ae05-43b04736c096	Jun 14 2021 00:17:04	--	--	RUNNING	
lr01_m01	02c86c47-c1d0-4ee4-9820-2a57dcc5bef2	Jun 14 2021 00:19:45	--	--	RUNNING	
lr01_m005	f1e22ad2-b1c4-4dc2-9a6d-a07ae42feb87	Jun 14 2021 00:19:48	--	--	RUNNING	
lr001_m05	a6ca5b9b-1178-497b-9651-5e936aec7b1e	Jun 14 2021 00:17:04	--	--	RUNNING	
lr001_m01	ed2e8cf8-21d6-4c00-b82f-3acf909104d4	Jun 14 2021 00:17:04	--	--	RUNNING	
lr001_m005	c52466ea-afcc-4ac8-9523-cc7ca3703cd8	Jun 14 2021 00:20:13	--	--	RUNNING	
lr0001_m05	367723ea-5d95-4263-ad9f-30d69e2c3dee	Jun 14 2021 00:19:45	--	--	RUNNING	
lr0001_m01	f3a4146d-9b5a-43f2-97a4-102f360bf6e6	Jun 14 2021 00:19:47	--	--	RUNNING	
lr0001_m005	e67eda8d-0088-450e-9dd0-b02335162080	Jun 14 2021 00:20:15	--	--	RUNNING	

Figure 65. 複数のジョブを同時投入したときの Job 一覧

今度は EC2 コンソールを見てみよう. 左のメニューから **Instances** を選択すると, Figure 66 に示すような起動中のインスタンスの一覧が表示される. **g4dn.xlarge** が 9 台稼働しているのが確認できる. Batch がジョブの投下に合わせて必要な数のインスタンスを起動してくれたのだ!

Instances (11) Info			Connect	Instance state ▾	Actions ▾	Launch instances	▼	
		<input type="text"/> Filter Instances						
	Name ▾	Instance ID	Instance state ▾	Instance type ▾	Status check	Alarm status	Availability Zone ▾	Public IPv4 DNS
□	[REDACTED]	[REDACTED]						3
□	[REDACTED]	[REDACTED]						[REDACTED]
□	-	i-0ba247396a4a863b1		Running		g4dn.xlarge		2/2 checks passed No alarms + ap-northeast-1a ec2-52-196-156-
□	-	i-0eb5e9cf4c680cdb7		Running		g4dn.xlarge		2/2 checks passed No alarms + ap-northeast-1a ec2-18-183-239-
□	-	i-01a568c61751d48f8		Running		g4dn.xlarge		2/2 checks passed No alarms + ap-northeast-1a ec2-13-114-41-7
□	-	i-0965f5df949d6d64d		Running		g4dn.xlarge		2/2 checks passed No alarms + ap-northeast-1a ec2-52-69-53-10
□	-	i-071424513eff9b111		Running		g4dn.xlarge		2/2 checks passed No alarms + ap-northeast-1a ec2-52-68-193-1
□	-	i-0b3b7b6b58728558b		Running		g4dn.xlarge		2/2 checks passed No alarms + ap-northeast-1a ec2-3-113-30-84
□	-	i-04548920d31a2e134		Running		g4dn.xlarge		2/2 checks passed No alarms + ap-northeast-1a ec2-3-112-221-2
□	-	i-00e1428460a61d067		Running		g4dn.xlarge		2/2 checks passed No alarms + ap-northeast-1a ec2-3-112-72-23
□	-	i-025a30bf863674374		Running		g4dn.xlarge		2/2 checks passed No alarms + ap-northeast-1a ec2-18-183-170-

Figure 66. 複数のジョブを同時投入したときの EC2 インスタンスの一覧

ここまで確認できたら、それぞれの Job が終了するまでしばらく待とう（だいたい 15 分くらいで終わる）。すべてのジョブが終了すると、ダッシュボードの **SUCCEEDED** が 9 となっているはずだ (<>>). また、Compute environment の **Desired vCPUs** も 0 に落ちていることを確認しよう。最後に EC2 コンソールに行って、すべての g4dn インスタンスが停止していることを確認しよう。

以上から、AWS Batch を使うことで、ジョブの投入に応じて自動的に EC2 インスタンスが起動され、ジョブの完了とともに直ちにインスタンスの停止が行われる一連の挙動を観察することができた。

さて、再び `run_sweep.ipynb` に戻ってこよう。

[5] 以降のセルでは、グリッドサーチの結果を可視化している。

```

# [5]
import pandas as pd
import numpy as np
import io
from matplotlib import pyplot as plt

# [6]
def read_table_from_s3(bucket_name, key, profile_name=None):
    if profile_name is None:
        session = boto3.Session()
    else:
        session = boto3.Session(profile_name=profile_name)
    s3 = session.resource("s3")
    bucket = s3.Bucket(bucket_name)

    obj = bucket.Object(key).get().get("Body")
    df = pd.read_csv(obj)

    return df

# [7]
grid = np.zeros((3,3))
for (i, lr) in enumerate([0.1, 0.01, 0.001]):
    for (j, m) in enumerate([0.5, 0.1, 0.05]):
        key = f"metrics_lr{lr:0.4f}_m{m:0.4f}.csv"
        df = read_table_from_s3("simplebatch-bucket43879c71-mbqaltx441fu", key)
        grid[i,j] = df["val_accuracy"].max()

# [8]
fig, ax = plt.subplots(figsize=(6,6))
ax.set_aspect('equal')

c = ax.pcolor(grid, edgecolors='w', linewidths=2)

for i in range(3):
    for j in range(3):
        text = ax.text(j+0.5, i+0.5, f"{grid[i, j]:0.1f}",
                      ha="center", va="center", color="w")

```

最終的に出力されるプロットが [Figure 67](#) である。

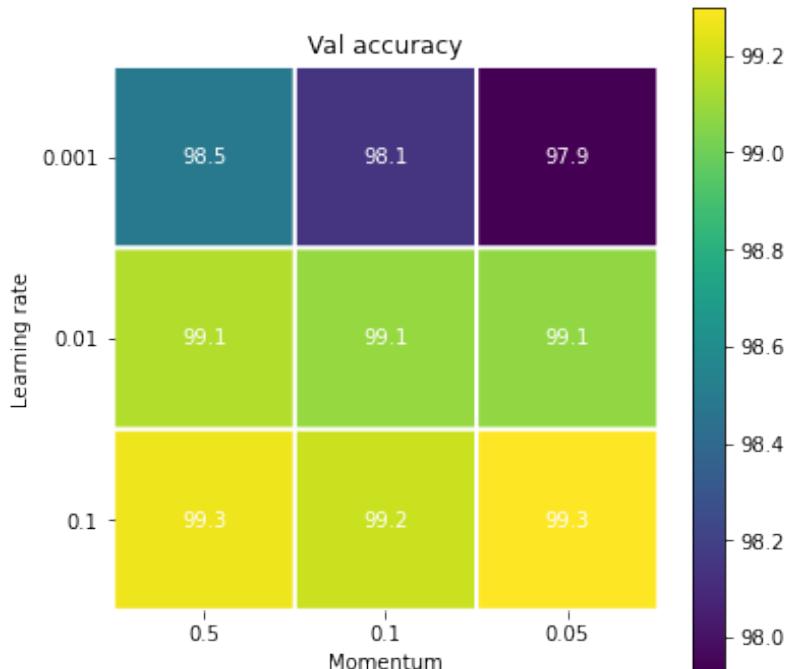


Figure 67. ハイパーパラメータのグリッドサーチの結果

このプロットから、差は僅かであるが、学習率が 0.1 のときに精度は最大となることがわかる。また、学習率 0.1 のときはモメンタムを変えても大きな差は生じないことが見て取れる。

今回のパラメータサーチは学習用として極めて単純化されたものである点は承知いただきたい。



例えば、今回は学習率が 0.1 が最も良いとされたが、それは訓練のエポックを 100 に限定しているからかもしれない。学習率が低いとその分訓練に必要なエポック数が多くなる。訓練のエポック数をもっと増やせばまた違った結果が観察される可能性はある。

また、今回は MNIST の訓練データ 60,000 枚のうち、48,000 枚を訓練データ、残り 12,000 枚を検証データとして用いた。この分割は乱数を固定してランダムに行ったが、もしこの分割によるデータのバイアスを気にするならば、分割の乱数を変えて複数回モデルの評価を行う (**k-fold cross-validation**) のも、より精緻なアプローチとして考えられる。

以上のようにして、CNN を用いた MNIST 分類モデルのハイパーパラメータの最適化の一連の流れを体験した。今回紹介したスタックは、ECR に置く Docker イメージを入れ替えることで、任意のプログラムを実行することができる。興味にある読者は、ぜひ自分自身のアプリケーションを実行してもらいたい。

9.11. スタックの削除

これにて、本ハンズオンは終了である。最後にスタックを削除しよう。

今回のスタックを削除するにあたり、ECR に配置された Docker のイメージは手動で削除されなければならない。(これをしないと、`cdk destroy` を実行したときにエラーになってしまう。これは CloudFormation の仕様なのでどうしようもない)。

ECR の Docker image を削除するには、ECR のコンソールに行き、イメージが配置されたレポジトリを開く。そして、画面右上の **DELETE** ボタンを押して削除する (Figure 68)。

simplebatch-repository9f1a3f0b-zfcfedjaz8q30

[View push commands](#)[Edit](#)

Images (1)

[Delete](#)[Scan](#) Find images

< 1 >



<input checked="" type="checkbox"/>	Image tag	Pushed at	Size (MB)	Image URI	Digest	Scan status	Vulnerabilities
<input checked="" type="checkbox"/>	latest	Jun 07, 2021 11:44:27 PM	4676.39	Copy URI	sha256:1f44b04...	-	-

Figure 68. ECR から Docker image を削除する

これが完了したうえで、次のコマンドでスタックを削除する。

```
$ cdk destroy
```

Chapter 10. Web サービスの作り方

ここからが、第三回目の講義の内容になる。

これまでの講義では、仮想サーバーをクラウド上に起動し、そこで計算を走らせる方法について解説してきた。最初に、EC2上に個人で使用するためのサーバーを立ち上げ、機械学習の計算を実践した。第二回の最後では、大規模な機械学習システムのもっとも初歩的なものとして、ECSとDockerを使ったクラスターを作成する方法を解説した。ここまで紹介したクラウドの計算の利用は、個人的な用途に限られていたが、クラウドの重要な側面のひとつとして、広く一般に使ってもらえるような計算サービス・データベースを提供する、というものが挙げられるだろう。

今回の講義は、前回までとは少し方向性を変え、どのようにしてクラウド上にアプリケーションを展開し、広く一般の人に使ってもらうか、という点を講義したいと思う。講義を通じて、どのように世の中のウェブサービスが出来上がっているのかを知り、さらにどうやって自分でそのようなアプリケーションを作るのか、という点を学んでもらう。その過程で、Serverlessアーキテクチャという最新のクラウド設計手法を解説する。

10.1. ウェブサービスの仕組み – Twitter を例に

あなたがパソコンやスマートフォンからTwitter, Facebook, YouTubeなどのウェブサービスにアクセスしたとき、実際にどのようなことが行われ、ページがロードされているのだろうか？

ここは知っている人も多いと思うので簡潔な説明にとどめるが、Twitterを具体例として、背後にあるサーバーとクライアントの間の通信を概説しよう。概念図としてはFigure 69のような通信がクライアントとサーバーの間で行われていることになる。

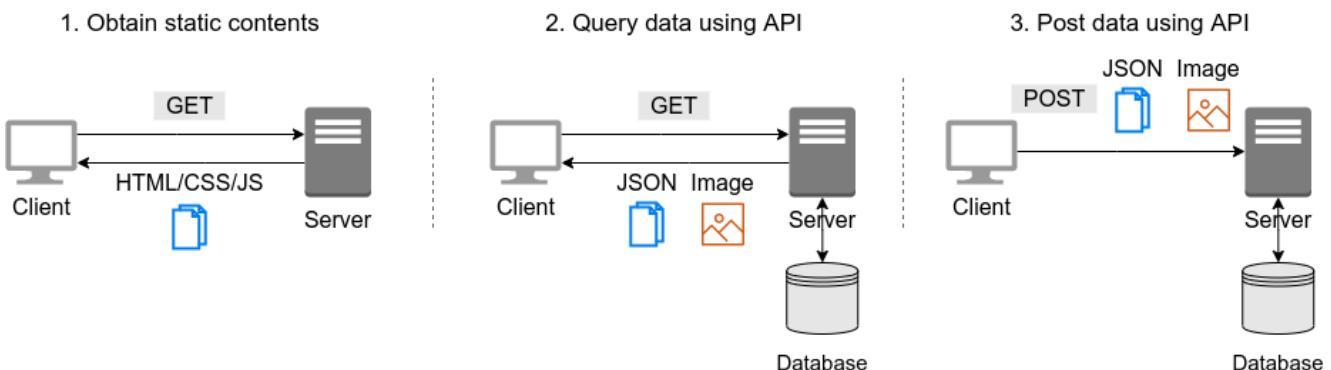


Figure 69. クライアントと Web サーバーの通信の概念図

前提として、クライアントとサーバーの通信は **HTTP (Hypertext Transfer Protocol)** を使って行われる。また、最近では、暗号化された HTTP である **HTTPS** を用いることがスタンダードになってきている。第一のステップとして、クライアントは HTTP(S) 通信によってサーバーから静的なコンテンツを取得する。静的なコンテンツとは、**HTML (Hypertext Markup Language)** で記述されたウェブページの文書本体、**CSS (Cascading Style Sheets)** で記述されたページのデザインやレイアウトファイル、そして **JavaScript (JS)** で記述されたページの動的な挙動を定義したプログラム、が含まれる。Twitterを含む現代的なウェブアプリケーションの設計では、この静的なファイル群はページの“枠”を定義するだけで、中身となるコンテンツ (e.g. ツイートの一覧) は別途 **API (Application Programming Interface)** によって取得されなければならない。そこで、クライアントは先ほど取得された JavaScript で定義されたプログラムに従って、サーバーに API を送信し、ツイートや画像データを取得する。この際、テキストデータのやり取りには **JSON (JavaScript Object Notation)** というフォーマットが用いられることが多い。画像や動画などのコンテンツも同様に API により取得される。このようにして取得されたテキストや画像が、HTMLの文書に埋め込まれることで、最終的にユーザーに提示されるページが完成するのである。また、新しいツイートを投稿するときにも、クライアントから API を通じてサーバーのデータベースにデータが書き込まれる。

10.2. REST API

API (Application Programming Interface) とはこれまで何度も出てきた言葉であるが、ここではよりフォーマルな定義付けを行う。API とはあるソフトウェア・アプリケーションが、外部のソフトウェアに対しコマンドやデータをや

りするための"媒介"の一般的総称である。とくに、ウェブサービスの文脈では、サーバーが外界に対して提示しているコマンドの一覧のことを意味する。クライアントは、提示されている API から適切なコマンドを使うことによって、所望のデータを取得したり、あるいはサーバーにデータを送信したりする。

