IaCにおける生成AI活用の可能性

1.1. 背景・目的

クラウド推進部40期目標としてIaCの実装検証を行った。 2024年後半からの時代背景として生成AIを活用したシステム開発のためのツールが急速に発展していたため、 IaCの実装検証でもTerraformのコーディングに生成AIを活用することを検討した。

1.2. 検証内容

生成AIを使用してTerraformのコード生成を実施する。

- 使用する基盤モデル、開発ツールは公開されている活用事例や基盤モデルのベンチマークを参考にして選定する。
- コード生成のために生成AIに与える指示などを含む生成方法は、複数の方法を検証する。検証した方法をメリット、デメリットを評価する。
  生成方法は、次の3ケースを想定する。
    - ① AWSの設計技術、Terraformコーディングの技術がある場合
    - ② AWSの設計技術のみある場合
    - ③ AWSの基礎知識のみある場合

1.3. 検証結果

1.3.1. 基盤モデル、ツールの選定結果

開発ツール

Cursor

基盤モデル

Claude 3.7 Sonnet

生成方法の指針

選定した開発ツール、基盤モデルを使用する場合は以下の指針に従って生成を行う。

- コード生成指示プロンプトと同時に、設計書とコーディングルールを与える。
- コード生成は開始から完了までを一回の指示で実施する。
    - コード生成指示から作業タスクへの分割、生成完了までの反復実行は開発ツールにより自動で実行する。
- コード生成が失敗する場合はプロンプト、設計書、コーディングルールのいずれかを修正して再実行する。

1.3.2. 検証より得られた考察

1.3.2.1. 開発プロセス

検証した3ケース全てにおいて、コードの生成からAWS上の環境生成完了までの開発プロセスは同一のプロセスになる。
以下に実施したプロセスを記載する。

- （1） 設計
  想定ケース毎に担当者の技術に応じた設計書を作成する。
  設計書の記載ルール、コーディングルールなどを作成する。
- （2） コード生成
  「(1) 設計」で作成した設計書とルールを与えて生成AIにTerraformのコードを生成させる。
- （3） IaCコード解析、レビュー
  生成されたTerraformのコードをチェックして、改善箇所を指摘する。指摘がある場合は、設計書とルールを修正して「(2)コード生成」を再実行する。
  Terraformのコードのチェックには以下の機能を使用する。 - ① コード解析ツール - ② 生成AIによるレビュー - ③ 担当者によるレビュー
- （4） 構築
  Terraformのコマンドを実行してAWS上に環境を構築する。
  生成AIはTerraformのコマンドを監視する。エラーが発生した場合はTerraformコードを修正してコマンドを再実行する。
- （5） 構築結果確認、動作検証
  担当者により、AWS上に構築された環境を目視確認と、システムの動作確認を行う。
  不具合がある場合は設計書とルールを修正して「(2)コード生成」から再実行する。

1.3.2.2. 開発方法毎のメリット、デメリット

（考え中）

1.3.2.3. 作業時間の短縮効果

- (1) Terraformのコーディングを行う作業時間は大幅に短縮される。
- (2) Terraformのコーディング以外の作業時間はほぼ増減なし。

1.3.2.4. 品質に与える影響

- (1) 以下の効果により、メンテナンスしやすい設計が保たれるようになる。
  - ① 設計書を修正してTerraformコードを生成する手順が守られる。
  - ② 設計書修正から環境の再構築のサイクルが早い。

1.3.2.4. 技術者の育成に与える影響

生成AIを活用してコード生成を行うと、Terraformの基礎学習を行わずともIaCを用いた学習プロセスが体験できる。

このことは、IaCに関する理解を促進するためには有効であるが、一方でTerraformに関する理解が不足したまま構築を行う危険がある。

対策として、開発リーダーによるメンバーの理解度のチェックを行う必要がある。

1.4. まとめ

IaCに生成AIを活用することで作業時間の短縮効果がある。また、技術者の育成にも一定の効果はあるが生成AIに依存するため必要な技術が習得できない危険がある。

生成AIの特徴をよく理解して使用することが重要となる。

2. 背景と目的

2.1. 背景

検証を計画した時点の背景として、生成AIの性能向上により特にシステム開発の分野においての活用範囲が急速に拡大している。 アプリケーション開発の活用事例が多数紹介されるなか、IaCコードの生成に特化したベンチマーク「DPIaC-Eval」ではIaC特有の課題が指摘されている。

2.2. 目的

- IaCの開発に生成AIを活用する際の「効果的な使用方法」を検証する。
- 生成AIを使用することの「有効性（どの程度有益か、どの条件で機能するか）」を検証する。
- 上記目的の達成のため、前提の異なるケース①〜③で比較検証を行う。

検証の実施内容

- IaCのコード生成に使用する生成AIの基盤モデル、開発ツールを選定する。
  基盤モデルの選定にあたって、生成AIのプログラミング性能を評価したベンチマークを対象に、基盤モデル選定の基準とするベンチマークを選択する。
- 検証を通じて、有効なIaCコードを生成するための方法を作成する。主に生成AIに与える指示とフィードバックのタイミングを調整する。

2.3. 範囲と対象外

- 範囲: 生成AIを活用したIaCコードの生成。Terraformのコードを生成し、AWS上に環境を構築する。
- 対象外: 生成AIの利用料と作業量の削減から算定される費用対効果。

2.4. 想定ケース

- ケース①： AWSの設計技術、Terraformコーディングの技術がある場合
- ケース②： AWSの設計技術のみある場合
- ケース③： AWSの基礎知識のみある場合

2.5. 成果物

ケース①、②、③それぞれにおいて環境構築が成功するまでの手順を作成する。
生成AI活用の有効性の観点からケース①、②、③の比較を行う。

3. 生成AI、ツールの選定

3.1. 生成AI基盤モデルの選定方針

IaCコードの生成を行うために適した基盤モデルを選定する。
選定にあたり、IaCの能力を測定するために適したベンチマークの選定を行い、ベンチマークの結果を参考にする。

3.2. プログラミング性能を比較する主要なベンチマーク

生成AIの性能を評価するベンチマークは多数存在する。ベンチマークの選定にあたりプログラミングの性能を測定するベンチマークを中心に評価を行う。 特にIaCコードを生成する能力を評価するベンチマークを重視する。
(1) DPIaC-Eval はIaCコードを生成する能力に特化して評価している。DPIaC-Evalの結果を最重視する。
(2) DPIaC-Eval は2025年1月が最終更新日となっている。SWE-benchは最新の基盤モデルが発表されるたびに即時更新されるため最新モデルの評価はSWE-benchを参考にする。

主要なベンチマークの概要

- HumanEval
  OpenAIが公開したプログラミング問題セット。AIが生成したコードが、指定されたテストケースをどの程度正しくパスできるかをスコア（%）で評価する。主にPythonで実装力・ロジックの正確性を測る指標として業界標準となっている。
- SWE-bench
  GitHub上の実際のイシュー（バグや機能追加）の自動修正課題を用いて、AIモデルが本当に実用レベルのコードを自律生成・修正できるかを評価する。難度が高く、より現場に近い「実践力」を測る信頼性の高いベンチマークである。
- MultiPL-E
  複数のプログラミング言語（Python、Java、C++など）で同じ課題を解かせることで、各AIモデルの"多言語対応力"や汎用的なコーディング能力を測定する。幅広い開発現場での適応力を可視化する。
- DPIaC-Eval（Deployability-centric Infrastructure-as-Code Eval）
  153の実際のインフラシナリオで「展開可能性（deployability）」まで踏み込んで評価し、syntax・意図達成・セキュリティ考慮も判定する。反復的なフィードバックでモデル精度がどこまで上がるかも計測する。

3.3. DPIaC-Evalの概要

- IaC分野においての初回正答率は20〜30%と著しく低く、一般的なプログラム生成に対して著しく低い。
    - 特にセキュリティ面においてはほぼ考慮されない。
- 人の指示を加えた反復的なフィードバックを繰り返すことで90%程度まで精度が向上。最高スコアはClaude 3.5を使用して25回反復した場合の98%。
- 基盤モデルの回答精度評価ではClaude 3.7、次いでClaude 3.5が高い回答精度を示した。

