AWSでのインフラコード管理への取り組み

ビジネス統括本部
デジタルソリューションビジネスユニット
クラウドソリューション部　第一グループ
宮崎　大輔

1. はじめに

昨今は、AWS（Amazon Web Services）やMicrosoft Azure、GCP（Google Cloud Platform）などの各IaaSの普及により、システムをクラウドに構築したり、オンプレミスからクラウドに移行したりする企業が増えている。そのため、システムを構築する上での過去の常識がそのままでは通用しなくなってきている。これまでのオンプレミスのサーバーは想定されるキャパシティの最大値相当のハードウェアをあらかじめ購入し、設計および構築をしておく必要があった。しかし、主なクラウドサービスは従量課金型であり、必要に応じて必要なリソースを構築もしくはスケールアウトし、不要になったサービスは削除することでコストを抑えることができる。これらの作業は適切な設計で作りこんでおけばクラウド側のフルマネージドで行うことも可能である。

そして、各クラウドベンダーは各種APIやCLIを用意しているため、その機能を使って作りこめば、オンプレミス時代には全て個別に管理運用する必要があったものを、ある程度自動化し、管理することも可能である。

この特性を活かして、当社でも2012年より、AWSを利用したサービス「enterpriseCloud+」を提供している。このサービスは主に、AWSの仮想サーバーサービスであるAmazon EC2の利用と運用をサポートする機能を付加した管理コンソールを提供している。サーバー起動・停止・再起動、バックアップ・リストア、スペック変更や、これらを自動で実行できるスケジュール機能、サーバー作成時に自動で監視設定がされるなど、AWS特有の知識や技術を必要とせずともEC2を利用できるようになっている。

しかし、ビジネスのニーズは様々で、複数のアカウントを同時に管理したい、複数のクラウドサービスをまたがって管理したい、開発したツールが対応していない機能を使いたいという要望が出てくるケースもある。当社が提供しているenterpriseCloud+管理コンソール（以下eC+コンソール）を利用している顧客でも、eC+コンソール対応外の様々なAWSサービスを利用していこうという変化も伺える。その場合、利用までに様々な調整事項や技術・人的コストが発生し、迅速に利用できるというクラウドの利点を活かせない可能性がある。

本稿では、そういった様々なニーズに対して広い範囲で対応できるTerraform（OSS版）の紹介を行う。

2. Terraformの紹介

2.1 Terraformとは

Terraformは、HashiCorp社が開発しているマルチクラウド対応のインフラ構成管理ツールである。コードによってインフラを管理する「Infrastructure as Code」を具現化したものだ。AWSはもちろん、様々なIaaS、SaaSに対応している。OSS版からEnterprise版まであり、OSS版を使用する場合でも有償だがサポートを得ることができる。

本稿では、Terraformをスモールスタートで利用する方法を紹介する。Terraformに初めて触れる、かつすでにある程度AWS上にリソースが作られている環境で実際に使うことを前提とする。

2.2 Terraformの特徴

TerraformはGo言語で作られており、実行環境に適したバイナリを1つ導入するだけで実行が可能である。HCLという独自の言語でインフラの設計をコードとして定義する。そのため、一見敷居が高そうに見えるが、Visual Studio CodeやIntelliJなど著名なエディタにTerraform用のプラグインが用意されているため、構文のミスも発生しづらい。またテンプレートになりうる初回コードさえできてしまえば、その後は一部のパラメータを変更することで簡単に同様の構築ができる。また、AWSだけでなく、Azure、GCPなどにも対応しており、複数のIaaSにまたがる業務であっても、ほとんど同じような設計、実行手順で管理できるようになる。

また、実行時のディレクトリにある拡張子が「tf」のファイルは全て読み込まれる仕様のため、運用の現場の文化に合わせてコードを分割、記述することで比較的に受け入れられやすい形で導入が可能である。Terraformで管理する単位はコードを置くディレクトリとなるため、ある程度の評価をした後、ディレクトリ設計を行って管理の方針と単位を決めれば、すでに別のツールで管理済みの環境にも導入が可能である。コードが正しく定義されていることが前提となるが、リソースの依存関係をコードから自動的に解析し、それに従って順番にリソースを作成する機能もTerraformには備わっている。

