

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	3
1.3. AWSアカウント	3
1.4. 環境構築	4
1.5. 前提知識	4
1.6. 講義に関連する資料	5
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?	8
3.AWS入門	11
3.1. AWSとは?	11
3.2. AWSの機能・サービス	11
3.3. Region と Availability Zone	13
3.4. AWSでのクラウド開発	14
3.5. CloudFormation と AWS CDK	18
4.Hands-on #1: 初めてのEC2インスタンスを起動する	21
4.1. 準備	21
4.2. SSH	21
4.3. アプリケーションの説明	22
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 による深層学習の高速化	33
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	41
6.4. ログイン	41
6.5. Jupyter Notebook の起動	43
6.6. PyTorchはじめの一歩	45
6.7. 実践ディープラーニング! MNIST手書き数字認識タスク	49
6.8. スタックの削除	53
7.Docker 入門	55
7.1. 機械学習の大規模化	55
7.2. Docker とは	56
7.3. Docker チュートリアル	58
7.4. Elastic Container Service (ECS)	61
8.Hands-on #3: AWS で自動質問回答ボットを走らせる	63
8.1. Fargate	63
8.2. 準備	64
8.3. Transformer を用いた question-answering プログラム	64
8.4. アプリケーションの説明	66
8.5. スタックのデプロイ	70

8.6. タスクの実行	71
8.7. タスクの同時実行	73
8.8. スタックの削除	75
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	76
9.1. Auto scaling groups (ASG)	76
9.2. AWS Batch	76
9.3. 準備	78
9.4. MNIST 手書き文字認識 (再訪)	78
9.5. アプリケーションの説明	81
9.6. スタックのデプロイ	84
9.7. Docker image を ECR に配置する	86
9.8. 単一のジョブを実行する	89
9.9. 並列に複数の Job を実行する	93
9.10. スタックの削除	97
9.11. 小括	99
10. Web サービスの作り方	100
10.1. ウェブサービスの仕組み – Twitter を例に	100
10.2. REST API	101
10.3. Twitter API	102
11. Serverless architecture	104
11.1. Serverful クラウド (従来型)	104
11.2. Serverless クラウドへ	105
11.3. サーバレスクラウドを構成するコンポーネント	106
12. Hands-on #5: サーバレス入門	111
12.1. Lambda ハンズオン	111
12.2. DynamoDB ハンズオン	116
12.3. S3 ハンズオン	120
13. Hands-on #6: Bashoutter	124
13.1. 準備	124
13.2. アプリケーションの説明	125
13.3. アプリケーションのデプロイ	131
13.4. API リクエストを送信する	134
13.5. 大量の API リクエストをシミュレートする	136
13.6. Bashoutter GUI を動かしてみる	136
13.7. アプリケーションの削除	137
13.8. 小括	138
14. まとめ	139
15. Appendix: 環境構築	140
15.1. AWS アカウントの取得	140
15.2. AWS のシークレットキーの作成	143
15.3. AWS CLI のインストール	145
15.4. AWS CDK のインストール	147
15.5. WSL のインストール	147
15.6. Docker のインストール	150
15.7. Python <code>venv</code> クイックガイド	151
15.8. ハンズオン実行用の Docker image の使い方	152
16. 謝辞	154
17. 著者紹介	155

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>

■告知■

各方面でご好評をいただいている本講義資料ですが、この度増補・改訂のうえ書籍として出版することが決定いたしました！書籍限定の書き下ろしの3章（約100ページ分！）を新たに追加して、2021年9月27日に発売予定です。この資料を気に入っていた方は、手に取っていただけるとありがとうございます。ここで公開している資料は引き続きオンラインで無料で読めますので、ご安心ください！

書籍名: AWSではじめる クラウド開発入門 (真野智之著,マイナビ出版,360ページ)

- Amazon (紙媒体 or Kindle) ⇒ <https://www.amazon.co.jp/dp/4839977607/>
- マイナビブックス (紙媒体 or PDF) ⇒ <https://book.mynavi.jp/ec/products/detail?id=124113>

■英語バージョン■ [こちら](#) のリンクにて鋭意作成中！

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 Services (以下、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。

とくに、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。読者が今後、研究などでクラウドを利用する際の、足がかりとなれば本書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いくつかの構成になっている。

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)	サーバレスアーキテクチャ入門	<ul style="list-style-type: none">Lambda, DynamoDB, S3の演習俳句を投稿する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、はじ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(とくに機械学習)を走らせるための入門となる知識・技術を解説する。あわせて、[Docker](#)とよばれる仮想計算環境の使用方法を紹介する。一つ目のハンズオンでは、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。二つ目のハンズオンでは、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。最後に、複数台のGPUインスタンスからなるクラスターを起動し、並列に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、サーバレスアーキテクチャとよ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(Serverfulとしばしばよばれる)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。ハンズオンでは、サーバレスクラウドの主要なコンポーネントであるLambda, DynamoDB, S3の演習を提供する。さらに、サーバレスの技術を使用して簡単なSNSを作成する。

これらの豊富なハンズオンにより、AWS上にクラウドシステムを開発するための知識と技術が身につくはずである。いずれのハンズオンも、実用性を重視したものになっており、これらをベースにカスタマイズを施すことで様々な応用が可能である。

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っ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"口ケット"とはこの講義のことである。この講義では, 口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, 口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)を深くは問わない。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際に, 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もって執筆された。

決して楽な搭乗体験ではないかもしれないが, この口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巻末付録 (Section 15.1) に簡単に記載したので, 必要に応じて参照していただきたい。

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, ほかのハンズオン(とく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。AWS Educate を利用することで金銭的な負担なしに AWS を体験することができる。また, 講義を経由せず個人でも 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, ゼひ活用してもらいたい。

1.4. 環境構築

本書では, AWS 上にクラウドアプリケーション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には, 以下の計算機環境が必要である。インストールの方法については, 巻末付録 ([Chapter 15](#)) に記してある。必要に応じて参照し, 環境構築を各自実施していただきたい。

- **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 \(WSL\)](#) を使い, Linux の仮想環境のインストールを推奨する ([Section 15.5](#) 参照)。
- **Docker:** 本書では Docker とよばれる仮想計算環境の利用方法を解説する。インストール手順については [Section 15.6](#) を参照のこと。
- **Python:** Version 3.6 以上をインストールする。とくに, ハンズオンでは `venv` モジュールを使用する。`venv` の使い方は [Section 15.7](#) 参照のこと。
- **Node.js:** version 12.0 以上をインストールする。
- **AWS CLI:** [Version 2](#) をインストールする。インストール手順については [Section 15.3](#) 参照のこと。
- **AWS CDK:** Version 1.100 以上をインストールする。Version 2 以降には未対応である。インストール手順については [Section 15.4](#) 参照のこと。
- **AWS 認証鍵の設定:** AWS API をコマンドラインから呼ぶには, 認証鍵 (secret key) が設定されている必要がある。認証鍵の設定については [Section 15.3](#) 参照のこと。

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 15.8](#) に記載している。

1.5. 前提知識

本書を読むにあたり, 要求する前提知識は大学初等程度の計算機科学の知識 (OS, プログラミングなど)のみである。それ以上の前提知識はとくに仮定しない。クラウドの利用経験もゼロで問題ない。が, 以下の事前知識があるとよりスムーズに理解をす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もつので、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。ここに記載されたクラウドの定義・モデルを図示したのが [Figure 2](#) である。

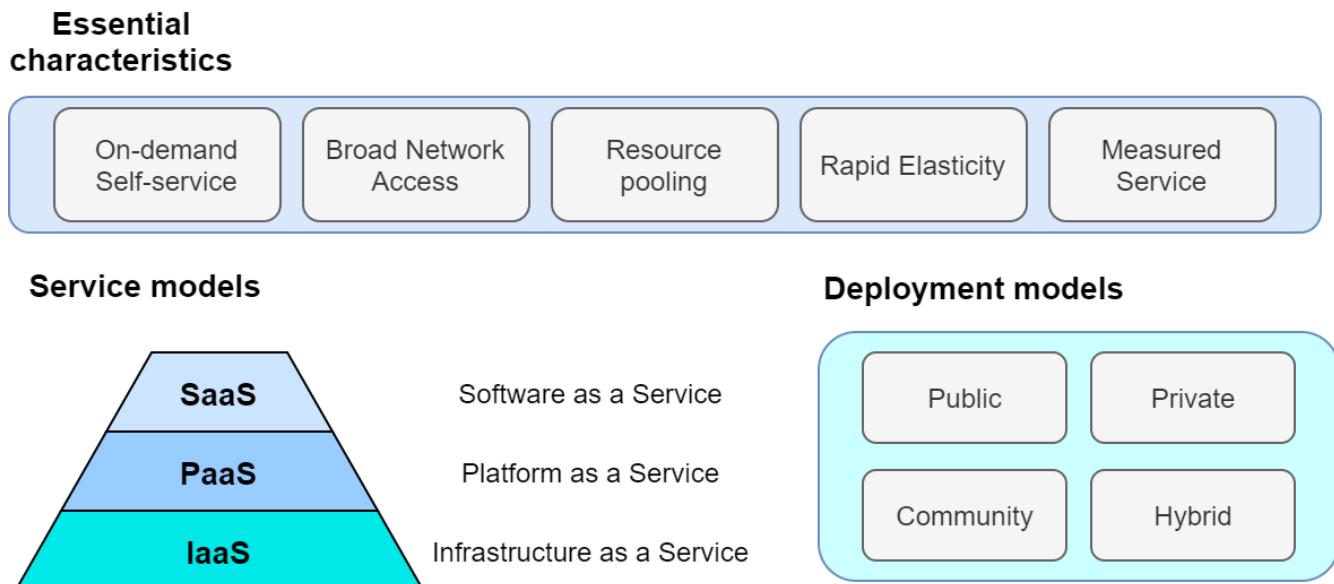


Figure 2. The NIST Definition of Cloud Computing

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時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.

先述の The NIST Definition of Cloud Computing に戻ると,クラウドプロバイダーによるクラウドサービスの形態としては,次の三つが定義されている([Figure 2](#)).

- Software as a Service (SaaS): クラウド上で実行されるアプリケーションをサービスとして利用者に提供する形態.例として,Google Drive や Slack などが挙げられる.利用者は,背後にあるクラウドのインフラ(ネットワークやサーバーなど)には直接触れず,アプリケーションとして提供されているクラウドサービスを享受する.
- Platform as a Service (PaaS): 顧客の作成したアプリケーション(多くの場合データベースとAPIリクエスト処理を行うサーバーのコードから構成される)をデプロイする環境をサービスとして利用者に提供する形態.PaaS では利用者はクラウドのインフラに直接触れるではなく,計算負荷が増減した際のサーバーのスケーリングはクラウドプロバイダーによってなされる.例としては,Google App Engine や Heroku などがある.
- Infrastructure as a Service (IaaS): クラウド上の計算インフラストラクチャーを従量課金制で利用者に提供する形態.利用者は必要なネットワーク・サーバー・ストレージをプロバイダーから借り受け,そこに自身のアプリケーションを展開し運用する.IaaS の例としては AWS EC2 などが挙げられる.

本書が扱うのは,主に IaaS におけるクラウド開発である.すなわち,開発者がクラウドのインフラを直接操作し,所望のネットワーク・サーバー・ストレージなどを一から構成し,そこにアプリケーションを展開するというクラウド開発である.この意味において,クラウドの開発とは**クラウドインフラストラクチャーを定義・展開するプログラムを構築するステップとインフラ上で実際に走るアプリケーションを作成するステップ**の二つに分けることができる.この二つは,プログラマーの技術としてはある程度分業を行なうことが可能であるが,最も効率化・最適化されたクラウドシステムを構築するためには両方の理解が必要である.本書では,前者(クラウドインフラの記述)に重きを置きつつ,アプリケーションレイヤーの話題を取り扱う.PaaS とは,開発者はアプリケーションレイヤーの開発に注力し,クラウドインフラの部分はクラウドプロバイダーに依存するという概念である.PaaS は,クラウドインフラの開発が不要になることで開発の時間が短縮されるが,細かなインフラの挙動はコントロールできないという限界がある.本書では PaaS についてはとくに取り扱わない.

SaaS は本書の文脈では開発による"成果物"と捉えられるだろう.すなわち,IaaS を構成するプログラムを作成し展開することによって,一般の人が利用できるようなウェブ上の計算サービスやデータベースを提供することが開発の最終ゴールである.本書のハンズオンではその実例として,シンプルな SNS の作成([Chapter 13](#))などの演習を提供する.

なお,最近では Function as a Service (FaaS) やサーバーレスコンピューティングなども新たなクラウドのカテゴリとして認知されている.これらの概念については[Chapter 12](#)などの章で詳しく触れていく.本書を読み進める中で明らかになるように,クラウドの技術は日進月歩である.本書では実用的・教育的な観点から,従来的なクラウドの設計概念に触れたあと,サーバーレスなどの最新の技術も網羅するので,楽しみにしながら読み進めていただきた

い。

最後に,The NIST Definition of Cloud Computing によると,クラウドの運用形態について次のような定義がなされている (Figure 2). 特定の組織・団体・企業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)** とよぶ. 例えば,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. 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上限は限られる場合が多く,拡張時の柔軟性に欠ける場合もある.

一方,商用のサービスとして一般の顧客に向けたクラウドのことを, **パブリッククラウド (public cloud)** とよぶ.有名なパブリック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 Services \(AWS\)](#) などがある. 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. その分, 巨大なデータセンターを運用する企業の計算資源にアクセスすることができるので, 計算のキャパシティは無尽蔵にあるといって過言でない.

第三のクラウドの運用形態として, **コミュニティクラウド (community cloud)** が挙げられる. これは, 例えば政府の省庁・機関など目的・役割を共有する団体・組織が共有して運用するクラウドを指す. 最後に, **ハイブリッドクラウド (hybrid cloud)** という形態もあり, これはプライベート・パブリック・コミュニティクラウドの二つ以上の組み合わせによって構成されるクラウドのことである. データ保護の観点から, いくつかの機密データやプライバシーに関わる情報はプライベートクラウドに保持し, 残りのシステムをパブリッククラウドに依存する, などの形態が想定される.

本書で説明するのは, 基本的にパブリッククラウドを使ったクラウド開発である. 特に, [Amazon Web Services \(AWS\)](#) を使用して, 具体的な技術と概念を学んでいく. 一方で, サーバーのスケーリングや仮想計算環境などのテクニックはすべてのクラウドに共通な概念であるので, クラウドのプラットフォームが変わろうと一般に通用する知識も同時に身につくはずだ.

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の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 3](#) 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.

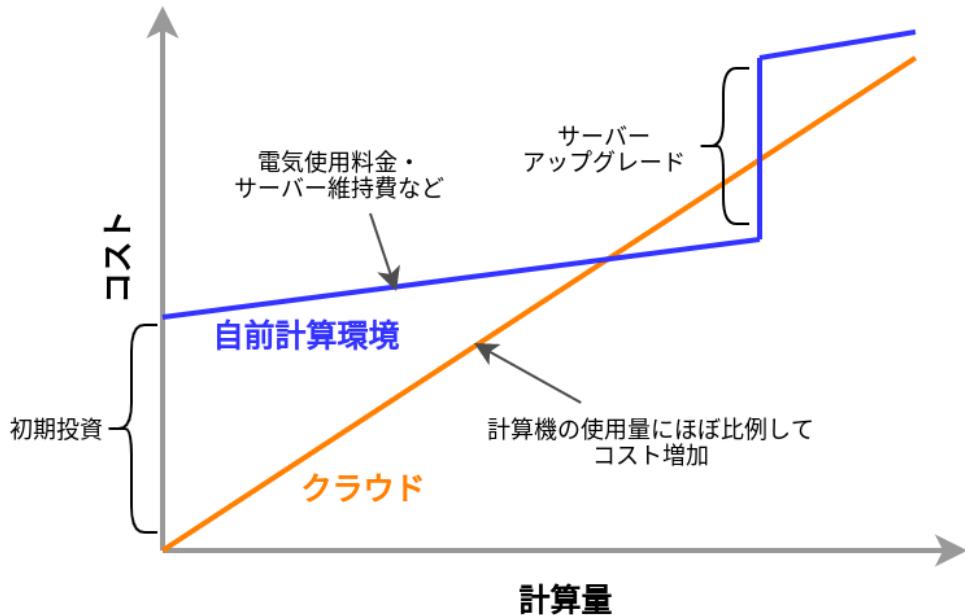


Figure 3. クラウドと自前計算機環境の経済的コストの比較

とくに、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 3 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

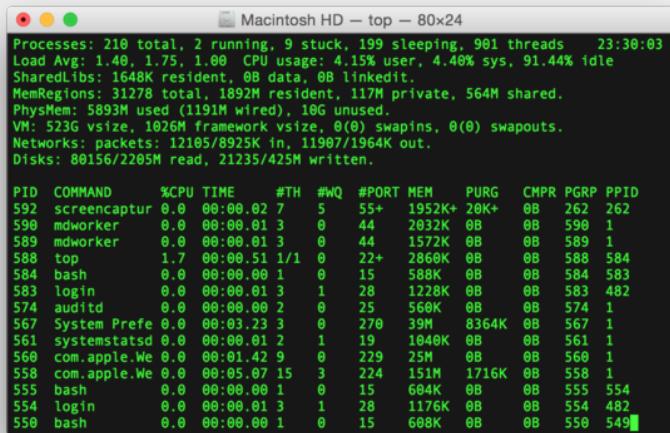
クラウドは、インターネットを通じて世界のどこからでもアクセスできる状態にあり、セキュリティ管理を怠ると簡単にハッキングの対象となりうる。ハッキングを受けると、情報流出だけでなく、経済的な損失を被る可能性がある。

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。

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 とよんだりする。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), 10G 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** とよんでいたのである。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 Services) 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。2021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32%)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リソースがAWSから提供されていることである。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、機能・サービスの幅広さはほかのクラウドプラットフォームと比べ抜きんでている。また、利用者数が多いことを反映して、公式あるいはサードパーティによる技術紹介記事が数多くウェブ上に存在しているだけでなく、ライブラリのユーザーコミュニティも大きく問題解決が渉るのも魅力の一つだ。初期のころウェビジネスを行う企業がユーザーの大半を占めていたが、最近は大学などでの科学的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 4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。

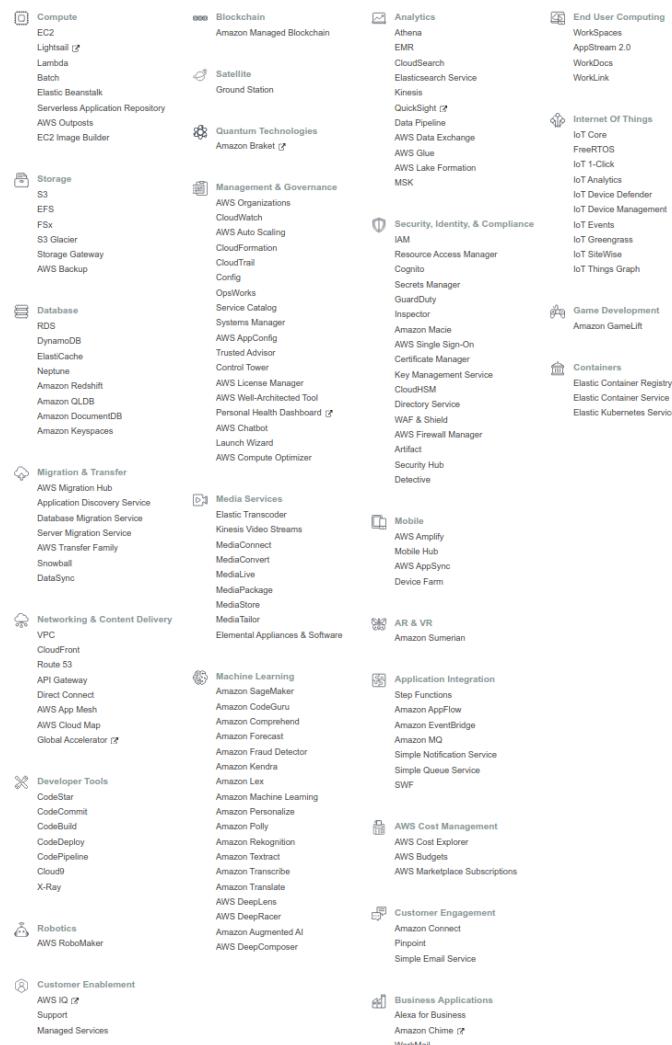


Figure 4. AWSで提供されている主要なサービス一覧

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一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 の初心者が陥りがちなのは,**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**,という状況である。たくさんのサービスの中から,どのサービスをどの順番で学んでいったらいいのか,その道筋すら明らかでなく,大きな参入障壁となっていることは間違いない。だが実のところ, AWS の**基本的な構成要素はそのうちの数個のみに限られる**。基本要素となる機能の使い方を知れば,AWS のおよそのリソースを使いこなすことが可能になる。ほかの機能の多くは,基本の要素を組み合わせて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。そのポイントを認知することが,AWS の学習の最初のステップである。

ここでは,AWS 上でクラウド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。

3.2.1. 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) とよばれる,小さな計算を**サーバーなし**で実行するためのサービス。サーバーレスアーキテクチャの章([Chapter 11](#))で詳しく解説する。

3.2.2. 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い。



S3 (Simple Storage Service) Object Storage とよばれる,APIを使ってデータの読み書きを行う,いわゆる"クラウド・ネイティブ"なデータの格納システムである。サーバーレスアーキテクチャの章([Chapter 11](#))で詳しく解説する。

3.2.3. 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い)。サーバーレスアーキテクチャの章([Chapter 11](#))で詳しく解説する。

3.2.4. 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。



API Gateway API のエンドポイントとバックエンドのサービス(Lambda など)を接続する際に用いる,リバースプロキシとしての役割を担う。[Chapter 13](#) で詳しく解説する。

3.3. 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, **リージョン (Region)** と **Availability Zone (AZ)** がある (Figure 5). 以下ではこの概念について簡単に記述する.

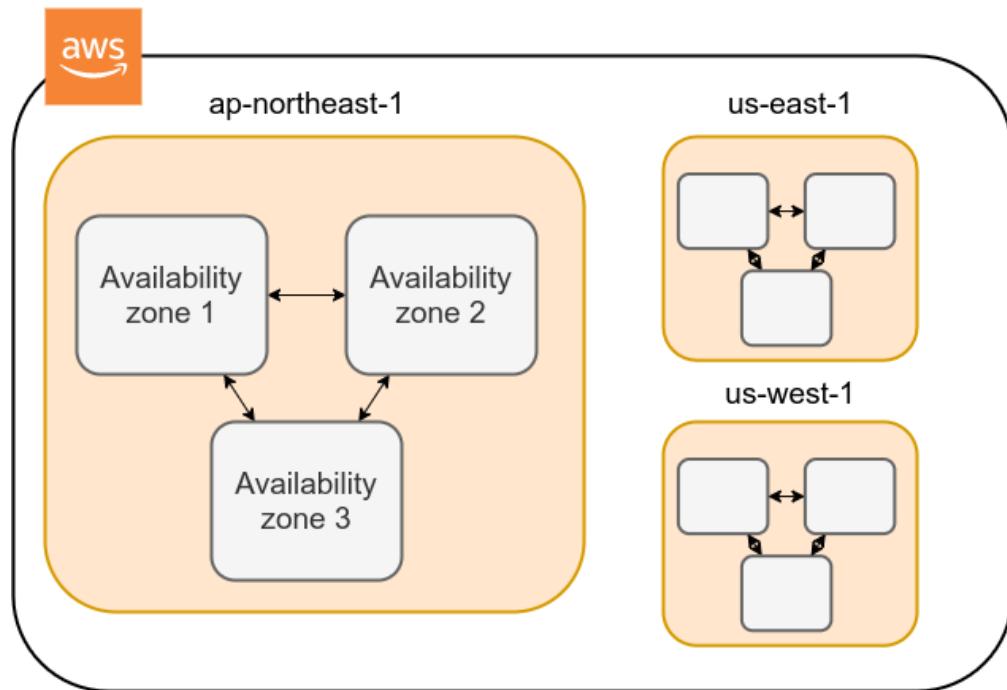


Figure 5. AWSにおける Region と Availability Zones

リージョン (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 6 は執筆時点で利用できるリージョン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. 各リージョンには固有の 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 6. 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 7, 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際, あるいはデプロイ済みのリソースを閲覧する際は, 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などは, 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.

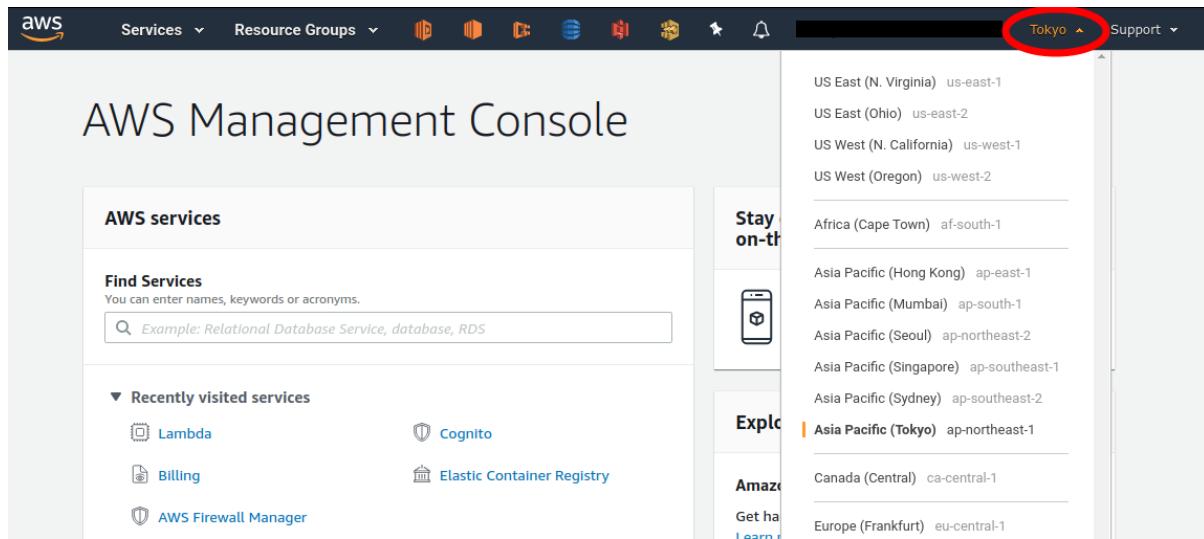


Figure 7. AWSコンソールでリージョンを選択

Avaialibity Zone (AZ) とは、リージョン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、もし一つのAZで火災や停電などが起きた場合でも、ほかの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を使用する際、どこのリージョン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の観点からは、地理的に一番近いリージョンを使用するのが一般的によいだろう。一方、EC2の利用料などはリージョンごとに価格設定が若干（10-20%程度）異なる。したがって、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、特定のリージョン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リージョンを決めると良い。

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"](#)

3.4. 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しよう。

AWSのリソースを追加・編集・削除する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。

3.4.1. 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 8）。

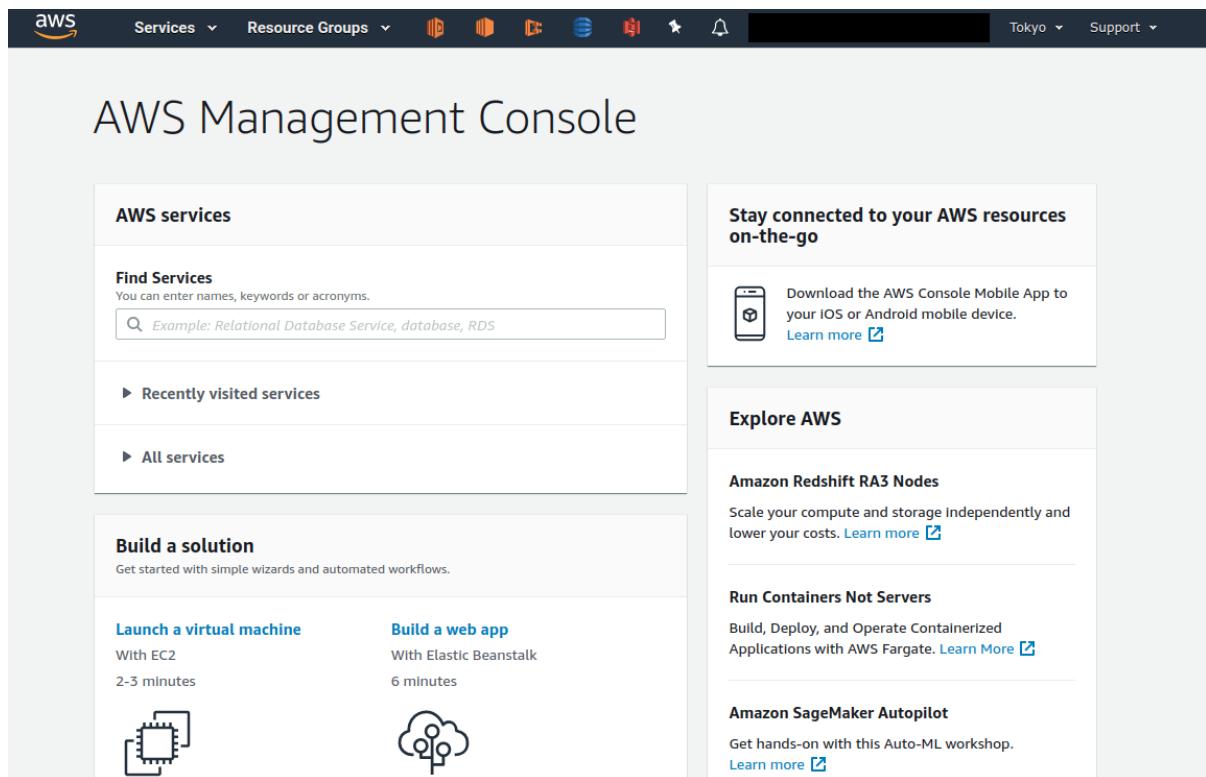


Figure 8. AWSマネージメントコンソール画面

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。

3.4.2. 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。

例えば、[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)** とよばれる) を追加したいとしよう。AWS CLI を使った場合は、次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` のリージョンに作成する。

Pythonからこれと同じ操作を実行するには、`boto3` ライブラリを使って、次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、次のような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
```

このコマンドにより、`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。

3.4.3. 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 は先述のとおり、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 15.3](#) 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 15.3](#)を参照.

まずは,S3にデータの格納領域(`Bucket`とよばれる.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ぬことから,前述のコマンド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できる.

Amazon Resource Name (ARN)

AWS 上のあらゆるリソースには、Amazon Resource Name (ARN) という固有の ID が付与されている。ARN は [arn:aws:s3:::my_bucket/](#) のようなフォーマットで記述され、ARN を使用することで、特定の AWS リソース (S3 のバケットや EC2 のインスタンス) を一意的に参照することができる。

S3 バケットや EC2 インスタンスなどには ARN に加えて、人間が読みやすい名前を定義することも可能である。この場合は、ARN または名前のどちらを用いても同じリソースを参照することが可能である。

3.5. CloudFormation と AWS CDK

3.5.1. 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一つある。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 9)。

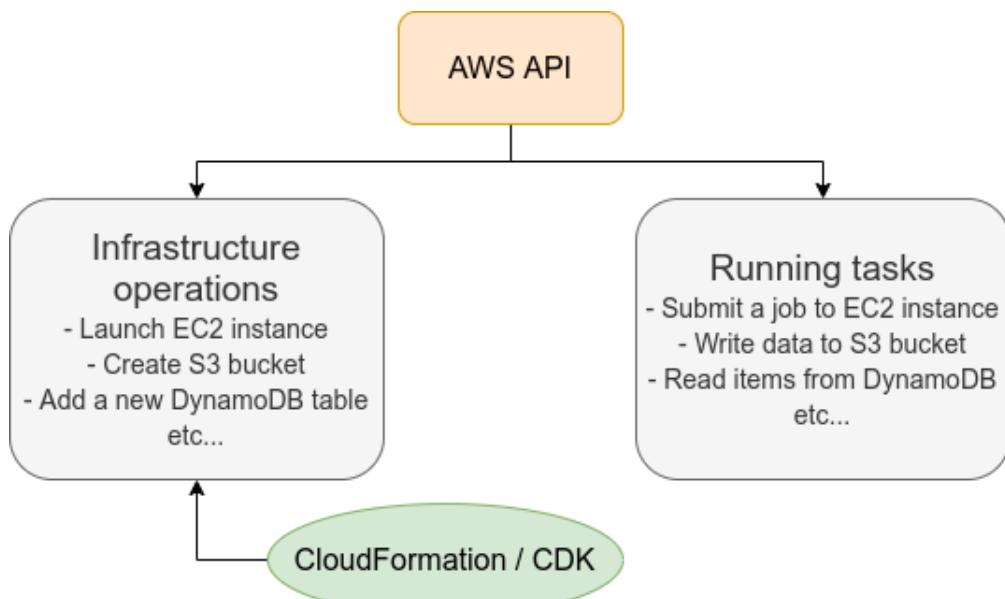


Figure 9. AWS APIはリソースを操作するコマンドとタスクを実行するコマンドに大きく分けられる。リソースを記述・管理するのに使われるのが、CloudFormation と CDK である。

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"を作る操作とよんでも良いだろう。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、たとえば、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 ファイルができ上がれば,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)**とよぶ.

CloudFormation を記述するには, 基本的に **JSON** (JavaScript Object Notation) とよばれるフォーマットを使う. 次のコードは, 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.

3.5.2. 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 リポジトリ](#) 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

ので、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         )
```