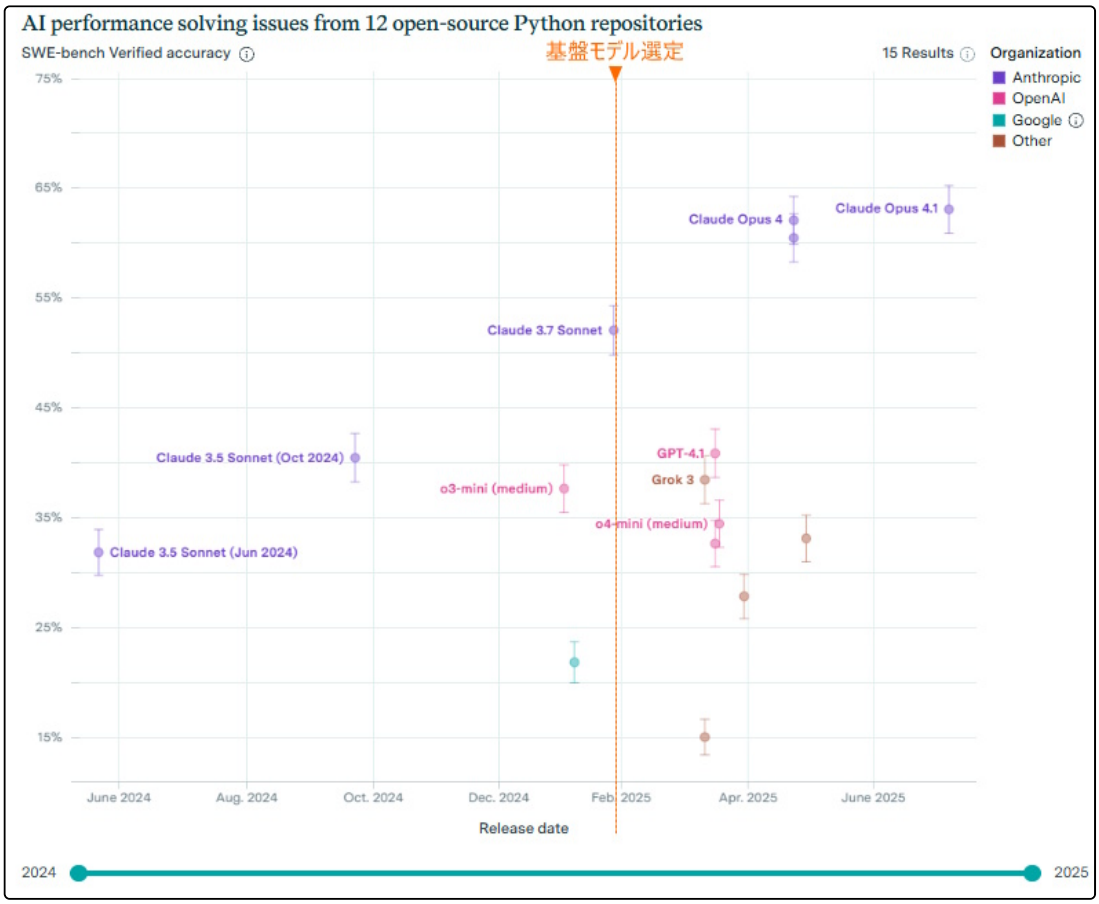
参考文献

Deployability-Centric Infrastructure-as-Code Generation: An LLM-based Iterative Framework
展開可能性を重視したIaCの生成：LLMベースの反復型フレームワーク
https://arxiv.org/html/2506.05623v1

3.4. DPIaC-Eval発表後に公開された基盤モデルの性能

2025/8/7時点のSWE-benchによるとClaude 4.1 Opusが最も高いスコアを記録している。

図1. 2025/08/14時点のSWE-bench



基盤モデルの性能は急速に進化している。実装検証では **DPIaC-Eval** の結果に従いClaude 3.7を使用するが、IaCのコーディングを開始する際には各種ベンチマークを参照して使用する時点で最も性能の高い基盤モデルを選択することを推奨する。

3.5. 開発環境

3.5.1. IDE

生成AIを活用してコーディングを行うツールを選定する。

選定条件

- 生成AIのモデルとしてClaude 3.7を選択できること
- エージェントモードを搭載していること。プロンプトの指示に従い、反復してコード生成が実行できること。

選定を行った2025年1月時点では2件の条件に合致するツールは次の2点であった。

(1) Cline

- 生成AIチャット機能、Visual Studio Codeの拡張機能として利用する。
- 入出力のトークン量に応じた従量課金制

(2) Cursor

- Visual Studio Codeを基本として生成AIの活用機能を追加したエディタ。
- リクエストの回数による課金

比較

エージェントを利用した反復実行の場合、消費トークン量が膨大になることがあるため、従量課金はリクエスト数に対する課金に比べて不利になる。

検証では使用料金で優位になるCursorを使用する。

> ※注記
> 2025年6月にCursorは消費トークン量による従量課金を導入した。
> ツール毎に料金体系は頻繁に更新されるため、使用する際に随時確認すること。

3.6. セキュリティとデータ取り扱い方針

生成AIを使用する場合に考慮する必要のあるセキュリティ上の条件を記載する。
使用する生成AI、開発ツールがセキュリティの条件に合致することを確認する。

3.6.1. 生成AIの学習によるデータ漏えい

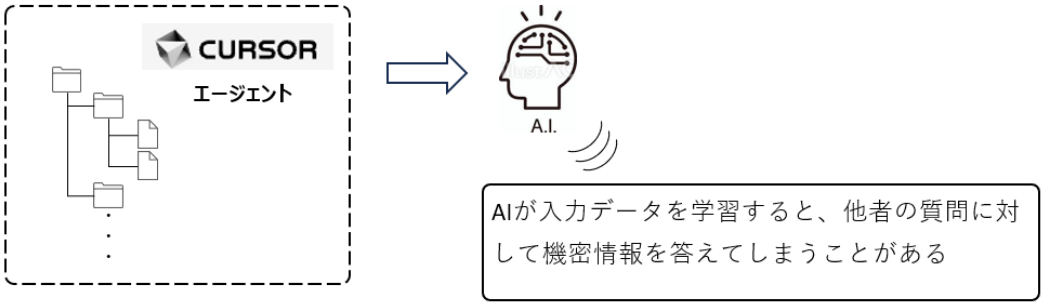
生成AIによっては、ユーザーが入力したデータを学習データとして利用することがある。

システム開発で生成AIを活用する場合には機密情報が生成AIに学習されないように考慮する必要がある。

機密情報の漏えい対策を考慮して、生成AIと開発ツールの選定とルールの作成を行う必要がある。

- Cursorのエージェントモードを使用することにより、意図せずローカルファイルの内容を送信することがある。
- 指示のミスにより生成AIへの指示にローカルのファイルが含まれることがある。
  ヒューマンエラーが原因になるため、完全に防ぐことは難しい。

図2. Cursorエージェントモード



3.6.2. データ漏えい対策状況

Claude

Anthropicの公式プライバシーポリシーに基づきClaudeの全てのモデルは特殊なケースを除き、入力データを学習に使用しない。

OpenAIのGPT-4.1など他の多くのモデルでは入力データを学習させないために追加の設定が必要になる。そのため、Claudeのモデルは他のモデルと比較して安全なモデルといえる。

- **Claudeの生成AIが学習に使用するケース**
  - 信頼性・安全性のレビューのためにフラグが立てられた会話
  - ユーザーが明示的に素材を報告し、学習に使うことに同意した場合
  - ユーザーが明示的に学習利用にオプトインした場合

Cursor

Cursorを使用する場合はPrivacy Modeの設定を"Privacy Mode"にすることでデータの漏えいを防ぐことが可能である。

Privacy Modeの設定値

- Share Data：
  - OpenAIなど生成AIモデルへの入力データの提供を許可。
  - 入力データと出力データをCursorのサービスに記録。記録したデータはCursorの機能向上のために活用する。
- Privacy Mode：
  - 「Share Data」で許可されるデータの提供を全て停止する。

まとめ

次の生成AIモデル、開発ツールを適切な設定で使用することでデータの漏えいを防ぐことが可能である。

- 生成AIのモデルはAnthropic社の提供するClaudeシリーズのモデルを使用する。
- CursorのPrivacy Modeの設定を"Privacy Mode"にする。

4. 検証の実施内容

3章までの検討結果を踏まえて、IaCコード生成の実装検証を実施する。 実装検証では作業者のスキル別に3ケースを想定し、それぞれケースでIaCコードを生成する最適な方法を作成する。

4.1. 対象範囲

- **システムを構築するプラットフォーム：**AWS
- **開発言語：**Terraform
- **システム構成の概要：**
    - コンテナ上で実行するWebアプリケーションを作成。
    - ECS上でFagate上で実行する。

4.2. 対象ケース

- ケース①： AWSの設計技術、Terraformコーディングの技術がある場合
    - 作業者がTerraformのリソース、設定を指示。
    - 生成AIが指示に従いTerraformコードを生成。
    - 作業者が生成されたTerraformコードを評価する。
- ケース②： AWSの設計技術のみある場合
    - 作業者がAWSのリソース、設定を指示。
    - 生成AIが指示に従いTerraformコードを生成。
    - 作業者はTerraformコードを実行してAWSに環境を作成し、AWS上の設定が指示どおりに作成されるか評価する。
- ケース③： AWSの基礎知識のみある場合
    - 作業者がアプリケーションの機能、AWSのサービスを指定。
    - 生成AIが指示に従いTerraformコードを生成。
    - 作業者はTerraformコードを実行してAWSに環境を作成、アプリケーションが正常に動作するか確認する。

4.3. コード生成方法の評価観点

生成方法は次の観点を考慮して最適な方法を作成する。

#	観点	説明
1	作業時間	作業者の手動でTerraformのコーディングを行った場合に比べて作業時間が短縮できること。
2	完成度	生成したコードから環境の構築がスムーズに行えること。
3	再現性	コード生成の再現性が高いこと、コード生成を繰り返し実行したときに同じコードが生成されること。
4	適合性	生成したコードはセキュリティや冗長化など、必要な要件を満たすコードになること。
5	可読性	生成したコードは読みやすく、人の手による継続的なメンテナンスが可能なこと。