2.3 eC+コンソールとTerraformの共存

Terraformは「data source」と呼ばれる機能がある。この機能を使えば、Terraform管理対象外のリソース情報を取得し、Terraform内でそのリソース情報を参照できる。この機能を有効活用することで、EC2インスタンスやセキュリティグループなどのEC2関連リソースはeC+コンソール、それ以外はTerraformで構築など住み分けが可能である。実際のコードは人間が記述するため、既存リソースを一意にフィルタリングするためにユニークなタグを必要に応じて追加しておくことで管理の負荷を下げることができる。実際の役割分担を以下の図1に示す。

eC+コンソールはEC2の利用・管理機能を主体としたサービスであるため、eC+コンソールが対応できない範囲をTerraformで補う形で導入する。まず、VPCやサブネット、ルートテーブルなどを先にTerraformで構築し、eC+コンソールでの利用の前提条件となるリソースを構築する。EC2に関連するリソースはeC+コンソールで構築を行うが、RDSなどeC+コンソールが対応していないリソースをTerraformでeC+コンソール利用開始後に構築する。この時の管理単位は顧客や実際に運用を行うチームのニーズに合わせて定義する。

さらに高度なクラウドのマネージドサービスを活用したシステムを構築したいという要望がある場合は、eC+コンソールでは対応しきれないため、Terraformで全てを構築する。その場合の管理単位を図2に示す。

AWSにおけるVPCなどの共通で使う基盤にあたるリソースは頻繁に変更を行うものではないため、単独の実行単位として切り出し、実際の業務システムが乗るシステムは他社でも行われているようなサービス単位で大きく分割し、さらに要件に応じて細かく実行単位を分ける。

次項では、すでに別の手段でAWS上のリソースを構築している環境にTerraformを導入することを前提とし、Terraformを順次導入して活用していく手法について解説する。

3. Terraform（OSS版）の利用

3.1 Terraformの導入

Terraformの導入は、「2.2Terraformの特徴」に記したとおり非常に簡単である。実行環境に対応したZipファイルをダウンロードして展開し、パスが通る場所にバイナリを1つ配置するだけである。

また、実行にあたっては、AWS CLIが実行可能なプロファイルの設定が完了しているか、実行環境に適切なIAMロールが付与されている必要がある。OSS版の仕様上、実際に使