このコードは、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！

4.1. 準備

ハンズオンのソースコードは GitHub の [handson/ec2-get-started](#) に置いてある。



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。

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。

- **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 15.1](#) を参照のこと。
- **Python と Node.js:** 本ハンズオンを実行するには、Python (3.6 以上), Node.js (12.0 以上) がインストールされていなければならない。
- **AWS CLI:** AWS CLI のインストールについては、[Section 15.3](#) を参照。ここに記載されている認証鍵の設定も済ませておくこと。
- **AWS CDK:** AWS CDK のインストールについては、[Section 15.4](#) を参照。
- **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 GitHub 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。

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 15.8](#) 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おう。

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のである。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を次に示す。`<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 <user name>@<host name>
```

SSH は平文のパスワードによる認証を行うこともできるが、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(読者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。たとえば、次のように使う。

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

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 10](#) に示す。

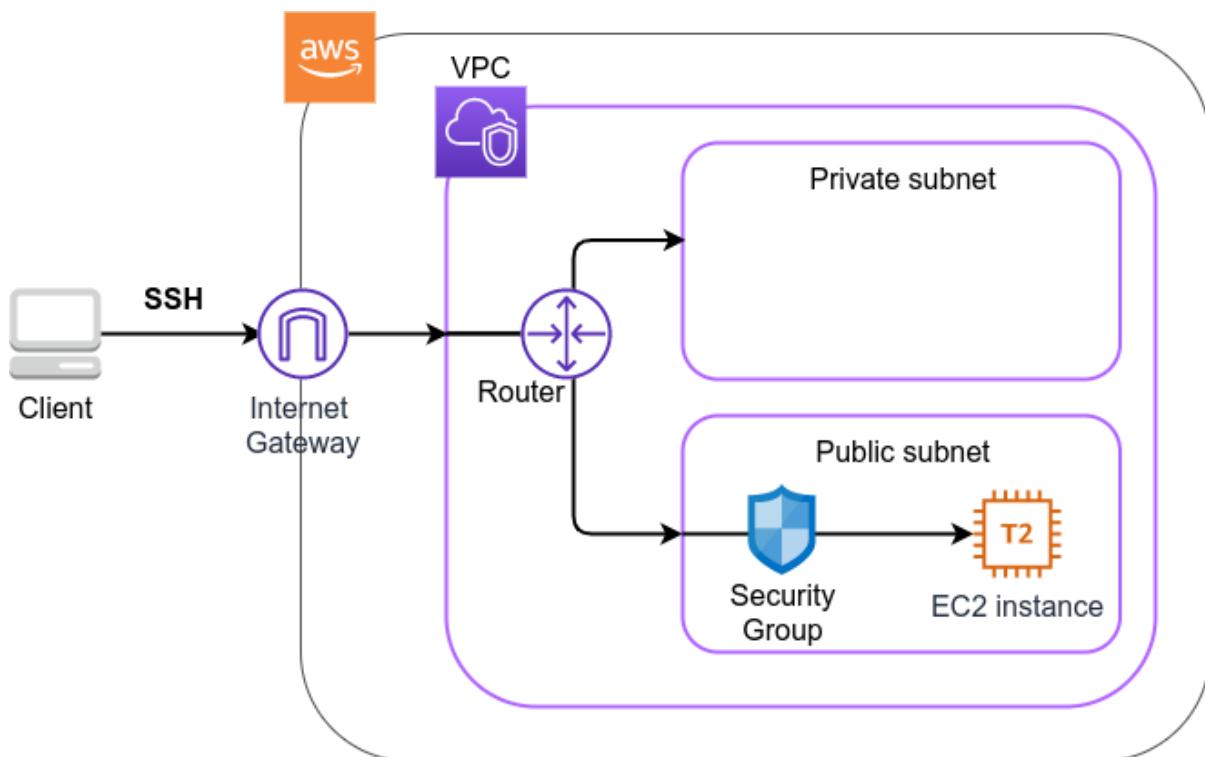


Figure 10. ハンズオン#1で作製するアプリケーションのアーキテクチャ

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。

[Figure 10](#) 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 のアイコン



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、そういった目的で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。たとえば、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。



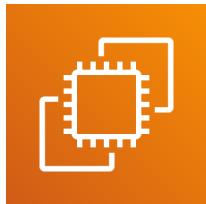
SSHはデフォルトでは22番ポートを使用するのが慣例である。



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のアイコン



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)とよぶ(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。[Table 2](#)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 "Amazon EC2 Instance Types"](#)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

[Table 2](#)からわかるように、CPUは1コアから96コアまで、メモリーは1GBから2TB以上まで、ネットワーク帯域は最大で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、 $0.0464 \times 10 = 0.464$ 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 ではほかに, [Spot instance](#) とよばれるインスタンスも存在しする。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。科学計算やウェブサーバーなどの用途でコストを削減する目的で, Spot Instance を活用する事例も多数報告されている。

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](#) として, [Amazon Linux](#) を選択している(Machine image は OS と似た概念である。Machine image については, [Chapter 6](#) 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 15.7](#) 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 15.2](#) を参照のこと。シークレットキーを発行したら、[Section 15.3](#) を参照し、コマンドラインの設定を行う。

手順をここに短く要約すると、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。

次の 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
```

こ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、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、EC2 インスタンスをデプロイするための準備が整った！早速、次のコマンドによりアプリケーションを AWS にデプロイしよう。`-c key_name="HirakeGoma"` というオプションで、先ほど生成した `HirakeGoma` という名前の鍵を使うよう指定している。

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

こ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 11](#) 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 11. 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,先ほど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> の部分は自身が作成した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 12 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 12. 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** とページをたどる。すると、Figure 13 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できる。同様に、VPC・SG についてもコンソールから確認できる。

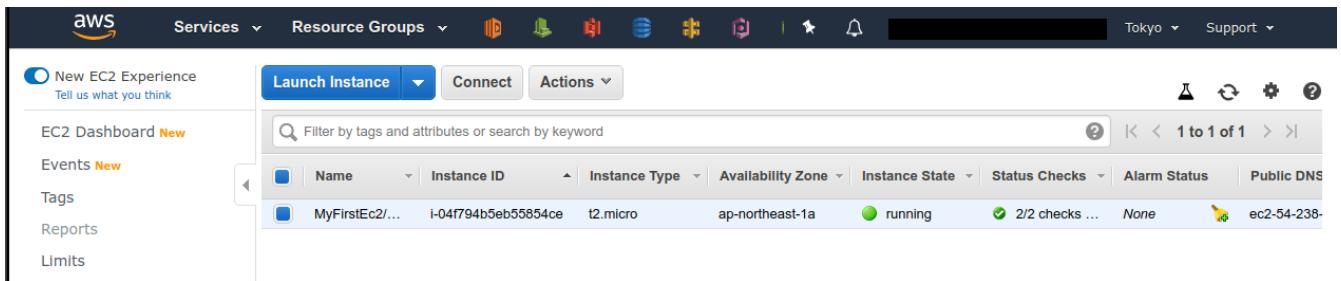


Figure 13. EC2 コンソール画面



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！

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 14）。

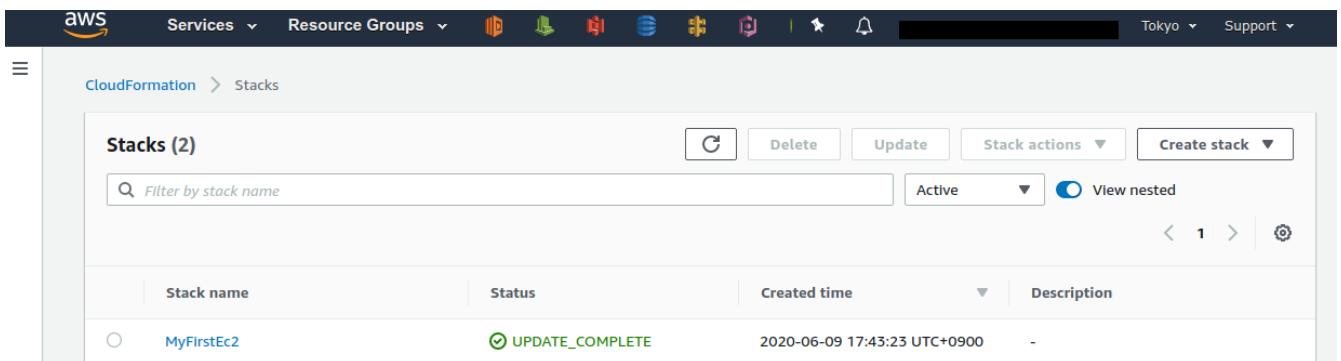


Figure 14. CloudFormation コンソール画面

"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、VPC 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、二つの方法がある。

一つ目の方法は、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 15）。すると、スタックの状態が "DELETE_IN_PROGRESS" に変わり、削除が完了すると CloudFormation のスタックの一覧から消える。

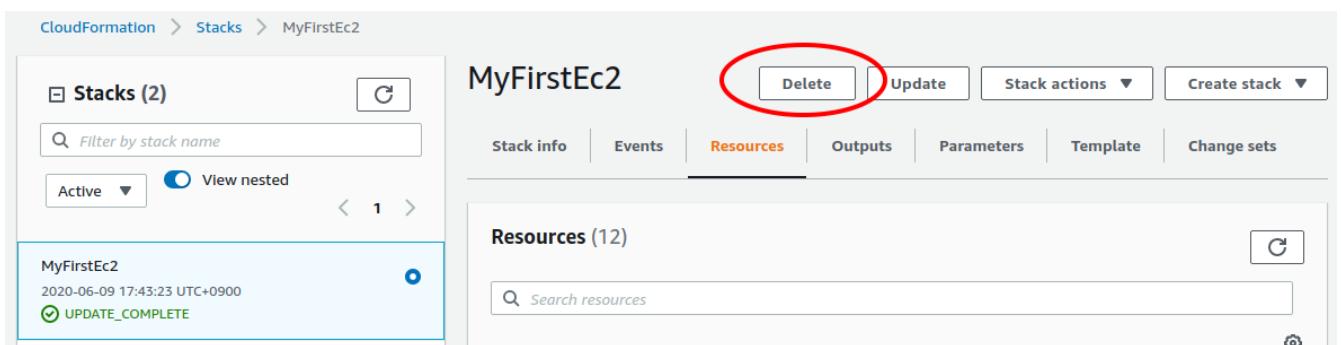


Figure 15. CloudFormation コンソール画面から、スタックを削除

二つ目の方法は、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、次のコマンドを実行する。

```
$ cdk destroy
```

このコマンドを実行すると、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とを自身で確かめよう。CloudFormation を用いることで関連するすべての AWS リソースを一度に管理・削除することができるので、大変便利である。



■ **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二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** を選択する。鍵の一覧が表示されるので、**HirakeGoma** とある鍵にチェックを入れ、画面右上の **Actions** から、**Delete** を実行する (Figure 16)。

The screenshot shows the AWS EC2 Key Pairs page. On the left sidebar, under the 'INSTANCES' section, 'Instances' is selected. In the main content area, the 'Key pairs (1/1)' section is displayed. A table lists one key pair: 'HirakeGoma'. The 'Actions' button next to this entry is highlighted in orange, indicating it is the target for deletion.

Figure 16. EC2でSSH鍵ペアを削除

コマンドラインから実行するには、次のコマンドを使う。

```
$ 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、クラウドの定義と用語の説明を行ったあと、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プラットフォームとして AWS を取り上げ、AWS を使用するにあたり最低限必要な知識と用語の説明を行った。さらに、**Chapter 4** のハンズオンでは AWS CLI と AWS CDK を使って、自身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、体験できただろう。筆者は、**Chapter 2** でクラウドの最も重要な側面はダイナミックに計算リソースを拡大・縮小できることである、と述べた。この言葉の意味が、ハンズオンを通じてより明らかにならんだろうか？ここで学んだ技術を少し応用するだけで、自分のウェブページをホストする仮想サーバーを作成したり、大量のコアを搭載した EC2 インスタンスを用意して科学計算を実行するなど、いろいろなアプリケーションが実現できる。

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年頃に始まった第三次 AI ブームのおかげで、学術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。とくに、**深層学習 (ディープラーニング)** とよ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。

深層学習の特徴は、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。たとえば、最新の言語モデルである [GPT-3](#) には**1750億個**も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、深層学習は高い表現力と汎化性能を実現しているのである。

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ワークでは、百万から億のオーダーのパラメータを内包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。結果として、ひとつの計算機では丸一日以上の時間がかかる場合も珍しくない。深層学習の発展の速度は目覚ましく、研究・ビジネス両方の観点からも、いかにスループットよくニューラルネットワークの最適化を行えるかが鍵となってくる。そのような問題を解決するのにとても有効な手段が、クラウドである！[Chapter 4](#) でその片鱗を見たように、クラウドを使用することでゼロから数千に至るまでの数のインスタンスを動的に起動し、並列に計算を実行することができる。さらに、深層学習を加速させる目的で、深層学習の演算に専用設計された計算チップ (GPU など) がある。クラウドを利用すると、そのような専用計算チップも無尽蔵に利用することができる。事実、先述した 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。

5.2. GPU による深層学習の高速化

深層学習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 (30 fps) 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。

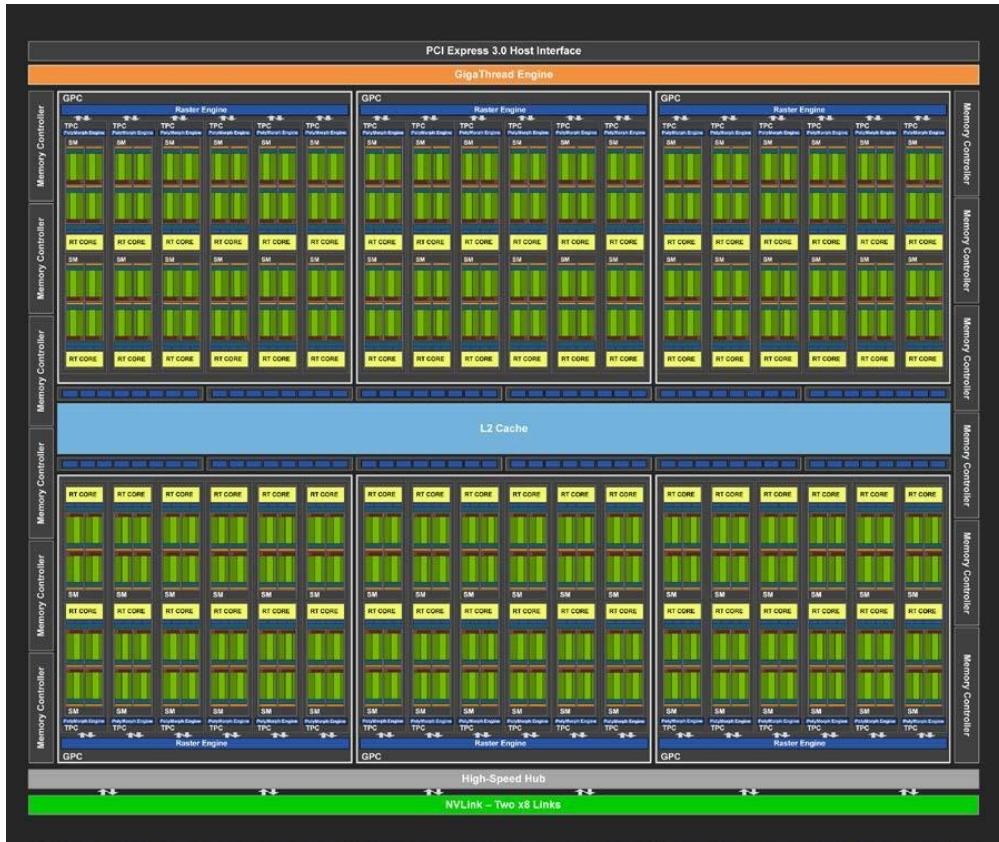


Figure 17. GPUのアーキテクチャ.GPUには数百から数千の独立した計算コアが搭載されている. (画像出典: <https://devblogs.nvidia.com/nvidia-turing-architecture-in-depth/>)

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もたない 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 18)。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。

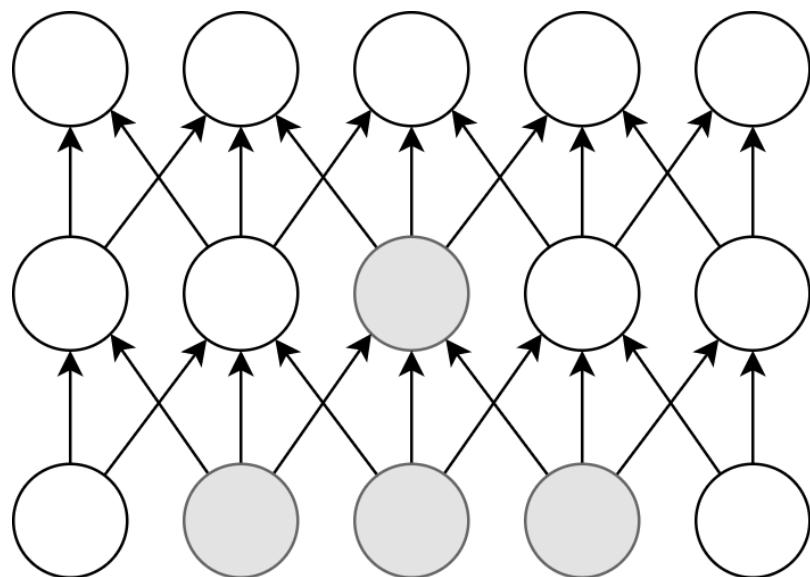


Figure 18. ニューラルネットワークにおける畳み込み演算.

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。たとえば、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とく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場合がほとんどである。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）。

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ってくるとクラウド技術の限界を攻めないと達成できないことは確かである。

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

6.1. 準備

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、深層学習モデル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 GitHub の [hands-on/mnist](#) に置いてある。

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。



初期状態の AWS アカウントでは、GPU 搭載の G タイプのインスタンスの起動上限が 0 になっていることがある。これを確認するには、AWS コンソールから EC2 の画面を開き、左のメニューから **Limits** を選択する。その中の **Running On-Demand All G instances** という数字が G インスタンスの起動上限を表している。

もし、これが 0 になっていた場合は、AWS の自動申請フォームから上限緩和のリクエストを送る必要がある。詳しくは [公式ドキュメンテーション "Amazon EC2 service quotas"](#) を参照のこと。



このハンズオンは、**g4dn.xlarge** タイプの EC2 インスタンスを使うので、東京 (**ap-northeast-1**) リージョンでは 0.71 \$/hour のコストが発生する。



AWS Educate Starter Account を使用している読者へ：執筆時点においては、Starter Account には GPU 搭載型インスタンスを起動できないという制限が設けられている。したがって、Starter Account のユーザーはこのハンズオンを実行することはできない。興味のある読者は、制限のない一般アカウントを自分自身で取得する必要があることに注意。

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 19 に示す。

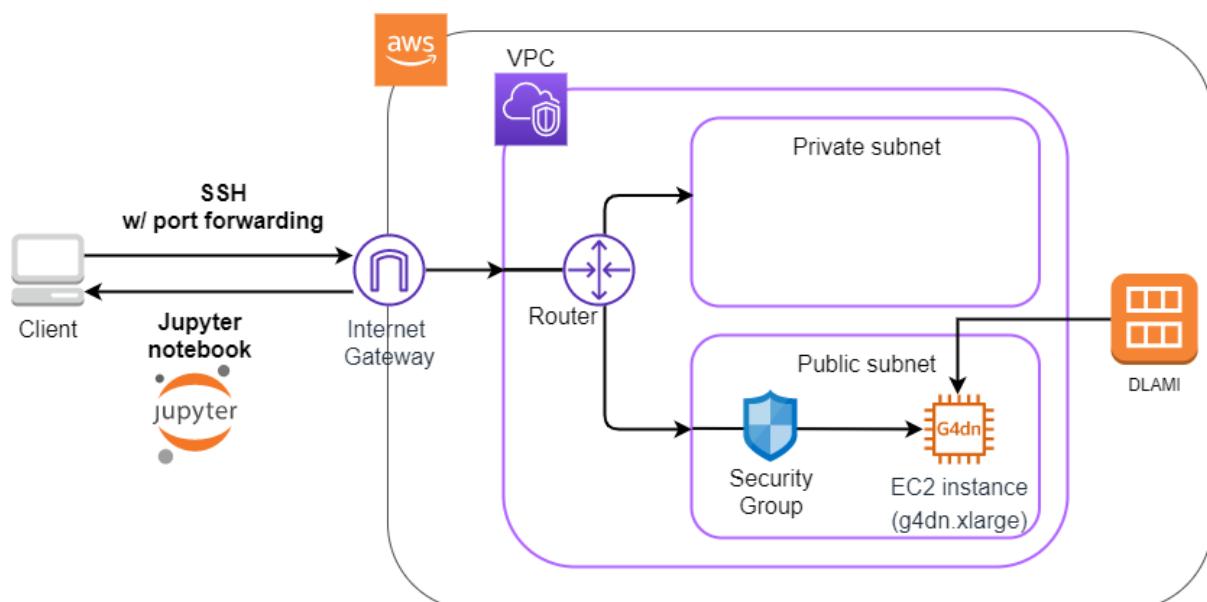


Figure 19. ハンズオン#2で作製するアプリケーションのアーキテクチャ

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あらかじ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                 "us-east-1": "ami-060f07284bb6f9faf",  
34                 "ap-northeast-1": "ami-09c0c16fc46a29ed9"  
35             }), ②  
36             vpc=vpc,  
37             vpc_subnets=ec2.SubnetSelection(subnet_type=ec2.SubnetType.PUBLIC),  
38             security_group=sg,  
39             key_name=key_name  
40         )
```

① ここで、`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）。`g4dn.xlarge` のインスタンスタイプは、Chapter 5 すでに触れた通り、NVIDIA T4 と呼ばれる廉価版モデルの GPU を搭載したインスタンスである。CPU は 4 core、メインメモリーは 16GB が割り当てられている。

② ここでは、Deep Learning 用の諸々のソフトウェアがプリインストールされた 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 リージョンごとに指定する必要があり、ここでは `us-east-1` と `ap-northeast-1` でそれぞれ定義して

いる。

DLAMI という新しい概念が出てきたので、説明しよう。



AMI が [us-east-1](#) と [ap-northeast-1](#) でしか定義されていないので、提供されているコードはこの二つのリージョンのみでデプロイ可能である。もしほかのリージョンを利用したい場合は、AMI の ID を自身で検索し、コードに書き込む必要がある。

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ときには必ずそれにかの 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、たとえば [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 を選択することもできるが、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したり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が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あらかじ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・ライブラリがすで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。この AMI は ap-northeast-1 でしか使用できない点に注意）。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" --region ap-northeast-1
```

```

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

Figure 20 のような出力が得られるはずである。得られた出力から、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 "What Is the AWS Deep Learning AMI?"）。

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。ここまでが、"Base" とよばれるタイプの DLAMI の中身である。

これに加えて、"Conda" とよばれるタイプには、"Base" 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・MxNet の環境など、フレームワークを簡単に切り替えることができる（これについては、後の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 15.3)。

```
# 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がすでに存在する場合は、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 21 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 21. 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(port forwarding)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。このコマンドの-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TCP/IPポートの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(figure 22)。このようなSSHによる接続方式をトンネル接続とよぶ。

```
ssh -L localhost:8931:localhost:8888 ec2-user@0.0.0.0
```

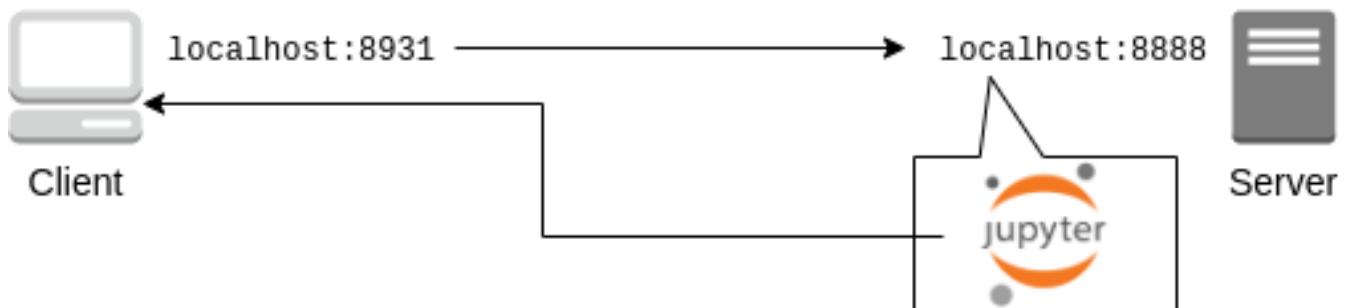


Figure 22. SSH のポートフォワーディングによる Jupyter Notebook へのアクセス



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、たとえばポート 22 (SSH) やポート 80 (HTTP) など、いくつかすで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> 部分は自身のインスタンスの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 の外にも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、出力結果をコピーしてホストマシンのファイルに書き込む方法である。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 [Docker 公式ドキュメンテーション "Use volumes"](#) 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。次のコマンドを実行する。

```
$ nvidia-smi
```

Figure 23 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、GPU の負荷・メモリー使用率などの情報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 23. 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 24)。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。

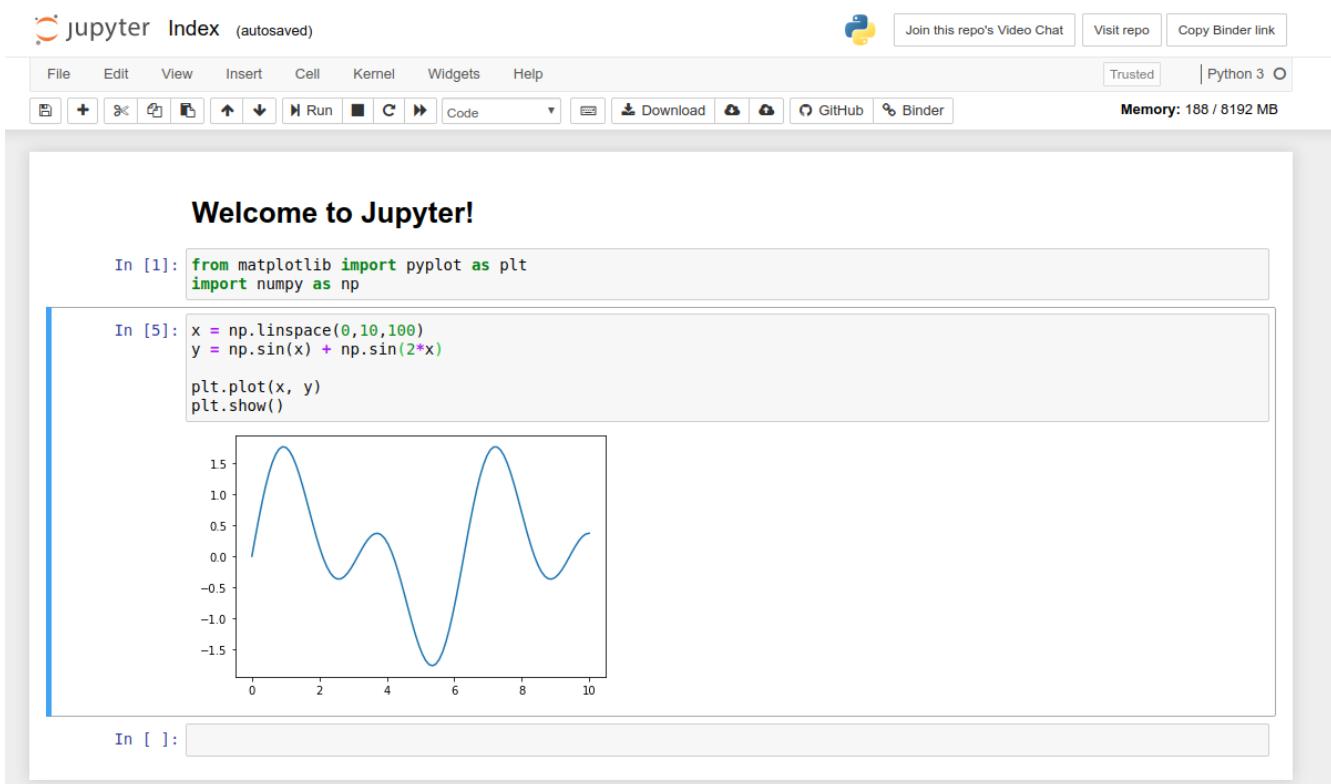


Figure 24. Jupyter Notebook の画面

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インタラクティブに実行していく。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 25 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=XXXX` 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 25. Jupyter Notebook サーバーを起動



Jupyter Notebook を初回に起動するときは, 起動に数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. これは, AWS の GPU 搭載型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=XXXX
```

`?token=XXXX` 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 26). これで, Jupyter の準備が整った!