特に、**REST (Representational State Transfer)** とよばれる設計思想に基づいた API が現在では最も一般的に使われている。REST に従った API のことを **REST API** あるいは **RESTful API** と呼んだりする。

REST API は、Figure 70 に示したような **Method** と **URI (Universal Resource Identifier)** の組からなる。



Figure 70. REST API

Method (メソッド) とは、"どのような操作を行いたいか"を抽象的に表す ("動詞"と捉えてもよい)。REST API では典型的には Table 5 に示したメソッドが用いられる。

一方、URI は、操作が行われる対象 (リソースとも呼ばれる) を表す。メソッドが動詞であることに対して、URI は"目的語"であると捉えても良い。Figure 70 の例で言えば、/status/home_timeline というリソース (ホームタイムラインのツイートの一覧) を取得せよ、という意味になる。

Table 5. REST API Methods

メソッド	動作
GET	要素を取得する
POST	新しい要素を作成する
PUT	既存の要素を新しい要素と置き換える
PATCH	既存の要素の一部を更新する
DELETE	要素を削除する

REST API のメソッドには、Table 5 で挙げたもの以外に、HTTP プロトコルで定義されている他のメソッド (OPTIONS, TRACE など) を用いることもできるが、あまり一般的ではない。



また、これらのメソッドだけでは動詞として表現し切れないこともあるが、URI のパスなどでより意味を明確にすることもある。メソッドの使い方も、要素を削除する際は必ず **DELETE** を使わなければならない、という決まりもなく、例えば、Twitter API でツイートを消す API は **POST statuses/destroy/:id** で定義されている。

最終的には、各ウェブサービスが公開している API ドキュメンテーションを読んで、それぞれの API がどんな操作をするのかをしっかりと調べる必要がある。

10.2.1. Twitter API

もう少し具体的にウェブサービスの API を体験する目的で、ここでは Twitter の API を見てみよう。

Twitter が提供している API の一覧は [このページ](#) で見ることができる。いくつかの代表的な API を Table 6 にまとめた。

Table 6. Twitter API

GET /statuses/home_time_line	ホームのタイムラインのツイートの一覧を取得する。
------------------------------	--------------------------

<code>GET statuses/show/:id</code>	<code>:id</code> で指定されたツイートの詳細情報を取得する.
<code>POST statuses/update</code>	新しいツイートを投稿する.
<code>POST statuses/retweet/:id</code>	<code>:id</code> で指定されたツイートをリツイートする.
<code>POST favorites/create/:id</code>	<code>:id</code> で指定されたツイートを"いいね"する.
<code>POST statuses/destroy/:id</code>	<code>:id</code> で指定されたツイートを削除する.

Twitter のアプリまたはウェブサイトを開くと、背後では上記のような API が実行され、結果として GUI のページがレンダリングされている。また、Twitter 上でボット (bot) を作るときは、開発者がこれらの API を自動で呼ぶようなプログラムを記述することで出来上がっている。

このように、API はあらゆるウェブサービスを作る上で一番基礎となる要素である。次からの章では、どのようにしてクラウド上に API を構築していくかを解説しよう。

Chapter 11. Serverless architecture

Serverless Architecture あるいは Serverless Computing とは、従来とは全くアプローチの異なるクラウドシステムの設計方法である。歴史的には、AWS が2014年に発表した [Lambda](#) がサーバレスアーキテクチャの最初の先駆けとされている。その後、Google や Microsoft などのクラウドプラットフォームも同様の機能の提供を開始している。サーバレスアーキテクチャの利点は、スケーラブルなクラウドシステムを安価かつ簡易に作成できる点であり、近年いたるところで導入が進んでいる。

Serverless とは、文字通りの意味としてはサーバーなしで計算をするということになるが、それは一体どういう意味だろうか？サーバレスについて説明するためには、まずは従来的な、"serverful" と呼ばれるようなシステムについて解説しなければならない。

11.1. Serverful クラウド（従来型）

従来的なクラウドシステムのスケッチを [Figure 71](#) に示す。クライアントから送信されたリクエストは、まず最初にAPIサーバーに送られる。API サーバーでは、リクエストの内容に応じてタスクが実行される。タスクには、API サーバーだけで完結できるものもあるが、多くの場合、データベースの読み書きが必要である。データベースには、データベース専用の独立したサーバーマシンが用いられることが一般的である。また、画像や動画などもデータは、また別のストレージサーバーに保存されることが一般的である。これらの API サーバー、データベースサーバー、ストレージサーバーはそれぞれ独立したサーバーマシンであり、AWS では EC2 を使った仮想インスタンスを想定してもらったら良い。

多くのウェブサービスでは、多数のクライアントからのリクエストを処理するため、複数のサーバーマシンがクラウド内で起動し、負荷を分散するような設計がなされている。クライアントから来たリクエストを計算容量に余裕のあるサーバーに振り分けるような操作を **Load balancing** とよび、そのような操作を担当するマシンのことを **Load balancer** という。

Load balancing の目的でたくさんのインスタンスを起動するのはよいのだが、それぞれがなんの計算もせず、ただ新しいタスクが来るのを待っているようではコストと電力の無駄遣いである。したがって、全てのサーバーが常に目標とする計算負荷を維持するよう、計算の負荷に応じてクラスター内の仮想サーバーの数を動的に増減させるような仕組みが必要である。そのような仕組みを **クラスターのスケーリング** とよび、負荷の増大に応答して新しい仮想インスタンスをクラスターに追加する操作を **scale-out**、負荷の減少に応答してインスタンスをシャットダウンする操作を **scale-in** と呼ぶ。クラスターのスケーリングは、各インスタンスを監視・統括するようなひとつ階層が上のサーバーを配置することで自動的に実行されるような設計がなされる。クラスターのスケーリングは、API サーバーではもちろんのこと、データベースサーバー・ストレージサーバーでも必要になることが多い。クラウドシステム内すべてのインスタンスで、負荷が均一になるような調整が必要なのである。

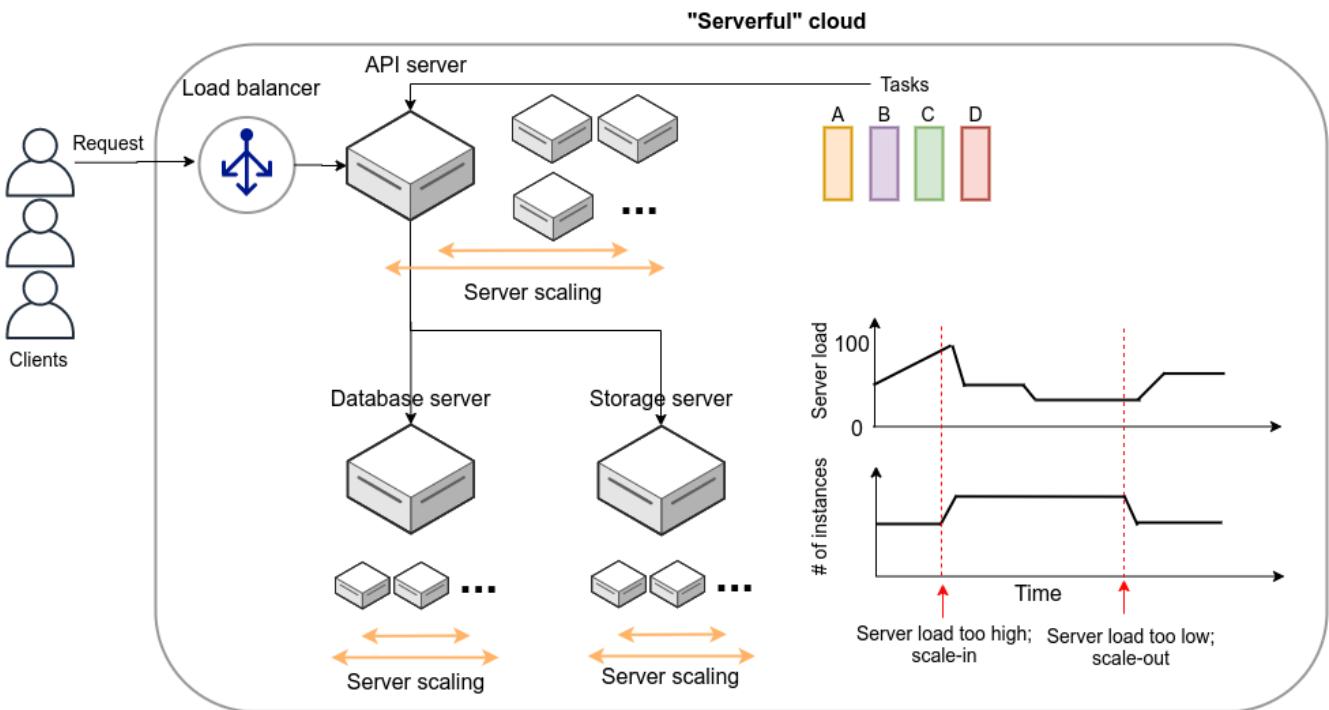


Figure 71. Serverful なクラウドシステム

11.2. Serverless クラウドへ

上述したように、従来のクラウドシステムの設計で非常に重要なのが、クラスターのスケーリングである。コストパフォーマンスを最大化するには、各サーバーの稼働率を100%に近づけるようなスケーリングのパラメータの調整が必要である。しかしながら、クラスターのスケーリングの最適化はかなり手間のかかる作業である。

さらに問題を複雑にするのは、APIサーバーで処理されるべきタスクが、非一様である点である。非一様であるとは、例えばタスクAは3000ミリ秒の実行時間と512MBのメモリーを消費し、別のタスクBは1000ミリ秒の実行時間と128MBのメモリーを消費する、というような状況を差している。一つのサーバーマシンが計算負荷が異なる複数のタスクを処理する場合、クラスターのスケーリングはより複雑になる。この状況をシンプルにするために、1サーバーで実行するタスクは1種類に限る、という設計も可能であるが、そうすると生まれる弊害も多い（ほとんど使われないタスクに対してもサーバー一台をまるまる割り当てなければならない=ほとんどアイドリング状態になってしまうなど）。

もっとシンプルで見通しの良いクラウドシステムのスケーリングの仕組みはないだろうか？

従来の serverful なシステムでの最大の問題点は、**サーバーをまるまる占有してしまう**という点にある。すなわち、EC2 インスタンスを起動したとき、そのインスタンスは起動したユーザーだけが使えるものであり、**計算のリソース (CPUやRAM)** が独占的に割り当てられた状態になる。固定した計算資源の割り当てがされてしまっているので、**インスタンスの計算負荷が0%であろうが100%であろうが、均一の使用料金が起動時間に比例して発生する**。

サーバーレスアーキテクチャは、このような **独占的に割り当てられた計算リソース** というものを完全に廃止する。サーバーレスアーキテクチャでは、計算のリソースは、クラウドプロバイダーが全て管理する。クライアントは、仮想インスタンスを一台まるごと借りるのでなく、**実行したいプログラムをクラウドに提出する**。クラウドプロバイダーは、自身の持つ巨大な計算リソースから空きを探し、提出されたプログラムを実行し、実行結果をクライアントに返す。以上を図示すると、Figure 72 のようになる。

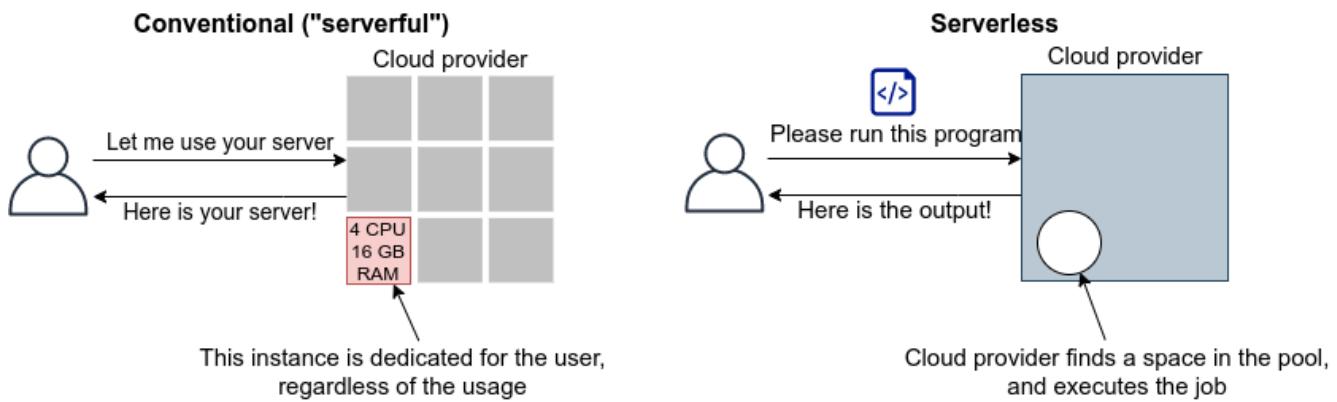


Figure 72. 従来のクラウドと Serverless クラウドの比較

サーバレスクラウドを利用することで、**クラウドのコストは実際に使用した計算の総量 (CPU稼働時間) で決定されること**になる。これは、計算の実行総量に関わらずインスタンスの起動時間で料金が決定されていた従来のシステムと比べて大きな違いである。一方で、クライアントが同時に大量のタスクを送信した場合でも、クラウドプロバイダー側はその需要に応えることのできるような計算リソースを瞬時に割り当てる所以ができるので、非常に高いスケーラビリティを実現することができる。

従来型の(仮想インスタンスをたくさん起動するような)クラウドシステムは、賃貸と似ているかもしれない。部屋を借りるというのは、その部屋でどれだけの時間を過ごそうが、月々の家賃は一定である。同様に、仮想サーバーも、それがどれほどの計算を行っているかに関わらず、一定の料金が時間ごとに発生する。

一方で、サーバレスクラウドは、電気・水道・ガス料金と似ている。こちらは、(ある程度の基本料金はあるかもしれないが) 実際に使用した分で料金が決定されている。サーバレスクラウドも、実際に計算を行った総時間で料金が決まる仕組みになっている。

11.3. Lambda



AWS でサーバレスコンピューティングの中心を担うのが、[Lambda](#) である。

Lambda の使い方を [Figure 73](#) に図示している。Lambda の仕組みはシンプルで、まずユーザーは実行したいプログラムを予め登録しておく。プログラムは、Python, Node.js, ruby などの主要な言語がサポートされている。そして、プログラムを実行したいときに、そのプログラムを実行 (invoke する) コマンドを Lambda に送信する。Lambda では、invoke のリクエストを受け取ると直ちに (数ミリセカンドから数百ミリセカンドのレイテンシーで) プログラムの実行を開始する。そして、実行結果をクライアントやその他の計算機に返す。

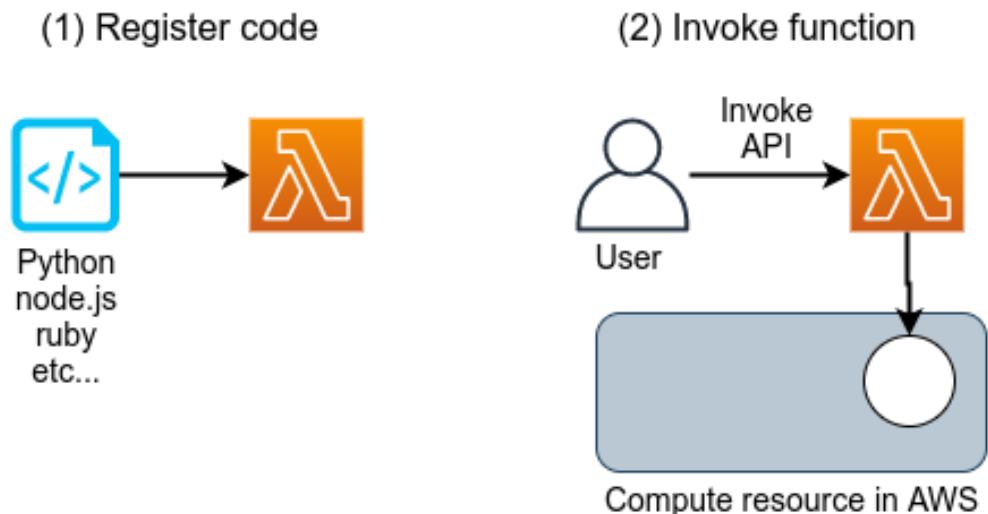


Figure 73. AWS Lambda

このように,Lambdaは仮想インスタンスを専有することはない. invokeのリクエストが来たときにのみ,動的に起動し,実行の終了とともに速やかにシャットダウンされる. また,同時に複数のリクエストが来た場合でも,AWSはそれらを実行するための計算リソースを割り当て,並列的に処理を行ってくれる. 原理上は,**数千から数万のリクエストが同時に来たとしても,Lambdaはそれらを同時に実行することができる**. このような,占有された仮想サーバーの存在なしに,動的に関数を実行するサービスを **FaaS (Function as a Service)** と呼ぶ.