図1 eC+コンソールとTerraformの役割分担

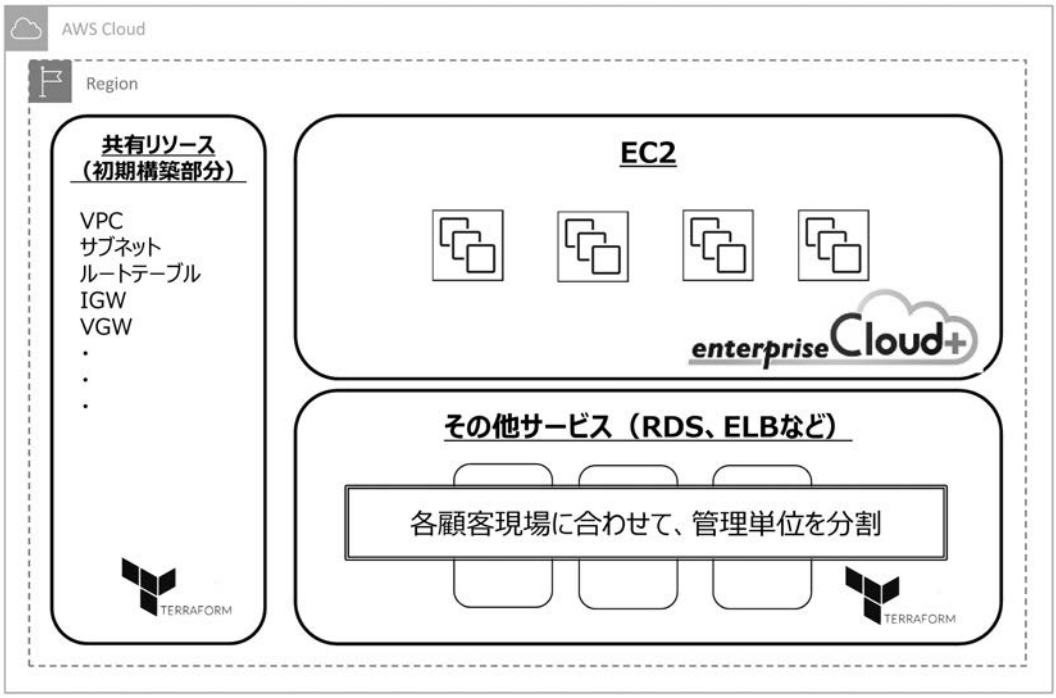
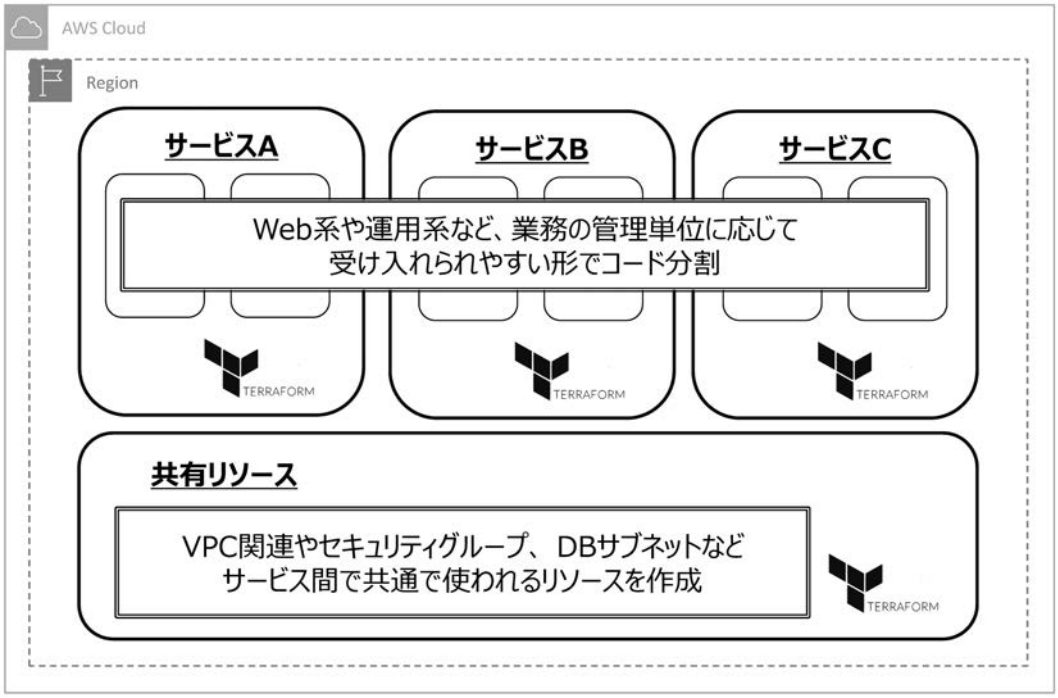


図2　Terraformのみでの管理単位



い込むにあたっては管理者相当の権限が付与されている必要がある。OSS版を実際の運用に乗せるには実行ユーザーを切り替えてIAMユーザーレベルで権限の絞り込みを行うか、専用の環境（ECSなど）を用意し、そこでのみ実行できるようにするなど設計の作りこみが必要になると考えられる。細かいアクセス権の設定を行いたい場合は、Terraform Enterpriseの導入検討を推奨する。

3. 2 tfstate管理用バケットの作成

今回はOSS版の利用となるため、terraform.tfstate（以下tfstate）を保存するためのバックエンドに利用するS3バケットを手動で作成する。Terraform Enterpriseを利用する場合はバックエンドの配慮は必要ない。デフォルトではtfstateは実行時のローカルディレクトリに保存される。作業者が一人の場合はデフォルトのままで問題ないが、本稿ではよく使われるS3をバックエンドとして指定する。参考までにTerraform Enterpriseを使用する場合は、専用のバックエンドの指定を行う。注意点として、S3バケット名はAWS上の既存バケット名の中でユニークである必要があるため、試験的に使用する場合は本番で使われる可能性のあるバケット名を避けることを推奨する。

3. 3 ミニマムなコードの作成と実行

Terraformでよく使うコマンドは以下となる。非常にシンプルであることがわかる。

表1のコマンドを用いて、今回はMySQLを使ったシンプルなRDSの作成を例とする。VPCやDBサブネット、セキュリティグループは作成済みの前提で作業を開始する。まず、挙動を理解するために、RDSを作成するのに必要なリソースの情報を取得するコードを記述する。リソースIDの動的取得には