The screenshot shows the Jupyter Notebook interface. At the top, there is a navigation bar with tabs for 'Files' (selected), 'Running', 'IPython Clusters', and 'Conda'. On the right side of the header, there are 'Quit' and 'Logout' buttons. Below the header, there is a search bar labeled 'Select items to perform actions on them.' and several action buttons: 'Upload', 'New', and a refresh icon. The main area displays a list of files and clusters in a table format. The columns are 'Name', 'Last Modified', and 'File size'. The table includes the following entries:

	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 26. 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 [Ventsislav Yordanov 氏によるブログ](#) 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 のほかに Caffe2 とよ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(詳しくは [プレスリリース](#) を参照)。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うえで、新規ノートブックを作成する([Figure 27](#))。"conda_pytorch_p36" の仮想環境には、PyTorch がインストール済みである。

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

ここでは、次のようなプログラムを書いて、実行していく。(Figure 28).

```
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 28. 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とき, 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、勉強のとき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。

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。

6.7. 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 29). これは,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 29. 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,多少の改変を行ったものである。

まずは,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 30). 画面右上の "Upload" ボタンをクリックし,ファイルを選択することでアップロードができる。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身でソースコードを読んでみるとよい。



Figure 30. `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、まずは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, evaluate

```

`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`, `evaluate`) 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 31 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()

```

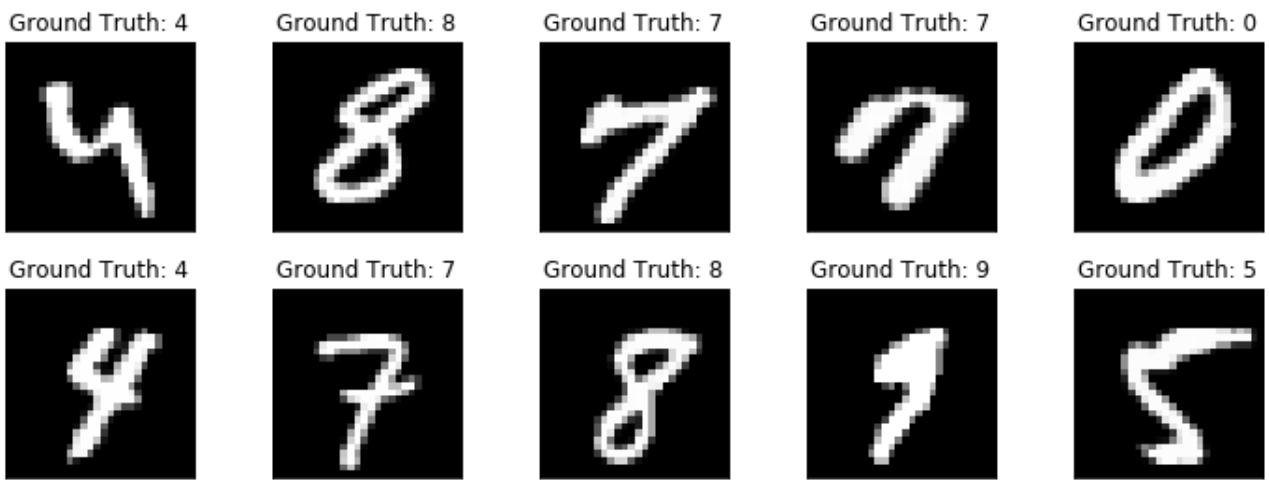


Figure 31. MNIST の手書き数字画像とその教師ラベル

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 32](#) 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.

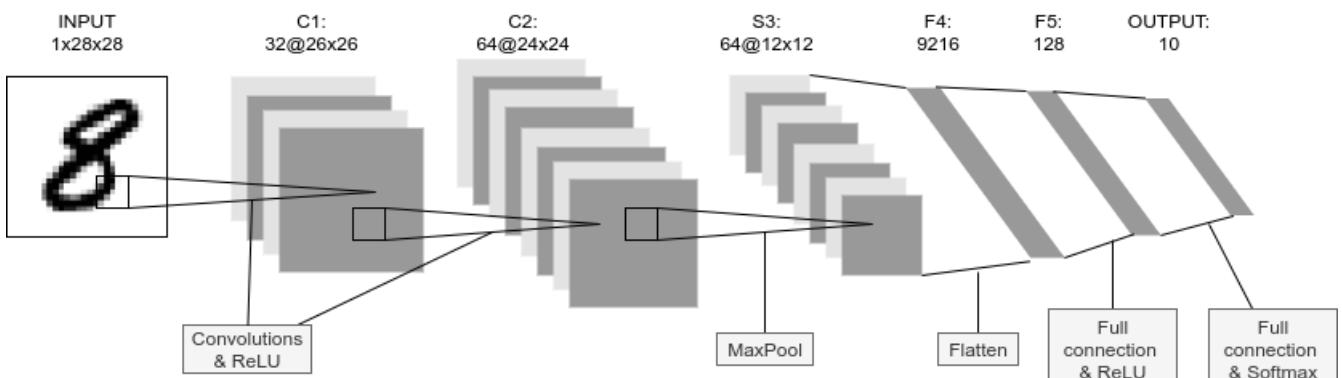


Figure 32. 本ハンズオンで使用するニューラルネットの構造.

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, 確率的勾配降下法 (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_loss, test_accuracy = evaluate(model, testloader)
6     print(f"\nTest set: Average loss: {test_loss:.4f}, Accuracy: {test_accuracy:.1f}%\n")
7
8 plt.figure(figsize=(7,5))
9 plt.plot(train_losses)
10 plt.xlabel("Iterations")
11 plt.ylabel("Train loss")
12 plt.show()
```

ここでは5エポック分の学習を行っている。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 33 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。

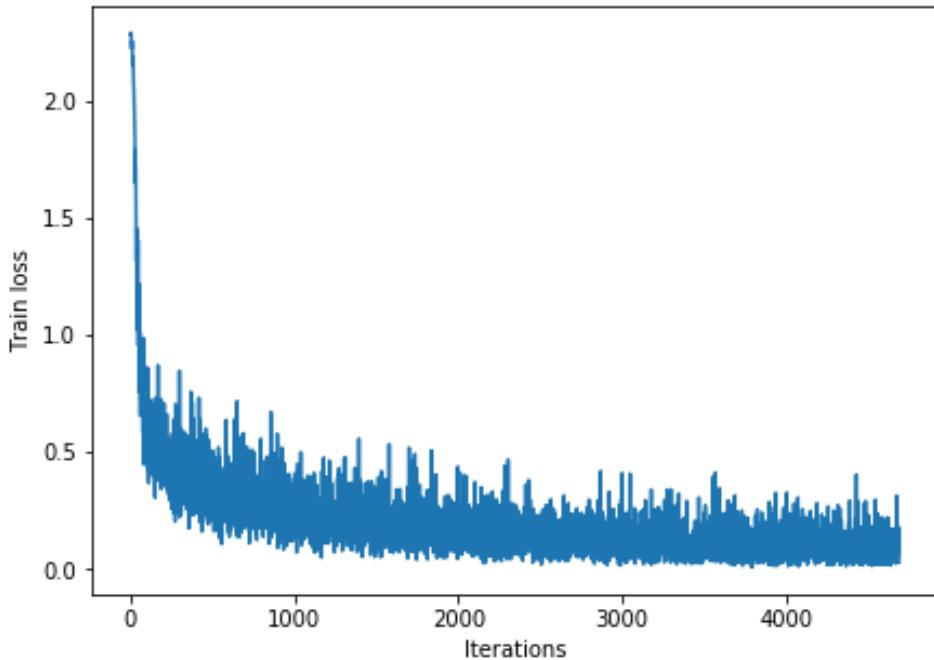


Figure 33. 学習の進行に対する Train loss の変化

出力にはテキスト形式で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 34)。

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

Figure 34. 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)

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 35 のような出力が得られるだろう。この図で、下段右から二番目は、"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()

```

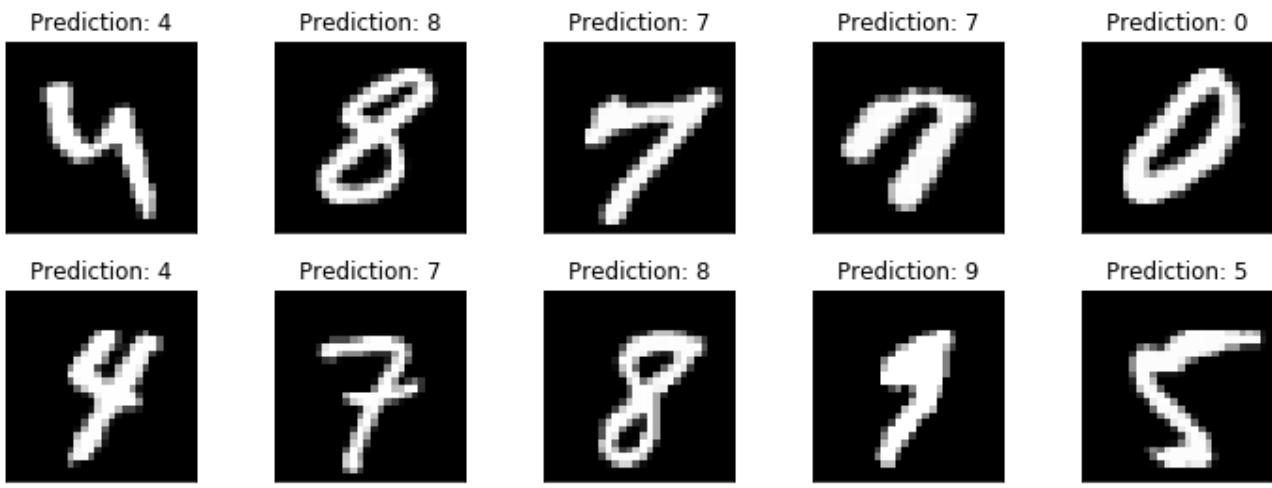


Figure 35. 学習した CNN による, MNIST 画像の推論結果

最後に,学習したニューラルネットワークのパラメータを `mnist_cnn.pt` というファイル名で保存しておこう. これで,将来いつでも今回学習したモデルを再現し,別の実験に使用することができる.

```

1 torch.save(model.state_dict(), "mnist_cnn.pt")

```

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.

6.8. 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 \$0.71 / hour の料金設定なので, 一起動しつづけると約\$17の請求が発生することになる!

AWS のバジェットアラート

AWS の初心者が (あるいは経験者も) しばしば陥る失敗が, インスタンスの停止忘れなどで無駄なリソースがクラウドで放置されてしまい, 巨大な額の請求が届く, というミスだ. 特に, 開発を行っている間はこのような事態は起こりうるものだと思って, 備えておかなければならない. このような事態を未然に防ぐため, AWS Budgets という機能が無料で提供されている. AWS Budgets を利用することで, 月の利用金額がある閾値を超えた場合にユーザーにメールが送信される, などのアラートを設定することができる. 詳細な手順は [AWS の公式ブログ "Getting Started with AWS Budgets"](#) を参照のこと. 本書の読者も, ぜひこのタイミングでアラートを設定しておくことを推奨する.

Chapter 7. Docker 入門

ここまでこの章で扱ってきたハンズオンでは、**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発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させ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章分 ([Chapter 7](#), [Chapter 8](#), [Chapter 9](#)) を使って、クラウドを利用する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。とくに、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** とよばれる計算機環境の仮想化ソフトウェアを紹介する ([Figure 38](#))。現代のクラウドは Docker なしには成り立たないといつても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。AWS 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思う。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 36](#) 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。

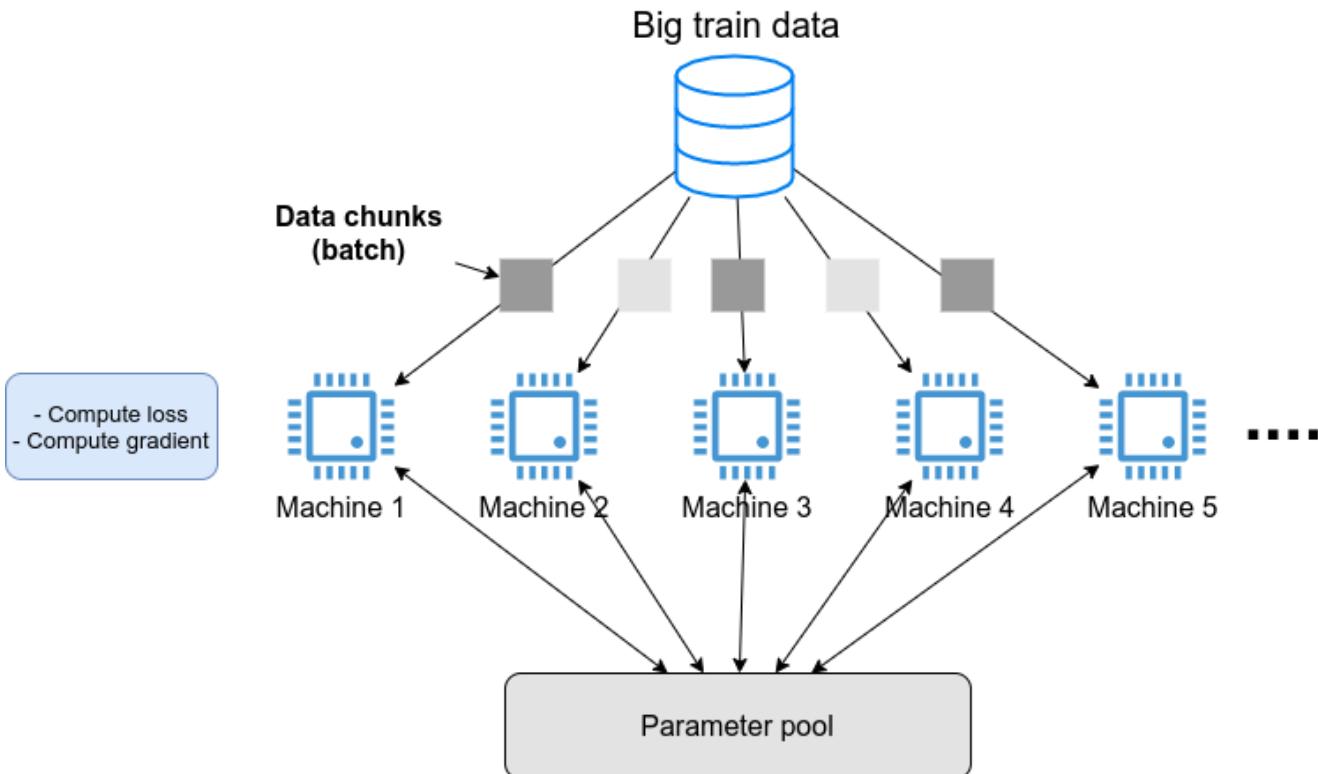


Figure 36. 複数の計算機を使った大規模な深層学習モデルの訓練

あるいは、学習済みのモデルを大量のデータに適用し、解析を行いたいと思う。たとえば、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 37](#) 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。

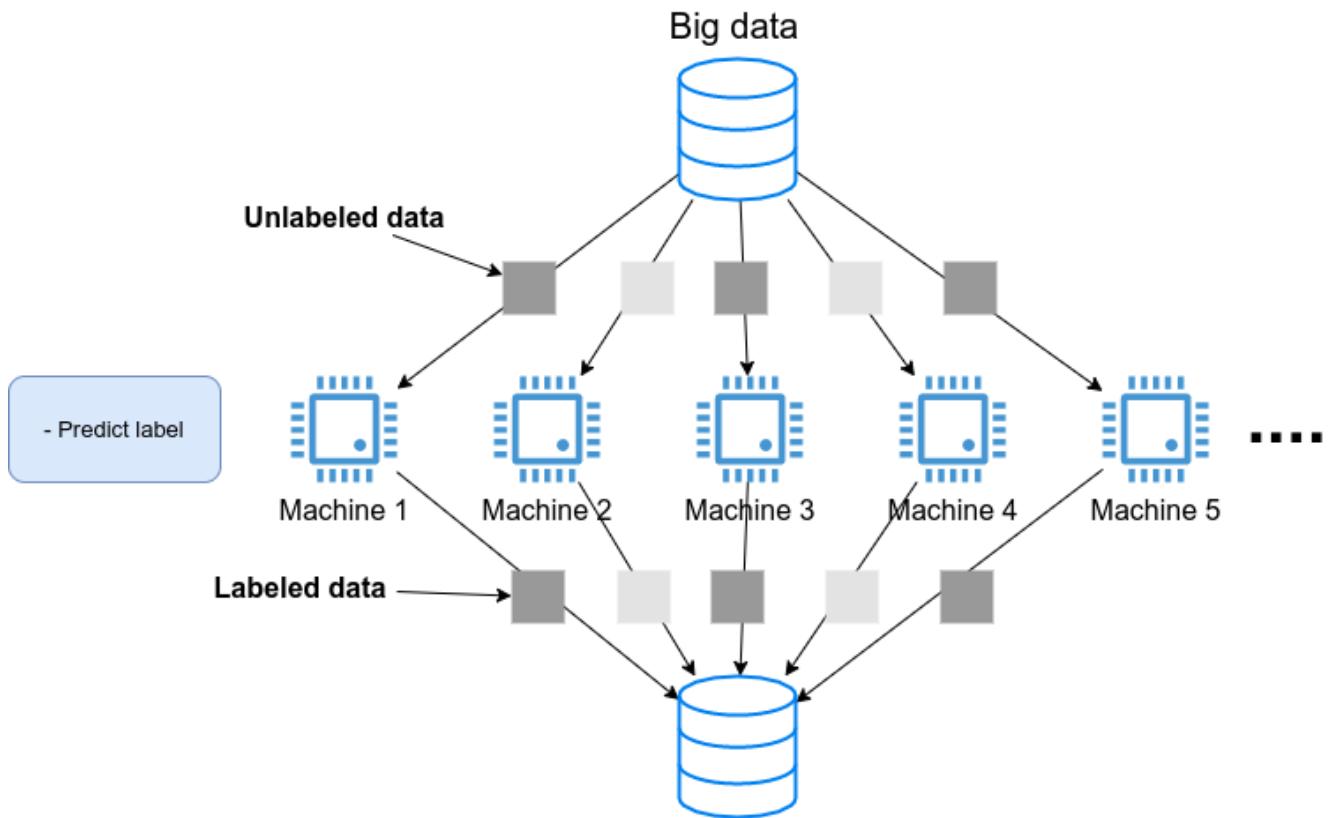


Figure 37. 複数の計算機による深層学習モデルを使った推論計算

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？

重要なポイントとして、Figure 36 や Figure 37 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行う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とよばれるソフトウェアである。

7.2. Docker とは



Figure 38. Docker のアイコン

Dockerとは、**コンテナ (Container)**とよ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を含めたすべ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（パッケージされた一つの計算環境のことを **イメージ (Image)** とよぶ）。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、Figure 37 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 らを中心に開発され、そ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も欠かすことのできないソフトウェアとなった。Dockerはエンタープライズ向けの製品を除き無料で使用することができ、コアの部分は **オープンソースプロジェクト** として公開されている。Dockerは Linux, Windows, Mac いずれの OS でも提供されている。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。

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 39). VM にはハイパーバイザ (Hypervisor) とよ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。たとえば、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、たとえば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 (これをisolationとよぶ)。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技術を使うことで所望のスペックをも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 39)。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 (=イメージ) 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、ハイパーバイザレイヤー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。

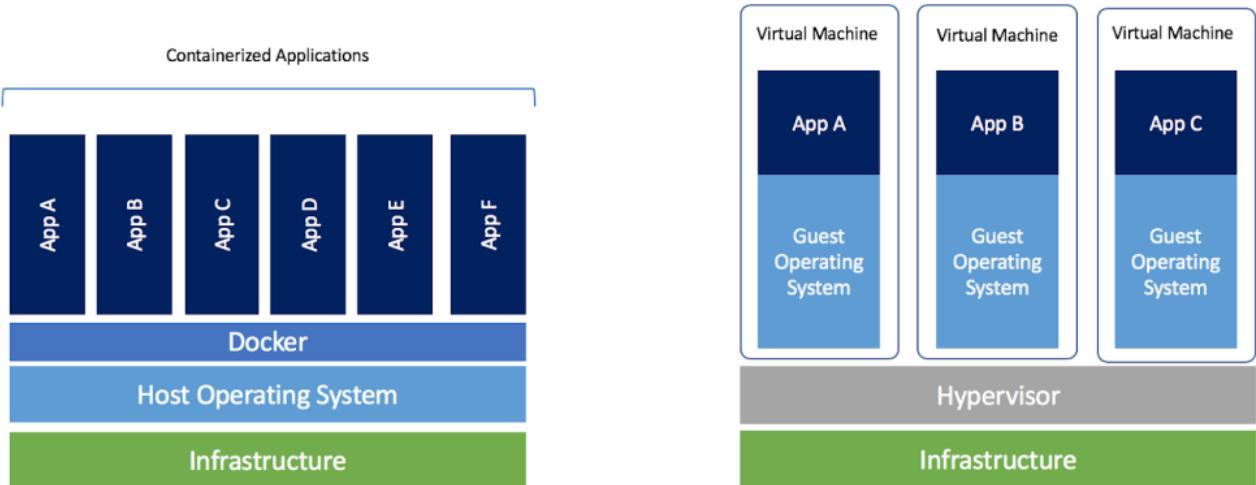


Figure 39. Docker (左) と VM (右) の比較 (画像出典: <https://www.docker.com/blog/containers-replacing-virtual-machines/>)

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. たとえば, プロジェクトごとに Docker イメージを作成することで, どの OS・コンピュータ 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](#) (Continuous Integration / Continuous Deployment) 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. さらにはサーバレスコンピューティング ([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, これから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 15.6](#) 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. Docker 用語集

Docker を使い始めるに当たり, 最初に主要な用語を解説しよう. 次のパラグラフで太字で強調された用語を頭に入れた上で, 続くチュートリアルに取り組んでいただきたい.

Docker を起動する際の大まかなステップを示したのが [Figure 40](#) である. パッケージされた一つの計算環境のことを **イメージ (Image)** とよぶ. イメージは, Docker Hub などのリポジトリで配布されているものをダウンロードするか, 自分でカスタムのイメージを作成することも可能である. イメージを作成するための"レシピ"を記述したファイルが **Dockerfile** である. Dockerfile からイメージを作成する操作を **build** とよぶ. イメージがホストマシンのメモリにロードされ, 起動状態にある計算環境のことを **コンテナ (Container)** とよぶ. Container を起動するために使用されるコマンドが **run** である.

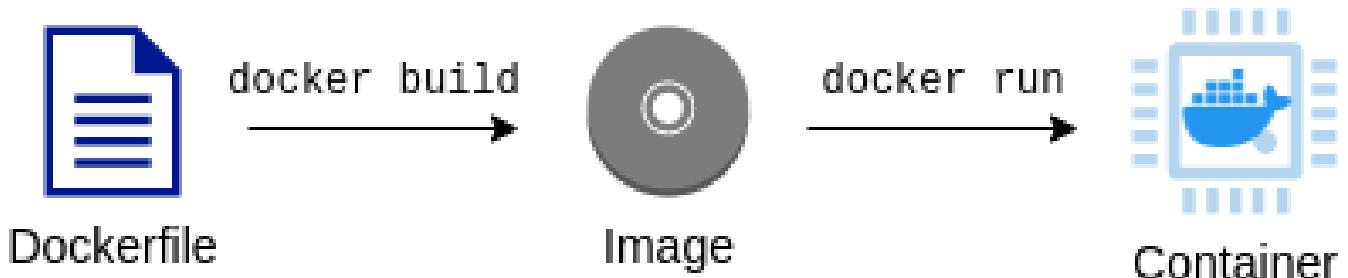


Figure 40. Image と Container

7.3.2. イメージをダウンロード

パッケージ化された Docker の仮想環境 (= **イメージ (Image)**) 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 イメージが集められており, GitHub などと同じ感覚で, オープンな形で公開されている。

たとえば, Ubuntu のイメージは [Ubuntu の公式リポジトリ](#) 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 イメージを公開するためのデータベース (レジストリ (registry) とよぶ) は Docker Hub だけではなく, たとえば GitLab や GitHub は独自のレジストリ機能を提供しているし, 個人のサーバーでレジストリを立ち上げることも可能である。Docker Hub 以外のレジストリから **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レジストリのアドレス (さらにオプションとしてポート番号) を指定する。

7.3.3. コンテナを起動

Pullしてきたイメージを起動するには, **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。

こ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 41)。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)** とよぶ。

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

Figure 41. Docker を使って ubuntu:18.04 イメージを起動

ここで使用した `ubuntu:18.04` のイメージは, 空の Ubuntu OS だが, すで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イメージは [PyTorch 公式の Docker Hub リポジトリ](#) 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イメージが見つからない場合は, 自動的に Docker Hub からダウンロードされる。

pytorch のコンテナが起動したら, 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。

7.3.4. 自分だけのイメージを作る

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、自分だけのイメージを作ることも可能である。

たとえば、[本書のハンズオン実行用に提供している docker イメージ](#)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 イメージを作るには、[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か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 イメージ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](#) の中身の説明は詳しくは行わないが、たとえば上のコードで <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 イメージを作成することができる。一度イメージを作成すれば、それを配布することで、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、ほか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。たとえば、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・エコシステムをもっているが、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、その名前にあるとおり pod とよ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 42. ECS のアイコン

ここまでに説明してきたように、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 (Figure 42)。ECS を使用することで、Docker にパッケージされたアプリケーションを計算機クラスターに投入したり、計算機クラスターのインスタンスを追加・削除する操作 (=スケーリング) を行うことができる。

ECS の概要を示したのが Figure 43 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、Docker Hub 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。あらかじ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（たとえば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）とよび、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）とよぶ。クラスターのスケーリングは、ECSがほかの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してしまえば、ユーザーは（ほとんどなにも考えずに）大量のタスク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タスクの量にちょうど十分なだけのインスタンスが起動し、タスクが完了した後は不要なインスタンスはすべて停止される。

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！

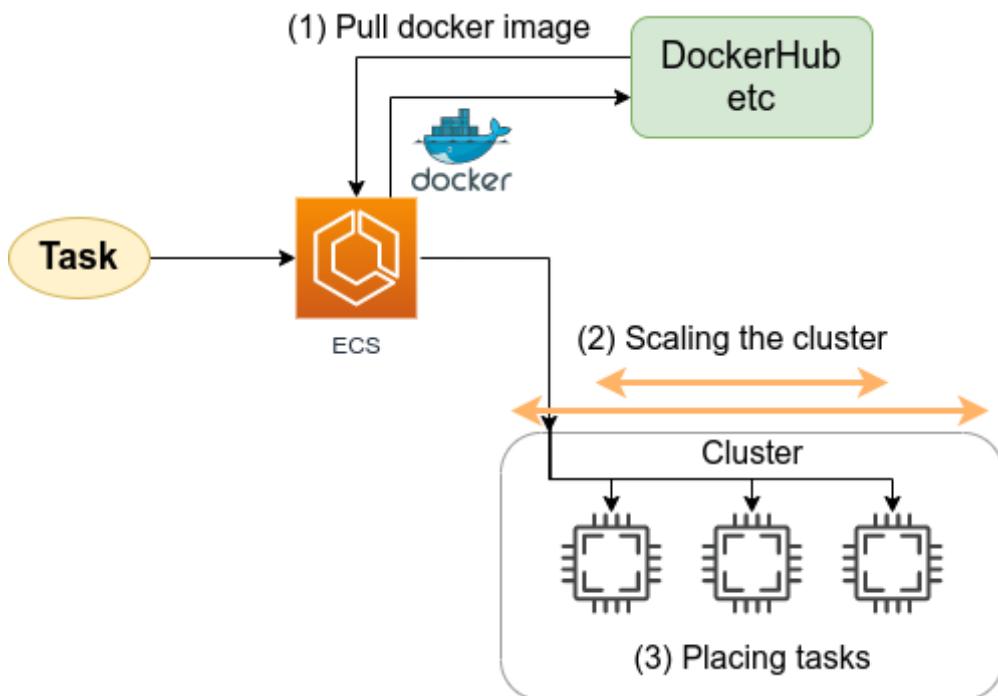


Figure 43. ECS の概要

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、Docker と ECS を駆使した機械学習アプリケーションを実装しよう。具体的には、深層学習による自然言語処理を行うことで、クライアントから与えられた文章題に対して回答を生成する、自動 Question & Answering ボットを作成しよう。ECS を利用することで、ジョブの数によって動的にインスタンスの数を制御し、並列にタスクを実行するようなシステムを構築しよう。



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 搭載型の EC2 クラスター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 Fargate クラスターを用いた推論計算の並列化を紹介する。

8.1. Fargate

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（Figure 44）。



Figure 44. Fargate のアイコン

ECS の概要を示した Figure 43 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 を用いた計算機クラスターの作成・管理は技術的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もたない。物理的実体をもたないというのは、たとえば 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。Fargate ではすべての計算は Docker コンテナを介して行われる。すなわち、Fargate を利用するには、ユーザーは最初に所望の Docker イメージを指定しておき、Fargate は `docker run` のコマンドを使用することで計算タスクを実行する。Fargate を用いる利点は、Fargate を ECS のクラスターに指定すると、スケーリングなどの操作が簡単な設定・プログラムで構築できる点である。

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（詳しくは [公式ドキュメンテーション "Amazon ECS on AWS Fargate"](#) 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 容量を単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。

8.2. 準備

ハンズオンのソースコードは GitHub の [handson/qa-bot](#) にある。

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。また、Docker が自身のローカルマシンにインストール済みであることも必要である。



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。

8.3. 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るものとする。上の問題では、次のような回答を返すべきである。

answer: 1921

人間にとっては、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解かせるのは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** とよ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 45)。このプログラムを、Docker にパッケージしたものが [著者の Docker Hub リポジトリ](#) 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。

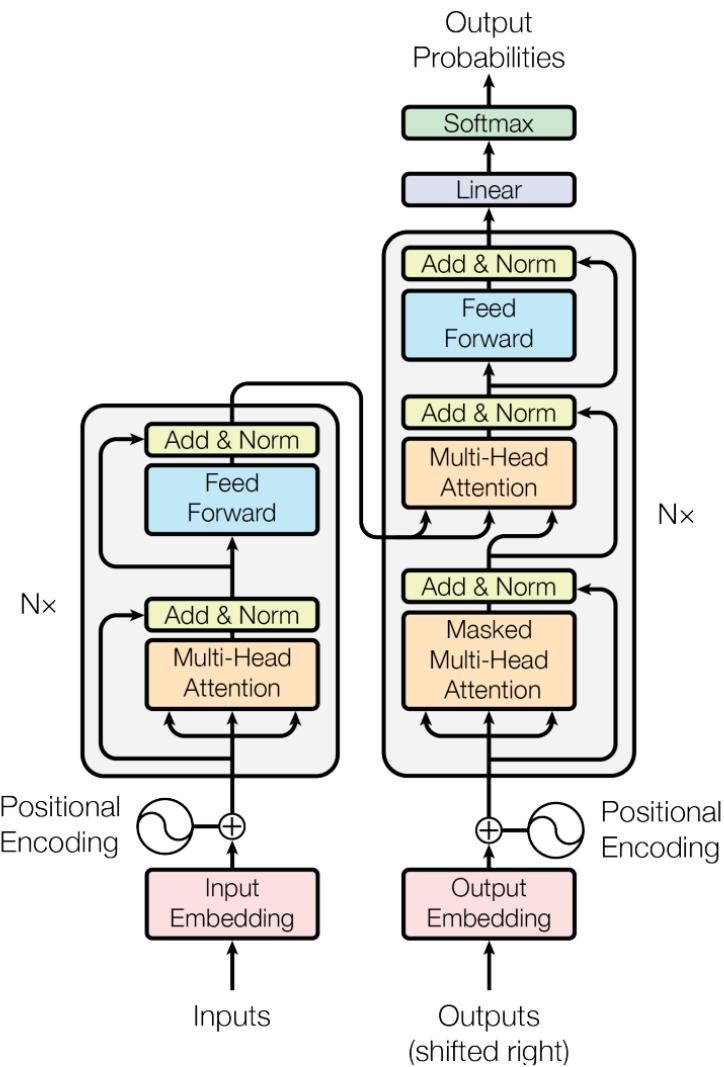


Figure 45. Transformer モデルアーキテクチャ (画像出典: [Vaswani+ 2017](#))



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るので、コストの削減と、実装をシンプルにする目的で、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。

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。まずは context と question をコマンドラインの変数として定義する。

```
$ 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 = 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 ?"
```

そうしたら、次のコマンドによってコンテナを実行する。

```
$ 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, いまは気にしなくてよい.

このコマンドを実行すると, 次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いてい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, 深層学習に支えられた言語モデルを用いることで, 実用にも役に立ちそうな 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](#) による DistilBert の実装のドキュメンテーションは [公式ドキュメンテーション](#) を参照のこと.



今回提供する Q-A ボットの Docker のソースコードは <https://github.com/tomomano/learn-aws-by-coding/blob/main/handson/qa-bot/docker/Dockerfile> 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 46](#) に示す.

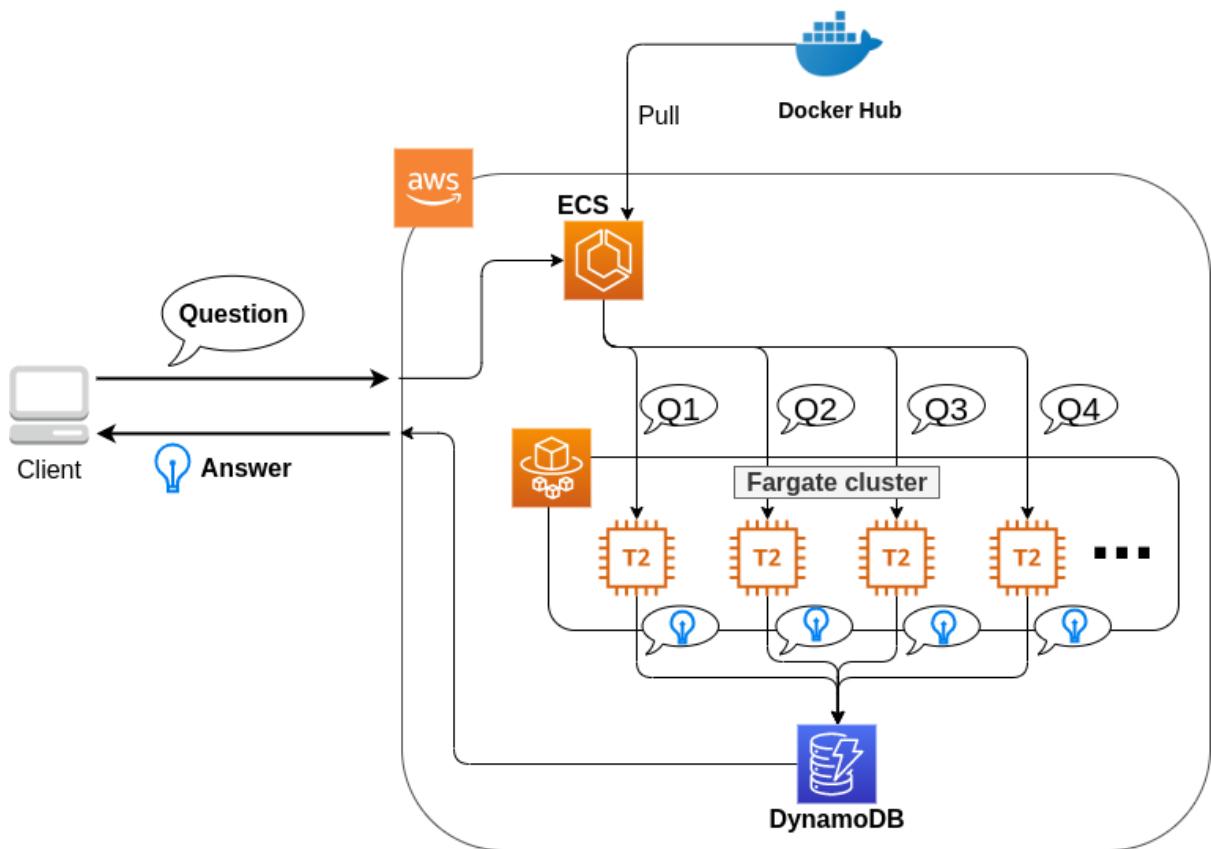


Figure 46. アプリケーションのアーキテクチャ

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、Docker Hub から、イメージをダウンロードする。
- ・ 次に、ECS はクラスター内に新たな Fargate インスタンスを立ち上げ、ダウンロードされた Docker イメージをこの新規インスタンスに配置する。
 - このとき、一つの質問に対し一つの Fargate 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/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                 "tomomano/qabot: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、サーバーレスアーキテクチャの章で扱うので、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 イメージを定義している。

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         "tomomano/qabot: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、とくにここでは 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、Docker Hub 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、これだけで前述したような、タスクのスケジューリングなどが自動で実行される。

このコードで `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。たとえば、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 "Amazon ECS on AWS Fargate"](#) 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。シークレットキーの設定も忘れずに(Section 15.3)。

```
# プロジェクトのディレクトリに移動
$ cd handson/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

# 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 47 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/EcsClusterQaBotTaskDef

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

Figure 47. 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の中身を確認してみよう。コンソールから、ECS のページに行くと Figure 48 のような画面が表示されるはずである。EcsClusterQaBot-XXXXという名前ついたクラスターを見つけよう。

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。Figure 48 で、FARGATE という文字の下に 0 Running tasks, 0 Pending tasks と表示されていることを確認しよう。この時点では一つもタスクが走っていないので、数字はすべて0になっている。

Figure 48. ECS コンソール画面

続いて、この画面の左のメニューバーから **Task Definitions** という項目を見つけ、クリックしよう。移動した先のページで **EcsClusterQaBot-EcsClusterQaBotTaskDefXXXX** という項目が見つかるので、開く。開いた先のページをスクロールすると Figure 49 に示したような情報が見つかるだろう。使用する CPU・メモリーの量や、Docker container の実行に関する設定などが、この Task Definition の画面から確認することができる。

Container ...	Image	CP...	GP...	Infe...	Hard/Soft memory limits (MiB)	Ess...
EcsClu...	tomomano/qabot:latest	0	--	--	--/--	true

Figure 49. Task definition の確認

8.6. タスクの実行

それでは、質問をデプロイしたクラウドに提出してみよう。

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 ([hands-on/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, コマンドラインで 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 (question) を引数として渡している.

こ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 イメージ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 50](#)). すると, 実行中のタスクの一覧が表示されるだろう.

	Task	Last status	Description	Task de...	Revision	Starte...	Started at
<input type="checkbox"/>	aad78e...	Pending	-	EcsClusterQaBo	8	-	-
<input type="checkbox"/>	b1a869...	Stopped	CannotPullCon...	EcsClusterQaBo	6	-	-

Figure 50. ECS のタスクの実行状況をモニタリング

[Figure 50](#) で見て取れるように, "Last status = Pending" となっていることから, この時点では, タスクを実行する準備を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 50](#) 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 51](#)). "Last status", "Platform version" 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, コンテナの吐き出した実行ログを閲覧することができる.

48ce33531af149bd8747064ae830535e

[Stop](#)[Configuration](#)[Logs](#)[Networking](#)[Tags](#)

Status

Last status
⌚ Deprovisioning

Desired status
⊖ Stopped

Stopped reason
 Essential container in task exited

Started at
 2021-06-29T15:30:46.572Z

Created at
 2021-06-29T15:28:20.534Z

Configuration

Platform version
 1.4.0

Task definition
[EcsClusterQaBotEcsClusterQaBot](#)
 tTaskDef54F4C2A5: 8

CPU
 1 vCPU

Memory
 4 GB

Containers (1)

Container name	▲	Image URI	▼	Status	▼	CPU	▼	Memory hard/soft limit	▼
EcsClusterQaBot-Container		tomomano/qabot:latest		⊖ Stopped		-		- / -	

Figure 51. 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 52 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@ammonite: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/EcsClusterQaBot-EcsClusterQaBotCluster6488E31F-txu0J9uYkRhf/48ce33531af149bd8747064ae830535e  

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: 0.9704799652099609
```

Figure 52. 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/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 53). 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。

The screenshot shows the AWS ECS console with the 'Tasks' tab selected. At the top, there are buttons for 'Stop' and 'Run new task'. Below is a search bar and a pagination area showing page 1 of 2. A table lists 12 tasks:

Task	Last started at	Description	Task definition	Revision	Started at	Started at
1fed63...	Pending	-	EcsClusterQaBotEcsC	8	-	-
34fece...	Running	-	EcsClusterQaBotEcsC	8	-	7 seconds ago
aa2c82...	Running	-	EcsClusterQaBotEcsC	8	-	4 seconds ago
eefec8...	Running	-	EcsClusterQaBotEcsC	8	-	8 seconds ago
2005a...	Deprovisioning	Essential container	EcsClusterQaBotEcsC	8	-	25 seconds ago
48ce33...	Stopped	Essential container	EcsClusterQaBotEcsC	8	-	9 minutes ago
53ca8...	Deprovisioning	Essential container	EcsClusterQaBotEcsC	8	-	23 seconds ago
60333...	Deprovisioning	Essential container	EcsClusterQaBotEcsC	8	-	22 seconds ago
aad78...	Stopped	Essential container	EcsClusterQaBotEcsC	8	-	12 minutes ago
b1a86...	Stopped	CannotPullContainerError	EcsClusterQaBotEcsC	6	-	-

Figure 53. 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. それには,次のコマンドを実行する.