Lambdaでは128MBから3008MBのメモリーを使用することができる(2020/06時点). 実行時間は100ミリ秒の単位で記録され,実行時間に比例して料金が決定される. [Table 7](#)はLambdaの利用料金表である.

Table 7. Lambda の料金表

Memory (MB)	Price per 100ms
128	\$0.0000002083
512	\$0.0000008333
1024	\$0.0000016667
3008	\$0.0000048958

例えば,128MBのメモリーを使用する関数を,それぞれ200ミリ秒,合計で100万回実行した場合,0.0000002083 * 2 * 10^6 = **\$0.4**の料金となる. ウェブサーバーのデータベースの更新など簡単な計算であれば,200ミリ秒程度で実行できる関数も多いことから,100万回データベースの更新を行ったとしても,たった \$0.4 しかコストが発生しないことになる.

11.4. サーバーレスストレージ: S3



サーバーレスの概念は,ストレージにも拡張されている.

従来的なストレージ(ファイルシステム)では,必ずホストとなるマシンとOSが存在しなければならない. 従って,それほどパワーは必要ないまでも,ある程度のCPUリソースを割かなければならない. また,従来的なファイルシステムでは,データ領域のサイズは最初に作成するときに決めなければならず,後から容量を増加させることはしばしば困難である(ZFSなどのファイルシステムを使えばある程度は自由にファイルシステムのサイズを増減できるが). よって,従来的なクラウドでは,ストレージを借りるときには予めディスクのサイズを指定せねばならず,ディ

スクの容量が空であろうと満杯であろうと、同じ利用料金が発生することになる。

Simple Storage Service (S3) は、サーバーレスなストレージシステムを提供する。S3 では、予めデータ保存領域の上限は定められていない。データを入れれば入れた分だけ、保存領域は拡大していく（仕様上はペタバイトスケールのデータを保存することが可能である）。ストレージにかかる料金も、保存してあるデータの総容量で決定される。

その他、データの冗長化やバックアップなど、通常ならば CPU が介在しなければならない操作も、API を通じて行うことができる。これらの観点から、S3 も サーバーレスクラウドの一部として取り扱われることが一般的である。

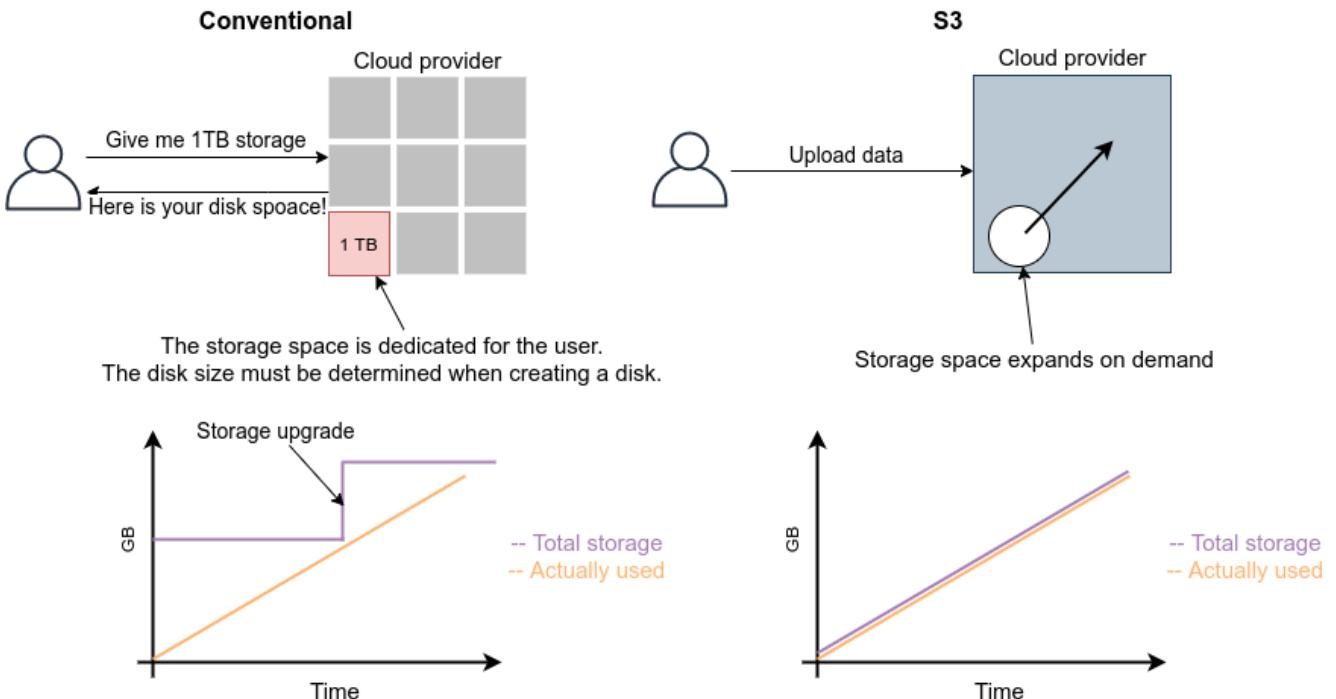


Figure 74. S3 と従来的なファイルシステムの比較

S3 の料金は、保存してあるデータの総容量と、外部へのデータ転送の総量で決定される（参考）。執筆時点では、データの保存には \$0.025 per GB per month のコストが発生する。従って、1000GB のデータを S3 に一ヶ月保存した場合、\$25 の料金が発生することになる。また、S3 はデータを外に取り出す際の通信にもコストが発生する。執筆時点では、S3 からインターネットを通じて外部にデータを転送すると \$0.114 per GB のコストが発生する。データを S3 に入れる（data-in）通信は無料で行える。また、AWS の同じ Region 内のサービス（Lambda など）にデータを転送するのは無料である。AWS の Region をまたいだデータの転送には、\$0.09 per GB のコストが発生する。

11.5. サーバーレスデータベース: DynamoDB



サーバーレスの概念は、データベースにも適用することができる。

ここでいうデータベースとは、Web サービスなどにおけるユーザー情報を記録しておくための保存領域のことを指している。従来的に有名なデータベースとしては MySQL, PostgreSQL, MongoDB などが挙げられる。データベースと普通のストレージの違いは、データの検索機能にある。普通のストレージではデータは単純にディスクに書き込まれるだけだが、データベースでは検索がより効率的になるようなデータの配置がされたり、頻繁にアクセスされるデータはメモリーにキャッシュされるなどの機能が備わっている。これにより、巨大なデータの中から、興味のある要素を高速に取得することができる。

このような検索機能を実現するには,当然CPUの存在が必須である.従って,従来的なデータベースを構築する際は,ストレージ領域に加えて,たくさんのCPUを搭載したマシンが用いられることが多い.また,格納するデータが巨大な場合は複数マシンにまたがった分散型のシステムが設計される.分散型システムの場合は,Section 11.1で議論したようにデータベースへのアクセス負荷に応じて適切なスケーリングがなされる必要がある.

DynamoDBは,サーバーレスなデータベースである.

DynamoDBは分散型のデータベースであるが,データベースのスケーリングはAWSによって行われる.ユーザーとしては,特に考えずに,送りたいだけのリクエストをデータベースに送信すればよい.データベースへの負荷が増減したときのスケーリングは,DynamoDBが自動で行ってくれる.

11.6. その他のサーバーレスクラウドの構成要素

その他,サーバーレスクラウドを構成するための構成要素を以下にあげる. API Gatewayについては,ハンズオン#5で触れる.

- [API Gateway](#): APIを構築する際のルーティングを担う.
- [Fargate](#): ハンズオン第三回で触れたFargateも,サーバーレスクラウドの要素の一部である. Lambdaでは実行できないような,メモリーや複数CPUを要するような計算などを行うために用いる.
- [Simple Notification Service \(SNS\)](#): サーバーレスのサービス間(LambdaとDynamoDBなど)でイベントをやり取りするためのサービス.
- [Step Functions](#): サーバーレスのサービス間のオーケストレーションを担う.

サーバーレスアーキテクチャは万能か?

この問への答えは,筆者はNOであると考える.

ここまで,サーバーレスの利点を強調して説明してきたが,まだまだ新しい技術なだけに,欠点,あるいはサーバーフルなシステムに劣る点は,数多くある.



ひとつ大きな欠点をあげるとすれば,サーバーレスのシステムは各クラウドプラットフォームに固有なものなので,特定のプラットフォームでしか運用できないシステムになってしまう点であろう. AWSで作成したサーバーレスのシステムを,Googleのクラウドに移植するには,かなり大掛かりなプログラムの書き換えが必要になる.一方,serverfulなシステムであれば,プラットフォーム間のマイグレーションは比較的簡単に行うことができる. クラウドプロバイダーとしては,自社のシステムへの依存度を強めることで,顧客を離さないようにするという狙いがあるのだろう…

その他,サーバーレスコンピューティングの欠点や今後の課題などは,次の論文で詳しく議論されている.興味のある読者は読んでみると良い.

- Hellerstein et al., "Serverless Computing: One Step Forward, Two Steps Back"
arXiv (2018)

Chapter 12. Hands-on #4: サーバーレス入門

第四回目のハンズオンでは、サーバーレスクラウドを構成する主要な構成要素について、実際に動かしながら学んでもらう。

今回のハンズオンで触れるのは以下の3つである。

- Lambda: サーバーレスの計算エンジン
- DynamoDB: サーバーレス・データベース

ハンズオンのソースコードはこちらのリンクに置いてある ⇒ <https://gitlab.com/tomomano/intro-aws/-/tree/master/handson/04-serverless>

12.1. Lambda ハンズオン

ここでは、ショートハンズオンとして、実際に AWS 上に Lambda を使った関数を定義し、計算を実行してみよう。ここでは、AWS CDK を利用してとてもシンプルな Lambda の関数を作成する。

ハンズオンのソースコードはこちらに置いてある ⇒ <https://gitlab.com/tomomano/intro-aws/-/tree/master/handson/04-serverless/lambda>



このハンズオンは、基本的に AWS Lambda の無料枠 の範囲内で実行することができる。

app.py にデプロイするプログラムが書かれている。中身を見てみよう。

```
1 ①
2 FUNC = """
3 import time
4 from random import choice, randint
5 def handler(event, context):
6     time.sleep(randint(2,5))
7     pokemon = ["Charmander", "Bulbasaur", "Squirtle"]
8     message = "Congratulations! You are given " + choice(pokemon)
9     print(message)
10    return message
11 """
12
13 class SimpleLambda(core.Stack):
14
15     def __init__(self, scope: core.App, name: str, **kwargs) -> None:
16         super().__init__(scope, name, **kwargs)
17
18     ②
19         handler = _lambda.Function(
20             self, 'LambdaHandler',
21             runtime=_lambda.Runtime.PYTHON_3_7,
22             code=_lambda.Code.from_inline(FUNC),
23             handler="index.handler",
24             memory_size=128,
25             timeout=core.Duration.seconds(10),
26             dead_letter_queue_enabled=True,
27         )
```

① ここで、Lambda で実行されるべき関数を定義している。これは非常に単純な関数で、2-5秒のランダムな時間スリープした後、["Charmander", "Bulbasaur", "Squirtle"] のいずれかの文字列をランダムに返す（これらは初代ポケットモンスターのゲームでオーキド博士にもらうヒトカゲ・フシギダネ・ゼニガメのことだ）。

② 次に, Lambda の関数の諸々のパラメータを設定している. それぞれのパラメータの意味は, 文字通りの意味なので明瞭であるが, 以下に解説する.

- `runtime=_lambda.Runtime.PYTHON_3_7`: ここでは, Python3.7 を使って上記で定義された関数を実行せよ, と指定している. Python3.7 の他に, Node.js, Java, Ruby, Go などの言語を指定することが可能である.
- `code=_lambda.Code.from_inline(FUNC)`: 実行されるべき関数が書かれたコードを指定する. ここでは, `FUNC=…` で定義した文字列を渡しているが, 文字列以外にもファイルのパスを渡すことも可能である.
- `handler="index.handler"`: これは, コードの中にいくつかのサブ関数が含まれているときに, メインとサブを区別するためのパラメータである. `handler` という名前の関数をメイン関数として実行せよ, という意味である.
- `memory_size=128`: メモリーは 128MB を最大で使用することを指定している. メモリーオーバーした場合は
- `timeout=core.Duration.seconds(10)` タイムアウト時間を10秒に設定している. 10秒以内に関数の実行が終了しなかった場合, エラーが返される.
- `dead_letter_queue_enabled=True`: アドバンストな設定なので説明は省略する.

上記のプログラムを実行することで, Lambda 関数がクラウド上に作成される. 早速デプロイしてみよう.