表1　Terraformでよく使うコマンドの一覧

#	コマンド	説明
1	terraform init	Terraformの初回処理やモジュールのダウンロード時に実行するコマンド
2	terraform init-upgrade	Terraformのモジュールを最新版に更新する際に実行するコマンド
3	terraform plan	コードの内容を反映した場合の変更内容を出力、確認するコマンド
4	terraform apply	コードの内容を反映するコマンド
5	terraform plan-destroy	管理下のリソースを削除した場合の変更内容を出力、確認するコマンド
6	terraform destroy	管理下のリソースを削除するコマンド
7	terraform state list	管理下のリソース一覧を出力するコマンド
8	terraform state show	管理下のリソースの内容を表示するコマンド。最後にリソース名を付与する
9	terraform state rm	管理下のリソースの内容を削除するコマンド。最後にリソース名を付与する
10	terraform state mv	管理下のリソース名を変更するコマンド。最後に変更前のリソース名と変更後のリソース名を付与する
11	terraform import	コードに定義した内容に既存リソースをインポートするコマンド　※リソースタイプによってオプションは異なる

図3 「main.tf」のコマンド記述例

```
# main.tf
// バックエンドの指定
terraform {
  required_version = ">= 0.11.8"

  backend "s3" {
    bucket = "cac-terraform-test-tfstate" // 作成したS3バケット名
    key    = "RDS/terraform.tfstate"
    region = "ap-northeast-1"
    // profile = "" # AWS CLIのプロファイルを指定する場合はここで指定
  }
}

// プロバイダの指定
provider "aws" {
  region  = "ap-northeast-1"
  // profile = "" # AWS CLIのプロファイルを指定する場合はここで指定
}


// セキュリティグループの情報を取得
data "aws_security_group" "rds_sg" { // Terraform上での変数名を決める
  filter {
    name   = "group-name"
    values = ["rdsSG"] // あらかじめ作成しているセキュリティグループ名を入力
  }
}
```

図4 「terraform init」コマンドの実行例

```
$ terraform init

Initializing the backend...

Successfully configured the backend "s3"! Terraform will automatically
use this backend unless the backend configuration changes.

Initializing provider plugins...
- Checking for available provider plugins on https://releases.hashicorp.com...- Downloading plugin
for provider "aws" (1.53.0)...

The following providers do not have any version constraints in configuration, so the latest version
was installed.

To prevent automatic upgrades to new major versions that may contain breaking changes, it is
recommended to add version = "..." constraints to the
corresponding provider blocks in configuration, with the constraint strings
suggested below.

* provider.aws: version = "~> 1.53"

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see any changes that
are required for your infrastructure. All Terraform commands should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

data source機能を使う。同時に前述のバックエンドとプロバイダの指定も行う。

この内容で保存し、コードを保存したディレクトリに移動して「terrafrom init」コマンドを実行する。

コードが自動的に解析され、実行したディレクトリに「.terraform」というディレクトリが作成され、必要なモジュールがダウンロードされてくる。

次に「terraform plan」コマンドを実行し、作成したコードで実行される内容を確認する。

図5のとおり「No changes.」と出力され、何も変更が行われないことが確認できるが、「data.aws_security_group.rds_sg: Refreshing state...」と出力されているとおり、data source機能で既存リソースの情報を取得してきていることがわかる。特にリソースの変更は行わないが、正しい情報が取

得できていることを確認するため「terraform apply」コマンドでコードの内容を実行する。

ここでAWSマネジメントコンソールからS3の画面を確認すると、まだ何もリソースは作成されないが、S3バケットにtfstateファイルが作られていることが確認できる。

次に「terraform state list」コマンドを実行し、取得した情報の内容を確認する。

リソースの作成は行っていないが、セキュリティグループの情報を取得してきていることが確認できる。今回はフィルタリングの定義を簡易なものにしているが、実際に使用する場合は複数のフィルタを定義し、ユニークのIDを取得するようにする。

次に「terraform state showリソース名」コマンドを実行し、情報が自動的に正しく取得できていることを確認する。

図9のとおり、目的のセキュリティグループのIDが取得できていることが確認できる。実際にこのセキュリティグループのIDを参照する場合は「" ${data.aws_security_group.rds_

図5 「terraform plan」コマンドの実行例

```
$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

data.aws_security_group.rds_sg: Refreshing state...

------------------------------------------------------------------------

No changes. Infrastructure is up-to-date.