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

結果として, Figure 54 のような出力が得られるだろう. 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: Никола Тесла; 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: Normands; 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: Normands; 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 54. \$ 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、深層学習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もったシステムである! 今回は GUI (Graphical User Interface) を用意することはしなかったが、このシステムに簡単な GUI を追加すればなかなか立派なウェブサービスとして運用できるだろう。

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、今回の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、前回までと同様に、AWS コンソールにログインし CloudFormation の画面から DELETE ボタンをクリックするか、コマンドラインからコマンドを実行する。コマンドラインから行う場合は、次のコマンドを使用する。

```
$ cdk destroy
```

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、すで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、一般的に言って、機械学習のワークフローで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。

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、勾配下降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、少しず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。研究・開発においては、スループットよくたくさんのモデルの可能性を探索することが生産性を決める重要なファクターであり、ハイパープラメータ探索を高速に解くという問題は極めて関心が高い。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て並列的にニューラルネットの訓練を実行することで、この問題を解く方法を学んでいこう。

9.1. Auto scaling groups (ASG)

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

ECS の概要を示した [Figure 43](#) 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。Fargate を用いると、自在にスケールする計算環境をとても簡単な設定で構築することができた。しかし、GPU を利用することができないなど、いくつかの制約があった。EC2 を使用した計算環境を指定することで、プログラミングの複雑度は増すが、GPU やその他のより高度かつ複雑な設定を伴ったクラスターを構築することができる。

EC2 クラスターには **ASG** と呼ばれるサービスが配置される。ASG は複数の EC2 インスタンスをロジカルな単位でグループ化することでクラスターを構成する。ASG はクラスター内に新しいインスタンスを起動する、あるいは不要になったインスタンスを停止するなどのスケーリングを担う。ASG で重要な概念として、**desired capacity**, **minimum capacity**, **maximum capacity** というパラメータがある。**minimum capacity**, **maximum capacity** は、それぞれクラスター内に配置できるインスタンスの数の最小値・最大値を指定するパラメータである。前者は、クラスターに負荷がかかっていない場合でもアイドリング状態にあるインスタンスを維持することで、急に負荷が増大した時などのバッファーとして作用することができる。後者は、負荷が急に増えたときに、過剰な数のインスタンスが起動する事態を防ぎ、経済的なコストの上限を定める役割を果たす。

desired capacity が、その時々でシステムが要求するインスタンスの数を指定する。**desired capacity** は、例えば24時間のリズムに合わせてインスタンスの数を増減させる(昼は多く夜は少なくなど)などの決まったスケジュールに基づいた設定を適用することができる。あるいはクラスター全体にかかる負荷に応じて、**desired capacity** を動的に制御することも可能である。どのような基準でクラスターのスケーリングを行うかを定めるルールのことを、**スケーリングポリシー** とよぶ。たとえば、クラスター全体の稼働率(負荷)を常に 80% に維持する、などのスケーリングポリシーが想定できる。この場合、クラスター全体の負荷が 80% を下回ったときにはクラスターからインスタンスが削除され、80% を超える(あるいは超えると予測される) 場合はインスタンスを追加する、という操作が ASG によって自動的に行われる。

上記のようなパラメータを検討し、ユーザーは ASG を作成する。ASG を作成したのち、ECS との連携をプログラムしてあげることで、ECS を介して ASG による EC2 クラスターにタスクを投入することが可能になる。

9.2. AWS Batch



Figure 55. AWS Batch のアイコン

先に説明したように、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 はその名のとおりバッチ (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。たとえば、初期値のパラメータを変えて複数の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舞台裏の詳細を気にすることなく、大量のジョブを投入できるシステムが手に入る点である。が、知識として背後では ECS/ASG/EC2 の三つ巴が協調して動作しているという点は知っておいてほしい。

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 56)。まず、**ジョブ (Job)** というのが、AWS Batch によって実行される一つの計算の単位である。**Job definitions** とはジョブ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。Job definition に基づいて個々のジョブが実行される。ジョブが実行されると、ジョブは **Job queues** に入る。Job queues とは、実行待ち状態にあるジョブの列のことであり、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 に溜まったジョブが優先的に実行される（筆者はこれをディズニーランドの“ファストパス”を連想して捉えている）。**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、それに応じてジョブを Compute environment に投下する。

以上が AWS Batch を使用するうえで理解しておかなければならない概念であるが、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。

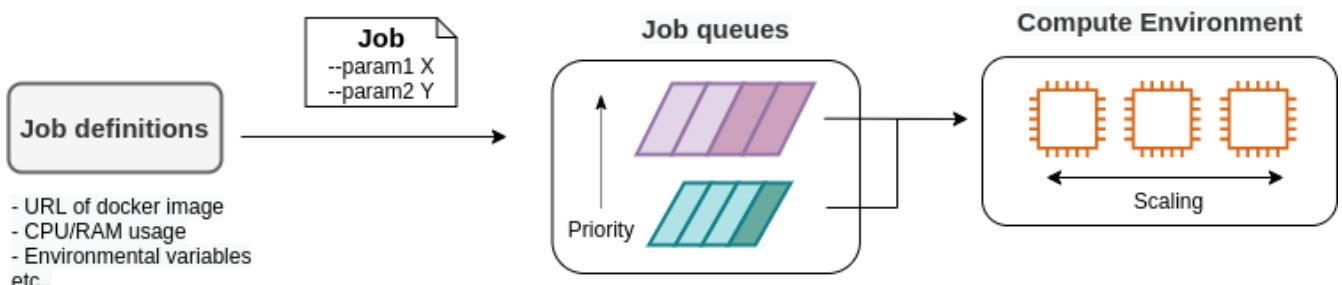


Figure 56. AWS Batch の主要な概念

EC2 or Fargate?

ECS でクラスターを構成する際, 計算を実行する場として EC2 と Fargate の二つの選択肢があることを説明した. それぞれ長所と短所を抱えているのだが, どのような場合にどちらを使うべきだろうか? それを検討するため, まずは [Table 5](#) を見てみよう. これは EC2 と Fargate の特徴をまとめたものである. 説明の都合上, 大幅な粗視化が行われている点は留意していただきたい.

Table 5. EC2 vs Fargate

	EC2	Fargate
Compute capacity	Medium to large	Small to medium
GPU	Yes	No
Launch speed	Slow	Fast
Task placement flexibility	Low	High
Programming complexity	High	Low



これまでに見てきたように, EC2 は最大の CPU 数・メモリーサイズが大きかったり, GPU を利用できたりするなど, 単一のインスタンスでの計算能力は高い. 対して, Fargate は単一インスタンスの最大 CPU 数は 4コアが上限である. その一方で, インスタンスの起動に要する時間は Fargate のほうが圧倒的に早く, より俊敏にクラスターのスケーリングを行うことができる. また, タスクをクラスターに投入する際のフレキシビリティも Fargate のほうが高い. フレキシビリティというのは, 例えば一つのインスタンスで 2つ以上のコンテナを走らせる, などの状況である. 単位 CPUあたりで処理されるタスクの数を最大化する際には, このような設計がしばしば採用される. プログラミングの複雑さという観点からは, Fargate のほうが一般的にシンプルな実装になる.

このように, EC2 と Fargate は互いに相補的な特性を有しており, アプリケーションによって最適な計算環境は検討される必要がある. また, EC2 と Fargate を両方用いたハイブリッドクラスターというのも定義可能であり, そのような選択肢もしばしば用いられる.

9.3. 準備

ハンズオンのソースコードは GitHub の [handson/aws-batch](#) にある.

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. また, Docker が自身のローカルマシンにインストール済みであることも必要である.



このハンズオンは, `g4dn.xlarge` タイプの EC2 インスタンスを使うので, アメリカ東部 (`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.



[Section 6.1](#) でも注意したが, このハンズオンを始める前に G タイプインスタンスの起動上限を AWS コンソールの EC2 管理画面から確認しよう. もし上限が 0 になっていた場合は, 上限緩和の申請を行う必要がある. [Section 9.5](#) にも関連した情報を記載しているので, 併せて参照されたい.

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 6.7](#) 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 6.7](#) 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, 次の行が該当す

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、ランダムサーチ、そしてベイズ最適化による方法である。

グリッドサーチとは、ハイパーパラメータの組が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が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、理論的にはグリッドサーチやランダムサーチ法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 な問題とよぶ（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ので便利である。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。

まずは、本ハンズオンで使用する Docker イメージをローカルで実行してみよう。

Docker イメージのソースコードは [handson/aws-batch/docker](#) にある。基本的に Section 6.7 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。

練習として、この Docker イメージを手元でビルドするところからはじめてみよう。`Dockerfile` が保存されているディレクトリに移動し、`mymnist` という名前 (Tag) をつけてビルドを実行する。

```
$ cd handson/aws-batch/docker  
$ docker build -t mymnist .
```



`docker build` でエラーが出たときは次の可能性を疑ってほしい。ビルトの中で、MNIST の画像データセットを <http://yann.lecun.com/exdb/mnist/> からダウンロードするのだが、ダウンロード先のサーバーがしばしばダウンしている。世界中の機械学習ユーザーがアクセスするので、これはしばしば発生するようである。サーバーがダウンしているとビルトも失敗してしまう。エラーメッセージにそれらしい文言が含まれていたら、この可能性を疑おう。

手元でビルトするかわりに、Docker Hub から pull することも可能である。その場合は次のコマンドを実行する。



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

イメージの準備ができたら、次のコマンドでコンテナを起動し、MNIST の学習を実行する..

```
$ docker run mymnist --lr 0.1 --momentum 0.5 --epochs 10
```

このコマンドを実行すると、指定したハイパーパラメータ (`--lr` で与えられる学習率と `--momentum` で与えられるモメンタム) 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 57)。

```
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 57. 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、次のコマンドにより GPU を使って計算を実行できる。

```
$ docker run --gpus all mymnist --lr 0.1 --momentum 0.5 --epochs 10
```

このコマンドでは、`--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 58 のようになるはずである。

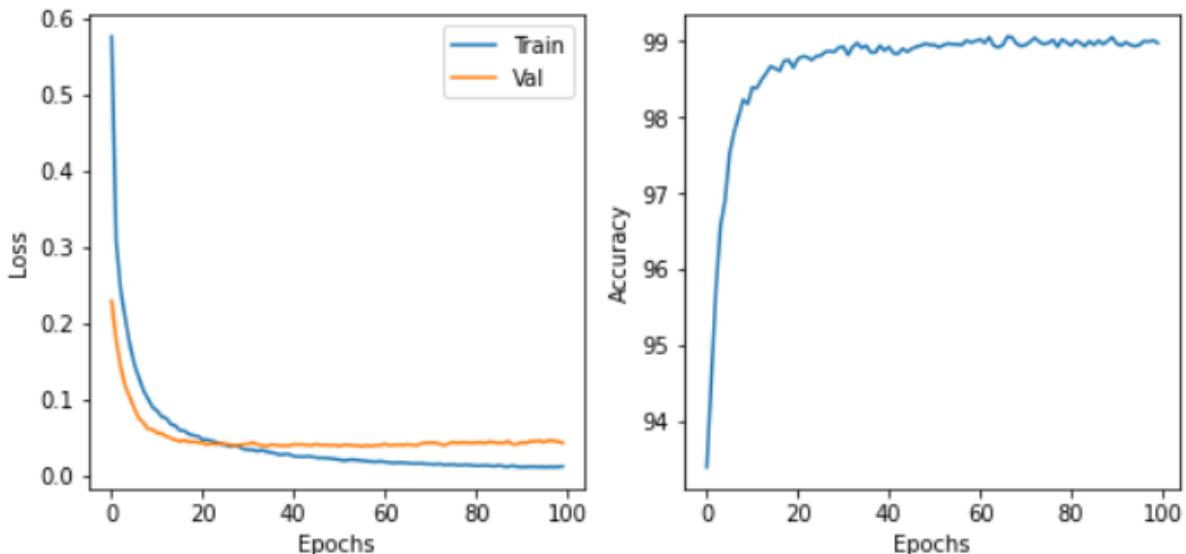


Figure 58. (左) Train/Validation データそれぞれの Loss のエポックごとの変化. (右) Validation データの Accuracy のエポックごとの変化

これはオーバーフィッティングとよばれる現象で、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** とよばれるテクニックが知られている。Early stopping とは、検証データの 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。



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。

9.5. 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 59 に示す。

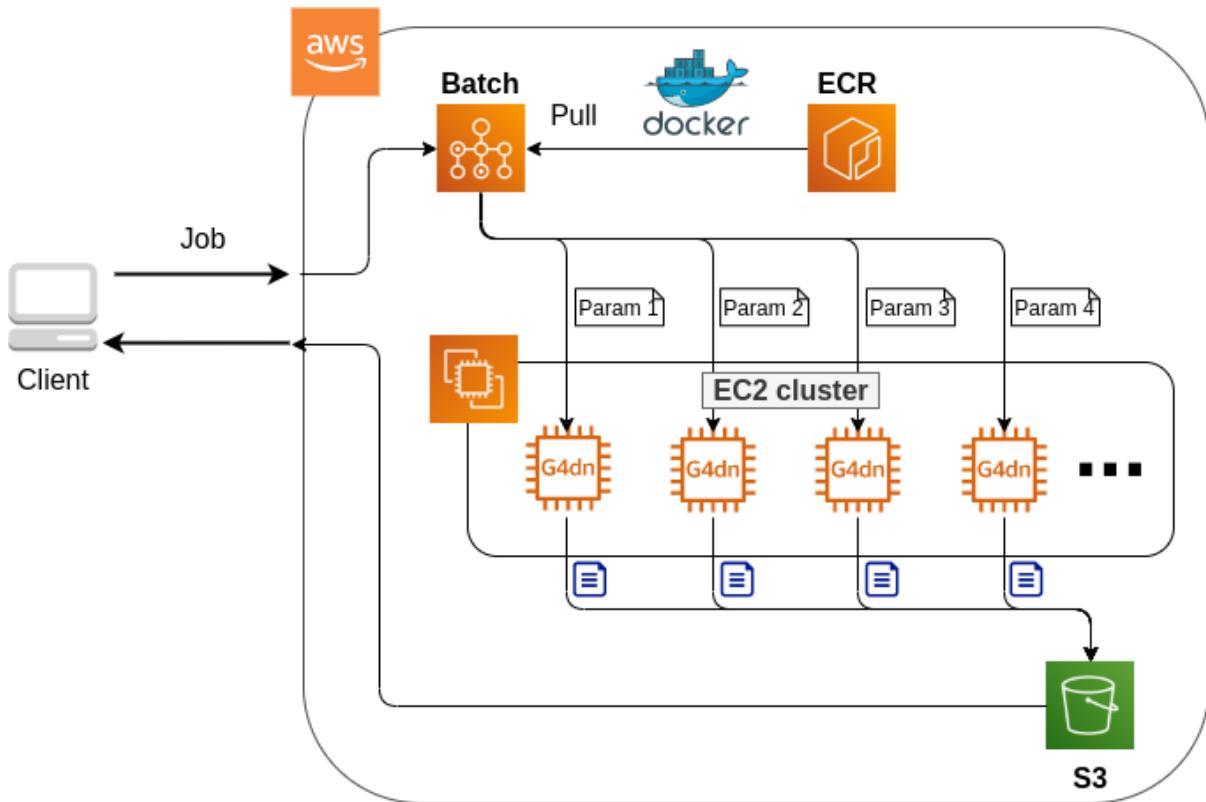


Figure 59. アプリケーションのアーキテクチャ

簡単にまとめると、次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 イメージは、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         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            # other parameters...
16        )
17
18        ②
19        managed_env = batch.ComputeEnvironment(

```

```

20         self, "managed-env",
21         compute_resources=batch.ComputeResources(
22             vpc=vpc,
23             allocation_strategy=batch.AllocationStrategy.BEST_FIT,
24             desiredv_cpus=0,
25             maxv_cpus=64,
26             minv_cpus=0,
27             instance_types=[
28                 ec2.InstanceType("g4dn.xlarge")
29             ],
30         ),
31         managed=True,
32         compute_environment_name=self.stack_name + "compute-env"
33     )
34
35     ③
36     job_queue = batch.JobQueue(
37         self, "job-queue",
38         compute_environments=[
39             batch.JobQueueComputeEnvironment(
40                 compute_environment=managed_env,
41                 order=100
42             )
43         ],
44         job_queue_name=self.stack_name + "job-queue"
45     )
46
47     ④
48     job_role = iam.Role(
49         self, "job-role",
50         assumed_by=iam.CompositePrincipal(
51             iam.ServicePrincipal("ecs-tasks.amazonaws.com")
52         )
53     )
54     # allow read and write access to S3 bucket
55     bucket.grant_read_write(job_role)
56
57     ⑤
58     repo = ecr.Repository(
59         self, "repository",
60         removal_policy=core.RemovalPolicy.DESTROY,
61     )
62
63     ⑥
64     job_def = batch.JobDefinition(
65         self, "job-definition",
66         container=batch.JobDefinitionContainer(
67             image=ecs.ContainerImage.from_ecr_repository(repo),
68             command=["python3", "main.py"],
69             vcpus=4,
70             gpu_count=1,
71             memory_limit_mib=12000,
72             job_role=job_role,
73             environment={
74                 "BUCKET_NAME": bucket.bucket_name
75             }
76         ),
77         job_definition_name=self.stack_name + "job-definition",
78         timeout=core.Duration.hours(2),
79     )

```

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。また、最小の vCPU は 0 である。今回は、負荷がかからないときにアイドリング状態にあるインスタンスを用意する利点は全くないので、ここは 0 にするのが望ましい。
- 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、ジョブが計算結果を 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。(IAM とはリソースがもつ権限を管理する仕組みである。詳しくは [Section 13.2.5](#) を参照)
- 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 1 台あたり 4 vCPU が割り当てられている。このプログラムでは Compute environment の maximum vCPUs を 64 と指定しているので、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 maximum vCPUs を 64 に限定しているのは、なんらかのミスで意図せぬジョブを大量にクラスターに投入してしまった事態で、高額の AWS 利用料金が発生するのを防ぐためである。もし、自分のアプリケーションで必要と判断したならば自己責任において 64 よりも大きな数を設定して構わない。

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 60)。**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 "Amazon EC2 service quotas"](#) を参照のこと。

The screenshot shows the AWS EC2 Limits page. The left sidebar has a 'New EC2 Experience' button and a 'Limits' section under 'Instances'. The main area is titled 'Limits' with a search bar and a 'Find limits' button. It lists various resource limits with columns for Name, Limit type, and Current limit. The listed items include Launch configurations, Auto Scaling groups, Running M5N Dedicated Hosts, Running M6G Dedicated Hosts, Running C3 Dedicated Hosts, Running C4 Dedicated Hosts, Running C5 Dedicated Hosts, and Running M6GD Dedicated Hosts. Most limits are set to 200 or 2, while some like Dedicated Hosts are set to 1.

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

Figure 60. EC2 コンソールから各種の上限を確認する

9.6. 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)。シークレットキーの設定も忘れずに ([Section 15.3](#))。

```

# 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

# 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 61)。

The screenshot shows the AWS Batch Dashboard with the following sections:

- Jobs overview:** Displays counts for four job states: RUNNABLE (0), RUNNING (0), SUCCEEDED (0), and FAILED (0).
- Job queue overview:** Shows two job queues: "SimpleBatchjob-queue" and another partially visible queue. For each, it lists counts for various states: SUBMITTED, PENDING, RUNNABLE, STARTING, RUNNING, SUCCEEDED, and FAILED.
- Compute environment overview:** Lists two compute environments: one with a partially visible name and "SimpleBatchcompute-env". Each entry includes columns for Name, Type, Provisioning model, Instance types, Status, State, Minimum vCPUs, Desired vCPUs, and Maximum vCPUs.

Figure 61. AWS Batch 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ててもよい）である。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、**Minimum vCPUs** が 0、**Maximum vCPUs** が 64 と設定されていることも見て取れる。加えて、この時点では一つもジョブが走っていないので、**Desired vCPUs** は 0 になっている。より詳細な Compute environment の情報を閲覧したい場合は、名前をクリックすることで詳細画面が開く。

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。**PENDING**, **RUNNING**, **SUCCEEDED**, **FAILED** などのカラムがあることが確認できる。ジョブが進行するにつれて、ジョブの状態がこのカラムにしたがって遷移していく。後でジョブを実際にサブミットしたときに戻ってこよう。

最後に、今回作成した Job definition を確認しよう。左側のメニューから **Job definitions** を選択し、次の画面で **SimpleBatchjob-definition** という項目を見つけて開く。ここから Job definition の詳細を閲覧することができる (Figure 62)。中でも重要な情報としては、**vCPUs**, **Memory**, **GPU** がそれぞれ Docker に割り当てられる

vCPU・メモリー・GPU の量を規定している。また、Image と書いてあるところに、ジョブで使用される Docker イメージが指定されている。ここでは、ECR のレポジトリを参照している。現時点ではこの ECR は空である。次のステップとして、この ECR にイメージを配置する作業を行おう。

The screenshot shows the 'Container properties' section of the AWS Batch job definition configuration. It includes fields for Command (["python3", "main.py"]), Image (redacted), vCpus (4), User (--), Read only filesystem (--), Memory (12000), Instance type (1), and Number of GPUs (1).

Figure 62. AWS Batch から Job definition を確認

9.7. 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 イメージをダウンロード (pull) してくる必要がある。前回のハンズオン (Chapter 8) では、公開設定にてある Docker Hub からイメージを 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レジストリである **ECR (Elastic Container Registry)** に image を配置するという設計を採用する。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イメージの置き場所を用意できる点である。Batch は ECR からイメージを pull することで、タスクを実行する (Figure 59)。

スタックのソースコードでいうと、次の箇所が 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、イメージを <1> で作った ECR から取得するように指定している。これとともに、Job definition には ECR へのアクセス権限が IAM を通じて自動的に付与される。

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 イメージを 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 63)。

The screenshot shows the AWS ECR console interface. At the top, there's a breadcrumb navigation: ECR > Repositories. Below it, a filter bar has 'Private' selected. The main area displays a table titled 'Private repositories (2)'. The table has columns: Repository name, URI, Created at, Tag immutability, Scan on push, and Encryption type. The first repository row is collapsed, showing a small icon and a truncated URI. The second repository row is expanded, showing the full URI: `https://REDACTED.dkr.ecr.REDACTED.amazonaws.com/simplebatch-repository9f1a3f0bzfc...`, and its details: Jun 07, 2021 10:24:14 PM, Disabled, Disabled, AES-256.

Private repositories (2)					
Repository name	URI	Created at	Tag immutability	Scan on push	Encryption type
<input checked="" type="checkbox"/> simplebatch-repository9f1a3f0bzfc...	https://REDACTED.dkr.ecr.REDACTED.amazonaws.com/simplebatch-repository9f1a3f0bzfc...	Jun 07, 2021 10:24:14 PM	Disabled	Disabled	AES-256
<input type="radio"/> simplebatch-repository9f1a3f0bzfc...	https://REDACTED.dkr.ecr.REDACTED.amazonaws.com/simplebatch-repository9f1a3f0bzfc...	Jun 07, 2021 10:24:14 PM	Disabled	Disabled	AES-256

Figure 63. 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 64 のようなポップアップ画面が立ち上がる.

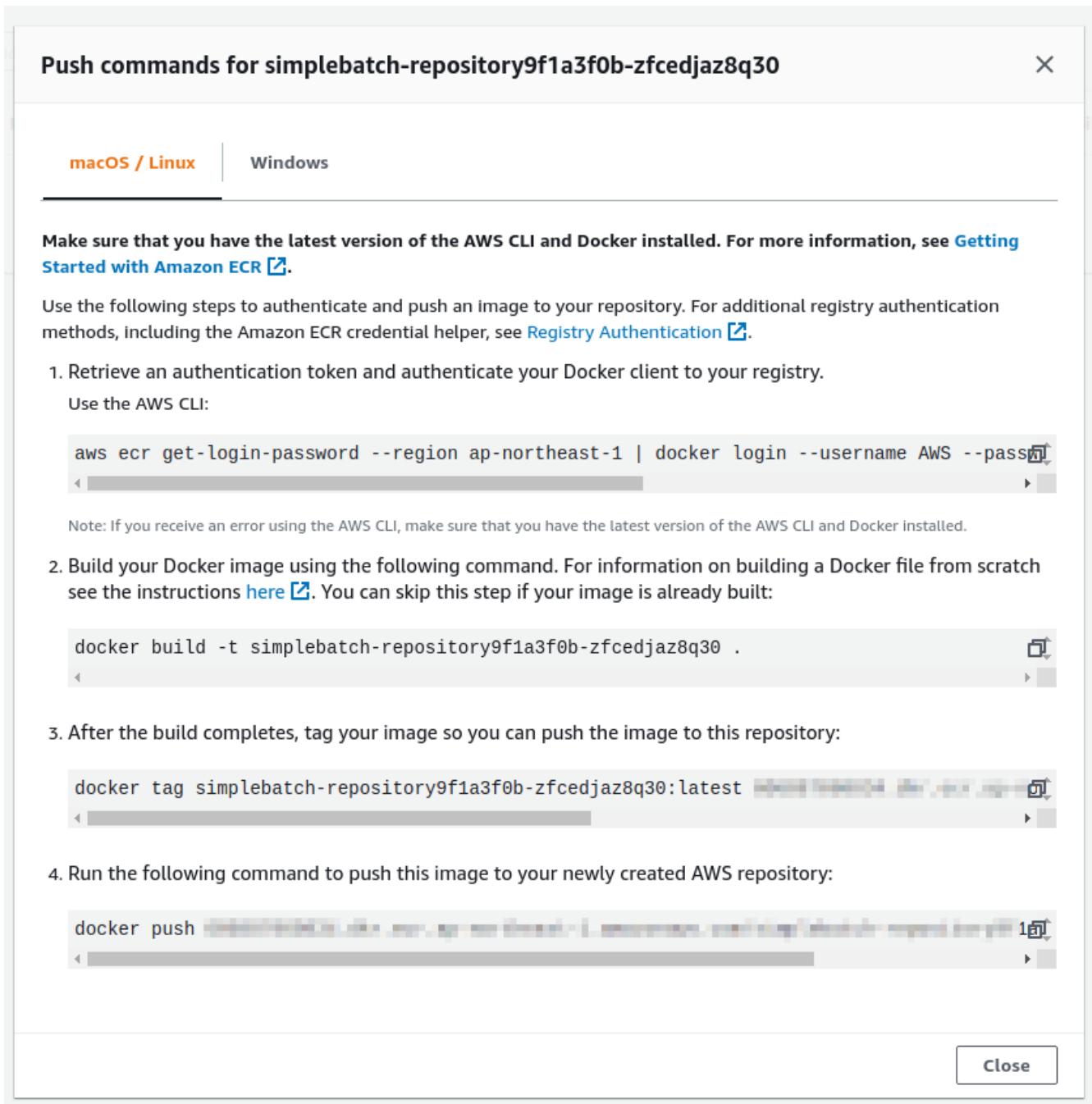


Figure 64. ECR への push コマンド

このポップアップ画面で表示されている四つのコマンドを順番に実行していくことで、手元の Docker イメージを 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。そのうえで、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 を使ってイメージ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。

四つ目のコマンドは、数GBあるイメージを ECR にアップロードするので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イメージが 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、確かにイメージが配置されていることが確認できる (Figure 65)。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。

simplebatch-repository9f1a3f0b-zfcfedjaz8q30

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

Images (1)

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

< 1 >

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

Figure 65. ECR へ image の配置が完了した

9.8. 単一のジョブを実行する

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, ローカルマシンから以下のコマンドで 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], [2], [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 # [1]
2 import boto3
3 import argparse
4
5 # [2]
6 # AWS 認証ヘルパー ...省略...
7
8 # [3]
9 def submit_job(lr:float, momentum:float, epochs:int, profile_name="default"):
10     if profile_name is None:
11         session = boto3.Session()
12     else:
13         session = boto3.Session(profile_name=profile_name)
14     client = session.client("batch")
15
16     title = "lr" + str(lr).replace(".", "") + "_m" + str(momentum).replace(".", "")
17     resp = client.submit_job(
18         jobName=title,
19         jobQueue="SimpleBatchjob-queue",
20         jobDefinition="SimpleBatchjob-definition",
21         containerOverrides={
22             "command": [
23                 "--lr", str(lr),
24                 "--momentum", str(momentum),
25                 "--epochs", str(epochs),
26                 "--uploadS3", "true"
27             ]
28         }
29     )
30     print("Job submitted!")
31     print("job name", resp["jobName"], "job ID", resp["jobId"])

```

`submit_job()` 関数について簡単に説明しよう。Section 9.4 で、MNIST の Docker をローカルで実行したとき、次のようなコマンドを使用した。

```
$ docker run -it mymnist --lr 0.1 --momentum 0.5 --epochs 10
```

ここで、`--lr 0.1 --momentum 0.5 --epochs 10` の部分が、コンテナ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、`ContainerOverrides` の `command` というパラメータを使用することで、コンテナに渡されるコマンドを指定することができる。コードでは以下の部分が該当する。

```

1 containerOverrides={
2     "command": [
3         "--lr", str(lr),
4         "--momentum", str(momentum),
5         "--epochs", str(epochs),
6         "--uploadS3", "true"
7     ]

```

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、エポック数 = 100 を指定したジョブを投入する。

```
# [4]
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 の認証情報入力を求め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、`submit_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 15.3](#)），`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 66 のような画面が表示されるだろう。

The screenshot shows the AWS Batch Dashboard with three main sections: Jobs overview, Job queue overview, and Compute environment overview.

- Jobs overview:** Displays counts for RUNNABLE (0), RUNNING (1), SUCCEEDED (0), and FAILED (0) jobs.
- Job queue overview:** Shows the status of jobs in the "SimpleBatchjob-queue". The "RUNNING" column is highlighted with a red box around the value "1".
- Compute environment overview:** Lists two environments. The "Desired vCPUs" column for the "SimpleBatchcompute-env" is highlighted with a pink box around the value "4".

Figure 66. AWS Batch でジョブが実行されている様子

Figure 66 で赤で囲った箇所に注目してほしい。一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 66 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-bucketXXXX` (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 67)。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。

The screenshot shows the Amazon S3 console interface. At the top, there's a navigation bar with 'Amazon S3' and a path 'simplebatch-bucket43879c71-mbqaltx441fu'. Below it, the bucket name 'simplebatch-bucket43879c71-mbqaltx441fu' is displayed. A horizontal menu bar has tabs: 'Objects' (which is selected and highlighted in orange), 'Properties', 'Permissions', 'Metrics', 'Management', and 'Access Points'. Under the 'Objects' tab, a section titled 'Objects (1)' shows one item: 'metrics_lr0.0100_m0.1000.csv'. This item is a CSV file, last modified on June 8, 2021, at 00:53:27 (UTC+09:00), with a size of 5.3 KB and a storage class of Standard. Below this table are several action buttons: 'Copy S3 URI', 'Copy URL', 'Download', 'Open', 'Delete', 'Actions', and 'Create folder'. There's also an 'Upload' button. A search bar at the bottom allows finding objects by prefix. Navigation arrows and a refresh icon are also present.

Figure 67. 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 [7] 番のセルでは、学習結果の CSV ファイルのダウンロードを行っている。

```
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 bucket_name = "simplebatch-bucket43879c71-mbqaltx441fu"
22 df = read_table_from_s3(
23     bucket_name,
24     "metrics_lr0.0100_m0.1000.csv"
25 )
```

[6] で S3 から CSV データをダウンロードし、pandas の `DataFrame` オブジェクトとしてロードする関数を定義している。[7] を実行する際、`bucket_name` という変数の値を、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）。

続いて, [9] 番のセルで, CSV のデータをプロットしている (Figure 68). 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 # [9]
2 fig, (ax1, ax2) = plt.subplots(1,2, figsize=(9,4))
3 x = [i for i in range(df.shape[0])]
4 ax1.plot(x, df["train_loss"], label="Train")
5 ax1.plot(x, df["val_loss"], label="Val")
6 ax2.plot(x, df["val_accuracy"])
7
8 ax1.set_xlabel("Epochs")
9 ax1.set_ylabel("Loss")
10 ax1.legend()
11
12 ax2.set_xlabel("Epochs")
13 ax2.set_ylabel("Accuracy")
14
15 print("Best loss:", df["val_loss"].min())
16 print("Best loss epoch:", df["val_loss"].argmin())
17 print("Best accuracy:", df["val_accuracy"].max())
18 print("Best accuracy epoch:", df["val_accuracy"].argmax())
```

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

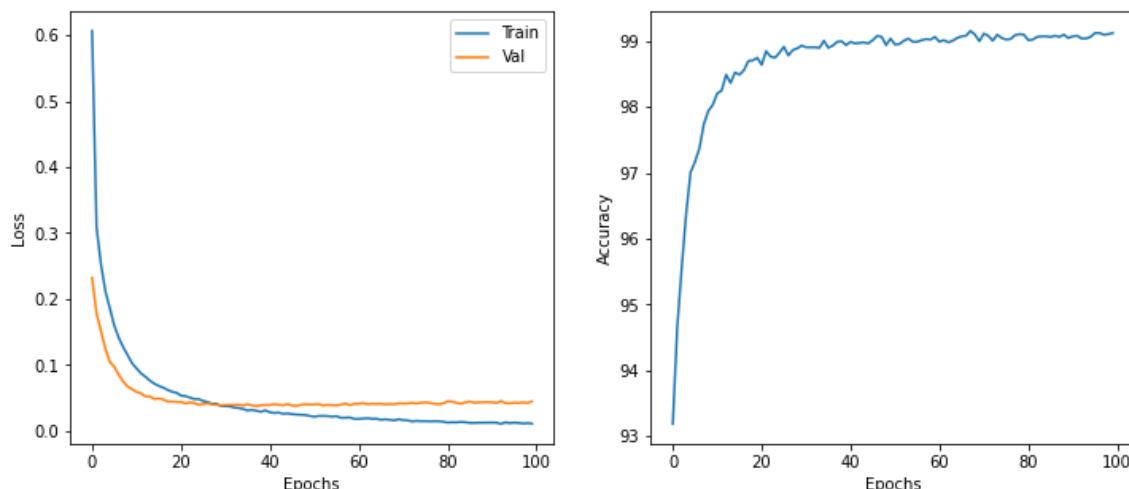


Figure 68. AWS Batch で行った MNIST モデルの学習の結果

9.9. 並列に複数の 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.

```

1 # [1]
2 import boto3
3 import argparse
4
5 # [2]
6 # AWS 認証ヘルパー ...省略...
7
8 # [3]
9 def submit_job(lr:float, momentum:float, epochs:int, profile_name=None):
10    # ...省略...