5. 実施結果 ケース① Terraformのコーディング技術がある場合

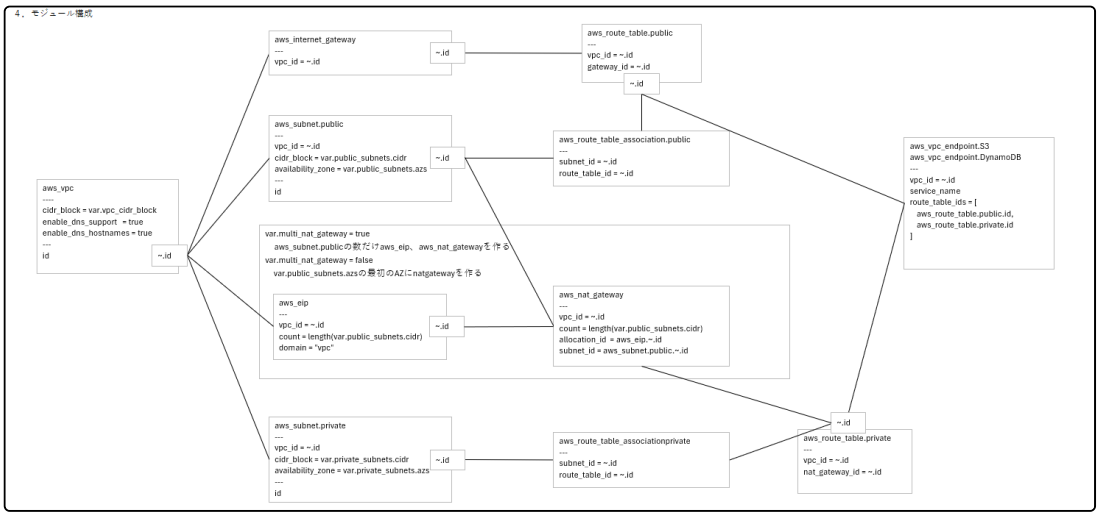
ケース①は作業者にTerraformのコーディングを行う十分な技術がある場合を想定する。作業者は生成するTerraformコードの詳細な指示を行う。生成されたTerraformコードのチェックを行う。

想定手順

- (1) AWSのシステム構成図および基本設計書を作成する。
- (2) Terraformのコードに記載するAWSのリソース名を指定する詳細な設計書を作成する。
- (3) 生成AIにTerraformのコードを生成させる。
- (4) 作業者が生成されたTerraformコードのチェックを行う。
  チェックは次の全ての作業を行う。 - a. コード解析ツールによるチェック - b. 生成AIによるコードレビュー - c. 作業者によるコードレビュー - d. AWSへのデプロイ実行時のエラー対応
- (5) 手順(4)で実施するいずれかのチェックで修正箇所が見つかった場合は、手順(2)の設計書を修正して生成AIによるTerraformコードの生成を再実施する。
- (6) 手順(4)の全てのチェックに合格するまで手順(3)から手順(5)を繰り返す。

作業者の作成する設計書

- (1) AWSシステム構成
  使用するAWSサービスと主な用途をテキストで記載。
- (2) 基本設計書
  システム構成図に記載したリソースの説明を記載する。システム内の役割やシステム構成図から読み取ることが困難な設定を記載する。
- (3) Terraformモジュール設計書 Terraformで作成するリソースを記載する設計書。リソースに設定する設定値とリソース間の相関を記載する。
- (4) Terraformコーディングルール **Terraformモジュール設計書**からTerraformのコードを生成するためのルールを記載する。



5.2. 作業内容と作業時間

作業者の作業内容と所要時間

作業内容	作業時間
AWS設計	8時間
Terraformモジュール設計	18時間
Terraformコーディングルール	4時間

生成AIの作業内容と所要時間

作業内容	作業時間
コード生成	0.5時間
コードレビュー	0.5時間

5.2. 方式の良い点

- Terraformモジュール設計書にTerraformコードで作成する全てのAWSリソースを記載した。
  生成するTerraformコードの細部まで指定するため、意図したとおりのTerraformコードを生成することができる。
- リソース間の相関を図で記載するため、リソースの関係を理解しやすい。

5.3. 課題

- Terraformモジュール設計書の作成のためにコーディングと同等の作業時間が必要になる。
- 変更管理が行いづらい。 検証の開始当初はリソース間の依存関係をテキストデータで表現するためER図や、UMLのクラス図、オブジェクト図などを使用することを検討した。 Terraformのリソースとリソースの相関を的確に表現できる記法が見つからなかったため、独自機能の図を作成することとした。 Terraformモジュール設計書はExcelの図で作成して、Cursorに生成を指示するときにはExcelから画像を作成して入力としている。 Excelと画像を使用しているためテキストデータに比べて変更の差分が確認しづらくなる。 テキストデータで作成する方法を見つけたい。
- 作成するAWSのリソースの設定を指示することに比べてTerraformモジュール設計書の記載内容が複雑になる。

総括

ケース①ではTerraformモジュール設計書を作成することによって作成者の意図した通りのTerraformコードを作成することができた。 一方で、作成するAWSリソースの設定を的確に指定するためには作業時間が大きくなり、コーディングと比べてメリットがあまりないとも感じる。 ケース②では作成するAWSリソースが的確に作成するための最低限の情報を指示する方法を模索する。

6. 実施結果 ケース② AWSの設計技術がある場合

ケース②は作業者がAWSの設計技術はあるが、Terraformのコーディングは未経験、または一部不安がある場合を想定する。
作業者はAWSのシステム構成と、AWS上に作成するリソースの設定を記載した設計書を作成し、生成AIにTerraformのコードを生成させる。

作業者の作成する設計書

- (1) システム構成図
  AWSのシステム構成図。
- (2) 基本設計書
  システム構成図に記載したリソースの説明を記載する。システム内の役割やシステム構成図から読み取ることが困難な設定を記載する。
- (3) 詳細設計書
  AWSリソースの詳細な設定を記載する。基本設計では記載していないパラメータなどを明示する。
- (4) Terraform設計書
  Terraformコードのディレクトリ構造などコーディングのルールを指定する。

想定手順

- (1) AWS環境の設計書を作成する。
- (2) 生成AIにTerraformのコードを生成させる。
- (3) 生成されたTerraformコードを実行してAWSの環境を作成する。コードの実行でエラーが発生した場合は手順(1)の設計書を修正する。
- (4) 作業者がAWS上に作成された環境を確認する。意図した環境が作成されない場合は手順(1)の設計書を修正して生成AIによるTerraformコードの生成を再実施する。
- (5) 作業者によるAWS環境のチェックに合格するまで手順(2)から手順(4)を繰り返す。

6.2. 作業内容と作業時間

(1) 設計書の記載内容、形式

作業者の作業内容と所要時間

作業内容	作業時間
AWS設計	12時間
Terraformコーディングルール	2時間

生成AIの作業内容と所要時間

作業内容	作業時間
設計書レビュー	0.5時間
コード生成	0.5時間
コードレビュー	0.5時間

6.3. 方式の良い点

(1) 設計書を全てテキストで記載するため以下のメリットがある。

- 変更箇所が把握しやすい。
- 設計書を作成するときに生成AIの支援が受けられる。設計書を全て作業者が作成する場合と比べて作業時間が短縮できる。

(2) 設計書の記載内容はシステム構成に必須で求める設定のみになる。システム構成の特徴が理解しやすい。

6.4. 課題

(1) 生成するAWSリソースの設定値が安定しない。

AWSリソースの設定を設計書で明示的に指定していない場合は生成AIの解釈により設定内容を決定する。

対策

- (対策①)：
    - 詳細設計書でAWSリソースの設定値を全て明示する。
- (対策②)：
    - 標準構成の設計書を作成する。標準的構成の設計書にはAWSリソースの設定値を全て明示的に設定する。
    - 作業者は標準構成との差分を記載した設計書を作成する。

(2) 生成するTerraformコードが安定しない。

Terraform設計書で明示的に指定していない箇所のコーディングについては生成AIが独自に解釈して行う。

対策

生成AIがコード生成を行うときに使用するTerraformコーディング標準の文書を作成する。
プロジェクト毎のTerraform設計書にはTerraformコーディング標準に記載されていないプロジェクト独自のルールを記載する。

注記

コーディング標準などは生成AIの基盤モデルにより異なる解釈をするために、生成するTerraformコードが安定しない減少が発生する。

コーディング標準などの文書は、基盤モデルを変更する度に出力が安定するように調整すること。

6.5 総括

ケース②では作成するAWSリソースを指示する方法を検討した。
明示的に指定したAWSリソースの設定は忠実に再現されたため、システムは問題なく動作させることができた。 課題として、明示的に指定した設定以外の出力が安定しなくなる。出力を安定させるために標準化の文書を作成させる必要があると感じた。 標準化文書の作成や、生成AIの基盤モデルの性能向上により生成されるコードが安定するか、引き続き検討課題とする。

7. 実施結果 ケース③ AWSの基礎知識がある場合

ケース③はAWSに関する基本的な知識のあるアプリケーション開発者が作業者となる場合を想定する。
作業者はアプリケーションの仕様からAWSサービスの選定を行う。AWS上のシステム構成は生成AIに作成させる。

作業者の作成する設計書

- (1) アプリケーション機能要件
  アプリケーションの機能に関する説明。
- (2) AWSシステム構成
  使用するAWSサービスと主な用途をテキストで記載。