12.1.1. デプロイ

デプロイの手順は,これまでのハンズオンとほとんど共通である. ここでは, コマンドのみ列挙する (# で始まる行はコメントである). それぞれの意味を忘れてしまった場合は, ハンズオン1, 2に戻って復習していただきたい.

```
# プロジェクトのディレクトリに移動
$ cd intro-aws/handson/04-serverless/lambda

# venv を作成し, 依存ライブラリのインストールを行う
$ python3 -m venv .env
$ source .env/bin/activate
$ pip install -r requirements.txt

# AWS の認証情報をセットする
# 自分自身の認証情報に置き換えること !
export AWS_ACCESS_KEY_ID=XXXXXX
export AWS_SECRET_ACCESS_KEY=YYYYYY
export AWS_DEFAULT_REGION=ap-northeast-1

# デプロイを実行
$ cdk deploy
```

デプロイのコマンドが無事に実行されれば, Figure 75 のような出力が得られるはずである. ここで表示されている `SimpleLambda.FunctionName = XXXX` の XXXX の文字列は後で使うのでメモしておこう.



```
✓ SimpleLambda

Outputs:
SimpleLambda.FunctionName = SimpleLambda-LambdaHandler212865DC-10W5VV6HTHZI
```

Figure 75. CDKデプロイ実行後の出力

AWS コンソールにログインして, デプロイされたスタックを確認してみよう. コンソールから, Lambda のページに行くと Figure 76 のような画面から Lambda の関数の一覧が確認できる.

The screenshot shows the AWS Lambda Functions console. On the left, there's a sidebar with 'AWS Lambda' at the top, followed by 'Dashboard', 'Applications', 'Functions' (which is highlighted in orange), and 'Additional Resources'. The main area is titled 'Functions (1)'. It has a search bar with 'Filter by tags and attributes or search by keyword'. Below the search bar is a table with columns: 'Function name', 'Description', 'Runtime', 'Code size', and 'Last modified'. A single row is shown, representing the function 'SimpleLambda-LambdaHandler212865DC-10W5VV6HTHZZI'. The runtime is listed as 'Python 3.7', code size as '306 bytes', and last modified as '4 minutes ago'.

Figure 76. Lambda コンソール - 関数の一覧

今回のアプリケーションで作成したのが **SimpleLambda-YYYY** という名前のついた関数だ。関数の名前をクリックして、詳細を見てみる。すると Figure 77 のような画面が表示されるはずだ。先ほどプログラムの中で定義したPythonの関数がエディターから確認することができる。また、下の方にスクロールすると、関数の各種設定も確認することができる。

The screenshot shows the configuration page for the function 'SimpleLambda-LambdaHandler212865DC-10W5VV6HTHZZI'. At the top, there are tabs for 'Throttle', 'Qualifiers', 'Actions', 'Select a test event', 'Test', and 'Save'. Below these are tabs for 'Configuration' (which is selected), 'Permissions', and 'Monitoring'. The main area is divided into sections: 'Designer' (selected), 'Function code' (selected), and 'Info'. In the 'Function code' section, there's a code editor with Python code. The code imports time and random, defines a handler function that sleeps for a random duration between 2 and 5 seconds, selects a randompokemon from a list, and prints a congratulatory message. The code editor has tabs for 'File', 'Edit', 'Find', 'View', 'Go', 'Tools', 'Window', 'Save', 'Test', and 'Actions'.

Figure 77. Lambda コンソール - 関数の詳細

12.1.2. Lambda 関数の実行

それでは、作成した Lambda 関数を実際に実行 (invoke) してみよう。AWS の API を使うことで、関数の実行をスタートすることができる。今回は、[invoke_one.py](#) に関数を実行するための簡単なプログラムを提供している。興味のある読者はコードを読んでもらいたい。

以下のコマンドで、Lambda の関数を実行する。コマンドの **XXXX** の部分は、先ほどデプロイしたときに **SimpleLambda.FunctionName = XXXX** で得られた XXXX の文字列で置換する。

```
$ python invoke_one.py XXXX
```

すると、"Congratulations! You are given Squirtle" という出力が得られるはずだ。とてもシンプルではあるが、クラウド上で先ほどの関数が走り、乱数が生成された上で、ポケモンが選択されて出力が返されている。上の

コマンドを何度か打ってみて、実行のごとに違うポケモンが返されることを確認しよう。

さて、上のコマンドは、一度につき一回の関数を実行したわけであるが、Lambda の本領は一度に大量のタスクを同時に実行できる点である。そこで、今度は一度に100個のタスクを同時に送信してみよう。

以下のコマンドを実行する。XXXX の部分は上と同様に置き換える。第二引数の **100** は 100個のタスクを投入せよ、という意味である。

```
$ python invoke_many.py XXXX 100
```

すると以下のような出力が得られるはずだ。

```
.....  
Submitted 100 tasks to Lambda!
```

実際に、100 個のタスクが同時に実行されていることを確認しよう。Figure 77 の画面に戻り、"Monitoring" というタブがあるので、それをクリックする。すると、Figure 78 のようなグラフが表示されるだろう。

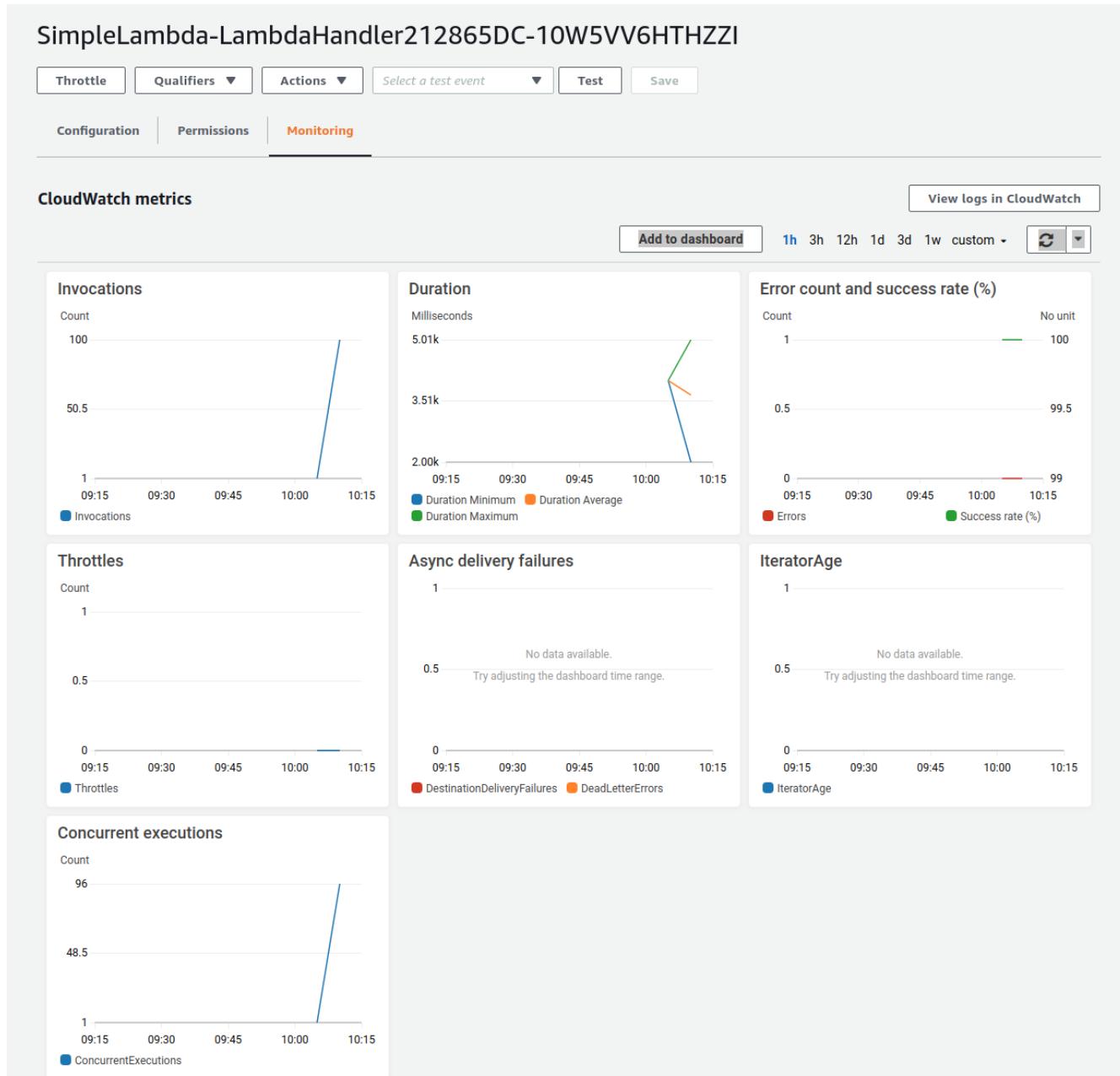


Figure 78. Lambda コンソール - 関数の実行のモニタリング



Figure 78 のグラフの更新には数分かかることがあるので, なにも表示されない場合は少し待つ.

Figure 78 で "Invocations" が関数が何度実行されたかを意味している. たしかに100回実行されていることがわかる. さらに, "Concurrent executions" が何個のタスクが同時に実行されたかを示している. ここでは 96 となっていることから, 96 個のタスクが並列的に実行されたことを意味している. (これが, 100 とならないのは, タスクの開始のコマンドが送られたのが完全には同タイミングではないことに起因する)

このように, 非常にシンプルではあるが, Lambda を使うことで, 同時並列的に処理を実行することのできるクラウドシステムを簡単に作ることができた.

もしこのようなことを従来的な serverful なクラウドで行おうとした場合, クラスターのスケーリングなど多くのコードを書くことに加えて, いろいろなパラメータを調節する必要がある.



興味がある人は, 一気に1000個などのジョブを投入してみると良い. が, あまりやりすぎると Lambda の無料利用枠を超えて料金が発生してしまうので注意.

12.1.3. スタックの削除

最後にスタックを削除しよう.

スタックを削除するには, 次のコマンドを実行すればよい.

```
$ cdk destroy
```

12.2. DynamoDB ハンズオン

ここでは, ショートハンズオンとして, 新しい DynamoDB のテーブルを作成する. そして実際にそこにデータの読み書きを行ってみる.

ハンズオンのソースコードはこちらに置いてある ⇒ <https://gitlab.com/tomomano/intro-aws/-/tree/master/handson/04-serverless/dynamodb>



このハンズオンは, 基本的に AWS DynamoDB の無料枠 の範囲内で実行することができる.

app.py にデプロイするプログラムが書かれている. 中身を見てみよう.

```
1 class SimpleDynamoDb(core.Stack):
2     def __init__(self, scope: core.App, name: str, **kwargs) -> None:
3         super().__init__(scope, name, **kwargs)
4
5         table = ddb.Table(
6             self, "SimpleTable",
7             partition_key=ddb.Attribute(
8                 name="item_id",
9                 type=ddb.AttributeType.STRING
10            ),
11            billing_mode=ddb.BillingMode.PAY_PER_REQUEST,
12            removal_policy=core.RemovalPolicy.DESTROY
13        )
```

以上のコードで, 最低限の設定がなされた空の DynamoDB テーブルを作成することができる. それぞれのパラメータの意味を簡単に解説しよう.

- **partition_key**: 全ての DynamoDB テーブルには Partition Key が定義されなければならない。Partition key とは、テーブル内のレコードごとに固有の ID のことである。同一の Partition key を持った要素はテーブルの中に一つしか存在することはできない。また、Partition key が定義されていない要素はテーブルの中に存在することはできない。ここでは、Partition key に `item_id` という名前をついている。
- **billing_mode**: `ddb.BillingMode.PAY_PER_REQUEST` を基本的に選択しておけばよい
- **removal_policy**: 省略

12.2.1. デプロイ

デプロイの手順は、これまでのハンズオンとほとんど共通である。ここでは、コマンドのみ列挙する (#で始まる行はコメントである)。それぞれの意味を忘れてしまった場合は、ハンズオン1, 2に戻って復習していただきたい。

```
# プロジェクトのディレクトリに移動
$ cd intro-aws/handson/04-serverless/dynamodb

# venv を作成し、依存ライブラリのインストールを行う
$ python3 -m venv .env
$ source .env/bin/activate
$ pip install -r requirements.txt

# AWS の認証情報をセットする
# 自分自身の認証情報に置き換えること！
export AWS_ACCESS_KEY_ID=XXXXXX
export AWS_SECRET_ACCESS_KEY=YYYYYY
export AWS_DEFAULT_REGION=ap-northeast-1

# デプロイを実行
$ cdk deploy
```

デプロイのコマンドが無事に実行されれば、Figure 79 のような出力が得られるはずである。ここで表示されている `SimpleDynamoDb.TableName = XXXX` の XXXX の文字列は後で使うのでメモしておこう。

```
✓ SimpleDynamoDb
Outputs:
SimpleDynamoDb.TableName = SimpleDynamoDb-SimpleTableC6BC762D-Y9KVLHG22DV9
```

Figure 79. CDKデプロイ実行後の出力

AWS コンソールにログインして、デプロイされたスタックを確認してみよう。コンソールから、DynamoDB のページに行き、左のメニューバーから "Tables" を選択する。すると、Figure 80 のような画面からテーブルの一覧が確認できる。

Name	Status	Partition key	Sort key
SimpleDynamoDb-SimpleTableC6BC762D-Y9KVLHG22DV9	Active	item_id (String)	-

Figure 80. CDKデプロイ実行後の出力

今回のアプリケーションで作成したのが `SimpleDynamoDb-YYYY` という名前のついたテーブルだ。テーブルの名前をクリックして、詳細を見てみると、Figure 81 のような画面が表示されるはずだ。"Items" のタブをクリッ

クすると、テーブルの中のレコードを確認することができる。現時点ではなにもデータを書き込んでいないので、空である。

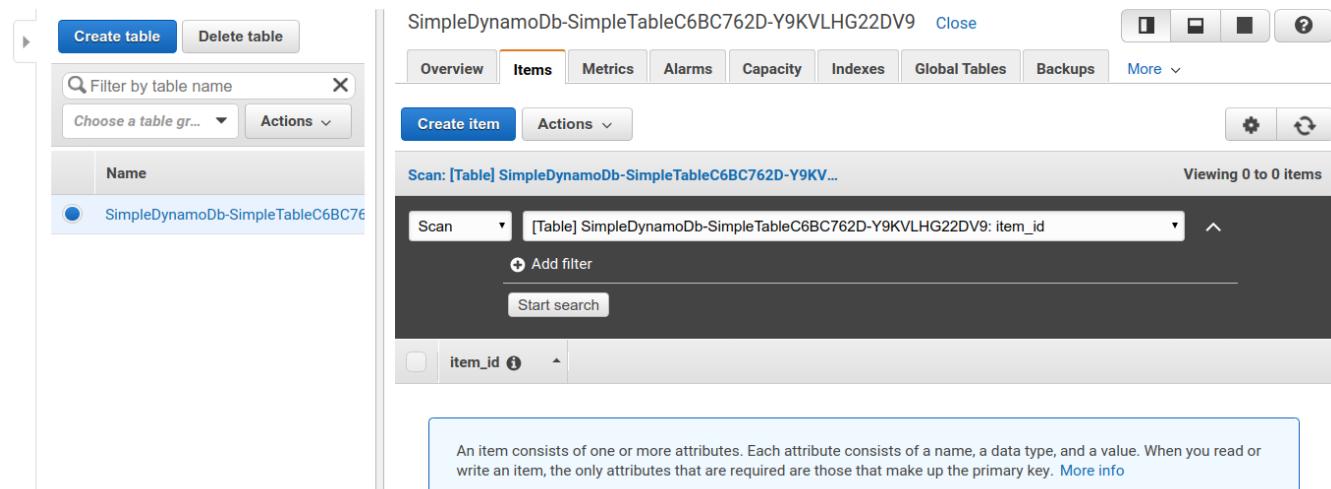


Figure 81. CDKデプロイ実行後の出力

12.2.2. データの読み書き

それでは、上で作ったテーブルを使ってデータの読み書きを実践してみよう。ここでは Python と [boto3](#) ライブラリを用いた方法を紹介する。

まず最初に、[boto3](#) ライブラリを用意する。次に、テーブルの名前から [Table](#) オブジェクトを作成する。"XXXX" の部分を自分がデプロイしたテーブルの名前 (Figure 79) に置き換えた上で、以下のコードを実行しよう。

```
1 import boto3
2 ddb = boto3.resource('dynamodb')
3
4 table = ddb.Table("XXXX")
```