```

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. ここでは $3 \times 3 = 9$ 個のジョブを作成した.

```

1 # [4]
2 for lr in [0.1, 0.01, 0.001]:
3     for m in [0.5, 0.1, 0.05]:
4         submit_job(lr, m, 100)

```

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 69). 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 69).

The screenshot shows the AWS Batch Dashboard with three main sections:

- Jobs overview:** A summary table with four categories: RUNNABLE (0), RUNNING (9, highlighted with a red box), SUCCEEDED (0), and FAILED (18).
- Job queue overview:** A table showing job queues and their status across various stages. It lists two queues: "SimpleBatchjob-queue" and another partially visible queue. The "SimpleBatchjob-queue" row shows 9 RUNNING jobs.
- Compute environment overview:** A table listing compute environments. It shows two environments: one with a name partially visible and another named "SimpleBatchcompute-env". The "SimpleBatchcompute-env" row shows a Desired vCPUs value of 36, highlighted with a pink box.

Figure 69. 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, 実行中のジョブの一覧が確認することができる (Figure 70). ジョブ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。

The screenshot shows the AWS Batch 'Jobs' page with 9 jobs listed. The top navigation bar includes buttons for 'Clone job', 'Cancel job', 'Terminate job', 'Submit new job', and a refresh icon. Below the navigation is a filter section for 'Job queue' (set to 'SimpleBatchjob-queue') and 'Status' (set to 'RUNNING'). A pagination control shows page 1 of 1. The main table has columns for Name, ID, Started at, Stopped at, Total run time, and Status. All 9 jobs are in the 'RUNNING' state.

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-cc7ca3703cde	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 70. 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 71 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！

The screenshot shows the AWS EC2 'Instances' page with 9 instances listed. The top navigation bar includes buttons for 'Info', 'Connect', 'Instance state', 'Actions', 'Launch instances', and a refresh icon. Below the navigation is a search bar and a 'Clear filters' button. The main table has columns for Name, Instance ID, Instance state, Instance type, Status check, Alarm status, Availability Zone, and Public IPv4 DNS. All 9 instances are in the 'Running' state and are of type 'g4dn.xlarge'.

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS
[REDACTED]	[REDACTED]	[REDACTED]	[REDACTED]	[REDACTED]	[REDACTED]	[REDACTED]	[REDACTED]
[REDACTED]	[REDACTED]	[REDACTED]	[REDACTED]	[REDACTED]	[REDACTED]	[REDACTED]	[REDACTED]
-	I-0ba247386a4a863b1	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-0965f3df949d6d64d	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 71. 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 10-15 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、ただ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。一つのジョブの完了によると 10 分の時間がかかるので、9 個のハイパーパラメータの組を逐次的に計算していた場合は 90 分の時間を要することになる。AWS Batch を使ってこれらの計算を並列に実行することで、ジョブ一個分の計算時間 (=10 分) ですべての計算を終えることができた！

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。

```

1 # [5]
2 import pandas as pd
3 import numpy as np
4 import io
5 from matplotlib import pyplot as plt
6
7 # [6]
8 def read_table_from_s3(bucket_name, key, profile_name=None):
9     if profile_name is None:
10         session = boto3.Session()
11     else:
12         session = boto3.Session(profile_name=profile_name)
13     s3 = session.resource("s3")
14     bucket = s3.Bucket(bucket_name)
15
16     obj = bucket.Object(key).get().get("Body")
17     df = pd.read_csv(obj)
18
19     return df
20
21 # [7]
22 grid = np.zeros((3,3))
23 for (i, lr) in enumerate([0.1, 0.01, 0.001]):
24     for (j, m) in enumerate([0.5, 0.1, 0.05]):
25         key = f"metrics_lr{lr:0.4f}_m{m:0.4f}.csv"
26         df = read_table_from_s3("simplebatch-bucket43879c71-mbqaltx441fu", key)
27         grid[i,j] = df["val_accuracy"].max()
28
29 # [8]
30 fig, ax = plt.subplots(figsize=(6,6))
31 ax.set_aspect('equal')
32
33 c = ax.pcolor(grid, edgecolors='w', linewidths=2)
34
35 for i in range(3):
36     for j in range(3):
37         text = ax.text(j+0.5, i+0.5, f"{grid[i, j]:0.1f}",
38                         ha="center", va="center", color="w")

```

最終的に出力されるプロットが [Figure 72](#) である。

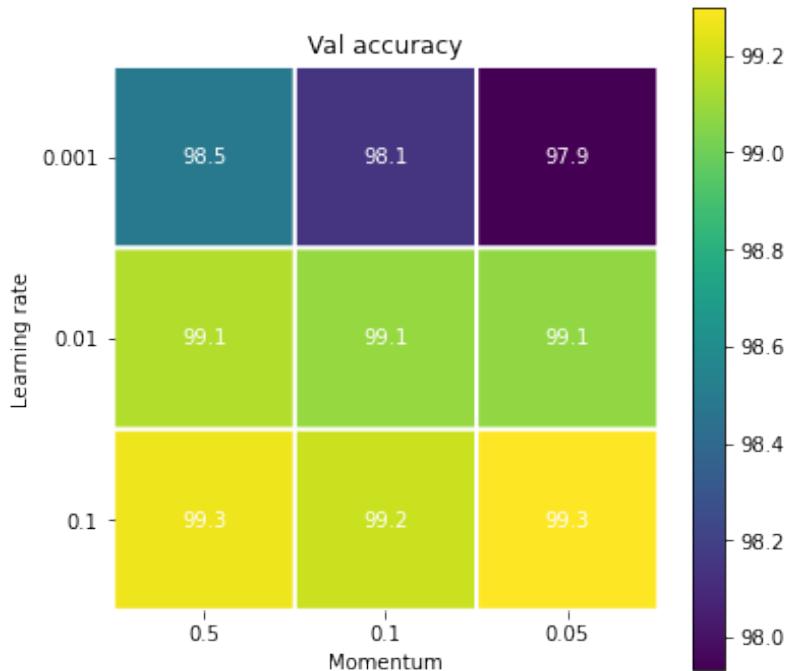


Figure 72. ハイパーパラメータのグリッドサーチの結果

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勉強用に極めて単純化されたものである点は承知いただきたい。

たとえば、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、 k 個の異なる学習・検証データの分割をあらかじめ用意して、複数回モデルの評価を行う (**k -fold cross-validation**) 方法も、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。AWS Batch を利用することで、比較的少ないプログラミングで、動的に EC2 クラスターを制御し、並列にジョブを処理するシステムが構築できた。ここまで EC2 を使いこなすことができれば、多くの問題を自力で解くことが可能になるだろう！

9.10. 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 の仕様なので従うしかない）。

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 73)。

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 73. ECR から Docker image を削除する

あるいは、AWS CLI から同様の操作を行うには、以下のコマンドを用いる (XXXX は自分の ECR レポジトリ名に置き換える)。

```
$ aws ecr batch-delete-image --repository-name XXXX --image-ids imageTag=latest
```

image の削除が完了したうえで、次のコマンドでスタックを削除する。

```
$ cdk destroy
```

[sec:batch_development_and_debug] === クラウドを用いた機械学習アプリケーションの開発とデバッグ

本章で紹介したハンズオンでは、AWS Batch を使用することでニューラルネットの学習を複数並列に実行し、高速化を実現した。本章の最後の話題として、クラウドを用いた機械学習アプリケーションの開発とデバッグの方法について述べよう。

ローカルに GPU を搭載した強力なマシンがなく、クラウドを利用する予算が確保されているのであれば、Figure 74 のような開発のスキームが理想的であると考える。最初の段階では、Chapter 6 で見たような方法で、GPU 搭載型の EC2 インスタンスを作成し、Jupyter Notebook などのインタラクティブな環境で様々なモデルを試し実験を行う。Jupyter である程度アプリケーションが完成してきたタイミングで、作成したアプリケーションを Dockerイメージにパッケージングする。そして、EC2 上で docker run を行い、作成したイメージがバグなく動作するか確認を行う。その後に、ハイパーパラメータの最適化などのチューニングを、Chapter 9 のハンズオンで学んだ AWS Batch などの計算システムを利用して行う。よい深層学習モデルが完成したら、仕上げに大規模データへの推論処理を行うシステムを Chapter 8 を参考に構築する。

実際、本書ではこの流れに沿って演習を進めてきた。MNIST タスクを解くモデルを、最初 Jupyter Notebook を使用して実験し、そのコードをほとんどそのまま Docker にパッケージし、AWS Batch を用いてハイパーパラメータサーチを行った。このサイクルを繰り返すことで、クラウドを最大限に活用した機械学習アプリケーションの開発を進めることができる。

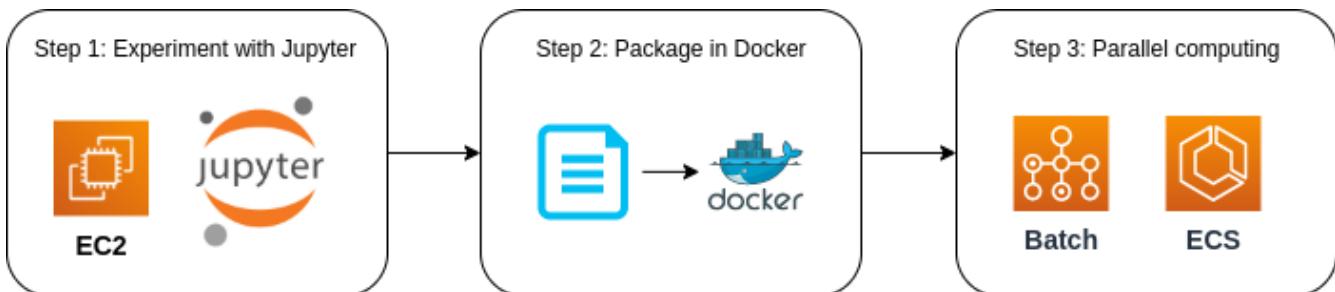


Figure 74. クラウドを活用した機械学習アプリケーションの開発フロー

9.11. 小括

ここまでが,本書第二部の内容である.第一部に引き続き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?

第二部ではまず最初に,深層学習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.さらに,ハンズオンでは,クラウドに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訓練した ([Chapter 6](#)).

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 によるクラスターの初歩を説明した ([Chapter 7](#)). 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 ([Chapter 8](#)). タスクの投入に応じて動的に計算リソースが作成・削除される様子を実際に体験できただろう.

さらに, [Chapter 9](#) では AWS Batch を用いてニューラルネットの学習を並列に実行する方法を紹介した.ここで紹介した方法は,ミニマムであるが,計算機システムを大規模化していくためのエッセンスが網羅されている.これらのハンズオン体験から,クラウド技術を応用してどのように現実世界の問題を解いていくのか,なんとなくイメージが伝わっただろうか?

本書の第三部では,さらにレベルアップし,サーバーレスアーキテクチャという最新のクラウドの設計手法について解説する.その応用として,ハンズオンでは簡単な SNS サービスをゼロから実装する.引き続きクラウドの最先端の世界を楽しんでいこう!

Chapter 10. Web サービスの作り方

ここからが、本書第三部の内容になる。これまでのセクションでは、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。EC2, ECS, Fargate, Batch などを利用して、動的にスケールするクラスターを構成し、並列にタスクを実行するクラウドシステムを実装してきた。振り返ると、これまで紹介してきた内容は、**自分自身が行いたい計算をクラウドを駆使することで実現する**、という用途にフォーカスしていたことに気がつくだろう。一方で、広く一般の人々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、というのもクラウドの重要な役割として挙げられる。

本章から始まる第三部では、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開発し、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。これを通じて、どのようにして世の中のウェブサービスができ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ゼロから構築するのか、という点を学んでもらう。その過程で、サーバーレス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。

その前準備として、本章ではどのようにしてウェブサービスが出来上がっているのか、その背後にある技術の概要を解説する。用語の解説が中心となるが、後のハンズオンを実装するために必須の知識であるので、理解して前に進むよう心がけよう。

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、コンテンツが提示されているのだろうか？

HTTP を通じたサーバーとクライアントのデータのやり取りは、すでに知っている読者も多いだろうし、逆にすべて解説しようとすると紙面が足りないので、ここではエッセンスの説明のみにとどめる。以降では [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 75](#) 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。

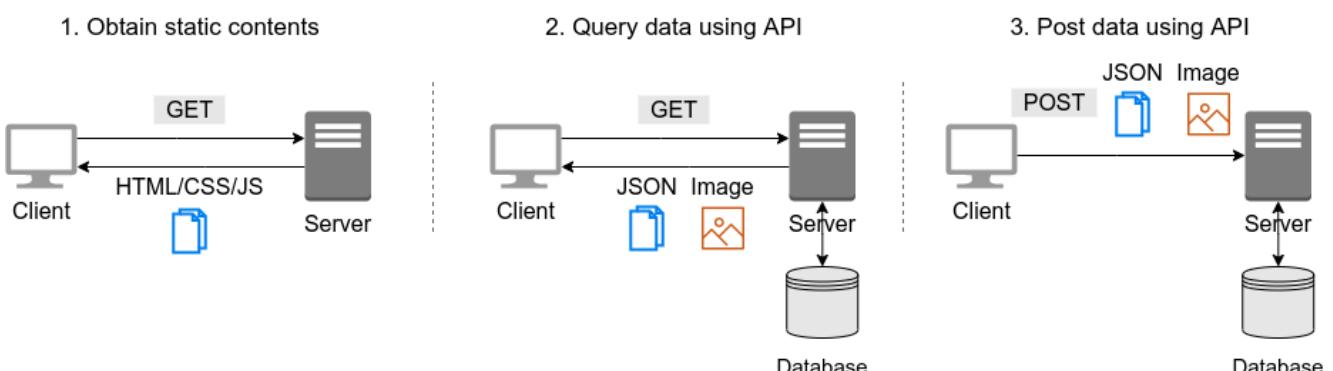


Figure 75. クライアントと Web サーバーの通信の概念図

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 (HTTPS (Hypertext Transfer Protocol Secure))** 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（例：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、クライアントは先のステップで取得された 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てコマンドやデータをやり取りするための媒介の一般的な総称である. 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.

とくに, ウェブの文脈では **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 の設計指針に従った API のことを **REST API** あるいは **RESTful API** とよんだりする.

REST API は, Figure 76 に示したような **Method** と **URI (Universal Resource Identifier)** の組からなる.



Figure 76. 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" として捉えることができる. メソッドには HTTP 規格で定義された9個の動詞 (verb) を使用することができる. この中でも, **GET**, **POST**, **PUT**, **PATCH**, **DELETE** の5個が最も頻繁に使用される (Table 6). この5つのメソッドによる操作を総称して **CRUD** (create, read, update, and delete) とよぶ.

Table 6. REST API Methods

メソッド	意図される動作
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

一方, URI は操作が行われる対象, すなわち "目的語" を表す. ウェブの文脈では操作が行われる対象のことをしばしば **リソース** とよぶ. URI は多くの場合 http または https から始まるウェブサーバーのアドレスから始まり, / (スラッシュ) 以降に所望のリソースのパスが指定される. Figure 76 の例で言えば, <https://api.twitter.com> というアドレスの [/1.1/statuses/home_timeline](https://api.twitter.com/1.1/statuses/home_timeline) というリソースを取得 (GET) せよ, という意味になる (なお, ここで **1.1** という数字は API のバージョンを示している). この API リクエストによって, ユーザーのホームのタイムラインのツイートの一覧が取得される.

REST API のメソッドには, Table 6 で挙げたもの以外に, HTTP プロトコルで定義されているほかのメソッド (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きれないこともあるが, URI の名前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ぬ, という決まりもなく, たとえば, 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調べる必要がある.



REST の概念は2000年代初頭に確立され, 今日の API 設計のスタンダードとなった. 一方で, ウェブのテクノロジーが進歩するにつれて, 新たな API の設計アプローチの需要も高まっている. 近年とくに人気を集めているのが, [GraphQL](#) と呼ばれる API の設計方法である. GraphQL は Facebook 社によって最初に作られ, 現在は GraphQL Foundation によって維持と更新がされている. GraphQL を使用すると, クライアントは REST と比較してより柔軟性の高いデータのクエリを行うことができるなど, いくつかの利点がある. キーワードだけでも知っておくと, 今後役に立つだろう.

10.3. 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 [Twitter の Developer Documentation](#) で見ることができる. いくつかの代表的な API を Table 7 にまとめた.

Table 7. Twitter API

エンドポイント	動作
<code>GET statuses/home_timeline</code>	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>GET search</code>	ツイートの検索を実行する.
<code>POST statuses/update</code>	新しいツイートを投稿する.
<code>POST media/upload</code>	画像をアップロードする
<code>POST statuses/destroy/:id</code>	<code>:id</code> で指定されたツイートを削除する.
<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 statuses/unretweet/:id</code>	<code>:id</code> で指定されたツイートのリツイートを取り消す.
<code>POST favorites/create</code>	選択したツイートを"いいね"する.
<code>POST favorites/destroy</code>	選択したツイートを"いいね"を取り消す.

この API リストをもとに, Twitter のアプリまたはウェブサイトを開いたときに起こるクライアントとサーバーの通信をシミュレートしてみよう.

ユーザーが Twitter を開くと, まず最初に `GET statuses/home_timeline` の API リクエストによって, ユーザーのホームのタイムラインのツイートのリストが取得される. 個々のツイートは JSON 形式のデータになっており, `id`, `text`, `user`, `coordinates`, `entities` などの属性を含む. `id` はツイートに固有な ID を表し, `text` はツイートの本文を含んでいる. `user` はツイートを投稿したユーザーの名前やプロフィール画像の URL などを含んだ JSON データになっている. `coordinates` にはツイートが発信された地理的な座標が記録されている. また, `entities` にはツイートに関連するメディアファイル(画像など)のリンクなどの情報が埋め込まれている. `GET statuses/home_timeline` からは直近のツイートのリスト(リストが長すぎる場合は途中で切られたもの)が取得される. もしついでの ID を知っている場合は `GET statuses/show/:id` を呼ぶことによって, `:id` パラメータで指定された特定のツイートを取得することができる.

ツイートの検索を行うためには `GET search` API を使用する. この API には, ツイートに含まれる単語や, ハッシュタグ, ツイートの発信された日時や場所など, 様々なクエリの条件を渡すことができる. API からは, `GET statuses/home_timeline` などと同様, JSON 形式のツイートのデータが返される.

ユーザーが新しいツイートを投稿するには `POST statuses/update` のエンドポイントを利用する. `POST statuses/update` には, ツイートの文章や, リプライの場合はリプライ先のツイートの ID などのデータを送信する. また, ツイートに画像データを添付したい場合は, `POST media/upload` を併せて使用する. ツイートの削除を行うには, `POST statuses/destroy/:id` を用いる.

そのほか, 頻繁に行われる操作としては, `POST statuses/retweet/:id` と `POST statuses/unretweet/:id` がある. これらは, `:id` で指定されるツイートに対して, それぞれリツイートを実行あるいは取り消すための API である.

また, `POST favorites/create`, `POST favorites/destroy` を使用することによって, 選択されたツイートに"いいね"を追加したり, 取り消したりする操作を行う.

このような一連の操作が, Twitter のアプリの背後では行われている. また, 自分自身でボットを作成したい場合は, これらの API を適切に組み合わせ, カスタムのプログラムを書くことで実現される.

このように, API はあらゆるウェブサービスを作るうえで一番基礎となる要素である. 次からの章では本章で紹介した用語が何度も出てくるので, 頭の片隅に置いたうえで読み進めていただきたい.

Chapter 11. Serverless architecture

サーバーレスアーキテクチャ (Serverless architecture) あるいは サーバーレスコンピューティング (Serverless computing) とは、従来とは全く異なるアプローチに基づく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](#) がサーバーレスアーキテクチャ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。サーバー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、文字どお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？サーバーレスについて説明するためには、まずは従来的な、"serverful" とよ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 77](#) に示す。クライアントから送信されたリクエストは、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、画像や動画などの容量の大きいデータは、また別のストレージサーバーに保存されることが多い。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 の言葉では EC2 による仮想インスタンスを想定してもらつたらよい。

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。

計算負荷を分散する目的で多数のインスタンスを起動するのはよいのだが、計算負荷が小さすぎてアイドリング状態にあるようではコストと電力の無駄遣いである。したがって、すべ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** とよぶ。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。ストレージサーバーでは、例えば頻繁にアクセスされるデータはキャッシュ領域に保存したり、データのコピーを複数作るなどのスケーリングが行われる。データベースサーバーも同様に、頻繁にアクセスされるデータのアクセスがパンクしてしまわないよう、分散的な処理が必要となる。このように、**クラウドシステム内すべての箇所で、負荷が均一になるような調整が必要であり、開発者は多くの時間をそのチューニングに費やすなければならない**。また、サービスの利用者の数などに応じてスケーリングの設定は常に見直される必要があり、継続的な開発が要求される。

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、非一様な点である。非一様であるとは、たとえばタスク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指している。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）。

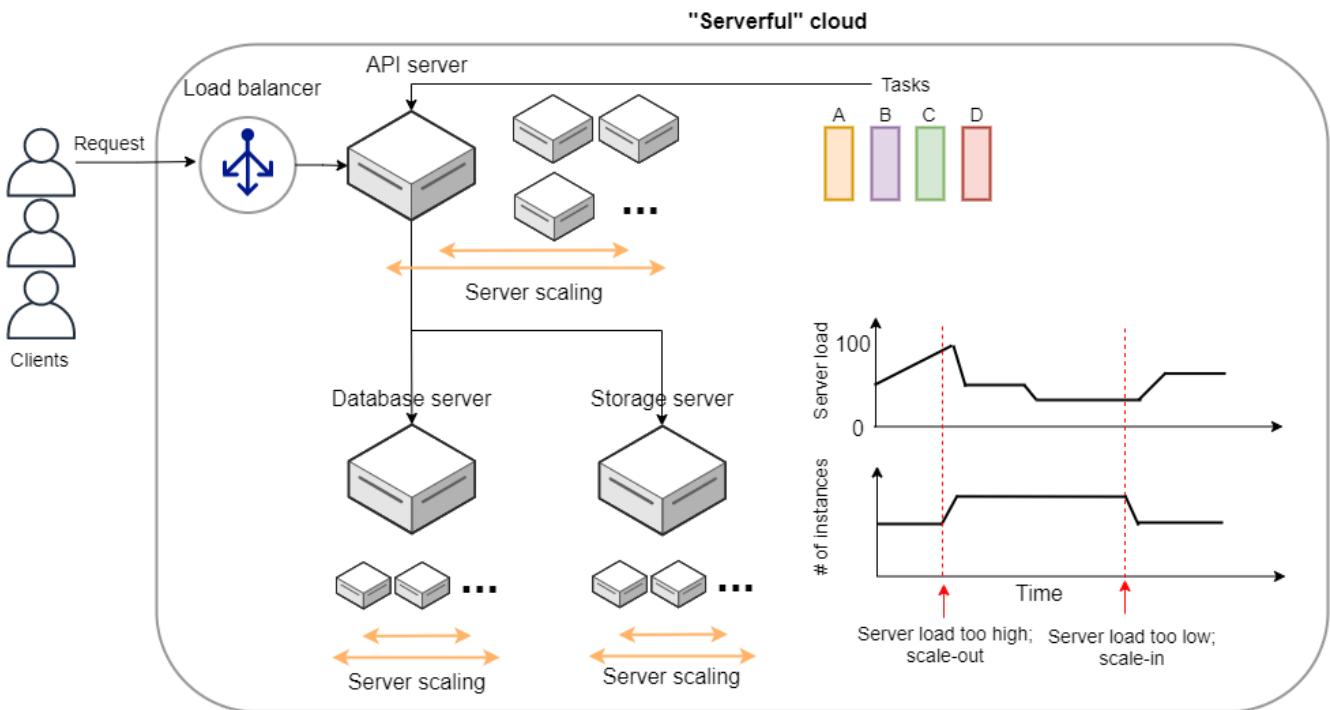


Figure 77. Serverful なクラウドシステム

11.2. Serverless クラウドへ

Section 11.1 で議論したように、クラスターのスケーリングはクラウドシステムの経済的効率とシステムの安定性を最大化するために必須の作業である。それを反映して、多くの開発者の時間が投資されてきた。

クラスターのスケーリングはすべての開発者が何度も繰り返し行ってきた作業であり、いくつかの側面をテンプレート化し、共通化することができたならば開発のコストを大幅に削減できるだろう。それを実現するには、根本的なレベルからクラウドシステムの設計を考え直す必要がある。スケーリングを前提として考えることで、もっとシンプルで見通しがよいクラウドシステムの設計の仕組みはないだろうか？そのような動機が、サーバーレスアーキテクチャが誕生する背後にあった。

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ことを出発点とする。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すべて管理する。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はなく、計算のタスクの需要が生まれる毎に、**実行したいプログラム・コマンドをクラウドに提出する**。クラウドプロバイダーは、自身のも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。言い換えると、**計算リソースのスケーリングやアロケーションなどはクラウドプロバイダーが一手に引き受け、ユーザーはジョブをサブミットすることに注力する**、という枠組みである。これを図示すると、Figure 78 のようになる。

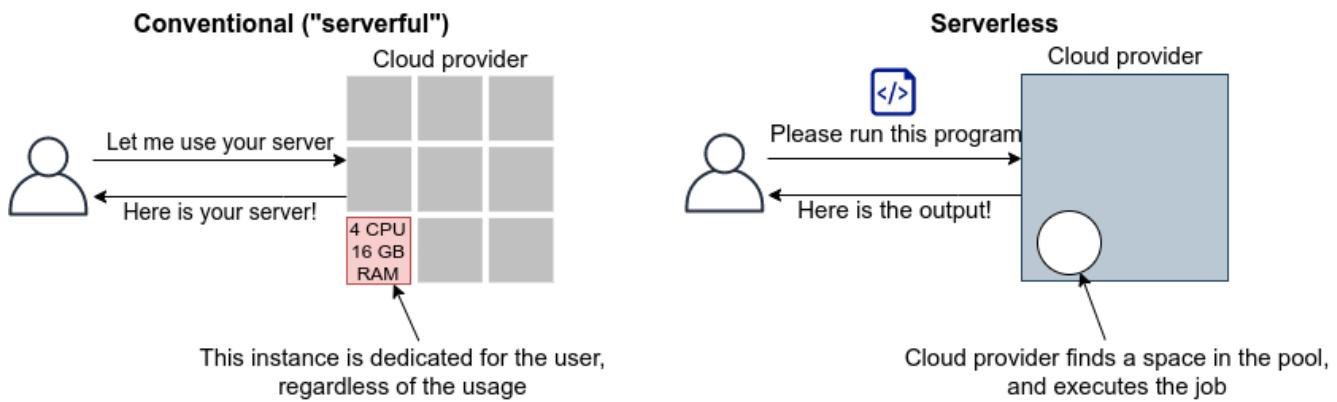


Figure 78. 従来のクラウドと Serverless クラウドの比較

サーバーレスクラウドでは、スケーリングはすべてクラウドプロバイダーが引き受けるので、スケーラビリティーが保証されている。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の独自の仕組みによってすべてのタスクが遅延なく実行される。また、サーバー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(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。

サーバーレスクラウドは、従来のクラウドとは根本から異なったアプローチなので、コードの書き方やシステムの設計が大きく異なる。サーバーレスクラウドを開発・運用するには、サーバーレス固有の概念や用語に精通している必要がある。以降では、実際にクラウドを動かしながら、サーバーレスをより具体的に体験していこう。

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、サーバーレスクラウドは、電気・水道・ガス料金と似ている。こちらは、実際に使用した量に比例して料金が決定されている。サーバーレスクラウドも、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. サーバーレスクラウドを構成するコンポーネント

サーバーレスアーキテクチャの概要がわかってきたところで、ここでは AWSにおいてサーバーレスクラウドを構成する様々なコンポーネントを紹介していこう。特に、**Lambda**, **S3**, **DynamoDB**を取り上げ、解説する (Figure 79)。サーバーレスクラウドは、これらのコンポーネントを統合することで一つのシステムが出来上がる。ここでは、Lambda, S3, DynamoDB を利用する際に押さえておかなければならない知識を一通り説明しきる都合上、具体的なイメージがわきにくいかもしれない。が、続く Chapter 12 でそれについてハンズオン形式で演習を行うので、そこでさらに理解を深めれば大丈夫である。



Figure 79. Lambda, S3, DynamoDB のアイコン

11.3.1. Lambda

AWSでサーバー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 80 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のコードを事前に登録しておく。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。Lambdaに登録されたひとつひとつのプログラムを関数(Function)とよぶ。そして、関数を実行したいときに、invoke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ただちに(数ミリセカンドから数百ミリセカンド程度の時間で)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。