This means that Terraform did not detect any differences between your
configuration and real physical resources that exist. As a result, no
actions need to be performed.
```

図6 「terraform apply」コマンドの実行例

```
$ terraform apply
data.aws_security_group.rds_sg: Refreshing state...

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
```

図7 S3バケットの確認

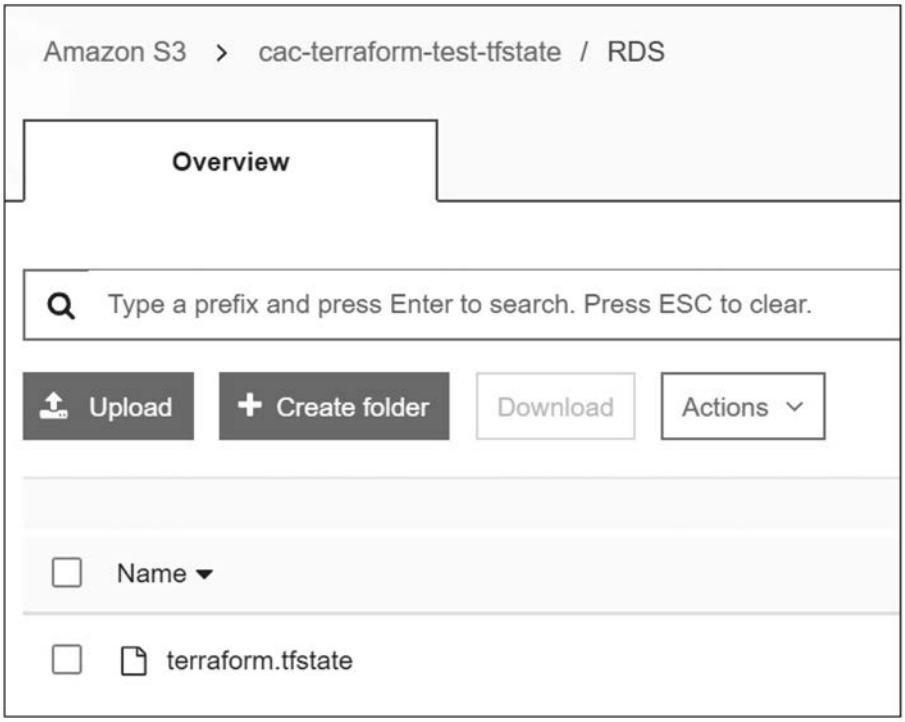


図8 「terraform state list」コマンドの実行例

```
$ terraform state list
aws_security_group.rds_sg
```

図9 「terraform state showリソース名」コマンドの実行例

```
$ terraform state show aws_security_group.rds_sg
id                              = sg-xxxxxxxxxxxxxxxxx
arn                             = arn:aws:ec2:ap-northeast-1:xxxxxxxxxxx:security-group/sg-
xxxxxxxxxxxxxxxxx
description                     = demo rdsSG
filter.#                        = 1
filter.1085127198.name          = group-name
filter.1085127198.values.#      = 1
filter.1085127198.values.2195288985 = rdsSG
name                            = rdsSG
tags.%                          = 0
vpc_id                          = vpc-xxxxxxxxxxxxxxxxx
```

sg.id}"」と指定する。詳細は「3.4 RDSの作成」で述べる。既存リソースの情報を取得する機能は豊富に用意されているため、実際に必要なリソースを作成するコードを記述する前に、本稿の例のような参照のみのコードを完成させ、関連リソースの情報が正しく取得できていることを確認する。

3.4 RDSの作成

次に以下のコードを「main.tf」に追記する。RDSを作成する定義の他にランダムでパスワードを生成する機能と、「terraform apply」コマンドの実行時にRDSのエンドポイントと生成したパスワードの標準出力を行う定義を追加している。

本題からは外れるが、Terraformには自動でインデントを揃える機能が備わっており、保存完了後に「terraform fmt」コマンドを実行してインデントを揃える。

本題に戻る。図10に示したように「main.tf」に新しいモジュール「random」を追加したため、そのままでは「terra-

図10 「main.tf」のコード追記例