想定手順

- (1) アプリケーション機能要件を作成する。
- (2) AWSシステム構成を作成する
- (3) 生成AIにAWS環境の作成を依頼する。
- (4) 生成AIはTerraformコード生成を行う、生成AIがTerraformのコードを実行してAWSの環境を作成する。
- (5) AWS上にアプリケーションをデプロイして動作確認を行う。動作確認でエラーが発生したらエラーメッセージを生成AIに入力してエラーの修正を指示する。生成AIは手順(4)から再実行する。
- (6) 動作確認が完了するまで手順(4)から手順(5)を繰り返す。

作業内容と所要時間

作業者の作業内容と所要時間

作業内容	作業時間
AWSシステム構成	2時間

生成AIの作業内容と所要時間

作業内容	作業時間
コード生成	0.5時間
コード実行AWS環境作成	0.5時間

7.3. 方式の良い点

(1) 設計書を全てテキストで記載するため以下のメリットがある。

- 変更箇所が把握しやすい。
- 設計書を作成するときに生成AIの支援が受けられる。設計書を全て作業者が作成する場合と比べて作業時間が短縮できる。

(2) 設計書の記載内容はシステム構成に必須で求める設定のみになる。システム構成の特徴が理解しやすい。

7.4. 課題

(1) 生成するAWSリソースの設定値が安定しない。

AWSリソースの設定を設計書で明示的に指定していない場合は生成AIの解釈により設定内容を決定する。

対策

- (対策①)：
  - 詳細設計書でAWSリソースの設定値を全て明示する。
- (対策②)：
  - 標準構成の設計書を作成する。標準的構成の設計書にはAWSリソースの設定値を全て明示的に設定する。
  - 作業者は標準構成との差分を記載した設計書を作成する。

(2) 生成するTerraformコードが安定しない。

Terraform設計書で明示的に指定していない箇所のコーディングについては生成AIが独自に解釈して行う。

対策

生成AIがコード生成を行うときに使用するTerraformコーディング標準の文書を作成する。
プロジェクト毎のTerraform設計書にはTerraformコーディング標準に記載されていないプロジェクト独自のルールを記載する。

注記

コーディング標準などは生成AIの基盤モデルにより異なる解釈をするために、生成するTerraformコードが安定しない減少が発生する。
コーディング標準などの文書は、基盤モデルを変更する度に出力が安定するように調整すること。

7.5 総括

ケース③では作成するAWSリソースを指示する方法を検討した。
明示的に指定したAWSリソースの設定は忠実に再現されたため、システムは問題なく動作させることができた。 課題として、明示的に指定した設定以外の出力が安定しなくなる。出力を安定させるために標準化の文書を作成させる必要があると感じた。 標準化文書の作成や、生成AIの基盤モデルの性能向上により生成されるコードが安定するか、引き続き検討課題とする。

8. 効果検証

本章では5章から7章で行った検証実績の結果からIaCに生成AIを活用したときの評価について考察する。

8.1. 総合評価

8.1.1. 生成AI活用の評価

8.1.2. 生成AIを活用する用途

(1) 設計書作成支援

- 生成AIに上流設計書から下位の設計書を作成させる　① システム構成図　② 基本設計書　③ 詳細設計書
- 生成AIと対話を重ねて設計書の制度を向上させる。

（効果）

- 作業時間の短縮
- 作業者の経験が不足している構成の場合は生成AIが知識を補ってくれる。 → 作業者の理解が向上する。

(2) 設計書レビュー

① 生成AIに設計書に関するヒアリングシートを作成させて、作業者が回答する。 暗黙知になっていた項目を明文化することにより、

- 設計書の制度が向上する。
- 作業者の理解が向上する。

② 生成AIにセキュリティや可用性などの観点を与えて評価させる。 技術者

（3）コードレビュー

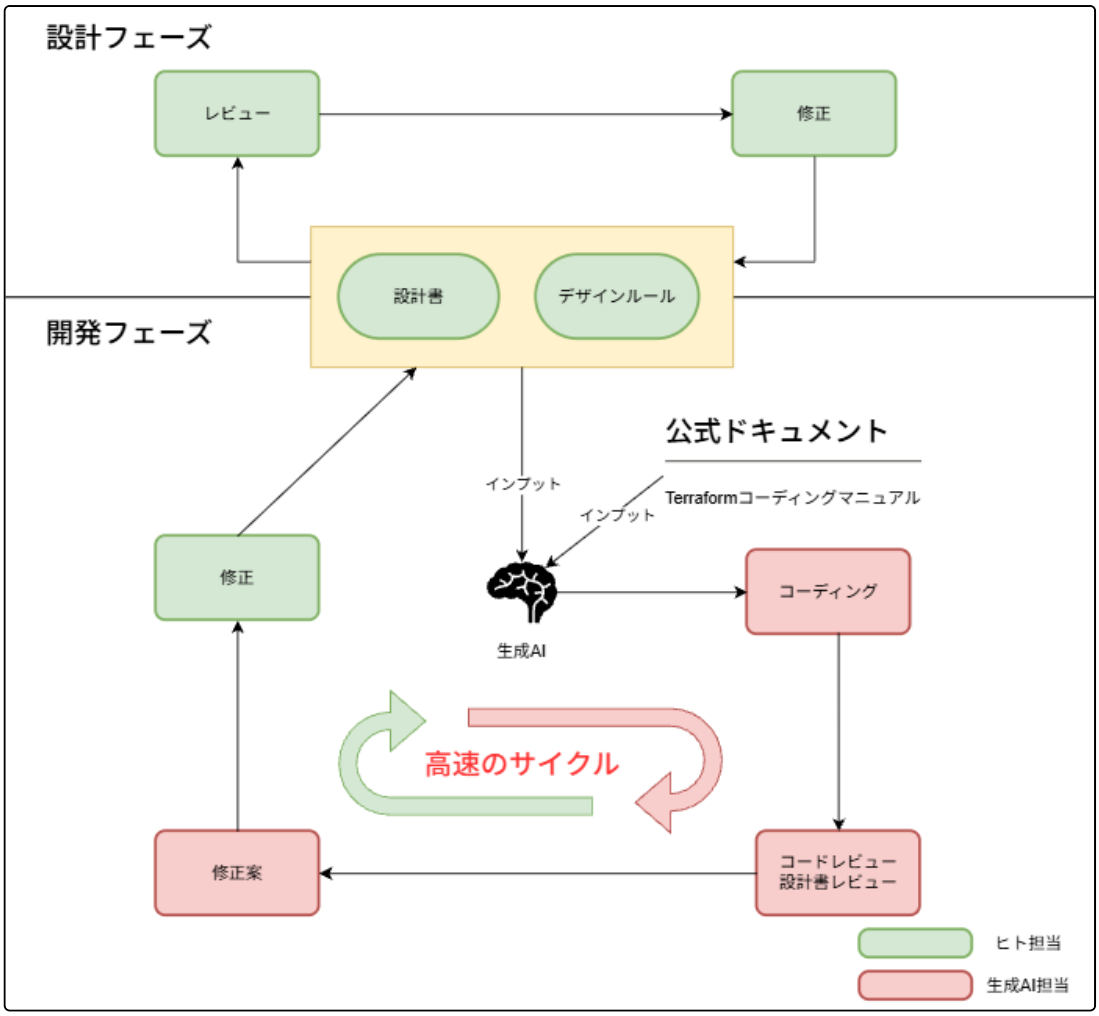
エラー発生時に、生成AIにエラーメッセージ、設計書、Terraformコードを入力して障害原因を推測させる。 障害原因の特定が迅速に行えることにより、対応までの時間が短縮される。

（4）障害原因調査

エラー発生時に、生成AIにエラーメッセージ、設計書、Terraformコードを入力して障害原因を推測させる。 障害原因の特定が迅速に行えることにより、対応までの時間が短縮される。

8.1.2. 効果が高いケース

設計変更からTerraformコーディング、環境構築までの時間が短縮されたため。開発のサイクルを高速で回せる



8.1.3. 適用条件と限界

生成AIは

- コーディング時間の大幅な短縮
- 幅広い知識により、技術者の知識不足の補填。
- コード、設計書の理解促進 など、多数の恩恵をもたらす。 一方で、最終的な成果物の精度は生成AIを使用する技術者の能力に依存する。

以下、典型的な課題

- 生成AIに観点を与えることのできる技術者が必要
  依頼内容の制度に応じて生成されるコードの品質にばらつきが出る。
  結局のところ、生成させる技術者が的確な観点を与えることができるかが重要になる。
- 全ての障害は対応できない。
  障害対応でコード修正、再試行を繰りかえしても解決しないことがある。 生成AIの解決できない問題に対応できる技術者が必要。

8.2. ケース別の評価

全てのケースでアプリケーションの動作が行える環境を作成できた。
|#|観点|ケース①|ケース②|ケース③| |---|---|---| |1|作業時間|△|○|◎| |2|完成度|◎|◎|○| |3|再現性|◎|○|△| |4|適合性|◎|◎|△| |5|可読性|◎|○|△|