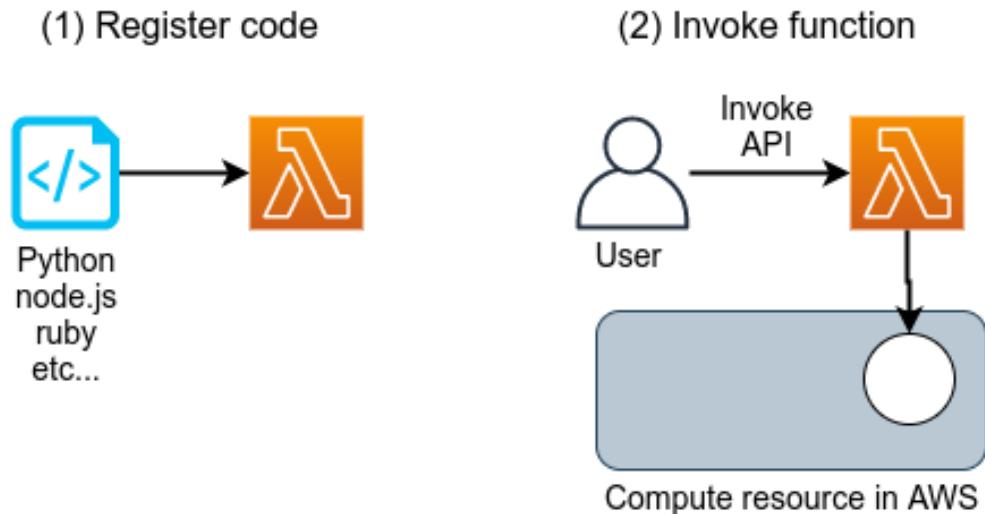


Figure 80. AWS Lambda

このように、Lambdaでは占有された仮想インスタンスは存在せず、実行を待っているプログラムだけがある状態である。invokeのリクエストに応じて、プログラムがAWSの巨大な計算機プールのどこかに配置され、実行される。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る、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のことを総称して **FaaS (Function as a Service)** とよぶ。

Lambdaではそれぞれの関数につき、128MBから10240MBのメモリーを使用することができる(執筆時点の仕様)。また、実効的なCPUのパワーはメモリーの量に比例する形で割り当てられる。すなわち、タスクに割り当てるメモリーの量が多いほど、より多くのCPUリソースが割り当てられることになる(しかし、RAMとCPUパワーの具体的な換算表はAWSからは公開されていない)。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 8](#)はLambdaの利用料金表である(執筆時点で [ap-north-east-1](#) リージョンを選択した場合)。

Table 8. Lambda の料金表

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

実行時間に比例する料金に追加して、リクエストを送信することに発生する料金が設定されている。これは、百万回のリクエストにつき \$0.2 である。たとえば、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 + 0.2 = \0.6 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 \$0.6 しかコストが発生しないことになる。また、コードが実行されず待機状態になっている場合は、発生する料金は0である。このように、実際に意味のある処理が行われた時間にのみ、料金が発生する仕組みになっている。

Lambdaは比較的短時間で完了する、反復性の高いタスクの実行に向いている。データベースの読み書きはその典型的な例であるが、そのほかにも、画像のサイズをトリミングしたり、サーバーサイドで定期的に実行されるメンテナンス処理などの利用が考えられる。また、複数のLambdaをリレー式に繋げることも可能で、シンプルな処理

を組み合わせることで複雑なロジックを表現することができる。



上述の Lambda の料金計算は、説明のためコストに寄与する要素をいくつか省いている点は承知いただきたい。例えば、DynamoDB の読み書きに関する料金や、ネットワークの通信にかかるコストが考慮されていない。

11.3.2. 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。したがって、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ぬ。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ディスクを初期化するときに決めなければならぬ、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変更することは可能である）。よって、従来的なクラウドでは、ストレージを借りる際にはあらかじめディスクのサイズを指定せねばならぬ、ディスクの中身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（Figure 81）。

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（Figure 81）。S3 は従来的なストレージシステムと異なり、OS に“マウントする”という概念はない。基本的に API を通じてデータの読み書きの操作が行われる。また、データの冗長化や暗号化、バックアップの作成など、通常ならば OS と 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。S3 では事前に決められたディスク領域のサイズではなく、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は、保存してあるデータの総容量で決定される。

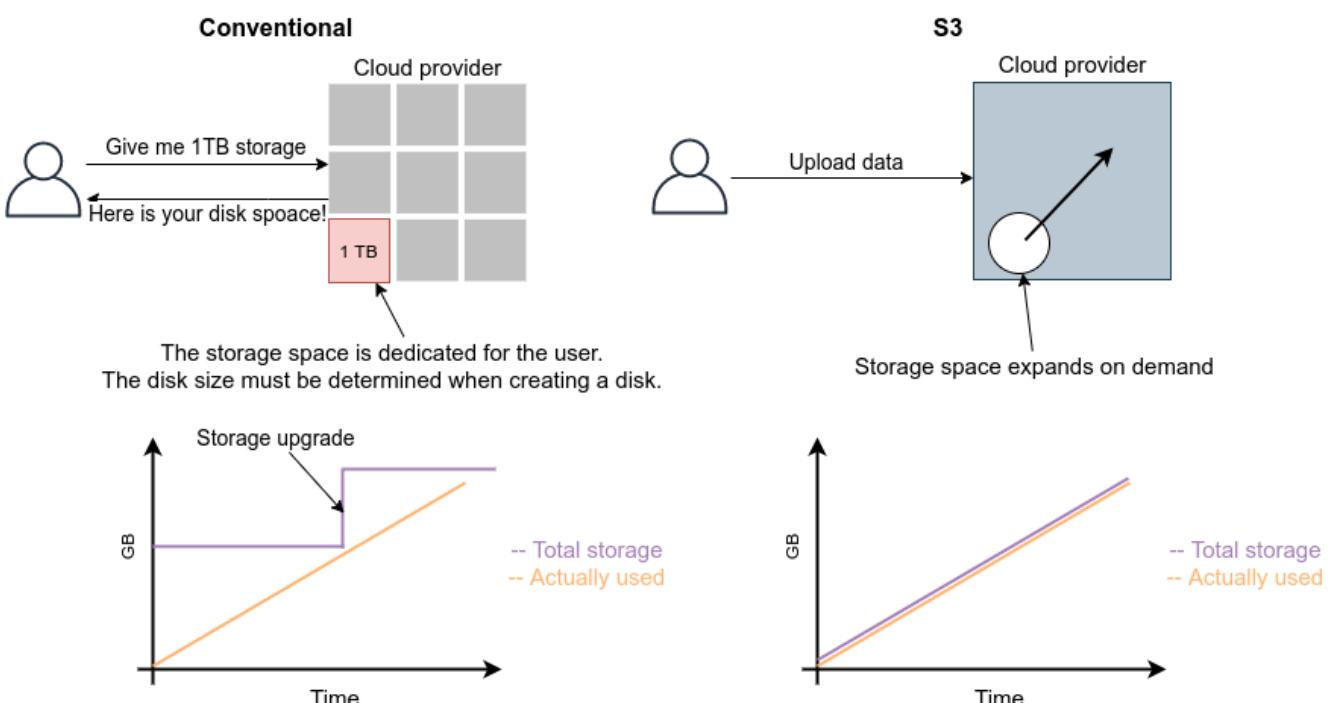


Figure 81. S3 と従来的なファイルシステムの比較

S3 を利用する際に、料金に関わってくる主要な事項をまとめたのが Table 9 である（us-east-1 リージョンのもの。説明のため主要な事項のみ取り出している。詳細は [公式ドキュメンテーション "Amazon S3 pricing"](#) を参照）。

Table 9. S3 の利用料金

項目	料金
Data storage (First 50TB)	\$0.023 per GB per month
PUT, COPY, POST, LIST requests (per 1,000 requests)	\$0.005

項目	料金
GET, SELECT, and all other requests (per 1,000 requests)	\$0.0004
Data Transfer IN To Amazon S3 From Internet	\$0
Data Transfer OUT From Amazon S3 To Internet	\$0.09 per GB

第一に,データの保存には \$0.025 per GB のコストが月ごとに発生する. したがって,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,**PUT, COPY, POST** などのリクエスト (=データを書き込む操作) に対しては,データ容量に関係なく,1000回ごとに \$0.005 のコストが発生する. **GET, SELECT** などのリクエスト (=データを読み込む操作) に対しては,1000回ごとに \$0.0004 のコストが発生する. また, S3 はデータを外に取り出す際の通信にもコストが生じる. 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 (data-out) すると \$0.09 per GB のコストが発生する. データをインターネットを通じて 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 や EC2 など) にデータを転送するのは無料である. AWS のリージョンをまたいだデータの転送にはコストが発生する. いずれにせよ, サーバーレスの概念に則り, すべての料金が従量課金制で決定される設定になっている.

11.3.3. 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や商品の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. したがって, 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 コアを搭載したマシンが用いられることが多い. また, データベース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, AWS が提供しているサーバーレスな分散型データベースである. サーバーレスであるので, 占有されたデータベース用仮想インスタンスは存在せず, API を通じてデータの書き込み・読み出し・検索などの操作を行う. S3 と同様に, データ保存領域の上限は定められておらず, 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. また, 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 が自動で行うので, ユーザーは心配する必要はない.

DynamoDB での利用料金の計算はやや複雑なのだが, "On-demand Capacity" というモードで使用した場合の料金に関する主要な事項をまとめたのが [Table 10](#) である ([us-east-1](#) リージョンのもの. 詳細は [公式ドキュメンテーション "Pricing for On-Demand Capacity"](#) を参照).

Table 10. DynamoDB の利用料金

項目	料金
Write request units	\$1.25 per million write request units
Read request units	\$0.25 per million read request units
Data storage	\$0.25 per GB-month

DynamoDB ではデータの書き込み操作の単位を **write request unit** とよび, データの読み込み操作の単位を **read request unit** とよぶ. 基本的に, 1kB 以下のデータを一度書き込むと 1 write request unit を消費し, 4kB 以下のデータを一度読み込むと 1 read request unit を消費する (詳しくは [公式ドキュメンテーション "Read/Write Capacity Mode"](#) を参照のこと). write request units は100万回ごとに \$1.25, read request units は100万回ごとに \$0.25 のコストが設定されている. また, 保存されたデータ容量に対して \$0.25 per GB の

コストが月ごとに発生する。DynamoDBは高速な検索機能などを備えたデータベースであるので、GBあたりのストレージコストはS3に比べ10倍程度高い。DynamoDBのデータの転送に関わるコストは、同じリージョン内ならばdata-in,data-outともに\$0である。リージョンをまたいだ通信には別途コストが発生する。

11.3.4. その他のサーバーレスクラウドの構成要素

以上で紹介したLambda,S3,DynamoDBがサーバーレスクラウドの中で最も使用する頻度が高いサービスになる。その他のサーバーレスクラウドの構成要素を以下に列挙する。いくつかについては、今後のハンズオンを行う中で改めて解説を行う。

- [API Gateway](#): APIを構築する際のルーティングを担う。[Chapter 13](#)で取り上げる。
- [Fargate](#): [Chapter 8](#)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との違いは、Lambdaよりも大容量のメモリーやCPUを要するような計算などを行うことができる点が挙げられる。
- [Simple Notification Service \(SNS\)](#): サーバーレスのサービス間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？

この問い合わせへの答えは、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をしてきたが、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。



大きな欠点を一つあげるとすれば、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。興味のある読者はぜひ読んでいただきたい。

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

Chapter 12. Hands-on #5: サーバーレス入門

前章ではサーバーレスアーキテクチャの概要の説明を行った。本章では、ハンズオン形式でサーバーレスクラウドを実際に動かしながら、具体的な使用方法を学んでいこう。今回のハンズオンでは Lambda, S3, DynamoDB の三つのサーバーレスクラウドの構成要素に触れていく。それぞれについて、短いチュートリアルを用意してある。

12.1. Lambda ハンズオン

まずは、Lambda を実際に動かしてみよう。ハンズオンのソースコードは GitHub の [hands-on/serverless/lambda](#) に置いてある。

このハンズオンで使用するアプリケーションのスケッチを Figure 82 に示す。STEP 1 では、AWS CDK を使用して Python で書かれたコードを Lambda に登録する。続いて STEP 2 では、Invoke API を使用して、同時にいくつも Lambda を起動し、並列な計算を行う。Lambda のワークフローを体験する目的で最小限の設定である。

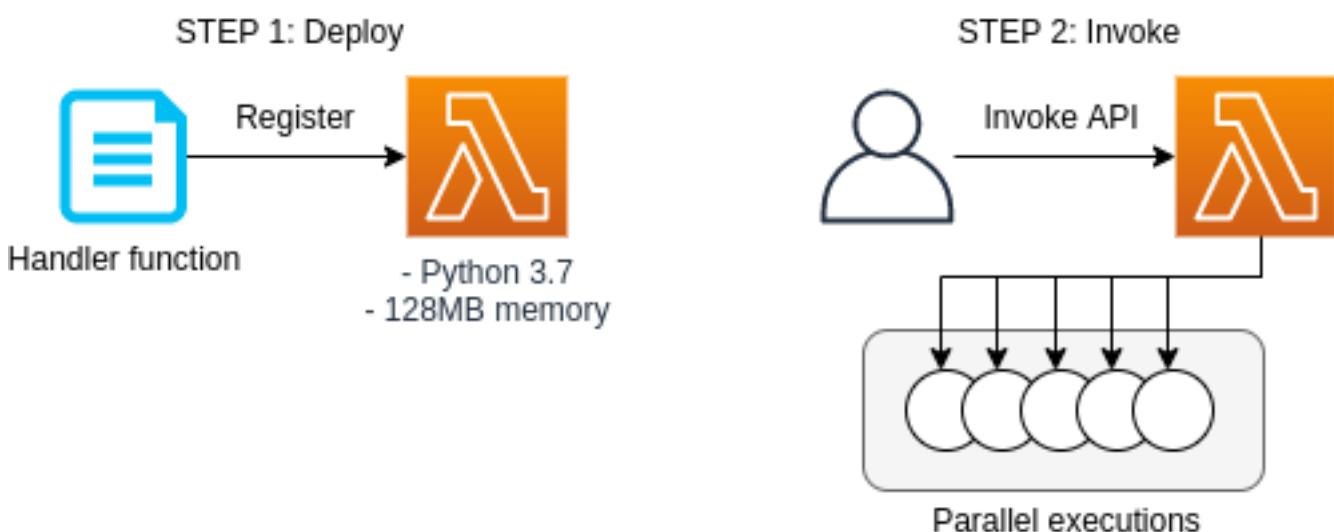


Figure 82. Lambda チュートリアルの概要



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     sushi = ["salmon", "tuna", "squid"]
8     message = "Welcome to Cloud Sushi. Your order is " + choice(sushi)
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、["salmon", "tuna", "squid"] のいずれかの文字列をランダムに選択し、"Welcome to Cloud Sushi. Your order is XXXX" (XXXXは選ばれた寿司のネタ) というメッセージをリターンする。
- ② 次に、Lambda に<1>で書いた関数を配置している。パラメータの意味は、文字どおりの意味なので難しくはないが、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 のほかに、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。
- `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。シークレットキーの設定も忘れずに (Section 15.3)。

```

# プロジェクトのディレクトリに移動
$ cd handson/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

# 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 83 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 83. 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 84 のような画面から Lambda の関数の一覧が確認できる。

Function name	Description	Runtime	Code size	Last modified
SimpleLambda-LambdaHandler212865DC-10W5VV6HTHZI		Python 3.7	306 bytes	4 minutes ago

Figure 84. 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` で始まるランダムな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 85 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できる。さらに下の方にスクロールすると、関数の各種設定も確認できる。

SimpleLambda-LambdaHandler212865DC-10W5VV6HTHZI

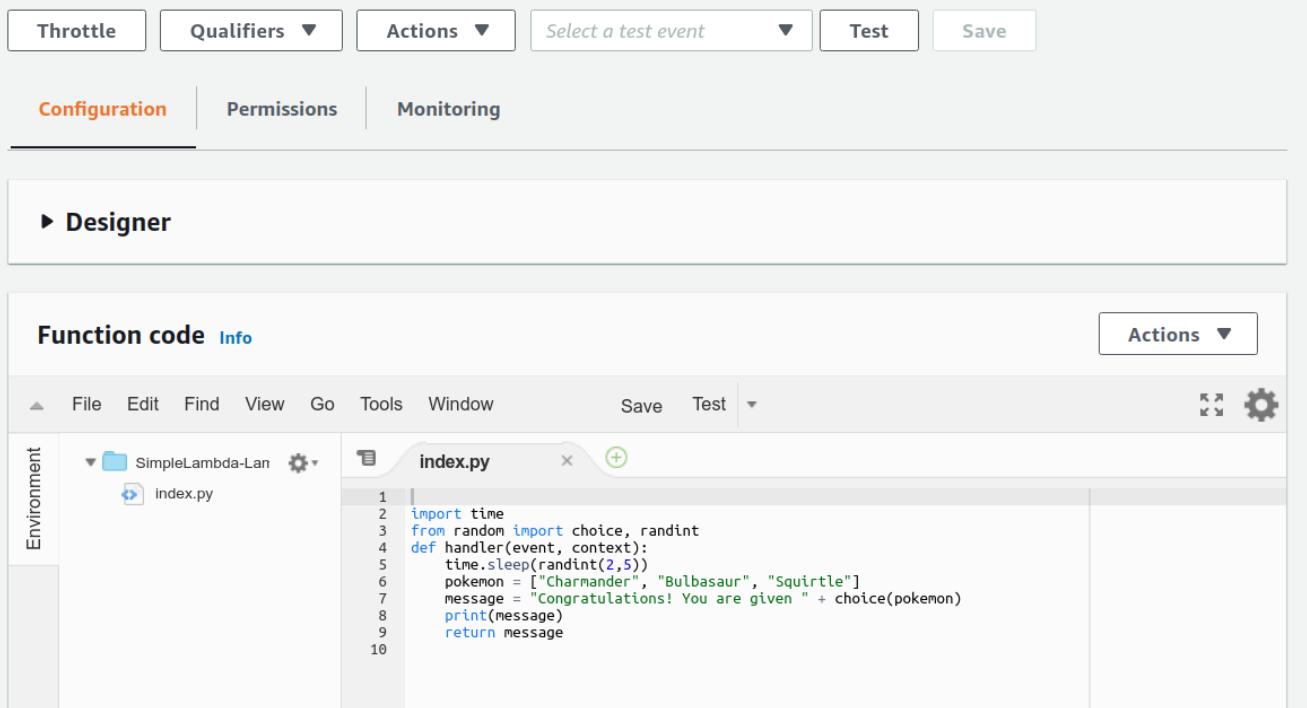


Figure 85. Lambda コンソール - 関数の詳細



Lambda で実行されるコードは、Lambda のコンソール画面 (Figure 85) のエディターで編集することもできる。デバッグをするときなどは、こちらを直接いじる方が早い場合もある。その場合は、CDK のコードに行った編集を反映させなおすことを忘れずに。

12.1.2. Lambda 関数の実行

それでは、作成した Lambda 関数を実行 (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、[handson/serverless/lambda/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、"Welcome to Cloud Sushi. Your order is salmon" 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、乱数が生成されたうえで、ランダムな寿司ネタが選択されて出力が返されている。このコマンドを何度か打ってみて、実行ごとに異なる寿司ネタが返されることを確認しよう。

さて、このコマンドは、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。[handson/serverless/lambda/invoke_many.py](#) のスクリプトを使用する。

次のコマンドを実行しよう。XXXX の部分は前述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
```

すると次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 85 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 86 のようなグラフが表示されるだろう。

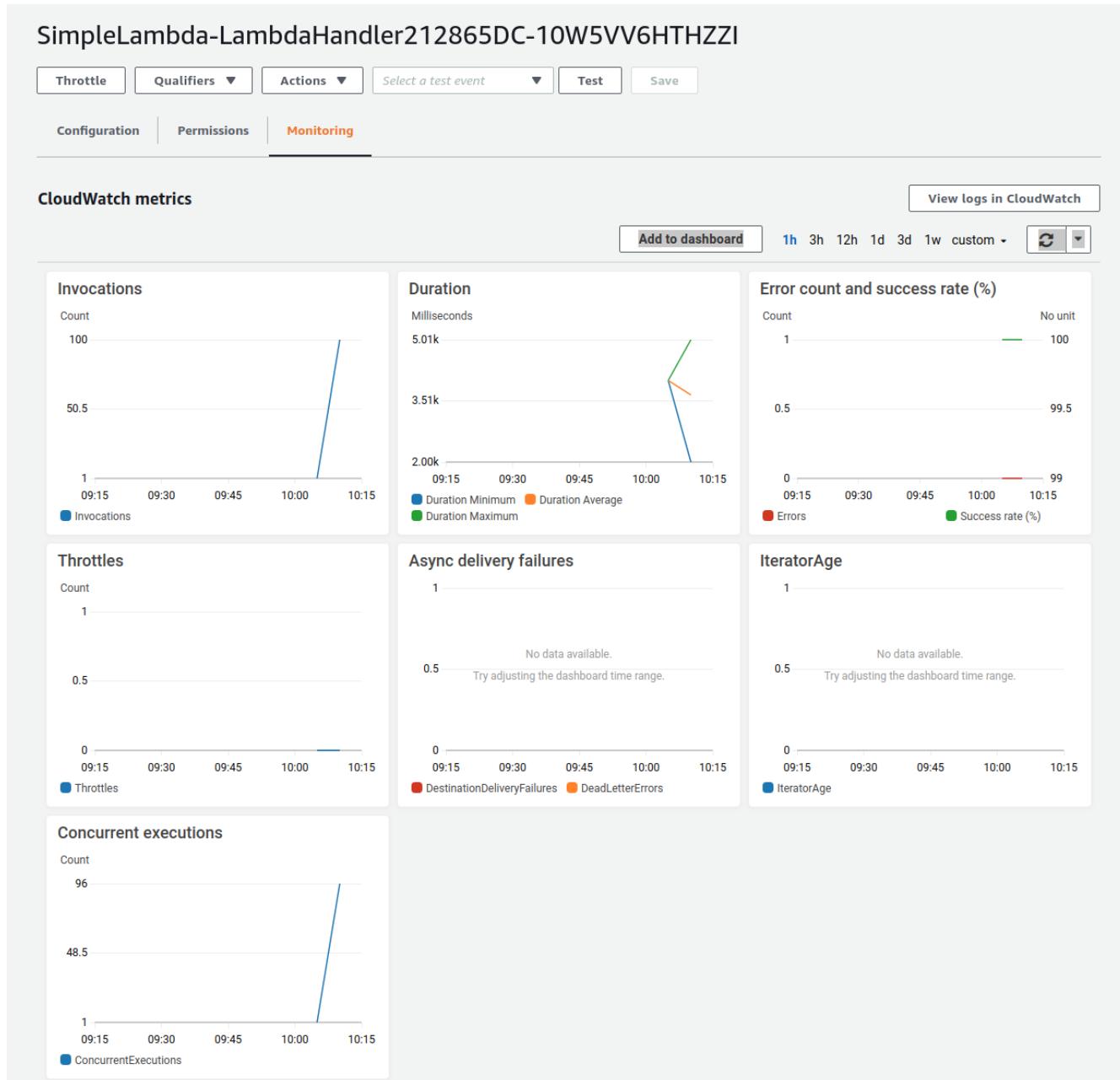


Figure 86. Lambda コンソール - 関数の実行のモニタリング



Figure 86 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 86 で "Invocations" が関数が何回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" は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、同時に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よい。Lambdaはそのような大量のリクエストにも対応できることが確認できるだろう。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

続いて、DynamoDB の簡単なチュートリアルをやってみよう。ハンズオンのソースコードは GitHub の [/handson/serverless/dynamodb](#) に置いてある。

このハンズオンで使用するアプリケーションのスケッチを Figure 87 に示す。STEP 1 では、AWS CDK を使用して DynamoDB のテーブルを初期化し、デプロイする。続いて STEP 2 では、API を使用してデータベースのデータの書き込み・読み出し・削除などの操作を練習する。

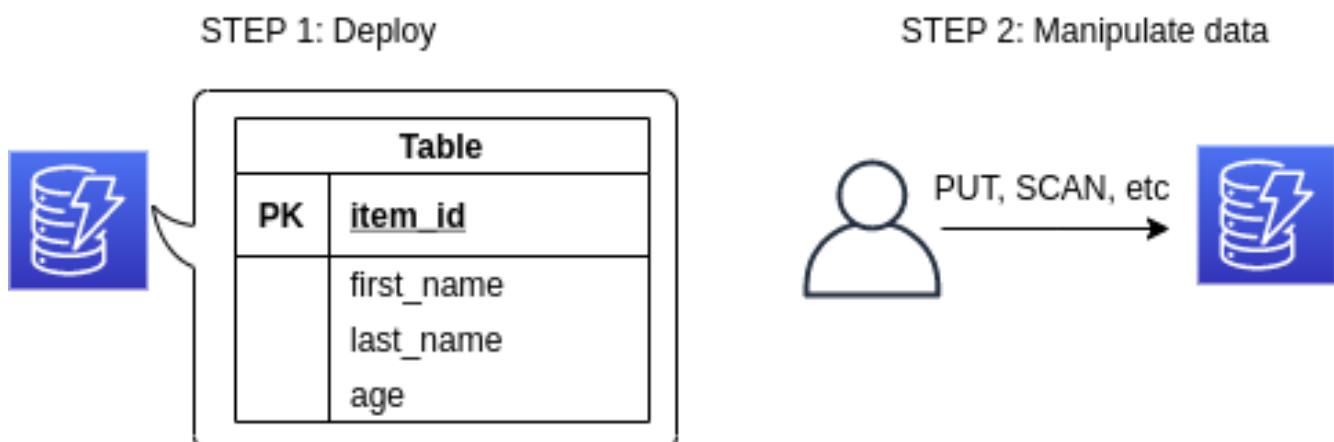


Figure 87. DynamoDB チュートリアルの概要



このハンズオンは、基本的に AWS DynamoDB の無料枠 の範囲内で実行できる。

[handson/serverless/dynamodb/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             ①
8                 partition_key=ddb.Attribute(
9                     name="item_id",
10                    type=ddb.AttributeType.STRING
11                ),
12                ②
13                billing_mode=ddb.BillingMode.PAY_PER_REQUEST,
14                ③
15                removal_policy=core.RemovalPolicy.DESTROY
16            )

```

このコードで、最低限の設定がなされた空の DynamoDB テーブルが作成される。それぞれのパラメータの意味を簡単に解説しよう。

- ① **partition_key**: すべての DynamoDB テーブルには Partition key が定義されていなければならない。Partition key とは、テーブル内の要素（レコード）ごとに存在する固有の ID のことである。同一の Partition key をもった要素がテーブルの中に二つ以上存在することはできない（注：Sort Key を使用している場合は除く）。詳しくは [公式ドキュメンテーション "Core Components of Amazon DynamoDB" 参照](#)。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` という名前をつけている。
- ② **billing_mode**: `ddb.BillingMode.PAY_PER_REQUEST` を指定することで、[On-demand Capacity Mode](#) の DynamoDB が作成される。ほかに `PROVISIONED` というモードがあるが、これはかなり高度なケースを除いて使用しないだろう。
- ③ **removal_policy**: CloudFormation のスタックが消去されたときに、DynamoDB も一緒に消去されるかどうかを指定する。このコードでは `DESTROY` を選んでいるので、すべて消去される。ほかのオプションを選択すると、スタックを消去しても DynamoDB のバックアップを残す、などの動作を定義することができる。

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）。シークレットキーの設定も忘れずに（[Section 15.3](#)）。

```

# プロジェクトのディレクトリに移動
$ cd handson/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

# 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 88](#) 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 88. 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 89 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 89. DynamoDB のコンソール (テーブルの一覧)

今回のアプリケーションで作成したのが **SimpleDynamoDb** で始まるランダムな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。すると Figure 90 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。

An item consists of one or more attributes. Each attribute consists of a name, a data type, and a value. When you read or write an item, the only attributes that are required are those that make up the primary key. [More info](#)

Figure 90. DynamoDB のコンソール (テーブルの詳細画面)

12.2.2. データの読み書き

それでは、Section 12.2.1 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。

まずは、テーブルに新しい要素を追加してみよう。ハンズオンのディレクトリにある `simple_write.py` を開いてみよう。中には次のような関数が書かれている。

```

1 import boto3
2 from uuid import uuid4
3 ddb = boto3.resource('dynamodb')
4
5 def write_item(table_name):
6     table = ddb.Table(table_name)
7     table.put_item(
8         Item={
9             'item_id': str(uuid4()),
10            'first_name': 'John',
11            'last_name': 'Doe',
12            'age': 25,
13        }
14    )

```

コードを上から読んでいくと、まず最初に boto3 ライブラリをインポートし、`dynamodb` のリソースを呼び出している。`write_item()` 関数は、DynamoDB のテーブルの名前（上で見た SimpleDynamoDb-XXXX）を引数として受け取る。そして、`put_item()` メソッドを呼ぶことで、新しいアイテムを DB に書き込んでいる。アイテムには `item_id`、`first_name`、`last_name`、`age` の4つの属性が定義されている。ここで、`item_id` は先ほど説明した Partition key に相当しており、UUID4 を用いたランダムな文字列を割り当てている。

では、`simple_write.py` を実行してみよう。"XXXX" の部分を自分がデプロイしたテーブルの名前（SimpleDynamoDb で始まる文字列）に置き換えたうえで、次のコマンドを実行する。

```
$ python simple_write.py XXXX
```

新しい要素が正しく書き始めたか、AWS コンソールから確認してみよう。Figure 90 と同じ手順で、テーブルの中身の一覧を表示する。すると Figure 91 のように、期待通り新しい要素が見つかるだろう。

	item_id	age	first_name	last_name
<input type="checkbox"/>	2b4874e5-aa5e-4e18-b303-72b61839da6f	25	John	Doe

Figure 91. DynamoDB に新しい要素が追加されたことを確認

boto3 を使ってテーブルから要素を読みだすことも可能である。ハンズオンのディレクトリにある `simple_read.py` を見てみよう。

```
1 import boto3
2 ddb = boto3.resource('dynamodb')
3
4 def scan_table(table_name):
5     table = ddb.Table(table_name)
6     items = table.scan().get("Items")
7     print(items)
```

`table.scan().get("Items")` によって、テーブルの中にあるすべての要素を読みだしている。

次のコマンドで、このスクリプトを実行してみよう ("XXXX" の部分を正しく置き換えることを忘れずに)。

```
$ python simple_read.py XXXX
```

先ほど書き込んだ要素が出力されることを確認しよう。

12.2.3. 大量のデータの読み書き

DynamoDB の利点は、最初に述べたとおり、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検索し拾ってくるには、次のコマンドを実行すればよい。

```
$ python batch_rw.py XXXX search_under_age 2
```

上の2つのコマンドを何回か繰り返し実行してみて、データベースに負荷をかけてみよう。とくに大きな遅延なく結果が返ってくることが確認できるだろう。

12.2.4. スタックの削除

DynamoDB で十分に遊ぶことができたら、忘れずにスタックを削除しよう。

これまでのハンズオンと同様、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
```

12.3. S3 ハンズオン

最後に、S3 の簡単なチュートリアルを紹介する。ハンズオンのソースコードは GitHub の [handson/serverless/s3](#) に置いてある。

[Figure 92](#) が今回提供する S3 チュートリアルの概要である。STEP 1 として、AWS CDK を用いて S3 に新しい空の

バケット (Bucket) を作成する。続いて STEP 2 では、データのアップロード・ダウンロードの方法を解説する。

STEP 1: Deploy



S3

STEP 2: Manipulate data

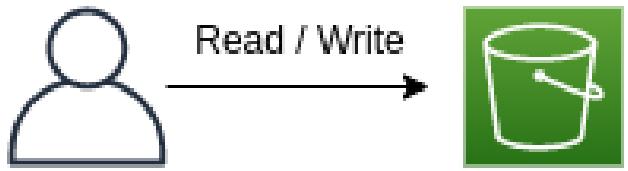


Figure 92. S3 チュートリアルの概要



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

[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 SimpleS3(core.Stack):
2     def __init__(self, scope: core.App, name: str, **kwargs) -> None:
3         super().__init__(scope, name, **kwargs)
4
5         # S3 bucket to store data
6         bucket = s3.Bucket(
7             self, "bucket",
8             removal_policy=core.RemovalPolicy.DESTROY,
9             auto_delete_objects=True,
10        )
```

`s3.Bucket()` を呼ぶことによって空のバケットが新規に作成される。上記のコードだと、バケットの名前は自動生成される。もし、自分の指定した名前を与えたい場合は、`bucket_name` というパラメータを指定すればよい。その際、バケットの名前はユニークでなければならない (i.e. AWS のデプロイが行われるリージョン内で名前の重複がない) 点に注意しよう。もし、同じ名前のバケットが既に存在する場合はエラーが返ってくる。



デフォルトでは、CloudFormation スタックが削除されたとき、S3 バケットとその中に保存されたファイルは削除されない。これは、大切なデータを誤って消してしまうことを防止するための安全策である。`cdk destroy` を実行したときにバケットも含めてすべて削除されるようにするには、`removal_policy=core.RemovalPolicy.DESTROY, auto_delete_objects=True` とパラメータを設定する。結果もよく理解したうえで、自分の用途にあった適切なパラメータを設定しよう。

12.3.1. デプロイ

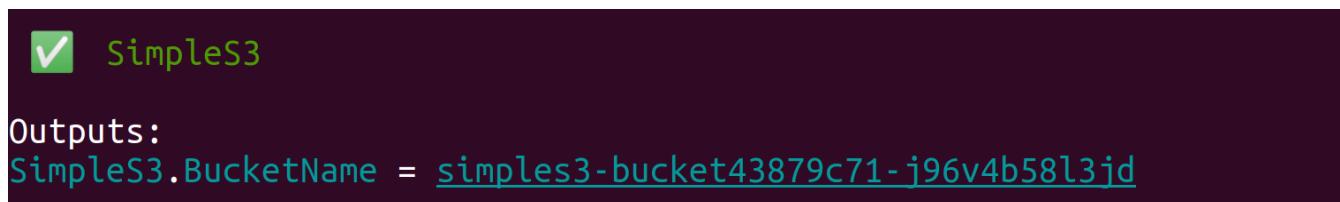
デプロイの手順は、これまでのハンズオンとほとんど共通である。ここでは、コマンドのみ列挙する (# で始まる行はコメントである)。シークレットキーの設定も忘れずに ([Section 15.3](#))。

```
# プロジェクトのディレクトリに移動
$ cd handson/serverless/s3

# 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

# デプロイを実行
$ cdk deploy
```

デプロイを実行すると, [Figure 93](#) 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 `SimpleS3.BucketName = XXXX` が, 新しく作られたバケットの名前である(今回提供しているコードを使うとランダムな名前がバケットに割り当てられる). これはあとで使うのでメモしておこう.