新しいデータを書き込むには次のコードを実行する。

```
1 table.put_item(
2     Item={
3         'item_id': 'bec7c265-46e2-4065-91d8-80b2e8dcc9c2',
4         'first_name': 'John',
5         'last_name': 'Doe',
6         'age': 25,
7     }
8 )
```

テーブルの中のデータを、そのデータの Partition key を使って読み出すには、次のコードを実行する。

```
1 table.get_item(
2     Key={"item_id": "bec7c265-46e2-4065-91d8-80b2e8dcc9c2"}
3 ).get("Item")
```

テーブルの中にあるデータを全て読み出したければ以下のコードを実行する。

```
1 table.scan().get("Items")
```

12.2.3. 大量のデータの読み書き

DynamoDB の利点は、最初に述べた通り、負荷に応じて自在にその処理能力を拡大できる点である。

そこで、ここでは一度に大量のデータを書き込む場合をシミュレートしてみよう。[batch_rw.py](#) に、一度に大量の書き込みを実行するためのプログラムが書いてある。

次のコマンドを実行してみよう (XXXX は自分のテーブルの名前に置き換える)。

```
$ python batch_rw.py XXXX write 1000
```

このコマンドを実行することで、ランダムなデータが1000個データベースに書き込まれる。

さらに、データベースの検索をかけてみよう。今回書き込んだデータには `age` という属性に1から50のランダムな整数が割り当てられている。`age` が2以下であるような要素だけを拾ってくるには、以下のコマンドを実行すればよい。

```
$ python batch_rw.py XXXX search_under_age 2
```

Chapter 13. Hands-on #5: Bashoutter

さて、最終回となるハンズオン第五回では、これまで学んできたサーバーレスクラウドの技術を使って、簡単なウェブサービスを作ってみよう。具体的には、人々が自分の作った俳句を投稿するSNSサービス (**Bashoutter**と名付ける) を作成してみよう。これまでの講義の集大成として、コードの長さとしては最も長くなっているが、頑張ってついてきてもらいたい。最終的には、Figure 82 のような、ミニマルではあるがとても現代風なSNSサイトが完成する！

ハンズオンのソースコードはこちらのリンクに置いてある ⇒ <https://gitlab.com/tomomano/intro-aws/-/tree/master/handson/05-bashoutter>



このハンズオンは、基本的に AWS の無料枠 の範囲内で実行することができる。

The screenshot shows the "Bashoutter" application interface. At the top, there is a blue header bar with the title "Bashoutter". Below it, the API Endpoint URL is displayed as <https://ig5181x0bd.execute-api.ap-northeast-1.amazonaws.com/prod/>. The main area has a white background with a light gray grid. In the center, there is a form titled "Post your Haiku!" with three input fields: "五 クラウドで" (5-line Cloud), "七 SNSを" (7-line SNS), and "五 つくったよ" (5-line I made). To the right of these fields is a "Your name" label and a "詠み人知らず" (Unknown Poet) link. Below the form is a "POST" button. To the right of the form, there is a "REFRESH" button. The bottom half of the screen displays a list of "People's Haikus" in a grid format. Each haiku card includes the poem, timestamp, and author. The cards are as follows:

Haiku ID	Timestamp	Author
dec2bd2bfcd144208201d652c703d992	2020/07/07 01:25	正岡子規
521b6a8ed4c34e6b98f4fc59cf556496	2020/07/07 01:26	松尾芭蕉
9cc9f01987bd4575a1741f0cb24a94ab	2020/07/07 01:26	松尾芭蕉
fe181b3ac913439aa1471c6a35cc3382	2020/07/07 01:28	与謝蕪村
6cdad2516ccb4a77b297c88690d9f7c1	2020/07/07 01:29	詠み人知らず

Figure 82. ハンズオン#5で作製するSNSアプリケーション "Bashoutter"

13.1. 準備

本ハンズオンの実行には、第一回ハンズオンで説明した準備 (Section 4.1) が整っていることを前提とする。それ以外に必要な準備はない。

13.2. アプリケーションの説明

13.2.1. API

今回のアプリケーションでは、人々からの俳句の投稿を受け付けたり、投稿された俳句の一覧を取得する、といった機能を実装したい。そこで、Table 8 に示すような4つのAPIを今回実装する。

Table 8. Hands-on #5 で実装するAPI

GET /haiku	俳句の一覧を取得する
POST /haiku	新しい俳句を投稿する
PATCH /haiku/{item_id}	{item_id} で指定された俳句にお気に入り票を一つ入れる
DELETE /haiku/{item_id}	{item_id} で指定された俳句を削除する

それぞれのAPIのパラメータおよび返り値の詳細は、<https://gitlab.com/tomomano/intro-aws/-/blob/master/handson/05-bashoutter/specs/swagger.yml> に定義してある。



Open API Specification (OAS; 少し以前は Swagger Specification と呼ばれていた) は、REST API のための記述フォーマットである。OAS に従って API の仕様が記述されると、簡単にドキュメンテーションを生成したり、クライアントアプリケーションを自動生成することができる。今回用意したAPI仕様も、OAS に従って書いてあるので、非常に見やすいドキュメンテーションを瞬時に生成することができる。詳しくは [このページ](#)などを参照。

13.2.2. アプリケーションアーキテクチャ

このハンズオンで作成するアプリケーションの概要を Figure 83 に示す。

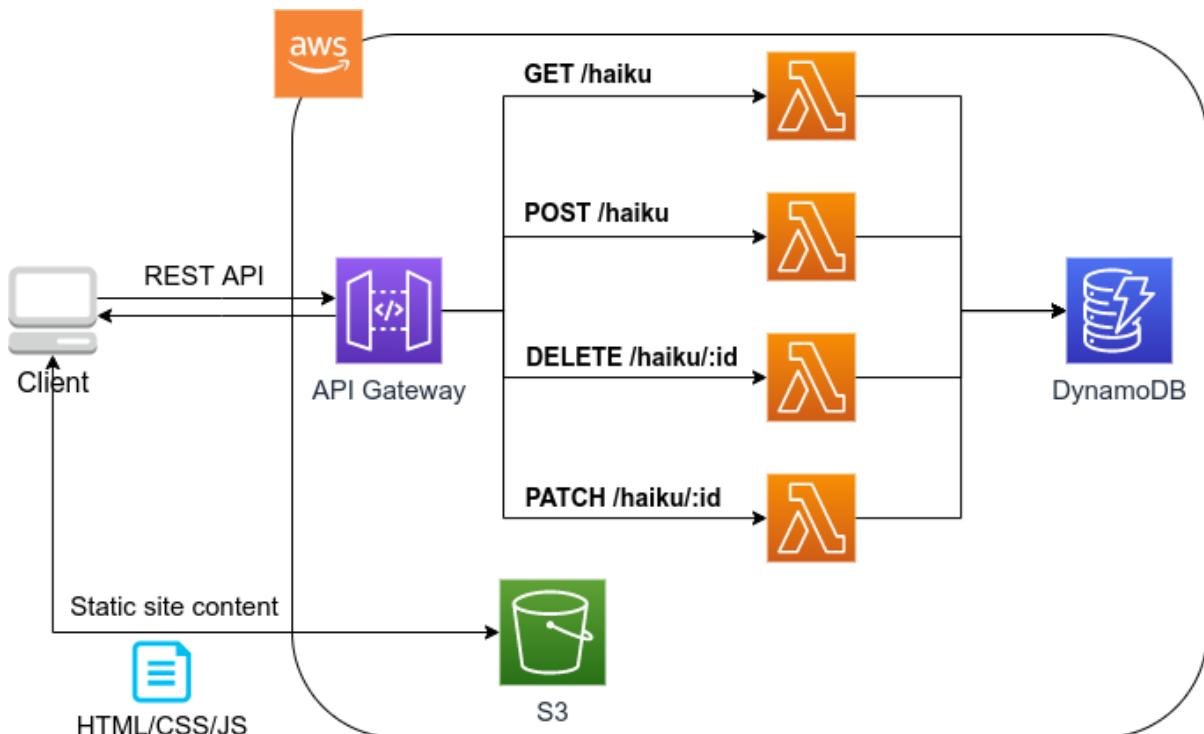


Figure 83. ハンズオン#5で作製するアプリケーションのアーキテクチャ

簡単にまとめると、以下のような設計である。

- ・ クライアントからの API リクエストは、API Gateway (後述) にまず送信され、API の URI に従って指定された Lambda 関数へ転送される。

- それぞれの API のパスごとに独立した Lambda が用意されている.
- 俳句の情報 (作者, 俳句本体, 投稿日時など) を記録するためのデータベース (DynamoDB) を用意する.
- 各 Lambda 関数には, DynamoDB へのアクセス権を付与する.
- 最後に, ウェブブラウザからコンテンツを表示できるよう, ウェブページの静的コンテンツを配信するための S3 バケットを用意する. クライアントはこの S3 バケットにアクセスすることで HTML/CSS/JS などのコンテンツを取得する.

それでは, プログラムのソースコードを見てみよう ([/handson/05-bashoutter/app.py](#)).

```

1 class Bashoutter(core.Stack):
2
3     def __init__(self, scope: core.App, name: str, **kwargs) -> None:
4         super().__init__(scope, name, **kwargs)
5
6         ①
7         # dynamoDB table to store haiku
8         table = ddb.Table(
9             self, "Bashoutter-Table",
10            partition_key=ddb.Attribute(
11                name="item_id",
12                type=ddb.AttributeType.STRING
13            ),
14            billing_mode=ddb.BillingMode.PAY_PER_REQUEST,
15            removal_policy=core.RemovalPolicy.DESTROY
16        )
17
18         ②
19         bucket = s3.Bucket(
20             self, "Bashoutter-Bucket",
21             website_index_document="index.html",
22             public_read_access=True,
23             removal_policy=core.RemovalPolicy.DESTROY
24         )
25         s3_deploy.BucketDeployment(
26             self, "BucketDeployment",
27             destination_bucket=bucket,
28             sources=[s3_deploy.Source.asset("./gui/dist")],
29             retain_on_delete=False,
30         )
31
32         common_params = {
33             "runtime": _lambda.Runtime.PYTHON_3_7,
34             "environment": {
35                 "TABLE_NAME": table.table_name
36             }
37         }
38
39         ③
40         # define Lambda functions
41         get_haiku_lambda = _lambda.Function(
42             self, "GetHaiku",
43             code=_lambda.Code.from_asset("api"),
44             handler="api.get_haiku",
45             memory_size=512,
46             **common_params,
47         )
48         post_haiku_lambda = _lambda.Function(
49             self, "PostHaiku",
50             code=_lambda.Code.from_asset("api"),
51             handler="api.post_haiku",

```

```

52         **common_params,
53     )
54     patch_haiku_lambda = _lambda.Function(
55         self, "PatchHaiku",
56         code=_lambda.Code.from_asset("api"),
57         handler="api.patch_haiku",
58         **common_params,
59     )
60     delete_haiku_lambda = _lambda.Function(
61         self, "DeleteHaiku",
62         code=_lambda.Code.from_asset("api"),
63         handler="api.delete_haiku",
64         **common_params,
65     )
66
67     ④
68     # grant permissions
69     table.grant_read_data(get_haiku_lambda)
70     table.grant_read_write_data(post_haiku_lambda)
71     table.grant_read_write_data(patch_haiku_lambda)
72     table.grant_read_write_data(delete_haiku_lambda)
73
74     ⑤
75     # define API Gateway
76     api = apigw.RestApi(
77         self, "BashoutterApi",
78         default_cors_preflight_options=apigw.CorsOptions(
79             allow_origins=apigw.Cors.ALL_ORIGINS,
80             allow_methods=apigw.Cors.ALL_METHODS,
81         )
82     )
83
84     haiku = api.root.add_resource("haiku")
85     haiku.add_method(
86         "GET",
87         apigw.LambdaIntegration(get_haiku_lambda)
88     )
89     haiku.add_method(
90         "POST",
91         apigw.LambdaIntegration(post_haiku_lambda)
92     )
93
94     haiku_item_id = haiku.add_resource("{item_id}")
95     haiku_item_id.add_method(
96         "PATCH",
97         apigw.LambdaIntegration(patch_haiku_lambda)
98     )
99     haiku_item_id.add_method(
100        "DELETE",
101        apigw.LambdaIntegration(delete_haiku_lambda)
102    )

```

- ① ここで、俳句の情報を記録しておくための DynamoDB テーブルを定義している。
- ② 続いて、静的コンテンツを配信するための S3 バケットを用意している。また、スタックのデプロイ時に、必要なファイル群を自動的にアップロードするような設定を行っている。
- ③ 続いて、それぞれの API で実行される Lambda 関数を定義している。関数は Python3.7 で書かれており、コードは [/handson/05-bashoutter/api/api.py](#) にある。
- ④ 次に、2で定義された Lambda 関数に対し、データベースへの読み書きのアクセス権限を付与している。
- ⑤ ここで、API Gateway により、各APIパスとそこで実行されるべき Lambda 関数を紐付けている。

それについて、もう少し詳しく説明しよう。

13.2.3. Public access mode の S3 バケット

S3 のバケットを作成しているコードを見てみよう。

```
1 bucket = s3.Bucket(  
2     self, "Bashoutter-Bucket",  
3     website_index_document="index.html",  
4     public_read_access=True,  
5     removal_policy=core.RemovalPolicy.DESTROY  
6 )
```

ここで注目してほしいのは `public_read_access=True` の部分だ。

前章で、S3 について説明を行った時には触れなかったが、S3 には **Public access mode** という機能がある。Public access mode をオンにしておくと、バケットの中のファイルは基本的にすべて認証無しで (i.e. インターネット上の誰でも) 閲覧できるようになる。この設定は、ウェブサイトの静的なコンテンツを置いておくのに最適であり、多くのサーバーレスによるウェブサービスでこのような設計が行われる。`public access mode` を設定しておくと、<http://XXXX.s3-website-ap-northeast-1.amazonaws.com/> のような固有の URL がバケットに対して付与される。そして、クライアントがこの URL にアクセスをすると、バケットの中にある `index.html` がクライアントに返され、ページがロードされる。(どのページがサーブされるかは、`website_index_document="index.html"` の部分で設定している。)