8.3. 技術者育成への影響

9. 生成AI活用における成功条件と主なリスク

9章では5章から7章で行った検証実績から得られた生成AI活用を成功させるための条件と、生成AIを使用することで発生する主なリスクについて記載する。

9.1. 成功条件（設計書、プロンプトの記載方法）

(1) 開発プロセスの作成、遵守

設計を修正してコーディング、デプロイの開発プロセスを遵守する。 設計書、コードの両方の精度が向上する。

(2) 開発過程のロギング

生成AIの指示応答内容は可能であれば全て記録する。 失敗の記憶を遡らせることで回答の精度が向上することがあるため、指示内容を生成AIの応答の記録は可能な限りログを保存する。

(3) ソース管理手法の最適化

生成AIによるコード生成は、失敗した場合に生成前に戻して再実行することが頻繁に起こる。 Gitを使用する前は、生成AIに指示する前には必ずコミットするなど生成AIを使用することを前提に最適化する。

(4) 開発標準文書の整備

開発標準文書の充実が作業の効率、成果物の精度に大きく影響する。
従来、人の育成の場合は協業や訓練により醸成された暗黙知が成果に大きく影響していたが、生成AIを活用するためにはそれら暗黙知の明文化が重要になる。

(5) リーダークラスのエンジニアの存在

技術者の能力がより重要になる。
コーディングの生産性、トラブルシュートなどの対応速度などは生成AIを活用することにより大幅に向上する。
セキュリティの知見、保守性の高い設計、計画の作成能力など、より高度な技術が要求される。
現在の生成AIでは実現できないそれらの技術を高めることが望まれる。

9.2. 主なリスクと対応策

(1) 生成AI依存

技術者が生成AIに依存してしまう。

対策

必ず対人レビューを行う。作業者の理解度をチェックする。

(2) 回答のブレ

生成AIは実行する度に回答が異なる。偶然成功した状況が再現できなくなることがある。

対策

全ての変更を記録する。

(3) 情報漏えい

生成AIが機密情報を学習してしまう。

対策

生成AIの選定、設定に気をつける。

(4) 破壊

生成AIが意図しない破壊をする。

対策

生成AIの権限を抑制する。

Git、AWSを直接操作する権限を与えない。 作業効率を重視して作業者と同等の権限を与えがちになる、常にAIにどの程度の権限を与えているか管理できるようにする。

- 操作可能なユーザーを与えない
- Cursorの設定で縛る
- Cursorのルールで縛る
- MCPの活用

10. 結論

本章では、9章までに行った実装検証の考察を総括する。

10.1. 生成AI活用の有効性に関する結論

生成AIは

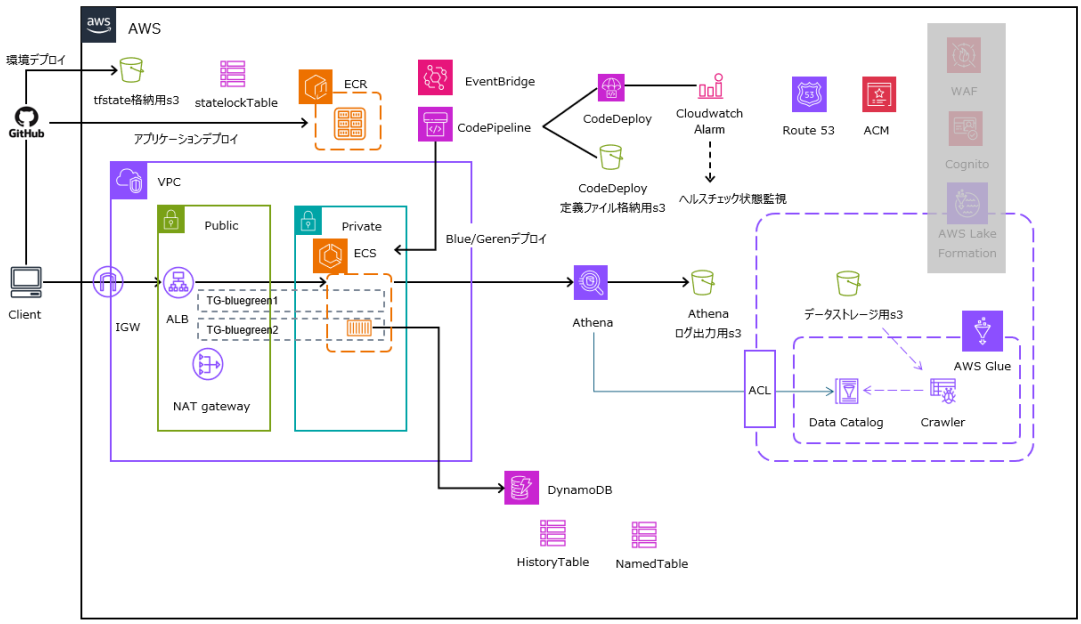
- コーディング時間の大幅な短縮
- 幅広い知識により、技術者の知識不足の補填。
- コード、設計書の理解促進 など、多数の恩恵をもたらす。 一方で、最終的な成果物の精度は生成AIを使用する技術者の能力に依存する。

10.2. 検証課題で行えなかった評価

生成結果を安定させるための標準化ドキュメントなどの整備は継続課題

付録A. ケース①で作成したドキュメント

(1) AWSシステム構成図



(2) rootモジュール設計書の記載ルール

```
# 基本ルール

TerraFormのrootモジュール設計を、プロンプトに画像で提供します。
生成AIは、その画像をもとに、サブモジュールを結合するTerraformのrootモジュールコードを生成します。

## 1. 宣言

このルールを使用する処理を実行する場合は"◆◆TerraFormルートモジュール設計に従い、rootモジュールコードを作成します◆◆"と言ってください。

## 2. rootモジュール設計書の解釈
### 2.1. Terraformコード出力仕様
1. 基本情報
    - 「パス」に「rootモジュールファイル」、「変数定義ファイル」、「パラメータ定義ファイル」を作成
    - 「rootモジュールファイル」に、「モジュールパス」のサブモジュールを結合するterraformのコードを作成
    - 「変数定義ファイル」に、rootモジュールの変数を定義する。
    - 「パラメータ定義ファイル」に、rootモジュールのパラメータを記載する。
    - 「パラメータ定義ファイル」の初回作成時は、サンプルパラメータを入力しておく

2. Terraform環境定義
    - 「Terraformバージョン」は、生成するコードのTerraformバージョン
    - 「プロバイダーソース」「プロバイダーソースバージョン」は、生成するコードのプロバイダー情報
```

```
### 2.1. モジュール間インターフェース構成
  -  オブジェクト
    - モジュールを四角いオブジェクトで表現する
    - 「モジュール：root」に付随する「variable」オブジェクトは、「変数定義ファイル」で定義する変数を意味する。
    - 「モジュール：root」に付随する「variable」のパラメータは、「パラメータ定義ファイル」に記載する。
    - 「サブモジュール：」から始まる四角いオブジェクトが、結合対象のサブモジュールである。
    - 「サブモジュール：」に付随する「variable」は、サブモジュールが要求するパラメータを意味する。
    - 「サブモジュール：」に付随する「output」は、サブモジュールがrootモジュールに返すパラメータを意味する。
  - モジュールの関係性
    - 「variable」方向に向かう線
      - rootモジュールが、サブモジュールを呼び出す際に与えるパラメータを意味する。

### 2.3. Terraformコードのコメント仕様
    - .tfファイルの最初に処理概要を記載
    - リソースの処理概要をリソース前に記載

## 3. コード生成時の挙動
    - 対象となるサブモジュールを、コード作成前に必ずスキャンする
    - 解釈が一意に定まらない場合、自己判断せず、ユーザーに対話形式で質問、提案すること
```

(3) TerraFormモジュール設計の記載ルール

```
# 基本ルール

TerraFormモジュール設計を、プロンプトに画像で提供します。
生成AIは、その画像をもとにTerraformコードを生成します。

## 1. 宣言

このルールを使用する処理を実行する場合は"◆◆TerraFormモジュール設計に従い、Terraformコードを作成します◆◆"と言ってください。

## 2. モジュール設計書の解釈
### 2.1. モジュール構成
    - リソースブロックを四角いオブジェクトで表現する
    - 四角いオブジェクト内の「---」で区切られたブロックは、以下の意味を持つ
        - 第1ブロック：リソースタイプ.リソース名
            - 「resource "リソース名" "リソース名" {}」 でコードブロックを生成する
            - data.で始まる場合は、「data "リソースタイプ" "リソース名"」で既存のリソースを参照する
            - リソース名の指定が省略されている場合は、任意の値を使う
            - 複数行書かれている場合は、同じ内容で行数分のリソースブロックを作成する
```

        - 第2ブロック：リソースブロックの属性
            - 主な属性を記載
            - 記載されていない必須の属性は、デフォルト値を設定する
            - 記載されていないOptionalの属性は、必要な場合、デフォルト値を設定する
            - 値が指定されている属性は、その値を設定する
            - 値が指定されていない属性は、デフォルト値を使用する
            - var.で始まる値は、変数定義ファイルから取得する
            - ~.で始まる値は、別リソースブロックのAttributeを参照していることを意味する
            - ~.は「別リソースブロック.リソース名」を省略したものである
            - ~.に続く文字列は、参照元リソースブロックのAttributeを意味する
        - 第3ブロック：出力
            - リソースブロックが出力定義ファイルに出力するAttributeを記載する
            - リソースブロックが省略されている場合は、Attributeを出力しない
    - リソースブロックに付属する四角いブロック
        - 別のリソースブロックが参照するAttributeを表現している
        - ~.は「リソースブロック.リソース名」を省略したものである
    - リソースブロックを繋ぐ線
        - 線は関係を示すものであるが、依存関係を示すものではない。
        - 明確な依存関係の設定が望ましい場合は、[depends_on]を指定してよい