```
// パスワードの自動生成
resource "random_id" "mysql_password" {
  byte_length = 8
}

// RDSの作成
resource "aws_db_instance" "mysql" {
  identifier        = "mysql01"
  engine            = "mysql"
  engine_version    = "5.7"
  allocated_storage = "20"
  storage_type      = "gp2"
  instance_class    = "db.t2.micro"

  skip_final_snapshot = true

  db_subnet_group_name = "rds-subnet"

  vpc_security_group_ids = [
    "${data.aws_security_group.rds_sg.id}", // セキュリティグループの指定
  ]

  name              = "mydb"
  username          = "cac"
  password          = "${random_id.mysql_password.id}"
  parameter_group_name = "default.mysql5.7"

  auto_minor_version_upgrade = "false"
}

output "mysql_adress" {
  value = "${aws_db_instance.mysql.address}"
}

output "mysql_password" {
  value = "${random_id.mysql_password.id}"
}
```

図11 「terrafrom fmt」コマンドの実行例

```
$ terraform fmt
main.tf
```

図12 「terraform init」コマンドの実行例

```
$ terraform init

Initializing the backend...

Initializing provider plugins...
- Checking for available provider plugins on https://releases.hashicorp.com...- Downloading plugin
for provider "random" (2.0.0)...

The following providers do not have any version constraints in configuration, so the latest version
was installed.

To prevent automatic upgrades to new major versions that may contain breaking changes, it is
recommended to add version = "..." constraints to the
corresponding provider blocks in configuration, with the constraint strings
suggested below.

* provider.aws: version = "~> 1.53"
* provider.random: version = "~> 2.0"

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see any changes that
are required for your infrastructure. All Terraform commands should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

form apply」コマンドを実行できない。そこで、「terraform init」コマンドを実行してモジュールをダウンロードする。

図12のとおり、新たにモジュールをダウンロードしてきたことがわかる。ここで再度「terraform plan」コマンドを実行する。

図13のとおり、「Plan:2to add, 0to change, 0to destroy.」と出力され、ランダムパスワードとRDSが作成されることが確認できる。次に「terraform apply」コマンドを実行して実際のリソースの作成を行う。デフォルトでは「terraform plan」コマンド相当の実行前結果の出力もなされるため、確認のうえ「yes」と入力することでリソース作成が開始される。

図15のとおり、RDSが作成された。リソース情報のアウトプットとしてランダム生成されたパスワードと、作成したRDSのエンドポイントが出力される。

以上で、Terraformを使ったRDS作成は完了である。

3.5 Terraform（OSS版）でよく起こしてしまうこととその対策

次に、利用中によく起こしてしまうこととその対策を述べる。OSS版のTerraformにはtfstateのバージョン管理機能はないため、前提条件として、定期的なバックアップを取得することを推奨する。

3.5.1 tfstateが壊れてしまった

「terraform state list」コマンドの実行が可能な場合は、「terraform state showリソースタイプ.リソース名」コマンドを実行して内容を確認しながら、壊れてしまったリソースを「terraform state rmリソースタイプ.リソース名」コマンドを実行し、「terraform plan」コマンドの結果を確認する。作成済みのリソースに対しては「terraform importリソースタイプ.リソース名リソース情報」コマンドを実行し、復元する。data sourceタイプの場合は「terraform state rm」コマンドを実行するのみでよい。「terraform import」コマンドは関連するリソースも自動的にインポートするものもあるため、必要に応じて「terraform state mv」コマンドでリソース名を意図したものに修正する。インポート完了後は「terraform plan」コマンドを実行し、変更の結果が「0」であることを確認し、念のために「terraform apply」コマンドを実行する。