Outputs:
`SimpleS3.BucketName = simples3-bucket43879c71-j96v4b58l3jd`

Figure 93. デプロイ実行後の出力

12.3.2. データの読み書き

スタックのデプロイが完了したら, 早速バケットにデータをアップロードしてみよう.

まずは, 以下のコマンドを実行して, `tmp.txt` という仮のファイルを生成する.

```
$ echo "Hello world!" >> tmp.txt
```

ハンズオンのディレクトリにある `simple_s3.py` に `boto3` ライブラリを使用した S3 のファイルのアップロード・ダウンロードのスクリプトが書いてある. `simple_s3.py` を使って, 上で作成した `tmp.txt` を以下のコマンドによりバケットにアップロードする. `XXXX` のところは, 自分自身のバケットの名前で置き換えること.

```
$ python simple_s3.py XXXX upload tmp.txt
```

`simple_s3.py` のアップロードを担当している部分を以下に抜粋する.

```
1 def upload_file(bucket_name, filename, key=None):
2     bucket = s3.Bucket(bucket_name)
3
4     if key is None:
5         key = os.path.basename(filename)
6
7     bucket.upload_file(filename, key)
```

`bucket = s3.Bucket(bucket_name)` の行で `Bucket()` オブジェクトを呼び出している. そして, `upload_file()` × ソードを呼ぶことでファイルのアップロードを実行している.

S3においてファイルの識別子として使われるのが **Key** である. これは, 従来的なファイルシステムにおけるパス(Path)と相同な概念で, それぞれのファイルに固有な Key が割り当てられる必要がある. Key という呼び方は, S3 が `Object storage` と呼ばれるシステムに立脚していることに由来する. `--key` のオプションを追加して `simple_s3.py` を実行することで, Key を指定してアップロードを実行することができる.

```
$ python simple_s3.py XXXX upload tmp.txt --key a/b/tmp.txt
```

ここではアップロードされたファイルに `a/b/tmp.txt` という Key を割り当てている.

ここまでコマンドを実行し終えたところで, 一度 AWS コンソールに行き S3 の中身を確認してみよう. S3 のコンソールに行くと, バケットの一覧が見つかるはずである. その中から, `simples3-bucket` から始まるランダムな名前のついたバケットを探し, クリックする. するとバケットの中に含まれるファイルの一覧が表示される([Figure 94](#)).

	Name	Type	Last modified	Size	Storage class
<input type="checkbox"/>	a/	Folder	-	-	-
<input type="checkbox"/>	tmp.txt	txt	July 7, 2021, 09:58:21 (UTC+09:00)	13.0 B	Standard

Figure 94. S3 バケットの中のファイル一覧

ここで実行した2つのコマンドによって、`tmp.txt` というファイルと、`a/b/tmp.txt` というファイルが見つかることに注目しよう。従来的なファイルシステムと似た体験を提供するため、S3 では Key が "/" (スラッシュ) によって区切られていた場合、ツリー状の階層構造によってファイルを管理することができる。



オブジェクトストレージには本来ディレクトリという概念はない。上で紹介した "/" による階層づけはあくまでユーザ一体験向上の目的のお化粧的な機能である。

次に、バケットからファイルのダウンロードを実行してみよう。`simple_s3.py` を使って、以下のコマンドを実行すればよい。`XXXX` のところは、自分自身のバケットの名前で置き換えること。

```
$ python simple_s3.py XXXX download tmp.txt
```

`simple_s3.py` のダウンロードを担当している部分を以下に抜粋する。

```

1 def download_file(bucket_name, key, filename=None):
2     bucket = s3.Bucket(bucket_name)
3
4     if filename is None:
5         filename = os.path.basename(key)
6
7     bucket.download_file(key, filename)

```

S3 からのダウンロードはシンプルで、`download_file()` メソッドを使って、ダウンロードしたい対象の Key を指定すればよい。ローカルのコンピュータでの保存先のパスを2個目の引数として渡している。

12.3.3. スタックの削除

以上のハンズオンで、S3 の一番基本的な使い方を紹介した。ここまでハンズオンが理解できたら、忘れずにスタックを削除しよう。これまでのハンズオンと同様、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
```

Chapter 13. Hands-on #6: Bashoutter

さて、最後のハンズオンとなる第六回では、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つてみよう。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。Lambda, DynamoDB, S3などの技術をすべて盛り込み、シンプルながらもサーバーレスの利点を生かしたスケーラブルなSNSアプリが誕生する。最終的には、Figure 95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！

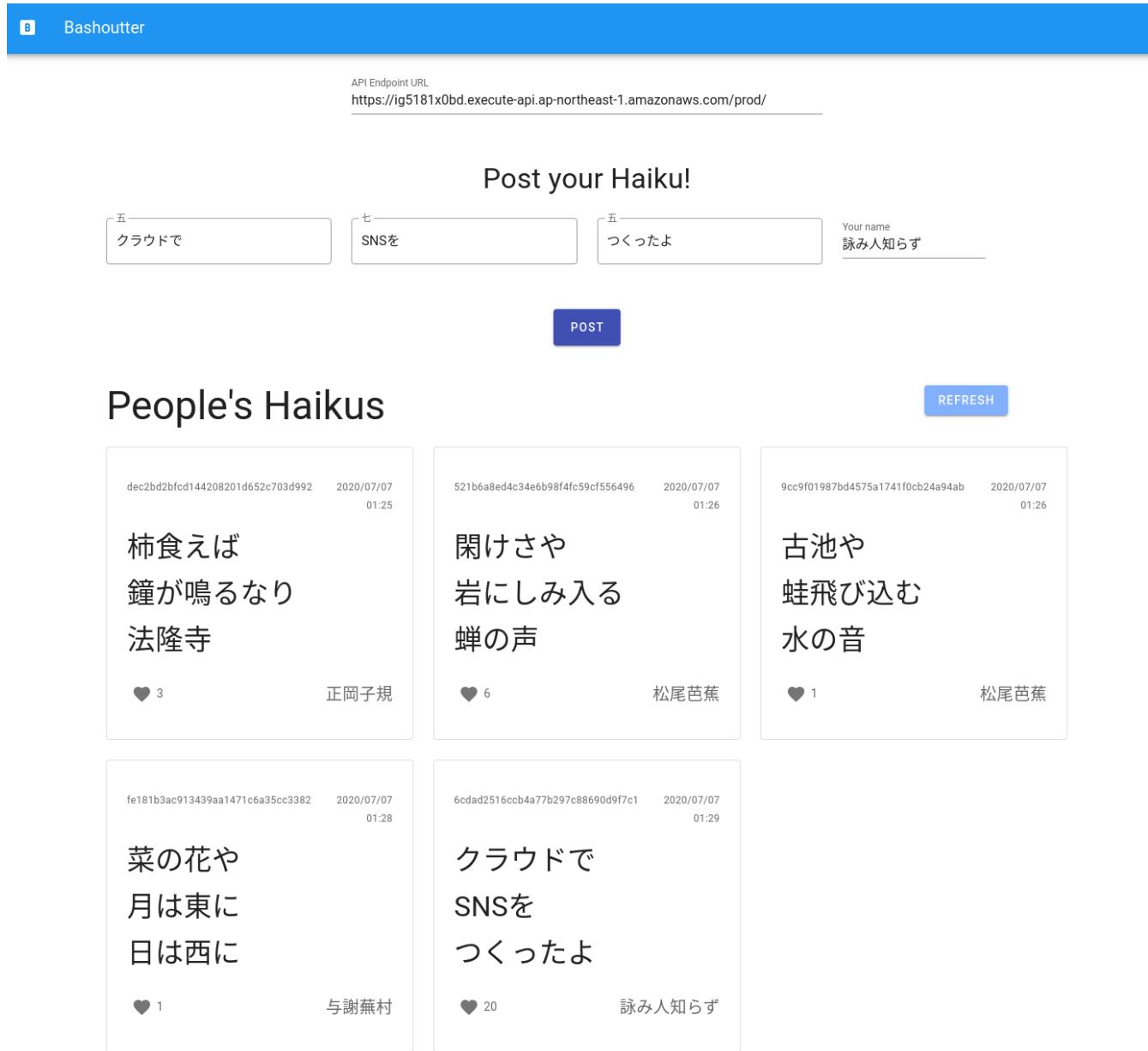


Figure 95. ハンズオン#6で作製するSNSアプリケーション "Bashoutter"

13.1. 準備

ハンズオンのソースコードは GitHub の [handson/bashoutter](#) に置いてある。

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。



このハンズオンは、基本的に AWS の無料枠 の範囲内で実行できる。

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。この機能を実現するための最小限の設計として、Table 11 に示すような四つの REST API を今回は実装する。俳句を投稿する、閲覧する、削除するという基本的なデータ操作を行うための API が完備されている。また、PATCH /haiku/{item_id} は、{item_id} で指定された俳句に”いいね”をするために使用する。

Table 11. Bashoutter 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、ハンズオンのソースコードの中の `swagger.yml` に定義してある。



Open API Specification (OAS; 少し前は Swagger Specification とよばれていた) 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。詳しくは [Swagger の公式ドキュメンテーション](#)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 96 に示す。

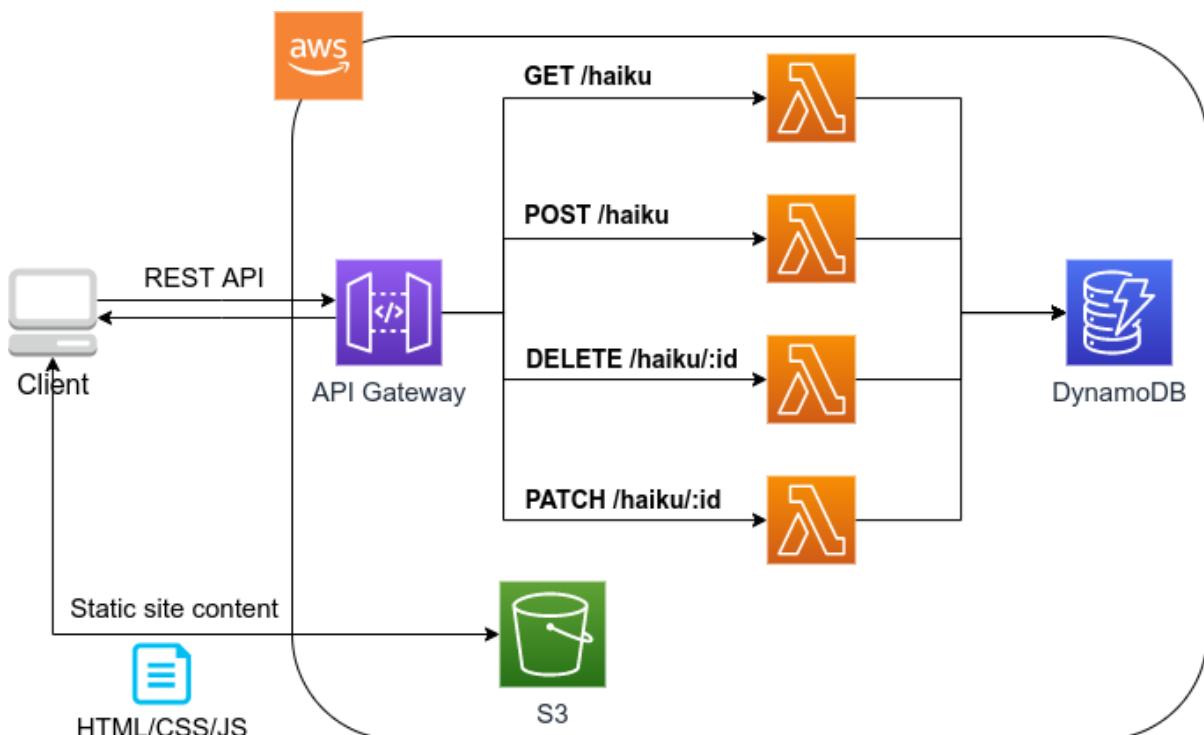


Figure 96. ハンズオン#5で作製するアプリケーションのアーキテクチャ

簡単にまとめると、次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 で指定された Lambda 関数へ転送される。
- ・それぞれの API のパス (リソース) ごとに独立した Lambda を用意する。
- ・俳句の情報 (作者,本文,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/bashouter/app.py](#))。

```

1 class Bashouter(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, "Bashouter-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, "Bashouter-Bucket",
21             website_index_document="index.html",
22             public_read_access=True,
23             removal_policy=core.RemovalPolicy.DESTROY
24         )
25
26         common_params = {
27             "runtime": _lambda.Runtime.PYTHON_3_7,
28             "environment": {
29                 "TABLE_NAME": table.table_name
30             }
31         }
32
33         ③
34         # define Lambda functions
35         get_haiku_lambda = _lambda.Function(
36             self, "GetHaiku",
37             code=_lambda.Code.from_asset("api"),
38             handler="api.get_haiku",
39             memory_size=512,
40             **common_params,
41         )
42         post_haiku_lambda = _lambda.Function(
43             self, "PostHaiku",
44             code=_lambda.Code.from_asset("api"),
45             handler="api.post_haiku",
46             **common_params,
47         )
48         patch_haiku_lambda = _lambda.Function(

```

```

49         self, "PatchHaiku",
50         code=_lambda.Code.from_asset("api"),
51         handler="api.patch_haiku",
52         **common_params,
53     )
54     delete_haiku_lambda = _lambda.Function(
55         self, "DeleteHaiku",
56         code=_lambda.Code.from_asset("api"),
57         handler="api.delete_haiku",
58         **common_params,
59     )
60
61     ④
62     # grant permissions
63     table.grant_read_data(get_haiku_lambda)
64     table.grant_read_write_data(post_haiku_lambda)
65     table.grant_read_write_data(patch_haiku_lambda)
66     table.grant_read_write_data(delete_haiku_lambda)
67
68     ⑤
69     # define API Gateway
70     api = apigw.RestApi(
71         self, "BashoutterApi",
72         default_cors_preflight_options=apigw.CorsOptions(
73             allow_origins=apigw.Cors.ALL_ORIGINS,
74             allow_methods=apigw.Cors.ALL_METHODS,
75         )
76     )
77
78     haiku = api.root.add_resource("haiku")
79     haiku.add_method(
80         "GET",
81         apigw.LambdaIntegration(get_haiku_lambda)
82     )
83     haiku.add_method(
84         "POST",
85         apigw.LambdaIntegration(post_haiku_lambda)
86     )
87
88     haiku_item_id = haiku.add_resource("{item_id}")
89     haiku_item_id.add_method(
90         "PATCH",
91         apigw.LambdaIntegration(patch_haiku_lambda)
92     )
93     haiku_item_id.add_method(
94         "DELETE",
95         apigw.LambdaIntegration(delete_haiku_lambda)
96     )

```

- 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 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。
- ③ 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/bashoutter/api/api.py](#) にある。
- ④ <3> 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。

それぞれの項目について、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とき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認証なしで (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、一般公開されている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 (どのファイルが返されるかは、`website_index_document="index.html"` の部分で設定している。)

なお、この時点ではバケットは空である。HTML/CSS/JS など静的コンテンツの配置は、デプロイを行った後ほどのステップで行う。

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により、**Content Delivery Network (CDN)** や暗号化された HTTPS 通信を設定することができる。CloudFrontについての詳細は [公式ドキュメンテーション "What is Amazon CloudFront?"](#)などを参照いただきたい。



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、興味のある読者は次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。

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ハンドラ (handler) 関数とよぶ。GET /haiku の API に対してのハンドラ関数を Lambda で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     memory_size=512,  
6     **common_params  
7 )
```

簡単なところから見ていくと、`memory_size=512` の箇所でメモリーの使用量を512MBに指定している。また、`code=_lambda.Code.from_asset("api")` によって外部のディレクトリ (`api/`) を参照せよと指定しており、`handler="api.get_haiku"` のところで `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,と定義している。

次に,ハンドラ関数として使用されている `get_haiku()` のコードを見てみよう([handson/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 return {
17     "statusCode": status_code,
18     "headers": HEADERS,
19     "body": json.dumps(resp, cls=DecimalEncoder)
20 }
```

`response = table.scan()` で,俳句の格納された DynamoDB テーブルから,すべての要素を取り出している。もしな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,ほかの 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()` メソッドをよぶ必要がある。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にあ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,あるリソースがほかのリソースに対してどのような権限をもっているか,を規定するものである。Lambdaは,デフォルトの状態ではほか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。同様に,CDK の `s3.Bucket` オブジェクトにも `grant_read_write()` というメソッドが備わっており,これによってバケットへの読み書きを許可することができる。このメソッドは,実は [Chapter 9](#) で AWS Batch によるクラスターを構成した際に使用した。興味のある読者は振り返ってコードを確認してみよう。

i 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、このコードでは `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 や EC2 などに接続を行うという機能を担う (Figure 97)。Lambda や EC2 によって行われた処理の結果は、再び API Gateway を経由してクライアントに返される。このように、クライアントとバックエンドサーバーの間に立ち、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、あるいはリバースプロキシとよんだりする。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。しかし、API Gateway はサーバーレスなルーターとして、固定されたサーバーを配置することなく、API のリクエストが来たときのみ起動し、API のルーティングを実行する。サーバーレスであることの当然の帰結として、アクセスの件数が増大したときにはそれにルーティングの処理能力を自動で増やす機能も備わっている。

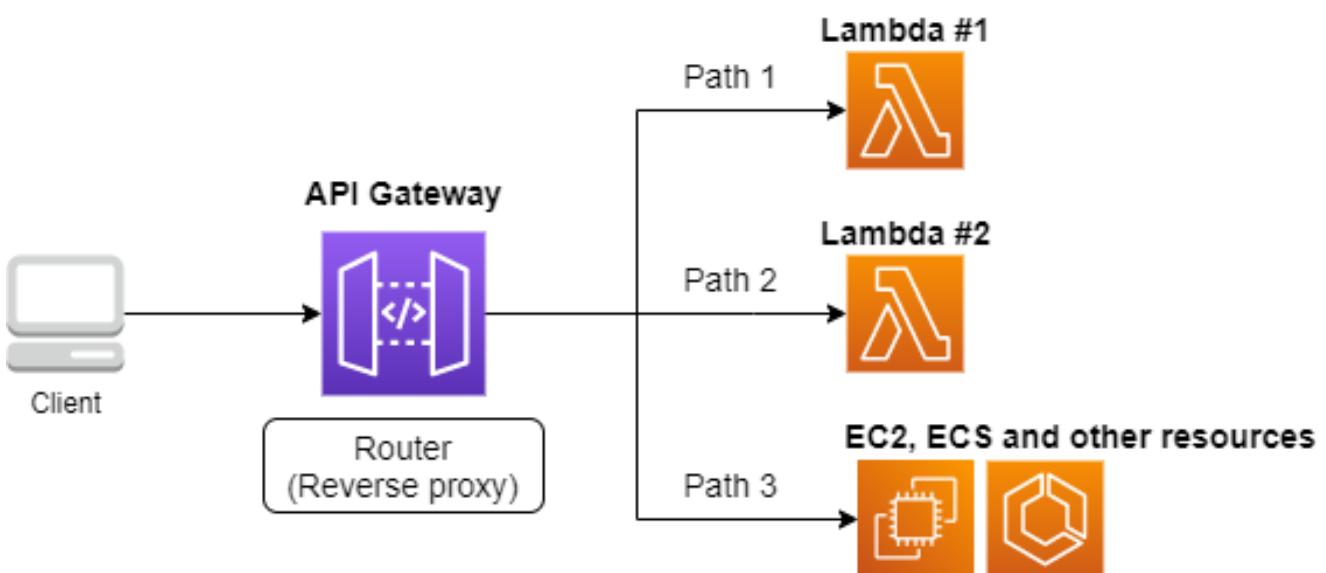


Figure 97. API Gateway

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が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 12](#) 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円で利用できる。

Table 12. 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 ①
2 api = apigw.RestApi(
3     self, "BashouterApi",
4     default_cors_preflight_options=apigw.CorsOptions(
5         allow_origins=apigw.Cors.ALL_ORIGINS,
6         allow_methods=apigw.Cors.ALL_METHODS,
7     )
8 )
9
10 ②
11 haiku = api.root.add_resource("haiku")
12 ③
13 haiku.add_method(
14     "GET",
15     apigw.LambdaIntegration(get_haiku_lambda)
16 )
17 haiku.add_method(
18     "POST",
19     apigw.LambdaIntegration(post_haiku_lambda)
20 )
21
22 ④
23 haiku_item_id = haiku.add_resource("{item_id}")
24 ⑤
25 haiku_item_id.add_method(
26     "PATCH",
27     apigw.LambdaIntegration(patch_haiku_lambda)
28 )
29 haiku_item_id.add_method(
30     "DELETE",
31     apigw.LambdaIntegration(delete_haiku_lambda)
32 )

```

- ① 最初に, `api = apigw.RestApi()` により, 空の API Gateway を作成している.
- ② 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.
- ③ 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.
- ④ 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.
- ⑤ 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, API Gateway の使い方は非常にシンプルで, 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.



このプログラムで新規 API を作成すると, 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 を接続するときに必要な設定である.

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). シークレットキーの設定も忘れずに (Section 15.3).

```

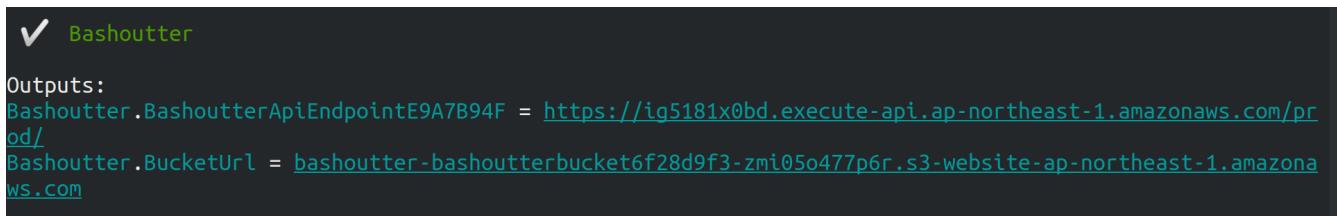
# プロジェクトのディレクトリに移動
$ cd intro-aws/handson/bashoutter

# 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

# 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 98 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` の二つ文字列はあとで使うのでメモしておこう。



```

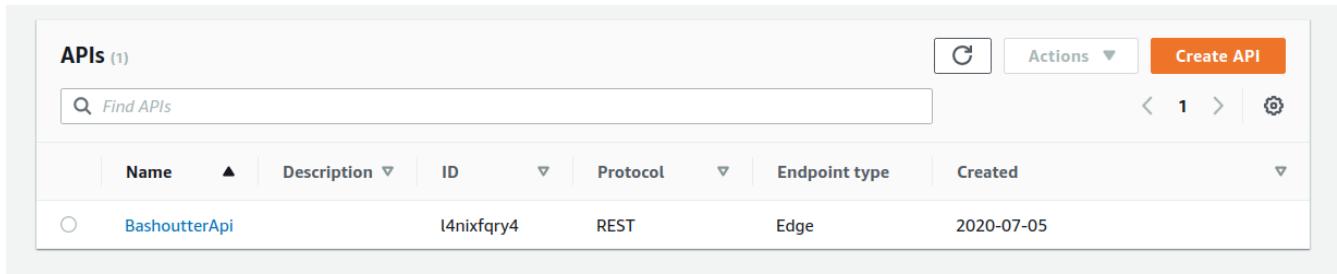
✓ Bashoutter

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 98. 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。まずは、コンソールから API Gateway のページに行く。すると、Figure 99 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
	BashoutterApi	l4nixfqry4	REST	Edge	2020-07-05

Figure 99. 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 100 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、前述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る。このハンズオンではとくにこれらの機能は使用しないが、"Method Request" と書いてある項目などがそれに相当する。次に、Figure 100 で画面右端の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ことに注目しよう。関数名をクリックと、該当する Lambda のコンソールに遷移し、関数の中身を閲覧することが可能である。

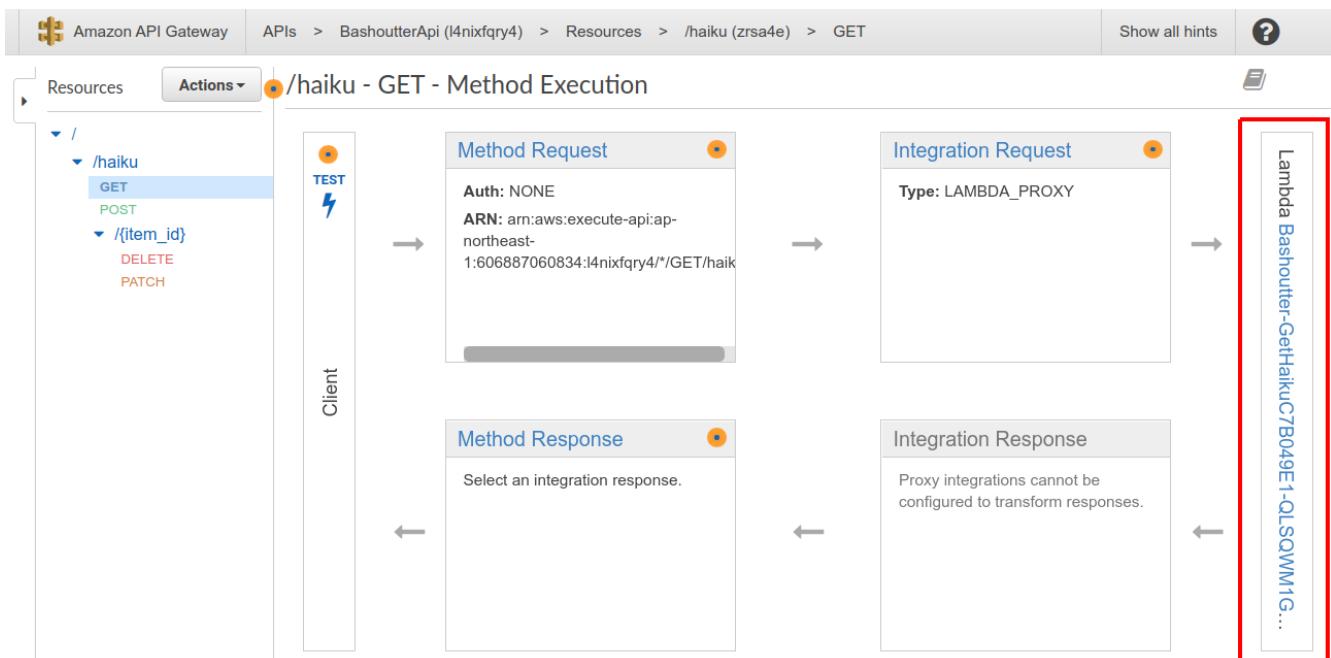


Figure 100. API Gateway コンソール画面 (2)

次に、S3 のコンソール画面に移ってみよう。bashouter- で始まるランダムな名前のバケットが見つかるはずである (Figure 101)。

The screenshot shows the AWS S3 console with the path: Buckets. The left sidebar includes options like Buckets, Batch Operations, Access analyzer for S3, Block public access (account settings), and Feature spotlight. The main area shows the following:

- Buckets (4)**: Buckets are containers for data stored in S3. Learn more
- Create bucket** button
- Buckets** table:

Name	Region	Access	Creation date
bashouter-bashouterbucket6f28d9f3-zml05o477p6r	Asia Pacific (Tokyo) ap-northeast-1	Public	July 7, 2020, 13:15 (UTC+09:00)

Figure 101. 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 102)。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 Public	Management	Access points
----------	------------	-----------------------	------------	---------------

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 102. S3 バケットの中身

13.4. API リクエストを送信する

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 リクエストを送信してみよう。まずはコマンドラインから API を送信する演習を行おう。S3 に配置した GUI は一旦おいておく。

ここではコマンドラインから HTTP API リクエスト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 (venv) を作成したとき、一緒にインストールされている。念のためインストールがうまくいっているか確認するには、仮想環境を立ち上げたあとコマンドラインに `http` と打ってみる。ヘルプのメッセージが出力されたら準備OKである。

まず、先ほどデプロイを実行したときに得られた 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=XXXX
```

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、`POST /haiku` を使って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"
```

次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。一つ前のコマンドで取得した俳句の item_id を、次のコマンドの XXXX に代入した上で実行しよう。

```
$ http PATCH "${ENDPOINT_URL}/haiku/XXXX"
```

{"description": "OK"} という出力が得られるはずである。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 の値で置き換えたうえで次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.

これで, 俳句の投稿・取得・削除そしていいねの追加, といった基本的な 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一つずつ俳句を投稿した. 多数のユーザーがいるような SNS では, 1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自動的に構築されている. このポイントを実証するため, 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/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. このプログラムを使用すると, POST /haiku の API リクエストを指定された回数だけ実行することができる.

テストとして, API を300回実行してみよう. 次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. したがって, 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1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?



先述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, 次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, これと同じことがウェブブラウザの背後で行われ, ページのコンテンツが表示されている ([Figure 75 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.

CDK のコードで, Public access mode の S3 バケットを作成したことを思い出してほしい. 最初のステップとして,

ここにウェブサイトのコンテンツをアップロードしよう。ハンズオンのソースコードの中に `gui/dist` というフォルダが見つかるはずである。ここにはビルド済みのウェブサイトの静的コンテンツ (HTML/CSS/JavaScript) が入っている。AWS CLI のコマンドを使うことでこれらのファイルを S3 にアップロードしよう。

```
$ aws s3 cp --recursive ./gui/dist s3://<BUCKET_NAME>
```

コマンドを実行する際は、Bashoutter ハンズオンのディレクトリから行うこと (`./gui/dist` に注目), そして `<BUCKET_NAME>` にはデプロイした自身のバケットの名前が入る点に注意。念のため, AWS コンソールにログインし, バケットにファイルがアップロードされている点を確認しておこう。



なお, 今回は GUI の説明はとくに行わないが, Bashoutter 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。Vue を使うことで, Single page application (SPA) の技術でウェブサイトの画面がレンダリングされる。ソースコードは [handson/bashoutter/gui](#) のディレクトリの中にあるので, 興味のある読者は確認してみるとよい。

アップトードが完了したところで, 続いてデプロイを実行したときにコマンドラインの出力を見直してみよう。
`Bashoutter.BucketUrl=` で与えられた URL が見つかるはずである ([Figure 98](#))。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, アドレスバーに S3 の URL を入力しへアクセスしてみよう。すると, [Figure 103](#) のようなページが表示されるはずである。

The screenshot shows the Bashoutter GUI. At the top, there is a blue header bar with the text "Bashoutter". Below it is a form with three input fields for a haiku (五, 七, 五) and a text field for "Your name". To the right of the name field is a "POST" button. Below the form is a section titled "People's Haikus" containing a list of haikus. A "REFRESH" button is located to the right of the list.

Figure 103. "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 GUI の仕様になっている)。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。

13.7. アプリケーションの削除

これで, Bashoutter プロジェクトが完成した! この SNS 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る。また, [Section 13.5](#) で見たように, 大量のユーザーの同時アクセスによる負荷がかかっても, 柔軟にスケールが行われ遅延なく処理を行うことができる。極めて簡素ながらも, 立派なウェブサービスとしてのスペックは満たしているのである!