2.2. Terraformコード出力仕様
1. 基本情報
    - 「パス」にモジュールディレクトリを作成
    - モジュールディレクトリに、メインコードファイル、変数定義ファイル、出力定義ファイルを作成

1. 変数定義ファイル
    - モジュールが引き受けるパラメータを記載する。

1. 出力定義ファイル
    - 出力名 = 出力の値
    - 出力の値の末尾に[]で型指された場合は、その型に従う。

1. モジュール構成
    - モジュールを構成するリソースを記載する。
    - 内容は「2. モジュール設計書の解釈」に従う。

1. その他ファイル
    - モジュールに付随するその他のファイルを記載する。
    - 指定がない限り、ファイルはモジュールディレクトリの直下に配置する。

2.3. Terraformコードのコメント仕様
    - .tfファイルの最初に処理概要を記載
    - リソースの処理概要をリソース前に記載

(4) TerraForm モジュール設計図

① モジュール：root

モジュール：root

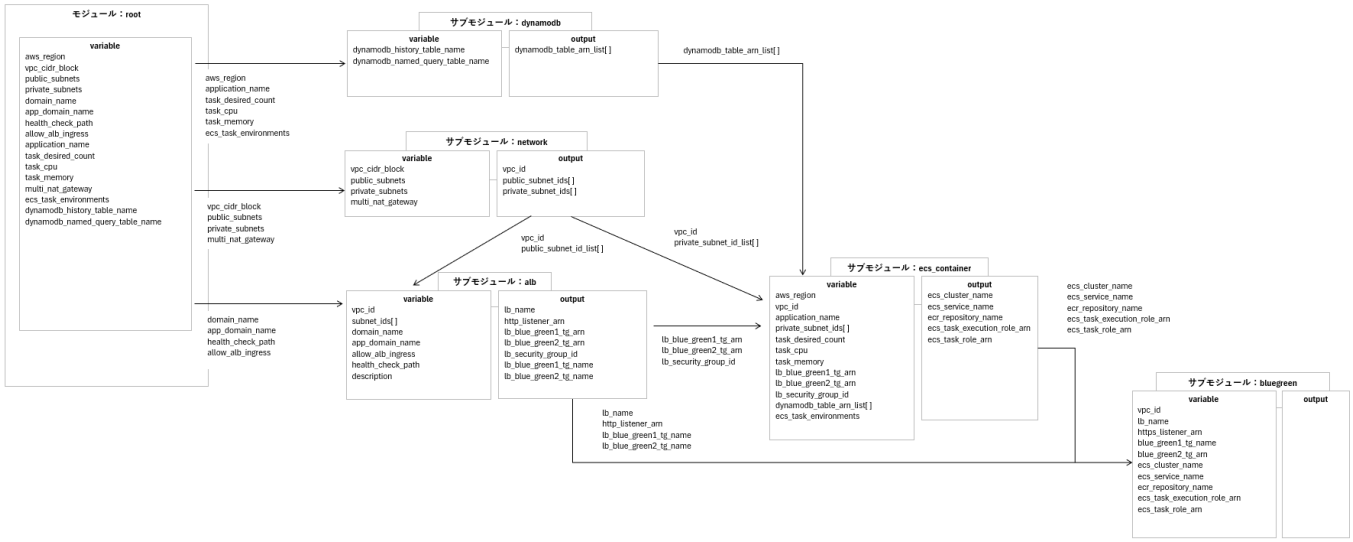
1. 基本情報

パス	=	infrastructure/ogura-generation
モジュールパス	=	infrastructure/ogura-generation/modules
rootモジュールファイル	=	main.tf
変数定義ファイル	=	variables.tf
パラメータ定義ファイル	=	env/ogura.tfvars
Terraform環境定義ファイル	=	provider.tf

2. Terraform環境定義ファイル情報

Terraformバージョン	=	1.9.8
プロバイダーソース	=	hashicorp/aws
プロバイダーソースバージョン	=	~> 5.74.0
バックエンド	=	S3
プロバイダー aws		
リージョン	=	var.aws_region
プロバイダー aws		
エイリアス	=	us-east-1
リージョン	=	us-east-1

3. モジュール間インターフェース構成



② モジュール：Network

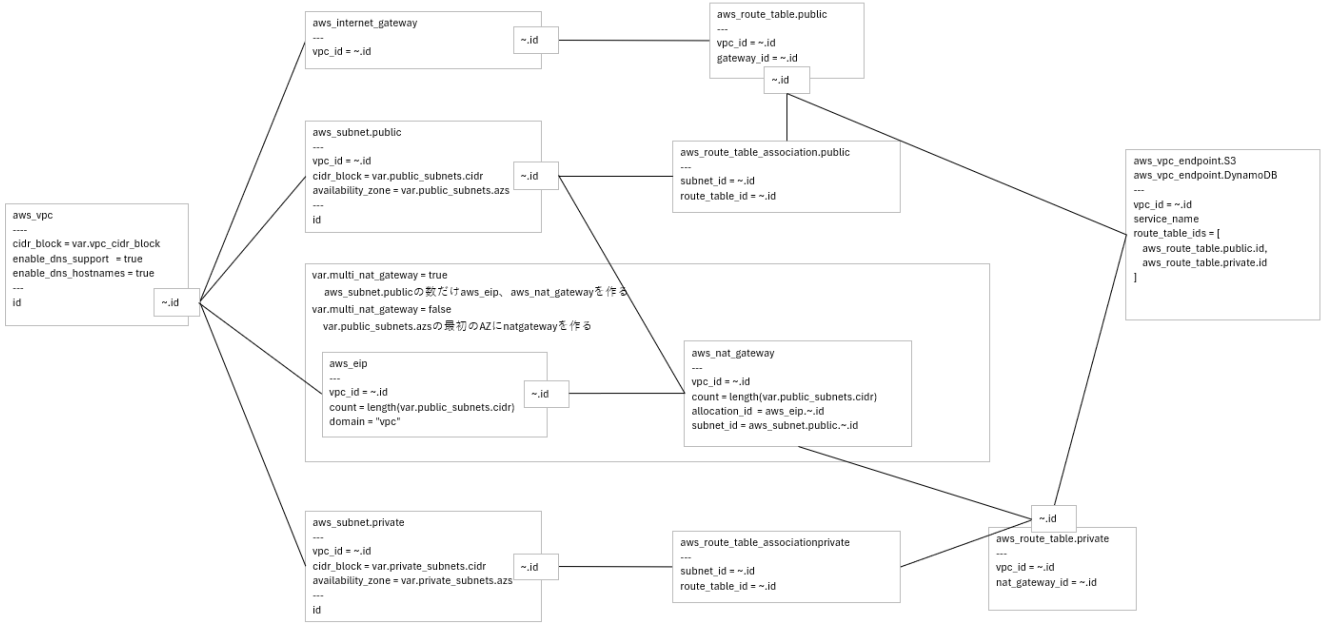
モジュール：Network

1．基本情報
- パス = infrastructure/ogura-genetaion-upstream/Terraform_code_v2/modules/network
- コードファイル = main.tf
- 変数定義ファイル = variables.tf
- 出力定義ファイル = outputs.tf

2．変数定義ファイル
- vpc_cidr_block[string]
- public_subnets(list(object))
  - cidr[string]
  - az[string]
- private_subnets(list(object))
  - cidr[string]
  - az[string]
- multi_nat_gateway[bool]

3．出力定義ファイル
- vpc_id = vpc_id.~.id[str]
- public_subnet_ids = aws_subnet.public[*].id[list]
- private_subnet_ids = aws_subnet.private[*].id[list]

4．モジュール構成



③ モジュール：ALB

モジュール：ALB

1．基本情報

パス	=	infrastructure/ogura-genetaion-upstream/Terraform_code_v2/modules/alb
コードファイル	=	main.tf
変数定義ファイル	=	variables.tf
出力定義ファイル	=	outputs.tf