より本格的なウェブページを運用する際には、public access mode の S3 バケットに、[CloudFront](#) という機能を追加することが一般的である。

CloudFront はいくつかの役割を担っているのだが、最も重要な機能が **Content Delivery Network (CDN)** である。CDN とは、頻繁にアクセスされるデータをメモリーなどの高速記録媒体にキャッシュしておくことで、クライアントがより高速にデータをダウンロードすることを可能にする仕組みである。また、世界各地のデータセンターにそのようなキャッシュを配置することで、クライアントと地理的に最も近いデータセンターからデータが配信する、というような設定も可能である。



また、CloudFront を配置することで、HTTPS 通信を設定することができる。(逆に言うと、S3 単体では HTTP 通信しか行うことができない。) 現代的なウェブサービスでは、秘匿情報を扱う扱わないに関わらず、HTTPS を用いることが標準となっている。

今回のハンズオンでは説明の簡略化のため CloudFront の設定を行わなかったが、興味のある読者は以下のリンクのプログラムが参考になるだろう。

- <https://github.com/aws-samples/aws-cdk-examples/tree/master/typescript/static-site>



今回の S3 バケットには、AWS によって付与されたランダムな URL がついている。これを `example.com` のような自分のドメインでホストしたければ、AWS によって付与された URL を自分のドメインの DNS レコードに追加すればよい。

Public access mode の S3 バケットを作成した後、バケットの中に配置するウェブサイトコンテンツを、以下のコードによりアップロードしている。

```

1 s3_deploy.BucketDeployment(
2     self, "BucketDeployment",
3     destination_bucket=bucket,
4     sources=[s3_deploy.Source.asset("./gui/dist")],
5     retain_on_delete=False,
6 )

```

ウェブサイトのコンテンツは [/handson/05-bashoutter/gui/](#) にある(特に,ビルド済みのものが `/dist/` 以下にある). 興味のある読者は中身を確認してみるとよい.



今回のウェブサイトは `Vue.js` と `Vuetify` という UI フレームワークを使って作成した. ソースコードは [/handson/05-bashoutter/gui/src/](#) にあるので,見てみるとよい.

13.2.4. API のハンドラ関数

API リクエストが来たときに,リクエストされた処理を行う関数のことを特にハンドラ (handler) 関数と呼ぶ. Lambda を使って `GET /haiku` の API に対してのハンドラ関数を定義している部分を見てみよう.

```

1 get_haiku_lambda = _lambda.Function(
2     self, "GetHaiku",
3     code=_lambda.Code.from_asset("api"),
4     handler="api.get_haiku",
5     ...
6 )

```

`code=_lambda.Code.from_asset("api"), handler="api.get_haiku"` のところで,外部のディレクトリ(`api/`)にある `api.py` というファイルの, `get_haiku()` という関数をハンドラ関数として実行せよ,と指定している. この `get_haiku()` のコードを見てみよう ([/handson/05-bashoutter/api/api.py](#)).

```

1 ddb = boto3.resource("dynamodb")
2 table = ddb.Table(os.environ["TABLE_NAME"])
3
4 def get_haiku(event, context):
5     """
6     handler for GET /haiku
7     """
8     try:
9         response = table.scan()
10
11         status_code = 200
12         resp = response.get("Items")
13     except Exception as e:
14         status_code = 500
15         resp = {"description": f"Internal server error. {str(e)}"}
16
17     return {
18         "statusCode": status_code,
19         "headers": HEADERS,
20         "body": json.dumps(resp, cls=DecimalEncoder)
21     }

```

`response = table.scan()` で,俳句の格納された DynamoDB テーブルから,全ての要素を取り出している. もしにもエラーが起きなければステータスコード200が返され,もしなにかエラーが起こればステータスコード500が返されるようになっている.

上記のような操作を,他の API についても繰り返すことで,すべての API のハンドラ関数が定義されている.



GET /haiku のハンドラ関数で, `response = table.scan()` という部分があるが, 実はこれは最善の書き方ではない. DynamoDB の `scan()` メソッドは, 最大で 1MB までのデータしか返さない. データベースのサイズが大きく, 1MB 以上のデータがある場合には, 再帰的に `scan()` メソッドを呼ぶ必要がある. 詳しくは [boto3ドキュメンテーション](#) を参照.

13.2.5. AWS における権限の管理 (IAM)

以下の部分のコードに注目してほしい.

```
1 table.grant_read_data(get_haiku_lambda)
2 table.grant_read_write_data(post_haiku_lambda)
3 table.grant_read_write_data(patch_haiku_lambda)
4 table.grant_read_write_data(delete_haiku_lambda)
```

これまで説明の簡略化のため敢えて触れてこなかったが, AWS には [IAM \(Identity and Access Management\)](#) という重要な概念がある. IAM は基本的に, あるリソースが他のリソースに対してどのような権限を持っているか, を規定するものである. Lambda は, デフォルトの状態では他のリソースにアクセスする権限をなにも有していない. したがって, Lambda 関数が DynamoDB のデータを読み書きするためには, それを許可するような IAM が Lambda 関数に付与されなければならない.

CDK による `dynamodb.Table` オブジェクトには `grant_read_write_data()` という便利なメソッドが備わっており, アクセスを許可したい Lambda 関数を引数としてこのメソッドを呼ぶことで, データベースへの読み書きを許可する IAM を付与することができる.



各リソースに付与する IAM は, 必要最低限の権限を与えるにとどめるというのが基本方針である. これにより, セキュリティを向上させるだけでなく, 意図していないプログラムからのデータベースへの読み書きを防止するという点で, バグを未然に防ぐことができる.

そのような理由により, 上のコードでは GET のハンドラー関数に対しては `grant_read_data()` によって, read 権限のみを付与している.

13.2.6. API Gateway

[API Gateway](#) とは, API の"入り口"として, API のリクエストパスに従って Lambda 関数などに接続を行うという機能を担う. このような API のリソースパスに応じて接続先を振り分けるようなサーバーをルーターと呼んだりする. 従来的には, ルーターにはそれ専用の仮想サーバーが置かれることが一般的であったが, API Gateway はその機能をサーバーレスで担ってくれる. すなわち, API のリクエストが来たときのみ起動し, API が来ていない間は完全にシャットダウンしている. 一方で, アクセスが大量に来た場合はそれに比例してルーティングの処理能力を増大してくれる.

API Gateway を配置することで, 大量 (1秒間に数千から数万件) の API リクエストに対応することのできるシステムを容易に構築することができる. API Gateway の料金は [Table 9](#) のように設定されている. また, 無料利用枠により, 月ごとに100万件までのリクエストは0円で使用できる.

Table 9. API Gateway の利用料金設定 ([参照](#))

Number of Requests (per month)	Price (per million)
First 333 million	\$4.25
Next 667 million	\$3.53
Next 19 billion	\$3.00
Over 20 billion	\$1.91

ソースコードの該当箇所を見てみよう.

```

1 api = apigw.RestApi(
2     self, "BashoutterApi",
3     default_cors_preflight_options=apigw.CorsOptions(
4         allow_origins=apigw.Cors.ALL_ORIGINS,
5         allow_methods=apigw.Cors.ALL_METHODS,
6     )
7 )
8
9 haiku = api.root.add_resource("haiku")
10 haiku.add_method(
11     "GET",
12     apigw.LambdaIntegration(get_haiku_lambda)
13 )
14 haiku.add_method(
15     "POST",
16     apigw.LambdaIntegration(post_haiku_lambda)
17 )
18
19 haiku_item_id = haiku.add_resource("{item_id}")
20 haiku_item_id.add_method(
21     "PATCH",
22     apigw.LambdaIntegration(patch_haiku_lambda)
23 )
24 haiku_item_id.add_method(
25     "DELETE",
26     apigw.LambdaIntegration(delete_haiku_lambda)
27 )

```

- `api = apigw.RestApi()` により、空の API Gateway を作成している。
- 次に、`api.root.add_resource()` のメソッドを呼ぶことで、`/haiku` という API パスを追加している。
- 続いて、`add_method()` を呼ぶことで、`GET, POST` のメソッドを `/haiku` のパスに定義している。
- さらに、`haiku.add_resource("{item_id}")` により、`/haiku/{item_id}` という API パスを追加している。
- 最後に、`add_method()` を呼ぶことにより、`PATCH, DELETE` のメソッドを `/haiku/{item_id}` のパスに定義している。

このように、逐次的に API パスとそこで実行されるメソッド・Lambda を記述していくだけよい。



上記のプログラムで新規 API を作成すると、AWS からランダムな URL がその API のエンドポイントとして割り当てられる。これを `.api.example.com` のような自分のドメインでホストしたければ、AWS によって付与された URL を自分のドメインの DNS レコードに追加すればよい。



API Gateway で新規 API を作成したとき、`default_cors_preflight_options`= というパラメータで [Cross Origin Resource Sharing \(CORS\)](#) の設定を行っている。これは、ブラウザで走る Web アプリケーションと API を接続する際に必要な設定である。興味のある読者は各自 CORS について調べてもらいたい。

13.3. アプリケーションのデプロイ

アプリケーションの中身が理解できたところで、早速デプロイを行ってみよう。

デプロイの手順は、これまでのハンズオンとほとんど共通である。ここでは、コマンドのみ列挙する (# で始まる行はコメントである)。それぞれの意味を忘れてしまった場合は、ハンズオン1, 2に戻って復習していただきたい。

```

# プロジェクトのディレクトリに移動
$ cd intro-aws/handson/05-bashoutter

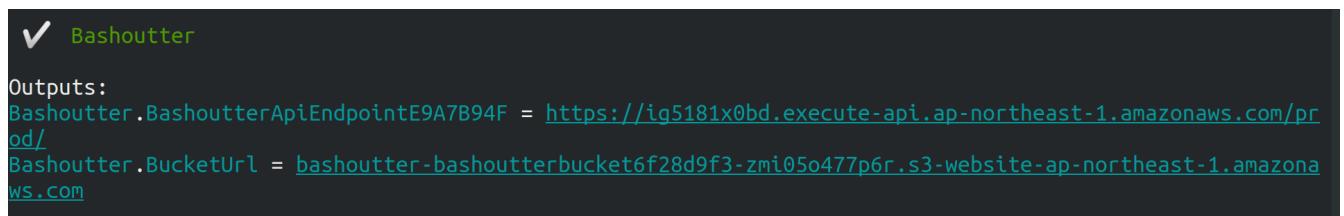
# venv を作成し、依存ライブラリのインストールを行う
$ python3 -m venv .env
$ source .env/bin/activate
$ pip install -r requirements.txt

# AWS の認証情報をセットする
# 自分自身の認証情報に置き換えること！
export AWS_ACCESS_KEY_ID=XXXXXX
export AWS_SECRET_ACCESS_KEY=YYYYYY
export AWS_DEFAULT_REGION=ap-northeast-1

# デプロイを実行
$ cdk deploy

```

デプロイのコマンドが無事に実行されれば、Figure 84 のような出力が得られるはずである。ここで表示されている `Bashoutter.BashoutterApiEndpoint = XXXX, Bashoutter.BucketUrl = YYYY` の二つ文字列は次に使うのでメモしておこう。



```

✓ Bashoutter

Outputs:
Bashoutter.BashoutterApiEndpoint = https://ig5181x0bd.execute-api.ap-northeast-1.amazonaws.com/prod/
Bashoutter.BucketUrl = bashoutter-bashoutterbucket6f28d9f3-zmi05o477p6r.s3-website-ap-northeast-1.amazonaws.com

```

Figure 84. CDKデプロイ実行後の出力

\$ cdk bootstrap のコマンドを実行していないと、上記のデプロイはエラーを出力する。

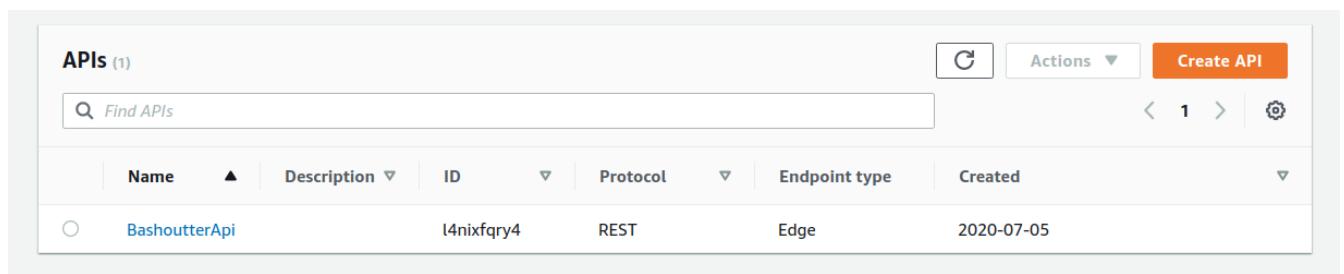


`bootstrap` のコマンドは1アカウントにつき1度実行されていればよい。Section 15.3 も参照のこと。



上記のデプロイで得られた API のエンドポイントは API Gateway によりランダムに作成されたアドレスである。このアドレスを DNS に登録することで、自分の好きなドメイン名（例：api.example.com）と結びつけることが可能である。

AWS コンソールにログインして、デプロイされたスタックを確認してみよう。コンソールから、API Gateway のページに行くと、Figure 85 のような画面が表示され、デプロイ済みの API エンドポイントの一覧が確認できる。



APIs (1)						
	Name	Description	ID	Protocol	Endpoint type	Created
○	BashoutterApi		l4nixfqry4	REST	Edge	2020-07-05

Figure 85. API Gateway コンソール画面 (1)

今回デプロイした "BashoutterApi" という名前の API をクリックすることで Figure 86 のような画面に遷移し、詳細情報を閲覧できる。GET /haiku, POST /haiku などが定義されていることが確認できる。

それぞれのメソッドをクリックすると、そのメソッドの詳細情報を確認できる。API Gateway は、上で説明したルーティングの機能だけでなく、認証機能などを追加することも可能であり、そのような理由で Figure 86 で画面右端

赤色で囲った部分に,この API で呼ばれる Lambda 関数が指定されている. 関数名をクリックすることで,関数の中身を閲覧することが可能である.