3.5.2 コードがデグレードしてしまった

「terraform plan」コマンドの実行結果を確認しながらコー

図13 「terraform plan」コマンドの実行例

```
$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

data.aws_security_group.rds_sg: Refreshing state...

------------------------------------------------------------------------

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  + aws_db_instance.mysql
      id:                               <computed>
      address:                          <computed>
      allocated_storage:                "20"
      apply_immediately:                <computed>
      arn:                              <computed>
      auto_minor_version_upgrade:       "false"
      availability_zone:                <computed>
      backup_retention_period:          <computed>
      backup_window:                    <computed>
      ca_cert_identifier:               <computed>
      character_set_name:               <computed>
      copy_tags_to_snapshot:            "false"
      db_subnet_group_name:             "rds-subnet"
      endpoint:                         <computed>
      engine:                           "mysql"
      engine_version:                   "5.7"
      hosted_zone_id:                   <computed>
      identifier:                       "mysql01"
      identifier_prefix:                <computed>
      instance_class:                   "db.t2.micro"
      kms_key_id:                       <computed>
      license_model:                    <computed>
      maintenance_window:               <computed>
      monitoring_interval:              "0"
      monitoring_role_arn:              <computed>
      multi_az:                         <computed>
      name:                             "mydb"
      option_group_name:                <computed>
      parameter_group_name:             "default.mysql5.7"
      password:                         <sensitive>
      port:                             <computed>
      publicly_accessible:              "false"
      replicas.#:                       <computed>
      resource_id:                      <computed>
      skip_final_snapshot:              "true"
      status:                           <computed>
      storage_type:                     "gp2"
      timezone:                         <computed>
      username:                         "cac"
      vpc_security_group_ids.#:         "1"
      vpc_security_group_ids.2255785808: "sg-xxxxxxxxxxxxxxxxx"

  + random_id.mysql_password
      id:                               <computed>
      b64:                              <computed>
      b64_std:                          <computed>
      b64_url:                          <computed>
      byte_length:                      "8"
      dec:                              <computed>
      hex:                              <computed>


Plan: 2 to add, 0 to change, 0 to destroy.

------------------------------------------------------------------------

Note: You didn't specify an "-out" parameter to save this plan, so Terraform
can't guarantee that exactly these actions will be performed if
"terraform apply" is subsequently run.
```

図14 「terraform apply」コマンドの実行例

```
$ terraform apply

<中略>

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value:

<中略>
```

図15 「terraform apply」コマンドの実行結果例

```
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

Outputs:

mysql_adress = mysql01.xxxxxxxx.ap-northeast-1.rds.amazonaws.com
mysql_password = F9oYvw6WT-U
```

ドを最新化し、「terraform plan」コマンド実行後の出力結果に差分が出なくなるまで更新を行う。複数人で作業する場合はよく起きることであるため、GitなどのVCS（Version Control System）を使い、そもそも発生しないような運用ルール決めをすることを推奨する。

3. 5. 3 AWSマネジメントコンソール、もしくは他のツールで設定変更をしてしまい差分が出た

前項と同様、「terraform plan」コマンドの実行結果を確認しながらコードを最新化し、「terraform plan」コマンド実行後の出力結果に差分が出なくなるまで更新を行う。AWSマネジメントコンソールで変更したリソースについては、それ相当のコードを記述して「terraform import」コマンドを実行する。現状、インポート機能はリソースによっては完全に対応していないため、必要に応じて「terraform state mv」コマンドなどを使ってリソース名を意図したものに合わせる。

「terraform destroy−target=リソース名」コマンドで任意のターゲットのみを削除し、再作成することも可能であるが、コードの書き方および依存内容によって意図しない関係リソースまで削除されてしまうため、この方法は推奨しない。「terraform state mv」、「terraform state rm」、「terraform import」コマンドを使ったtfstateの操作のみで差分の修正をし、リソースに影響が出ないように作業をすることを推奨する。

3. 5. 4 data sourceフィルタが意図したとおり動かない

data sourceのフィルタリングはAPIのフィルタリング機能を使って動作する。そのため、ユニークなリソースをフィルタリングしたい場合はあらかじめ可能な限りのタグ（もしくはそれ同等のもの）を付与し、複数の汎用的なキーワードと値で絞り込めるようにしておく。設計が必要なため初回構築時は手間を要するが、初回構築用のコード以外は汎用性のあるTerraformのコードを記述することが可能になる。

4. 今後の取り組み

Terraformの導入にあたっては、「1. はじめに」で述べたようにすでにクラウドへシステム構築・移行をして利用している状況も多々あるため、どこから使い始めるか、どこまでTerraformで管理するのかが課題となる。当社でもeC+コンソールを提供しているが、クラウド入門者の顧客にはこれまでどおりeC+コンソールを利用いただき、クラウドの特性をさらに活かして迅速にシステムの構築や業務との連携を行いたい場合はTerraformの導入も検討している。また、OSS版を本格的に使うためには非常に作りこみが必要なため、小規模ながらTerraform Enterpriseの導入も検討している。

顧客が本来の業務に集中して取り組めるように様々な選択肢を当社では検討しており、今回紹介したTerraformだけではなく、様々な技術を取り入れクラウド活用のニーズに対応できるように努めていきたい。