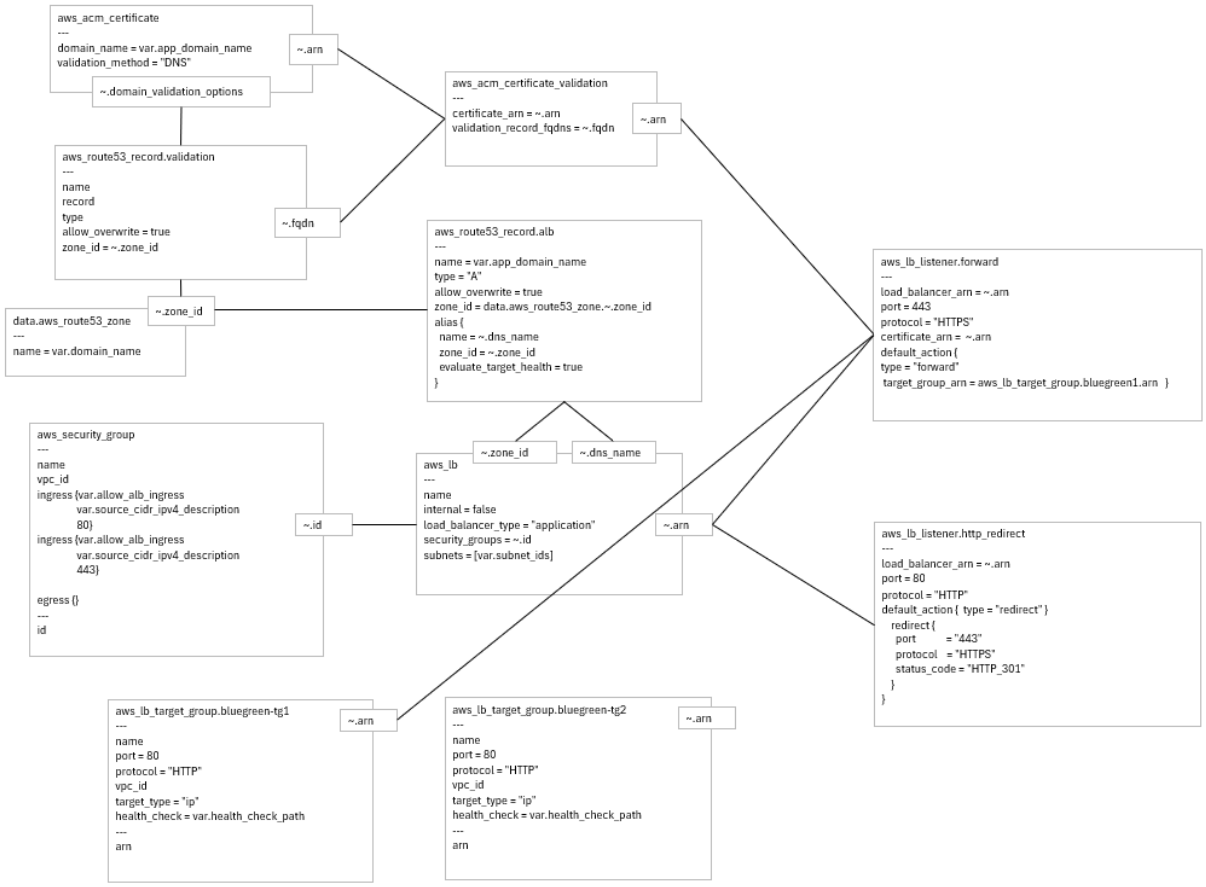
2．変数定義ファイル

vpc_id[string]
subnet_ids[list]
domain_name[string]
app_domain_name[string]
allow_alb_ingress(map(object))
  source_cidr_ipv4
  source_cidr_ipv4_description
health_check _path[string]

3．出力定義ファイル

lb_name[string]	=	aws_lb_name.~.name
http_listener_arn	=	aws_lb_listener.~.arn
https_listener_arn	=	aws_lb_listener.~.arn
lb_blue_green1_tg_arn	=	aws_lb_target_group.blue_green1.arn
lb_blue_green2_tg_arn	=	aws_lb_target_group.blue_green2.arn
lb_security_group_id[string]	=	aws_security_group.~.id
lb_blue_green1_tg_name	=	aws_lb_target_group.blue_green1.name
lb_blue_green2_tg_name	=	aws_lb_target_group.blue_green2.name

4．モジュール構成



④ モジュール：ECS_Container

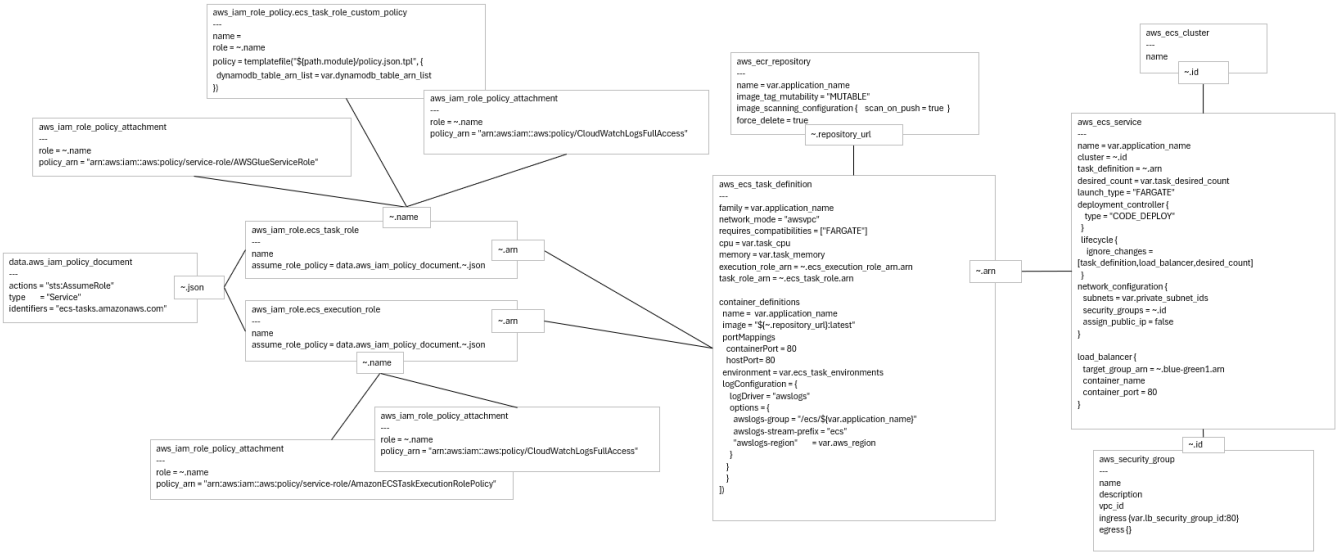
モジュール：ECS_Container

1．基本情報
- パス = infrastructure/ogura-generation/modules/ecs_container
- コードファイル = main.tf
- 変数定義ファイル = variables.tf
- 出力定義ファイル = outputs.tf

2．変数定義ファイル
- aws_region[string]
- vpc_id[string]
- application_name[string]
- private_subnet_ids[list]
- task_desired_count[num]
- task_cpu[num]
- task_memory[num]
- lb_target_group_arn[string]
- lb_security_group_id[string]
- dynamodb_table_arn_list[list]
- ecs_task_environments(list(object))
  - name[string]
  - value[string]

3．出力定義ファイル
- ecs_cluster_name = aws_ecs_cluster.~.name
- ecs_service_name = aws_ecs_service.~.name
- ecr_repository_name = aws_ecr_repository.~.name
- ecs_task_execution_role_arn = aws_iam_role.ecs_task_execution_role.arn
- ecs_task_role_arn = aws_iam_role.ecs_task_role.arn

4．モジュール構成



5．その他ファイル

IAMポリシーJSONは「配列」ではなく、「オブジェクト（{}）」で記載します。

1．ecsタスクロール用のポリシーファイル

- jsonファイル = policy.json.tpl
- ステートメント =
  - 1
    - Action : s3:PutObject, s3:ListBucket, s3:GetObject
    - Effect : Allow
    - Resource : arn:aws:s3:::*
  - 2
    - Action : dynamodb:UpdateItem, dynamodb:Scan, dynamodb:Query, dynamodb:PutItem, dynamodb:GetItem, dynamodb:DeleteItem, dynamodb:BatchWriteItem, dynamodb:BatchGetItem
    - Effect : Allow
    - Resource : ${jsonencode(dynamodb_table_arn_list)}
  - 3
    - Action : athena:StartQueryExecution, athena:GetQueryExecution, athena:GetQueryResults, athena:StopQueryExecution, athena:ListQueryExecutions, athena:GetWorkGroup
    - Effect : Allow
    - Resource : *

⑤ モジュール：DynamoDB

モジュール：DynamoDB

1．基本情報

パス	=	infrastructure/ogura-generation/modules/dynamodb
コードファイル	=	main.tf
変数定義ファイル	=	variables.tf
出力定義ファイル	=	outputs.tf

2．変数定義ファイル

dynamodb_history_table_name[string]
dynamodb_named_query_table_name[string]

3．出力定義ファイル

dynamodb_table_arn_list[lsit] = [
aws_dynamodb_table.query_history.arn
aws_dynamodb_table.named_query_table.arn
]

4．モジュール構成

```
aws_dynamodb_table.query_history_table
---
name = var.dynamodb_history_table_name
billing_mode = "PAY_PER_REQUEST"
hash_key = "query_id"
range_key = "created_at"
 attribute {
  name = "query_id"
  type = "S"
 }
 attribute {
  name = "created_at"
  type = "S"
 }
---
arn
```

```
aws_dynamodb_table.named_query_table
---
name =  var.dynamodb_named_query_table_name
billing_mode = "PAY_PER_REQUEST"
hash_key = "query_id"
range_key = "query_name"
 attribute {
  name = "query_id"
  type = "S"
 }
 attribute {
  name = "query_name"
  type = "S"
 }
---
arn
```