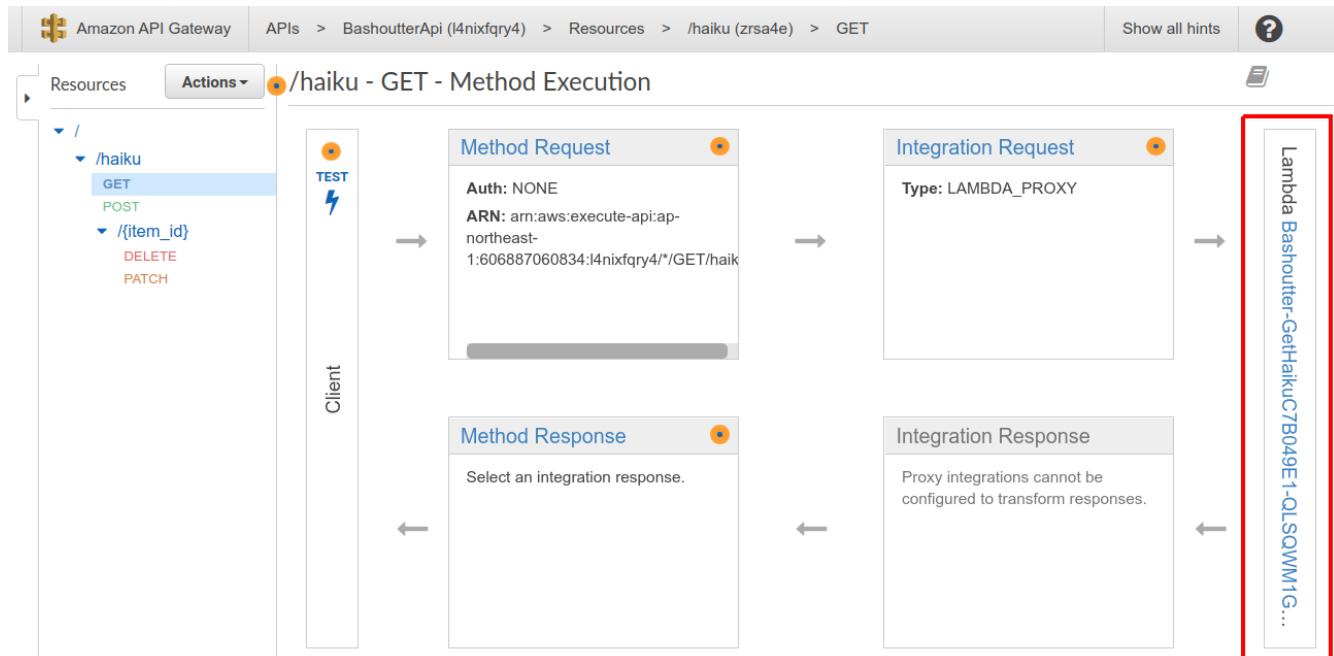


Figure 86. API Gateway コンソール画面 (2)

次に, S3 のコンソール画面に移ってみよう. "bashouter-XXXXXX" という名前のバケットが見つかるはずである (Figure 87).

The screenshot shows the AWS S3 console. The sidebar has "Buckets", "Batch Operations", and "Access analyzer for S3". Under "Feature spotlight", it says "While we continue to improve the new version of the S3 console, you can temporarily switch to the previous console experience for buckets. To help us improve the experience, give feedback." The main area shows "Buckets (4)" with a table. The table has columns: Name, Region, Access, and Creation date. One row is selected, showing the bucket name "bashouter-bashouterbucket6f28d9f3-zmli05o477p6r", Region "Asia Pacific (Tokyo) ap-northeast-1", Access "Public", and Creation date "July 7, 2020, 13:15 (UTC+09:00)".

Figure 87. S3 コンソール画面

バケットの名前をクリックすることで,バケットの中身を確認してみよう. `index.html` のほか, `css/`, `js/` などのディレクトリがあるのが確認できるだろう (Figure 88). これらが,ウェブページの"枠"を定義している静的コンテンツである.

Amazon S3 > bashoutter-bashoutterbucket6f28d9f3-zmi05o477p6r

bashoutter-bashoutterbucket6f28d9f3-zmi05o477p6r

Overview Properties Permissions Management Access points

Public

Type a prefix and press Enter to search. Press ESC to clear.

Upload Create folder Download Actions

Asia Pacific (Tokyo)

<input type="checkbox"/> Name	Last modified	Size	Storage class
<input type="checkbox"/> css	--	--	--
<input type="checkbox"/> fonts	--	--	--
<input type="checkbox"/> js	--	--	--
<input type="checkbox"/> index.html	Jul 7, 2020 1:16:56 PM GMT+0900	723.0 B	Standard

Viewing 1 to 4

Figure 88. S3 バケットの中身

13.4. API を送信する

それでは、デプロイしたアプリケーションに対し、実際に API を送信してみよう（ひとまずは、S3 にある GUI の方はおいておく。今回のアプリケーションでより本質的なのは API の方だからである）。

ここではコマンドラインから API を送信するためのシンプルな HTTP クライアントである [HTTPie](#) を使ってみよう。HTTPie は、スタックをデプロイするときに Python 仮想環境を作成した際、一緒にインストールした。コマンドラインに `http` と打ってみて、コマンドの使い方が出力されることを確認しよう。

まず最初に、先ほどデプロイを実行した際に得られた API のエンドポイントの URL (`Bashoutter.BashoutterApiEndpoint = XXXX` で得られた `XXXX` の文字列) をコマンドラインの変数に設定しておく。

```
$ export ENDPOINT_URL="https://0000.execute-api.ap-northeast-1.amazonaws.com/prod/"
```



上のコマンドで、URLは自分のデプロイしたスタックのURLに置き換える。

次に、俳句の一覧を取得するため、`GET /haiku` の API を送信してみよう。

```
$ http GET "${ENDPOINT_URL}/haiku"
```

現時点では、まだだれも俳句を投稿していないので、空の配列 (`[]`) が返ってくる。

それでは次に、俳句を投稿してみよう。

```
$ http POST "${ENDPOINT_URL}/haiku" \
username="松尾芭蕉" \
first="閑さや" \
second="岩にしみ入る" \
third="蟬の声"
```

以下のような出力が得られるだろう。

```
HTTP/1.1 201 Created
Connection: keep-alive
Content-Length: 49
Content-Type: application/json
...
{
  "description": "Successfully added a new haiku"
}
```

新しい俳句を投稿することに成功したようである。本当に俳句が追加されたか、再び GET リクエストを呼ぶことで確認してみよう。

```
$ http GET "${ENDPOINT_URL}/haiku"

HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 258
Content-Type: application/json
...
[
  {
    "created_at": "2020-07-06T02:46:04+00:00",
    "first": "閑さや",
    "item_id": "7e91c5e4d7ad47909e0ac14c8bbab05b",
    "likes": 0.0,
    "second": "岩にしみ入る",
    "third": "蟬の声",
    "username": "松尾芭蕉"
  }
]
```

素晴らしい！

次に、PATCH /haiku/{item_id} を呼ぶことでこの俳句にいいねを追加してみよう。上のコマンドで取得した俳句の item_id を、下のコマンドの XXXX

```
$ http PATCH "${ENDPOINT_URL}/haiku/XXXX"
```

再び GET リクエストを送ることで、いいね (likes) が 1 増えたことを確認しよう。

```
$ http GET "${ENDPOINT_URL}/haiku"
...
[{
  ...
  "likes": 1.0,
  ...
}]

```

最後に、DELETEリクエストを送ることで俳句をデータベースから削除しよう。XXXXはitem_idの値で置き換えた上で以下のコマンドを実行する。

```
$ http DELETE "${ENDPOINT_URL}/haiku/XXXX"
```

再びGETリクエストを送ることで、返り値が空([])になっていることを確認しよう。

以上のように、SNSに必要な基本的なAPIがきちんと動作していることが確認できた。

13.5. 大量のAPIリクエストをシミュレートする

さて、前節ではマニュアルでひとつづづ俳句を投稿した。多数のユーザーがいるようなSNSでは、一秒間に数千件以上の投稿がされている。今回はサーバーレスアーキテクチャを採用したこと、そのような瞬間的な大量アクセスにも容易に対応できるようなシステムが構築できている！

その点をデモンストレートするため、ここでは大量のAPIが送信された状況をシミュレートしてみよう。

[/handson/05-bashoutter/client.py](#)に、大量のAPIリクエストをシミュレートするためのプログラムが書かれている。このプログラムは基本的にPOST /haikuのAPIリクエストを指定された回数だけ実行する。

テストとして、APIを300回送ってみよう。以下のコマンドを実行する。

```
$ python client.py $ENDPOINT_URL post_many 300
```

数秒のうちに実行が完了するだろう。これがもし、単一のサーバーからなるAPIだったとしたら、このような大量のリクエストの処理にはもっと時間がかかるだろう。最悪の場合には、サーバーダウンにもつながっていたかもしれない。従って、今回作成したサーバーレスアプリケーションは、とてもシンプルながらも一秒間に数百件の処理を行えるような、スケーラブルなクラウドシステムであることがわかる。サーバーレスでクラウドを設計することの利点を垣間見ることができただろうか？



上記のコマンドにより、大量の俳句を投稿するとデータベースに無駄なデータがどんどん溜まってしまう。データベースを完全に空にするには、以下のコマンドを使用する。

```
$ python client.py $ENDPOINT_URL clear_database
```

13.6. Bashoutter GUIを動かしてみる

前節ではコマンドラインからAPIを送信する演習を行った。ウェブアプリケーションでは、これらのAPIはウェブブラウザ上のウェブページから送信され、コンテンツが表示されている([Figure 69](#)参照)。最後に、APIがGUIと統合されるとどうなるのか、見てみよう。

デプロイを実行したときにコマンドラインで出力された、ashoutter.BucketUrl=に続くURLを確認しよう([Figure](#)

84). これは,先述したとおり, Public access mode の S3 バケットの URL である.

ウェブブラウザを開き,そのURLへアクセスをしてみよう. すると, Figure 89 のようなページが表示されるはずである.

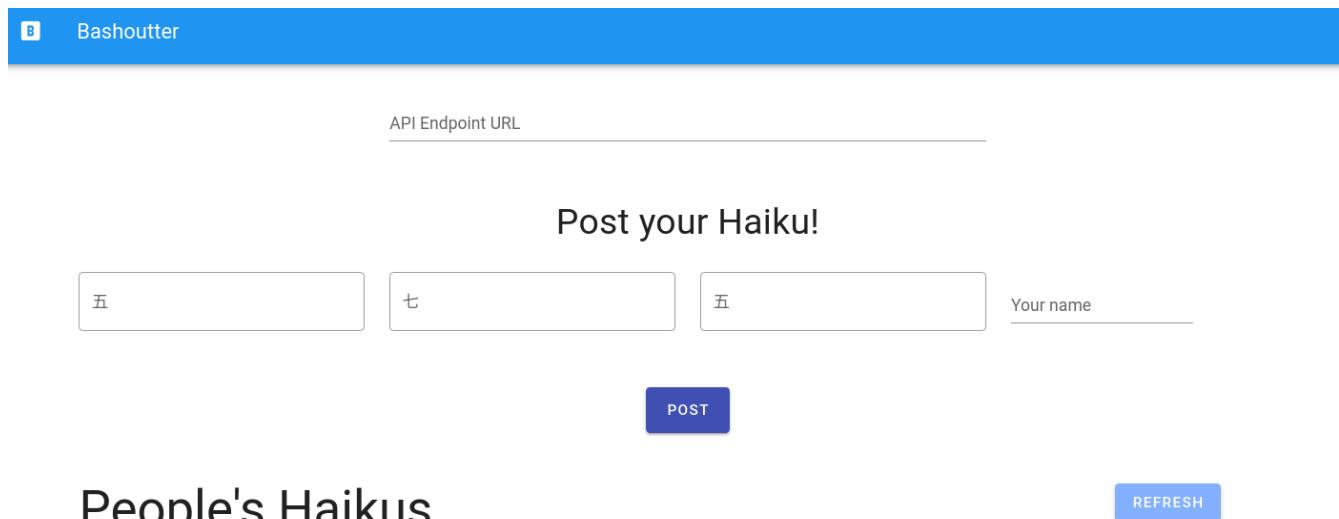


Figure 89. "Bashoutter" の GUI 画面

ページが表示されたら,一番上の "API Endpoint URL" と書いてあるテキストボックスに,今回デプロイした API Gateway の URL を入力する(今回のアプリケーションでは,API Gateway の URL はランダムに割り当てられるのでこのような仕様になっている). そうしたら,画面の "REFRESH" と書いてあるボタンを押してみよう. データベースに俳句が登録済みであれば,俳句の一覧が表示されるはずである. 各俳句の左下にあるハートのアイコンをクリックすることで, "like" の票を入れることができる.

新しい俳句を投稿するには,五七五と投稿者の名前を入力して, "POST" を押す. "POST" を押した後は,再び "REFRESH" ボタンを押すことで最新の俳句のリストをデータベースから取得する.

今回は,どうやって GUI を作成したかは触れないが,基本的にページの背後では `GET /haiku`, `POST /haiku` などの API がクラウドに送信されることで,コンテンツが表示されている. 興味のある読者は GUI のソースコードも読んでみるとよい ([/handson/05-bashoutter/gui/](#)).

13.7. アプリケーションの削除

これにて,第五回ハンズオンは終了である.最後にスタックを削除しよう.

スタックを削除するには,まず最初に S3 バケットの中身をすべて削除しなければならない. コンソールから実行するには, S3 コンソールに行き,バケットの中身を開いた上で,全てのファイルを選択し, "Actions" → "Delete" を実行すれば良い.

コマンドラインから実行するには,次のコマンドを使う. <BUCKET NAME> のところは,自分のバケットの名前 ("BashoutterBucketXXXX" というパターンの名前がついているはずである) に置き換えることを忘れずに.

```
$ aws s3 rm <BUCKET NAME> --recursive
```

S3 バケットを空にしたら,次のコマンドを実行してスタックを削除する.

```
$ cdk destroy
```

13.8. 講義第三回目のまとめ

ここまでが,講義第三回目の内容である.

今回は,クラウドの応用として,一般の人に使ってもらうようなウェブアプリケーション・データベースをどのようにして作るのか,という点に焦点を当てて,講義を行った. その中で,従来的なクラウドシステムの設計と,ここ数年の最新の設計方法であるサーバーレスアーキテクチャについて解説した. 特に, AWS でのサーバーレスの実践として, Lambda, S3, DynamoDB のハンズオンを行った. 最後に,ハンズオン第五回目では,これらの技術を統合することで,完全サーバーレスなウェブアプリケーション "Bashoutter" を作成した.

今回の講義を通じて,世の中のウェブサービスがどのようにして出来上がっているのか,少し理解が深まっただろうか? また,そのようなウェブアプリケーションを自分が作りたいと思ったときの,出発点となることができたならば幸いである.