Bashoutter アプリを存分に楽しむことができたら、最後に忘れずにスタックを削除しよう。

コマンドラインからスタックの削除を実行するには、次のコマンドを使う。

```
$ cdk destroy
```

CDK のバージョンによっては S3 のバケットが空でないと、`cdk destroy` がエラーを出力する場合がある。この場合はスタックを削除する前に、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うえで、すべてのファイルを選択し、"Actions" → "Delete" を実行すればよい。



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
```

13.8. 小括

ここまでが、本書第三部の内容であった。

第三部では、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、説明を行った。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。[Chapter 12](#) では、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、[Chapter 13](#) 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。

これらの演習を通じて、世の中のウェブサービスがどのようにしてでき上がっているのか、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、今回のハンズオンがそ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: 環境構築

本書を読み進めるにあたって、ハンズオンのプログラムを実行するための環境を自分のローカルマシンにセットアップしなければならない。ここでは、AWS やコマンドラインの初心者を想定して、本章で必要なソフトウェアやライブラリのインストールなどを簡単に解説する。以下に簡単な目次を示そう。既に環境構築が済んでいる場合は適宜読み飛ばしていただき、関係のある箇所のみ目を通せば良い。

- AWS アカウントの取得 ([Section 15.1](#))
- AWS シークレットキーの作成 ([Section 15.2](#))
- AWS CLI のインストール ([Section 15.3](#))
- AWS CDK のインストール ([Section 15.4](#))
- WSL のインストール ([Section 15.5](#))
- Docker のインストール ([Section 15.6](#))
- Python venv クイックガイド ([Section 15.7](#))
- ハンズオン実行用の Docker image の使い方 ([Section 15.8](#))

使用する OS は Linux/Mac/Windows のどれを用いても構わない。Windows のユーザーは、Windows Subsystem for Linux (WSL) を使用することを想定している ([Section 15.5](#))。

また、本書のハンズオンを実行するための [Docker イメージ](#) を提供している。これを用いると、AWS CLI/CDK や Python の設定などをスキップできるので、Docker の使用方法を知っている読者には便利だろう。

15.1. AWS アカウントの取得

本書で提供するハンズオンを実際に自分で試すには、読者自身で AWS のアカウントの作成をする必要がある。詳しいアカウントの作成の手順は [公式のドキュメンテーション](#) に書かれているので、そちらも参照していただきたい。以下の手順に従ってアカウントの作成を行う。

まず、ウェブブラウザから [AWS コンソール](#) にアクセスし、右上の [Create an AWS Account](#) をクリックする ([Figure 104](#) で実線で囲った部分)。

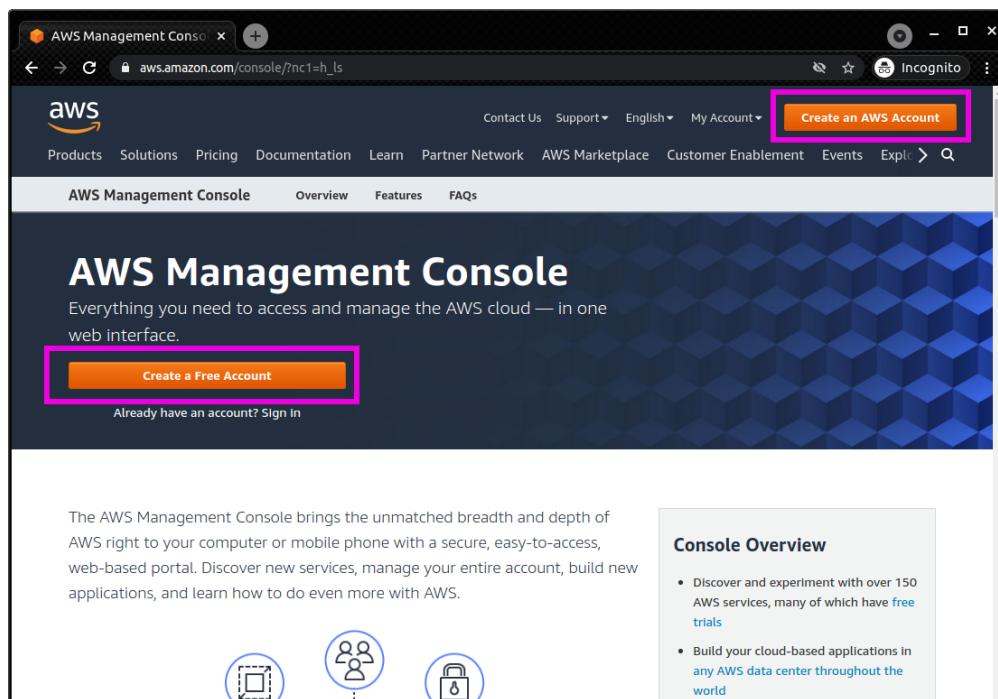


Figure 104. サインアップ (1): AWS コンソールにアクセス

次に,遷移した先のページでメールアドレスとパスワードなどの登録を行う(Figure 105).

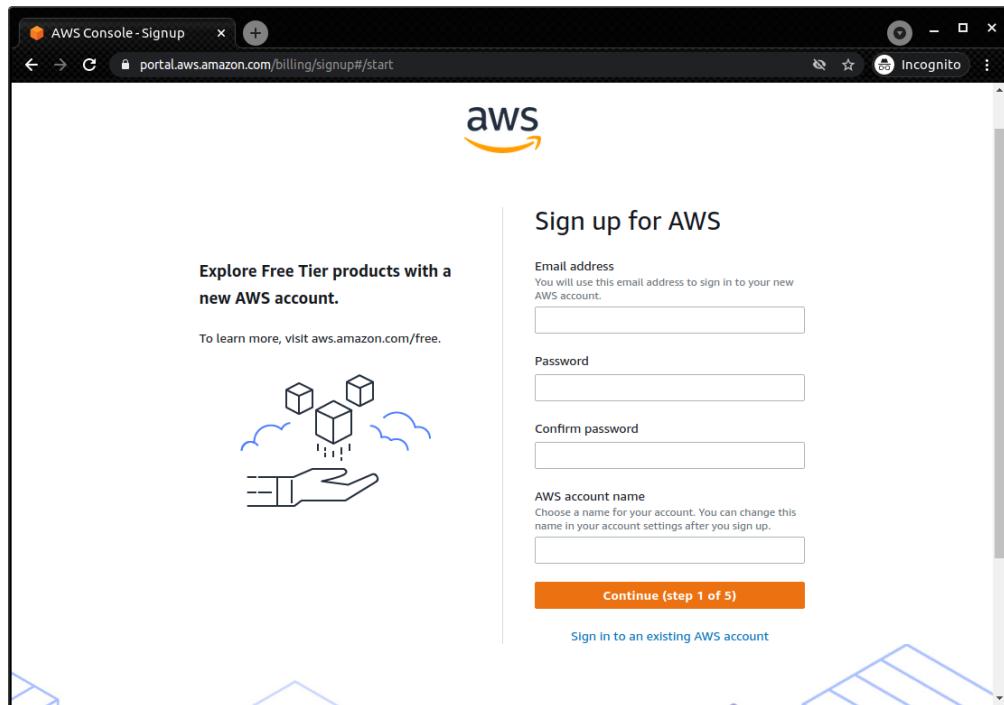


Figure 105. サインアップ(2): メールアドレス・パスワードなどの登録.

続いて,住所や電話番号などを訊かれるので,すべて入力しよう(Figure 106).

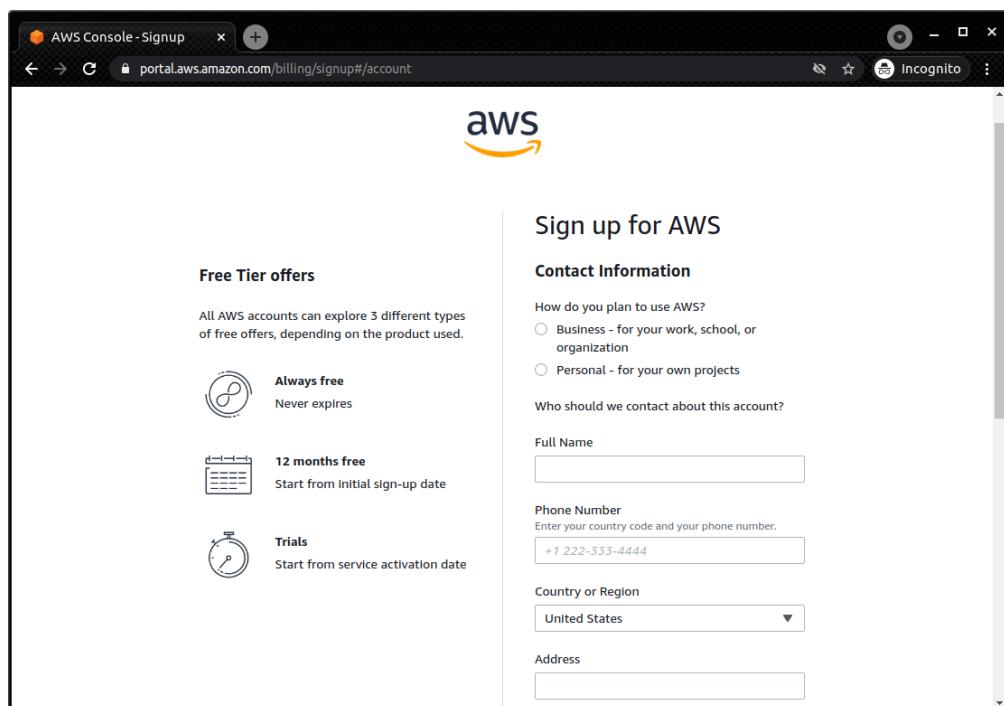


Figure 106. サインアップ(3): 住所・電話番号の入力

次に,クレジットカードの情報の登録を求められる(Figure 107). 個人でAWSを利用する場合は,利用料金の請求はクレジットカードを経由して行われる. クレジットカードの登録なしにはAWSを使い始めることはできないことに注意.

The screenshot shows the 'Sign up for AWS' page. On the left, there's a 'Secure verification' section with a note: 'We will not charge for usage below AWS Free Tier limits. We temporarily hold \$1 USD/EUR as a pending transaction for 3-5 days to verify your identity.' Below this is a shield icon with a checkmark. On the right, the 'Billing Information' section includes fields for 'Credit or Debit card number', payment method icons (VISA, MasterCard, AMEX, DISCOVER), 'Expiration date' (Month and Year dropdowns), 'Cardholder's name' (text input), 'Billing address' (radio button for 'Use my contact address' selected, showing '1919-1 Tancha, Onna-son, Kunigami-gun Okinawa 904-0412 JP'), and 'Use a new address' (radio button). The AWS logo is at the top.

Figure 107. サインアップ (4): クレジットカードの登録

次の画面では、携帯電話の SMS またはボイスメッセージを利用した本人確認が求められる (Figure 108). 希望の認証方法を選択し、自分の携帯電話番号を入力しよう。

The screenshot shows the 'Sign up for AWS' page with the 'Confirm your identity' section. It features a user icon with a checkmark. The text explains that before using the AWS account, the user must verify their phone number. It asks how to send the verification code: 'Text message (SMS)' (selected) or 'Voice call'. Below this are fields for 'Country or region code' (United States (+1)) and 'Mobile phone number'. At the bottom is a 'Security check' section with a CAPTCHA field containing 'w7xxcy' and two buttons.

Figure 108. サインアップ (5): 携帯電話による本人確認

無事に本人確認が完了すると、最後にサポートプランの選択を求められる (Figure 109). 無料の Basic support を選択しておけば問題ない。

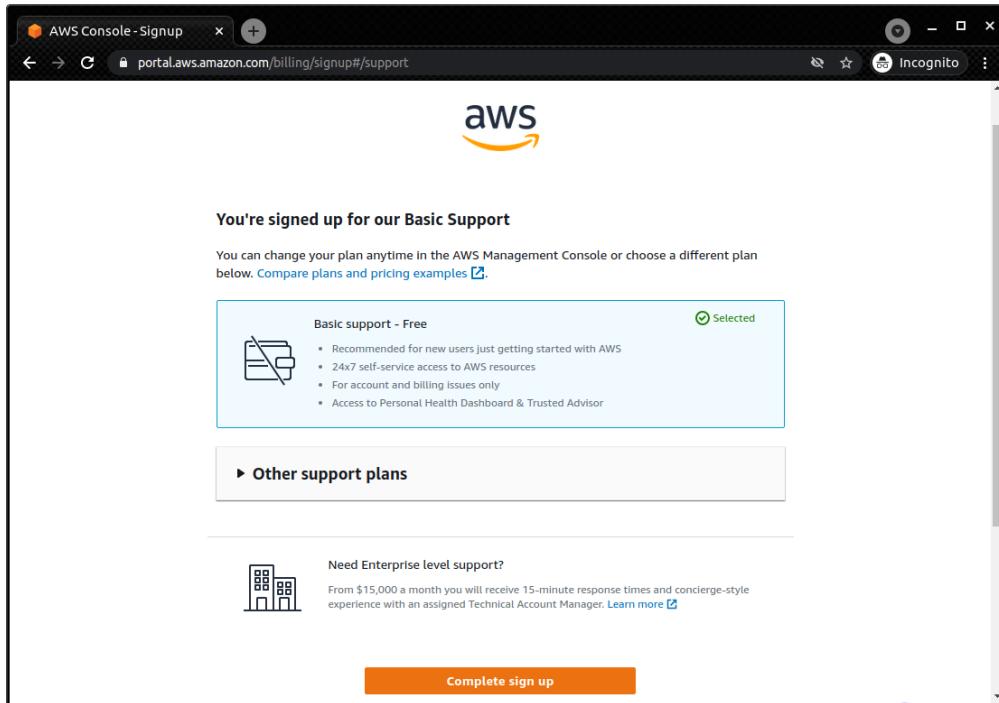


Figure 109. サインアップ (6): サポートプランの選択

以上のステップにより、アカウントの作成が完了する (Figure 110). 早速ログインをして、AWS コンソールにアクセスできるか確認しておこう。

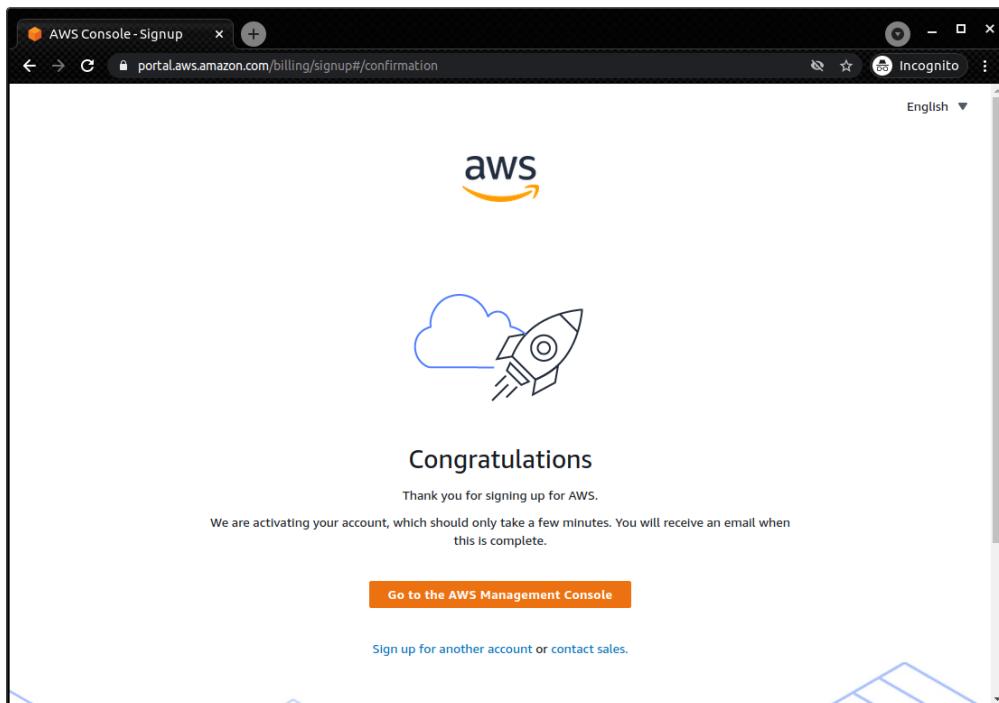


Figure 110. サインアップ (7): アカウントの作成が完了した

15.2. 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 を操作するときに、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 "Understanding and getting your AWS credentials"](#) 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 111)

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 112)
4. 表示された Access key ID, Secret access key を記録しておく(画面を閉じると二度と表示されない).
5. 鍵を忘れてしまった場合などは,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 15.3](#)).

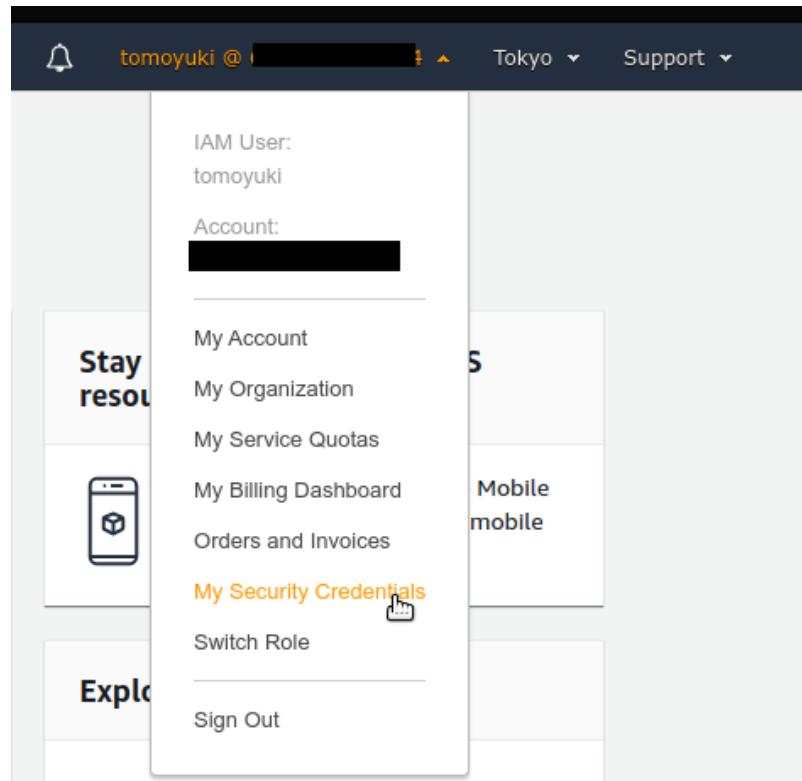


Figure 111. AWS シークレットキーの発行1

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 112. 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、次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 113).
- [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 114). ここで表示された内容を `~/.aws/credentials` にコピーする (Section 15.3 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。

```
[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よい。`default` 以外の名前を使用する場合は、コマンドを実行するときにプロファイル名を指定する必要がある (詳しくは Section 15.3).



The screenshot shows the 'Your AWS Account Status' section of the vocareum AWS Educate Starter Account. It displays the following information:

- Active**: full access (blurred)
- \$100**: remaining credits (estimated)
- 1:47**: session time

Below this, there are two buttons: 'Account Details' and 'AWS Console'.

Figure 113. vocareum コンソール

The screenshot shows the 'AWS CLI:' section of the vocareum AWS Educate Starter Account. It contains the following text:

```
AWS Access
Session started at: 2021-06-20T18:29:05-0700
Session to end at: 2021-06-20T21:29:05-0700
Remaining session time: 2h18m12s

AWS Starter account
Term: 364 days 23:13:23

AWS CLI:
Copy and paste the following into ~/.aws/credentials
[default]
aws_access_key_id=blurred
aws_secret_access_key=blurred
aws_session_token=blurred
```

Figure 114. vocareum から AWS シークレットキーの発行

15.3. 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、次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、次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 15.2](#) 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。

こ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。念のため、`cat` コマンドを使って中身を確認してみるとよい。

```
$ 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
```

`~/.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、たとえば `[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よい。

```
$ 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
```

これらの環境変数は、`~/.aws/credentials` よりも高い優先度をもつので、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。

15.4. 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、基本的に次のコマンドを実行すればよい。

```
$ 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、次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、次のコマンドにより 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 15.3](#) 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 15.3](#) を参照。

15.5. WSL のインストール

本書のハンズオンではコマンドラインから AWS CLI のコマンドを実行したり、Python で書かれたプログラムを実行する。コマンドは基本的に UNIX のターミナルを想定して書かれている。Linux や Mac のユーザーは OS に標準搭載されているターミナルを用いれば良い。Windows を利用している読者は、[Windows Subsystem for Linux \(WSL\)](#) を利用することで、仮想の Linux 環境を構築することを推奨する。[Cygwin](#) などの Linux 環境をエミュレートするほかのツールでも構わないが、本書のプログラムは WSL でのみ動作確認を行っている。

WSL とは、Windows の OS 上で Linux の仮想環境を起動するための、Microsoft 社が公式で提供しているソフトウェアである。Ubuntu など希望の Linux distribution が選択でき、基本的にすべての Linux 向けに作られたプログラム・ソフトウェアを使用することができる。

執筆時点では WSL 2 が最新版として提供されているので、以下では WSL 2 のインストール手順を簡単に説明する。細かな詳細などは、[公式ドキュメンテーション](#) を参照のこと。

前提として、使用される OS は Windows 10 (Pro または Home エディション) でなければならない。さらに、使用している Windows 10 のバージョンが WSL に対応するバージョンであるかを確認する。X64 のシステムでは Version 1903, Build 18362 以上でなければならない。バージョンが対応していない場合は、Windows のアップデートを行う。

まず最初に、Administrator 権限で PowerShell を起動する (Figure 115)。左下の Windows メニューの検索バーに `powershell` と入力すると、PowerShell のプログラムが見つかるはずである、これを右クリックし、`Run as administrator` を選択し起動する。

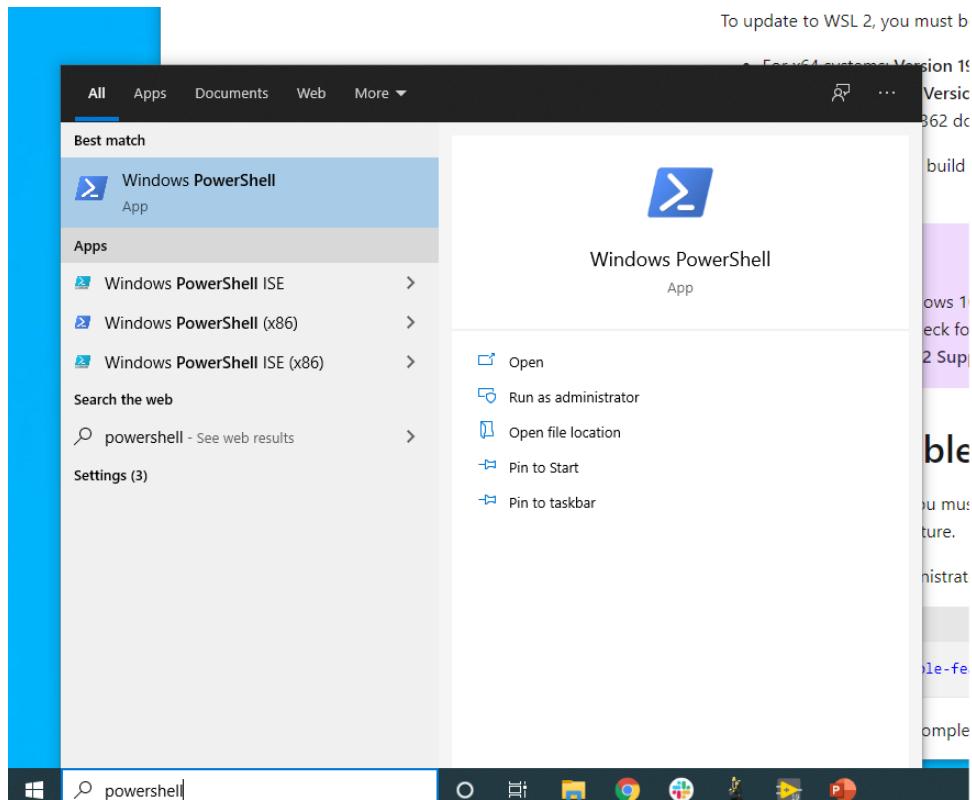


Figure 115. 管理者権限での PowerShell の起動

PowerShell が起動したら、次のコマンドを実行する。

```
dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /norestart
```

実行して、“The operation completed successfully.” と出力されるのを確認する。これで WSL が enable される。

次に、先ほどと同じ Administrator 権限で開いた PowerShell で次のコマンドを実行する。

```
dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /norestart
```

実行して、“The operation completed successfully.” と出力されるのを確認する。これが確認出来たら、一度コンピュータを再起動する。

続いて、Linux kernel update package を次のリンクからダウンロードする。

https://wslstorestorage.blob.core.windows.net/wslblob/wsl_update_x64.msi

ダウンロードしたファイルをダブルクリックして実行する。ダイアログに従ってインストールを完了させる。

そうしたら、再び PowerShell を開き次のコマンドを実行する。

```
wsl --set-default-version 2
```

最後に、自分の好みの Linux distribution をインストールする。ここでは Ubuntu 20.04 をインストールしよう。

Microsoft store のアプリを起動し、検索バーに **Ubuntu** と入力する。Ubuntu 20.04 LTS という項目が見つかるはずなので、それを開き、“Get” ボタンをクリックする (Figure 116)。しばらく待つと、Ubuntu 20.04 のインストールが完了する。

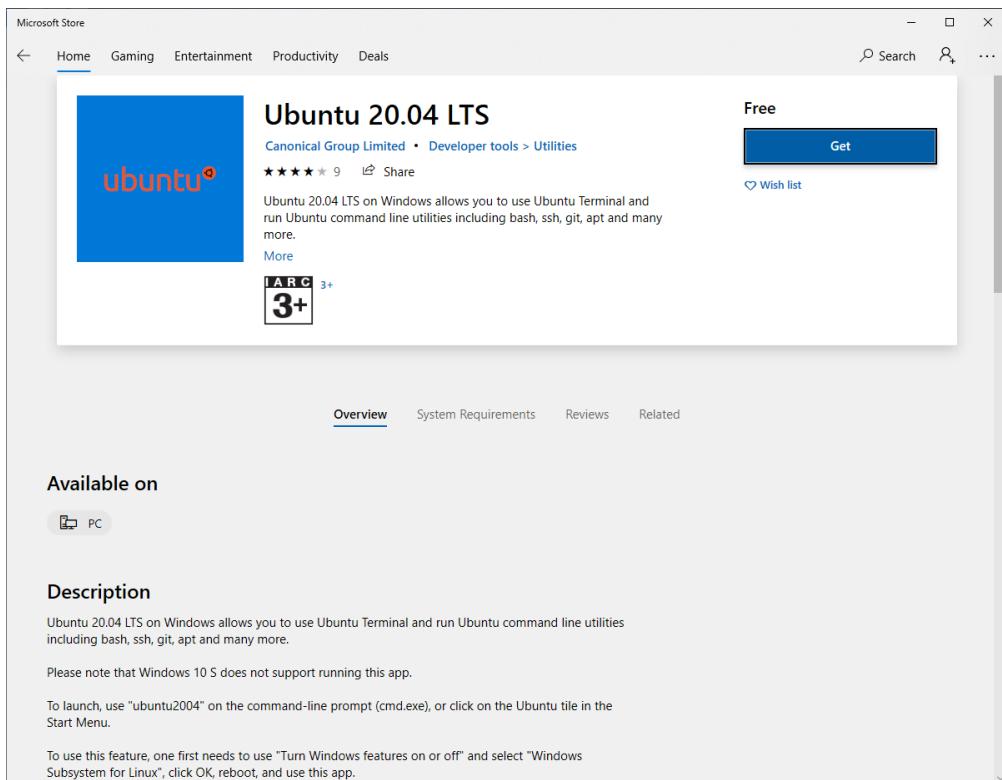


Figure 116. Microsoft store から Ubuntu 20.04 をインストール

Ubuntu 20.04 を初回に起動すると、初期設定が自動で開始され、数分待つことになる。初期設定が終わると、ユーザー名・パスワードを設定するようプロンプトが出るので、プロンプトに従い入力する。

これで WSL2 のインストールが完了した。早速 WSL2 を起動してみよう。左下の Windows メニューの検索バーに **Ubuntu** と入力すると、Ubuntu 20.04 のプログラムが見つかるはずである (Figure 117)。クリックして起動しよう。

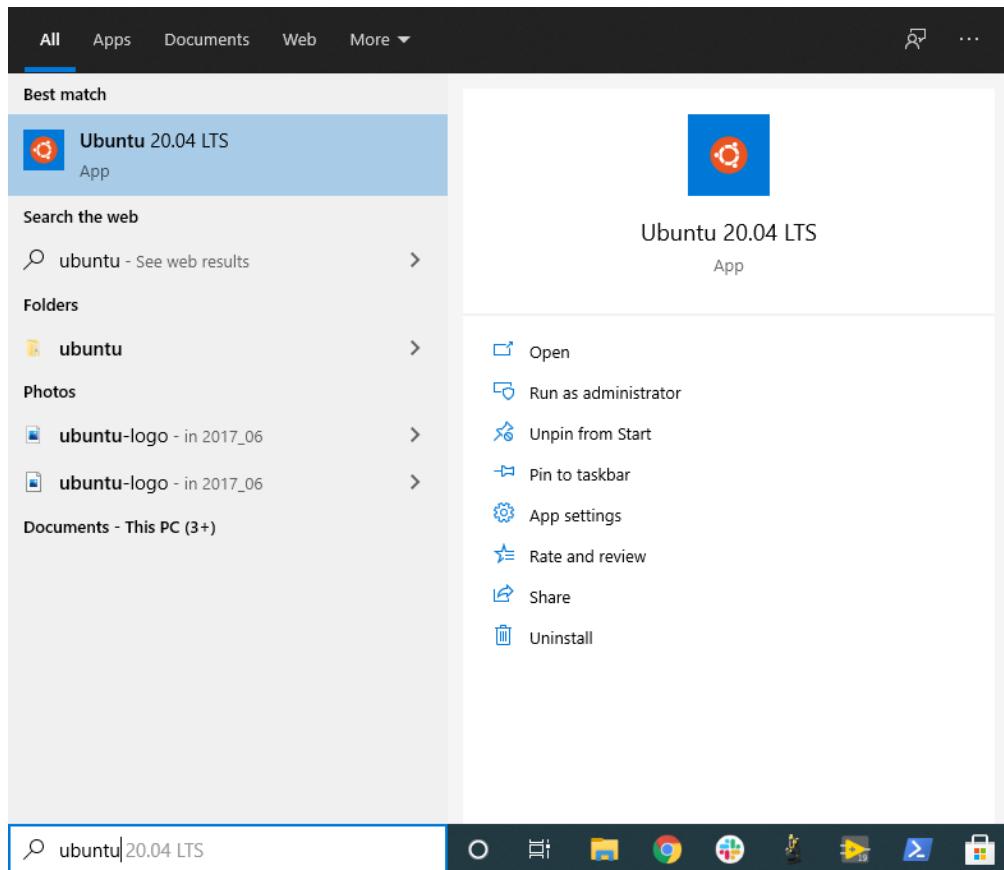


Figure 117. Ubuntu 20.04 の起動

すると、ターミナルの黒い画面が立ち上がるだろう (Figure 118). `ls`, `top`などのコマンドを打ってみて、WSL がきちんと動作していることを確認しよう。

```
tomoyuki@caspar: ~
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 4.19.128-microsoft-standard x86_64)

 * Documentation: https://help.ubuntu.com
 * Management: https://landscape.canonical.com
 * Support: https://ubuntu.com/advantage

System information as of Thu Jul 29 17:36:09 JST 2021

System load: 4.11      Processes: 21
Usage of /: 1.5% of 250.98GB   Users logged in: 0
Memory usage: 5%          IPv4 address for eth0: 172.20.127.126
Swap usage: 0%

=> There is 1 zombie process.

304 updates can be installed immediately.
136 of these updates are security updates.
To see these additional updates run: apt list --upgradable

The list of available updates is more than a week old.
To check for new updates run: sudo apt update

This message is shown once once a day. To disable it please create the
/home/tomoyuki/.hushlogin file.
tomoyuki@caspar:~$
```

Figure 118. WSL の起動画面

オプションとして、[Windows Terminal](#) というマイクロソフトから提供されているツールを使うと、より快適に WSL を使用することができる。興味のある読者はこちらのインストールも推奨する。

15.6. Docker のインストール

Docker のインストールの方法は OS によって異なる。

Mac ユーザーは、Docker Desktop をインストールする。インストールの方法は、[Docker のウェブサイト](#) から、Mac 版の Docker Desktop をダウンロードし、ダウンロードされたファイルをダブルクリックし、[Applications](#) のフ

オルダにドラッグするだけで良い。詳細は[公式ドキュメンテーション](#)を参照のこと。

Windows のユーザーは、Docker Desktop をインストールする。その際、WSL 2 が事前にインストールされなければならない。詳細は[公式ドキュメンテーション](#)を参照のこと。Docker Desktop をインストールすると、WSL からも `docker` コマンドが使用できるようになる。

Linux ユーザー（特に Ubuntu ユーザー）については、インストールの方法はいくつかのアプローチがある。[公式ドキュメンテーション](#)にいくつかのインストールの方法が示されているので、詳しい情報はそちらを参照いただきたい。

最も簡単な方法は、Docker が公式で提供しているインストールスクリプトを用いる方法である。この場合、次のコマンドを実行することで Docker がインストールされる。

```
$ curl -fsSL https://get.docker.com -o get-docker.sh  
$ sudo sh get-docker.sh
```

デフォルトのインストールでは、root ユーザーのみが `docker` コマンドを使用できる設定になっている。従って、コマンドには毎回 `sudo` を付け加える必要がある。これが面倒だと感じる場合は、次のステップにより、使用するユーザーを `docker` というグループに追加する（詳細は[公式ドキュメンテーション "Post-installation steps for Linux"](#) を参照）。

まず最初に、`docker` という名前にグループを追加する。インストールによっては、既に `docker` グループが作られている場合もある。

```
$ sudo groupadd docker
```

次に、現在使用しているユーザーを `docker` グループに加える。

```
$ sudo usermod -aG docker $USER
```

ここまでできたら、一度ログアウトし、再度ログインする。これによって、グループの変更がターミナルのセッションに反映される。

設定が正しくできているかを確認するため、次のコマンドを実行してみる。

```
$ docker run hello-world
```

`sudo` なしでコンテナが実行できたならば、設定は完了である。

15.7. 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 仮想環境とよ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 119)。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 119. `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。

15.8. 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 イメージは [Docker Hub](#) においてある。Docker イメージのビルドファイルは GitHub の [docker/Dockerfile](#) にある。

次のコマンドでコンテナを起動する。

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

初回にコマンドを実行したときのみ、イメージが 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イメージが使用される。

コンテナが起動すると、次のようなインター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 15.3 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コンテナにマウントすることで、同じ認証ファイルをコンテナ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、次のコマンドでコンテナを起動する。

```
$ docker run -it -v ~/.aws:/root/.aws:ro tomomano/labc:latest
```

これにより、ローカルマシンの `~/.aws` をコンテナの `/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/` がコンテナ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 鍵をコンテナ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

- ・香取真知子氏 - ハンズオンプログラムの動作確認、文章校閲

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。