Chapter 14. まとめ

Chapter 15. Appendix

15.1. AWS のシークレットキーの作成

AWS シークレットキーとは、AWS CLI や AWS CDK から AWS の API を操作する際に、ユーザー認証を行うための鍵のことである。AWS CLI/CDK を使うには、最初にシークレットキーを発行する必要がある。AWS シークレットキーの詳細は [公式ドキュメンテーション](#) を参照。

1. AWS コンソールにログインする。
2. 画面右上のアカウント名をクリックし、表示されるプルダウンメニューから "My Security Credentials" を選択 (Figure 90)
3. "Access keys for CLI, SDK, & API access" の下にある "Create accesss key" のボタンをクリックする (Figure 91)
4. 表示された Access key ID, Secret access key を記録しておく (画面を閉じると以降は表示されない)。
5. 鍵を忘れてしまった場合は、同じ手順で再発行が可能である。
6. 発行したシークレットキーは、`~/.aws/credentials` のファイルに書き込むか、環境変数に設定するなどして使う (詳しくは [Section 15.2](#))。

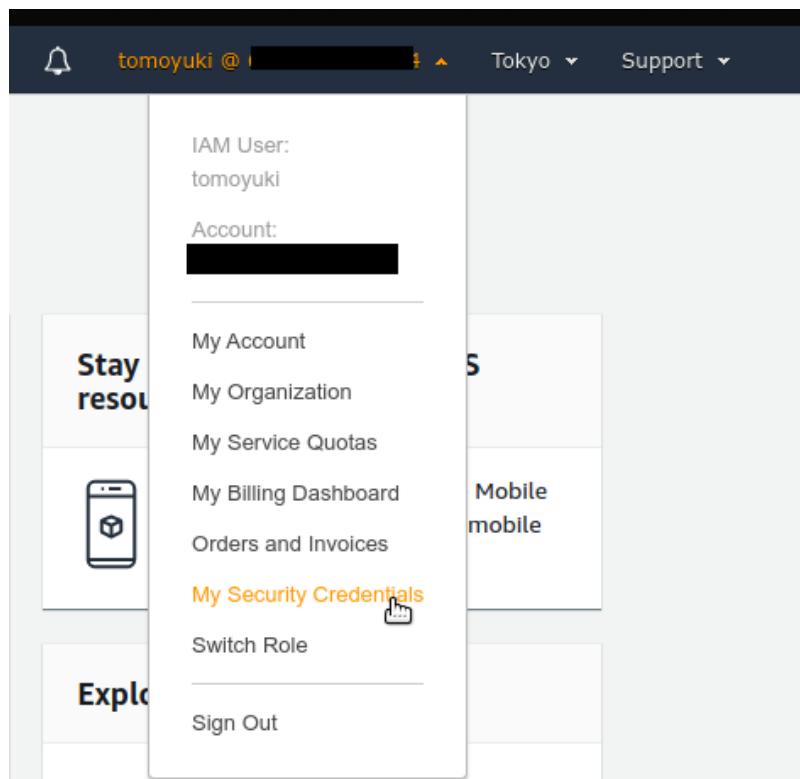


Figure 90. AWS シークレットキーの発行1

Access keys for CLI, SDK, & API access

Use access keys to make programmatic calls to AWS from the AWS Command Line Interface (AWS CLI), Tools for Windows PowerShell, the AWS SDKs, or direct AWS API calls. **If you lose or forget your secret key, you cannot retrieve it. Instead, create a new access key and make the old key inactive.** [Learn more](#)

[Create access key](#)

Figure 91. AWS シークレットキーの発行2

AWS Educate Starter Account を用いている場合は、以下の手順でシークレットキーを確認する。

- AWS Educate のコンソール画面から、[vocareum](#) のコンソールに移動する (Figure 92).
- [Account Details](#) をクリックし、続いて [AWS CLI: Show](#) をクリックする。
- `aws_access_key_id`, `aws_secret_access_key`, `aws_session_token` が表示される (Figure 93). ここで表示された内容を `~/.aws/credentials` にコピーする (Section 15.2 参照). `aws_session_token` の箇所も漏らさずコピーすること。
- 続いて、`~/.aws/config` というファイルを用意し、次の内容を書き込む。現時点では AWS Starter Account は `us-east-1` リージョンでしか利用できないためである。

```
[profile default]
region = us-east-1
output = json
```

- 上記の説明ではプロファイル名が `default` となっていたが、これは自分の好きな名前に変更しても良い。`default` 以外の名前を使用する場合は、コマンドを実行する際にプロファイル名を指定する必要がある (Section 15.2).



The screenshot shows the 'Welcome to your AWS Educate Account' page. It includes a warning message about session access and a link to the FAQ. On the right, the 'Your AWS Account Status' section displays account status (Active), remaining credits (\$100), and session time (1:47). Below these are 'Account Details' and 'AWS Console' buttons.

Figure 92. vocareum コンソール

The screenshot shows the AWS CLI command output. It includes session details (started at 2021-06-20T18:29:05-0700, term ends at 2021-06-20T21:29:05-0700, session time 2h18m12s), AWS Starter account information (Term: 364 days 23:13:23), and the AWS CLI command to copy credentials to `~/.aws/credentials`. The copied text is:
[default]
aws_access_key_id=
aws_secret_access_key=
aws_session_token=

Figure 93. vocareum から AWS シークレットキーの発行

15.2. AWS CLI のインストール

読者のために、執筆時点におけるインストールの手順 (Linux 向け) を簡単に記述する。将来のバージョンでは変更される可能性があるので、常に [公式のドキュメンテーション](#) で最新の情報をチェックすることを忘れずに。

```
$ curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"  
$ unzip awscliv2.zip  
$ sudo ./aws/install
```

インストールできたか確認するため、以下のコマンドを打ってバージョン情報が出力されることを確認する。

```
$ aws --version
```

インストールができたら、以下のコマンドにより初期設定を行う（[参照](#)）。

```
$ aws configure
```

コマンドを実行すると、AWS Access Key ID, AWS Secret Access Key を入力するよう指示される。シークレットキーの発行については [Section 15.1](#) を参照。コマンドは加えて、Default region name を訊いてくる。ここには自分の好きな地域（例えば ap-northeast-1=東京リージョン）を指定すればよい。最後の Default output format は JSON としておくとよい。

上記のコマンドを完了すると、`~/.aws/credentials` と `~/.aws/config` という名前のファイルが生成されているはずである。念の為、中身をしてみるとよい。

```
$ cat ~/.aws/credentials  
[default]  
aws_access_key_id = XXXXXXXXXXXXXXXXXXXX  
aws_secret_access_key = YYYYYYYYYYYYYYYYYYYYY  
  
$ cat ~/.aws/config  
[profile default]  
region = ap-northeast-1  
output = json
```

上記のような出力が得られるはずである。

`~/.aws/credentials` には認証鍵の情報が、`~/.aws/config` には AWS CLI の設定が記録されている。

デフォルトでは、`[default]` という名前でプロファイルが保存される。いくつかのプロファイルを使い分けたければ、`default` の例に従って、例えば `[myprofile]` などという名前でプロファイルを追加すればよい。

AWS CLI でコマンドを打つときに、プロファイルを使い分けるには、

```
$ aws s3 ls --profile myprofile
```

のように、`--profile` というオプションをつけてコマンドを実行する。

いちいち `--profile` オプションをつけるのが面倒だと感じる場合は、`AWS_PROFILE` という環境変数を設定するといい。

```
$ export AWS_PROFILE=myprofile
```

あるいは、認証情報などを環境変数に設定するテクニックもある。

```
export AWS_ACCESS_KEY_ID=XXXXXX  
export AWS_SECRET_ACCESS_KEY=YYYYYY  
export AWS_DEFAULT_REGION=ap-northeast-1
```

上の環境変数は、[~/.aws/credentials](#) よりも高い優先度を持つので、環境変数が設定されていればそちらの情報が使用される（参照）。



AWS Educate Starter Account は [us-east-1](#) のリージョンのみ利用可能である（執筆時点での情報）。よって、AWS Educate Starter Account を使用している場合は、default region を [us-east-1](#) に設定する必要がある。

15.3. AWS CDK のインストール

読者のために、執筆時点におけるインストールの手順（Linux 向け）を簡単に記述する。将来のバージョンでは変更される可能性があるので、常に [公式のドキュメンテーション](#) で最新の情報をチェックすることを忘れずに。

Node.js がインストールされていれば、基本的に以下のコマンドを実行すれば良い。

```
$ sudo npm install -g aws-cdk
```

本書のハンズオンは AWS CDK version 1.100.0 で開発した。CDK は開発途上のライブラリなので、将来的に API が変更される可能性がある。API の変更によりエラーが生じた場合は、version 1.100.0 を使用することを推奨する。



```
$ npm install -g aws-cdk@1.100
```

インストールできたか確認するため、以下のコマンドを打って正しくバージョンが表示されることを確認する。

```
$ cdk --version
```

インストールができたら、以下のコマンドにより AWS 側の初期設定を行う。これは一度実行すれば OK。

```
$ cdk bootstrap
```



[cdk bootstrap](#) を実行するときは、AWS の認証情報とリージョンが正しく設定されていることを確認する。デフォルトでは [~/.aws/config](#) にあるデフォルトのプロファイルが使用される。デフォルト以外のプロファイルを用いるときは [Section 15.2](#) で紹介したテクニックを使って切り替える。



AWS CDK の認証情報の設定は AWS CLI と基本的に同じである。詳しくは [Section 15.2](#) を参照。

15.4. Python `venv` クイックガイド

他人からもらったプログラムで、numpy や scipy のバージョンが違う！などの理由で、プログラムが動かない、という経験をしたことがある人は多いのではないだろうか。もし、自分の計算機の中に一つしか Python 環境がないとすると、プロジェクトを切り替えるごとに正しいバージョンをインストールし直さなければならず、これは大変な手間である。

コードのシェアをよりスムーズにするためには、ライブラリのバージョンはプロジェクトごとに管理されるべきである。それを可能にするのが Python 仮想環境と呼ばれるツールであり、[venv](#), [pyenv](#), [conda](#) などがよく使われる。

そのなかでも、[venv](#) は Python に標準搭載されているので、とても便利である。[pyenv](#) や [conda](#) は、別途インストールの必要があるが、それぞれの長所もある。

[venv](#) を使って仮想環境を作成するには、

```
$ python -m venv .env
```

と実行する。これにより [.env/](#) というディレクトリが作られ、このディレクトリに依存するライブラリが保存されることになる。

この新たな仮想環境を起動するには

```
$ source .env/bin/activate
```

と実行する。

シェルのプロンプトに [\(.env\)](#) という文字が追加されていることを確認しよう (Figure 94)。これが、"いまあなたは [venv](#) の中にいますよ" というしるしになる。



Figure 94. [venv](#) を起動したときのプロンプト

仮想環境を起動すると、それ以降実行する [pip](#) コマンドは、[.env/](#) 以下にインストールされる。このようにして、プロジェクトごとに使うライブラリのバージョンを切り分けることができる。

Python では [requirements.txt](#) というファイルに依存ライブラリを記述するのが一般的な慣例である。他人からもらったプログラムに、[requirements.txt](#) が定義されていれば、

```
$ pip install -r requirements.txt
```

と実行することで、必要なライブラリをインストールし、瞬時に Python 環境を再現することができる。



[venv](#) による仮想環境を保存するディレクトリの名前は任意に選べることができるが、[.env](#) という名前を用いるのが一般的である。

15.5. ハンズオン実行用の Docker image の使い方

ハンズオンを実行するために必要な、Node.js, Python, AWS CDK などがインストールされた Docker image を用意した。これを使用することで、自分のローカルマシンに諸々をインストールする必要なく、すぐにハンズオンのコードが実行できる。



ハンズオンのいくつかのコマンドは Docker の外 = ローカルマシンのリアル環境で実行されなければならない。それらについてはハンズオンの該当箇所に注意書きとして記してある。

Docker Image は [Docker Hub](#) においてある。Docker Image のビルドファイルは [こちら](#) にある。

次のコマンドで container を起動する。

```
$ docker run -it tomomano/labc:latest
```

初回にコマンドを実行したときのみ, image が Docker Hub からダウンロード (pull) される. 二回目以降はローカルにダウンロードされた image が使用される.

container が起動すると, 次のようなインタラクティブシェルが表示されるはずである (起動時に `-it` のオプションをつけたのがポイントである).

```
root@aws-handson:~$
```

この状態で `ls` コマンドを打つと, `handson/` というディレクトリがあるはずである. ここに `cd` する.

```
$ cd handson
```

すると, 各ハンズオンごとのディレクトリが見つかるはずである.

あとは, ハンズオンごとにディレクトリを移動し, ハンズオンごとの `virtualenv` を作成し, スタックのデプロイを行えばよい ([Section 4.4](#) など参照). ハンズオンごとに使用する依存ライブラリが異なるので, それぞれのハンズオンごとに `virtualenv` を作成するという設計になっている.

AWS の認証情報を設定することも忘れずに. [Section 15.2](#) で記述したように, `AWS_ACCESS_KEY_ID` などの環境変数を設定するのが簡単な方法である. あるいは, ローカルマシンの `~/.aws/credentials` に認証情報が書き込まれているなら, このディレクトリを container にマウントすることで, 同じ認証ファイルを container 内部から参照することが可能である. この選択肢を取る場合は, 以下のコマンドで container を起動する.

```
$ docker run -it -v ~/.aws:/root/.aws:ro docker.pkg.github.com/tomomano/learn-aws-by-coding/labc:latest
```

これにより, ローカルマシンの `~/.aws` を container の `/root/.aws` にマウントすることができる. 最後の `:ro` は `read-only` を意味する. 大切な認証ファイルが誤って書き換えられてしまわないように, `read-only` のフラグをつけることをおすすめする.



`/root/` が container 環境におけるホームディレクトリである. ここで紹介した認証ファイルをマウントするテクニックは, SSH 鍵を container に渡すときなどにも使える.

Chapter 16. 謝辞

本原稿の執筆にあたり、以下の方々からの協力を得た。この場を借りて、感謝を表したい。

2021年バージョンの contributors

- yet to come…

2020年バージョンの contributors

- 勝俣敬寛氏 - Docker イメージの作成
- 香取真知子氏 - ハンズオンプログラムの動作確認
- [@shuuji3](#) - MR !15
- [@takatama_jp](#) - MR !14

本書の執筆には [Asciidoctor](#) を使用した。

また、本書はオープンソースの教科書として、すべての読者・ディベロッパーからのフィードバックを受け付けています。誤植や記述の誤り、改善点など見つかったら、ぜひ [Issues](#) や [Pull request](#) を投稿していただきたい。

Chapter 17. 著者紹介

真野 智之 (Tomoyuki Mano)

情報理工学博士 (東京大学大学院情報理工学系研究科システム情報学専攻). 2021年より日本学術振興会特別研究員(PD) (現職). 沖縄科学技術大学院大学 (OIST) にてポスドク研究員として働く. 現在の研究分野は神経科学・神経情報学. 趣味は料理・ランニング・鉄道・アニメ, 村上春樹の熱烈な愛読家.

連絡先 tomoyukimano@gmail.com

GitHub <https://github.com/tomomano>

Chapter 18. ライセンス

本教科書およびハンズオンのソースコードは [CC BY-NC-ND 4.0](#) に従うライセンスで公開しています。

教育など非商用の目的での本教科書の使用や再配布は自由に行なうことが可能です。商用目的で本書の全体またはその一部を無断で転載する行為は、これを固く禁じます。