⑥ モジュール：BlueGreen

モジュール：BlueGreen

1．基本情報

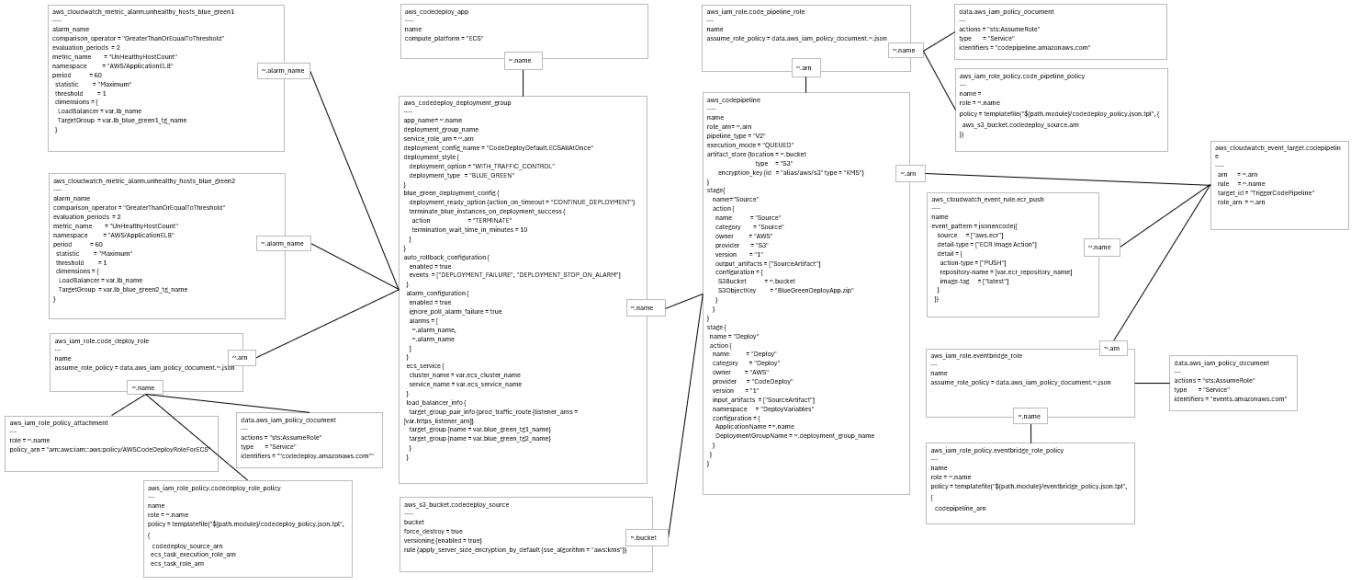
パス	＝	infrastructure/ogura-generation/modules/bluegreen
コードファイル	＝	main.tf
変数定義ファイル	＝	variables.tf
出力定義ファイル	＝	outputs.tf

2．変数定義ファイル

ecs_cluster_name[string]
ecs_service_name[string]
ecr_repository_name[string]
lb_name[string]
https_listener_arn[string]
lb_blue_green1_tg_name[string]
lb_blue_green2_tg_name[string]
ecs_task_execution_role_arn[string]
ecs_task_role_arn[string]

3．出力定義ファイル

4．モジュール構成



5．その他ファイル

IAMポリシーJSONは「配列」ではなく、「オブジェクト（{}）」で記載します。

1．Code_pipeline用のポリシーファイル（json形式）

jsonファイル	＝	codepipeline_policy.json.tpl		
ステートメント	＝	1	Action	： s3:ListBucket
			Effect	： Allow
			Resource	： *
		2	Action	： s3:PutObject
				s3:GetObject
				s3:GetObjectVersion
				s3:GetBucketVersioning
			Effect	： Allow
			Resource	： codedeploy_source_arn,
				codedeploy_source_arn/*
		3	Action	： codedeploy:CreateDeployment
				codedeploy:GetDeployment
				codedeploy:GetDeploymentConfig
				codedeploy:GetApplicationRevision
				codedeploy:RegisterApplicationRevision
				codedeploy:GetDeploymentGroup
				codedeploy:GetApplication
				codedeploy:GetDeploymentTarget
				codedeploy:ListDeploymentTargets
			Effect	： Allow
			Resource	： *
		4	Action	： kms:DescribeKey
				kms:GenerateDataKey
				kms:Encrypt
				kms:Decrypt
			Effect	： Allow
			Resource	： *
		5	Action	： ecs:DescribeServices
				ecs:DescribeTaskDefinition
				ecs:DescribeTasks
				ecs:ListTasks
				ecs:UpdateService
			Effect	： Allow
			Resource	： *
		6	Action	： ec2:DescribeNetworkInterfaces
				ec2:DescribeVpcs
				ec2:DescribeSubnets
				ec2:DescribeSecurityGroups
			Effect	： Allow
			Resource	： *

2．Event Bridge用のポリシーファイル（json形式）

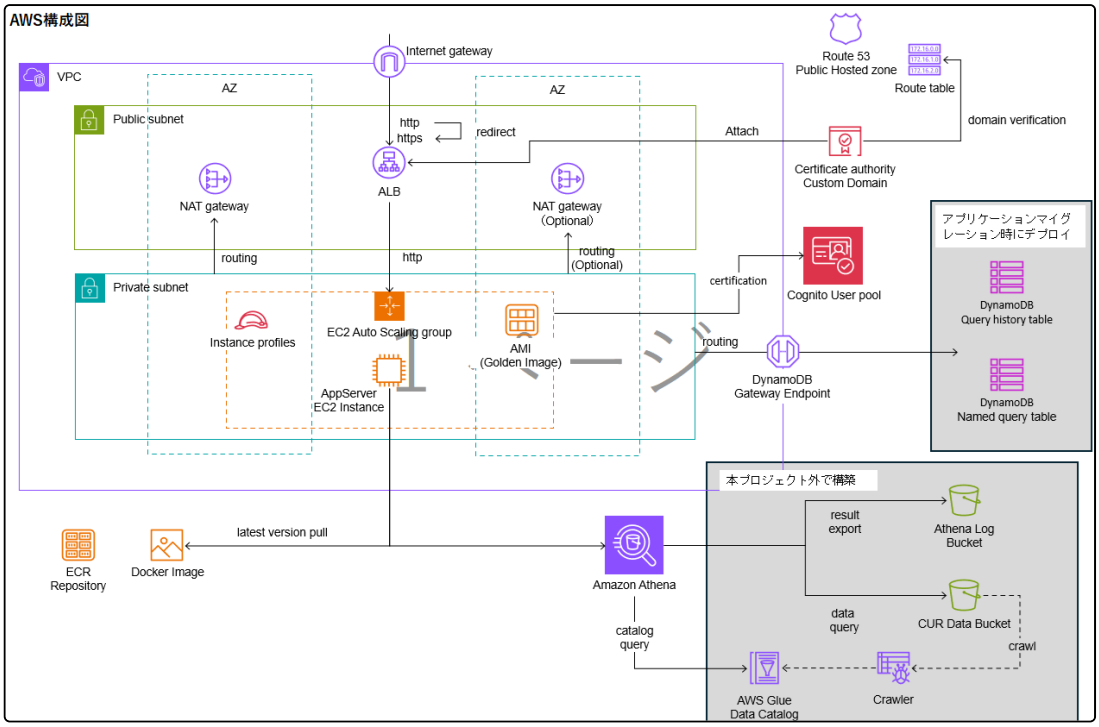
jsonファイル	＝	eventbridge_policy.json.tpl		
ステートメント	＝	1	Action	： codepipeline:StartPipelineExecution
			Effect	： Allow
			Resource	： codepipeline_arn

3．Code_Deploy用のポリシーファイル（json形式）

jsonファイル	＝	codedeploy_policy.json.tpl		
ステートメント	＝	1	Action	： s3:ListBucket
			Effect	： Allow
			Resource	： *
		2	Action	： s3:PutObject
				s3:GetObject
				s3:GetObjectVersion
				s3:GetBucketVersioning
			Effect	： Allow
			Resource	： codedeploy_source_arn,
				codedeploy_source_arn/*
		3	Action	： iam:PassRole
			Effect	： Allow
			Resource	： ecs_task_execution_role_arn
				ecs_task_role_arn

付録B. ケース②で作成したドキュメント

(1) システム構成図



(2) AWS基本設計

```
# AWS基本設計

## 1. 概要
- 本システムは、EC2 Auto Scalingグループを中心としたWebアプリケーション基盤である。
- アプリケーションはEC2でDockerコンテナとして稼働し、CURデータを保存したS3バケットにAthenaクエリを実行し、結果を返す。
- クエリ実行結果は、Athena実行結果保存用のS3バケットに出力する。
- クエリ文はDynammoDBに保存し、再利用可能とする。

## 2. ネットワーク構成
- **VPC**:
  - システム全体のネットワークを分離する。
  - IPv6は無効とする。
- **サブネット**:
  - パブリックサブネット: ALB、NAT Gatewayを配置する。
  - プライベートサブネット: EC2インスタンス（アプリケーションサーバー）を配置する。
- **AZ（アベイラビリティゾーン）**:
  - 冗長化のため複数AZにまたがって構成する。
- **Internet Gateway**:
  - インターネットアクセスを提供する。
- **NAT Gateway**:
  - プライベートサブネットからの外部アクセスを可能にする。
- **Route Table**:
  - 各サブネットのルーティングを制御する。
- **DynamoDB VPCエンドポイント**:
  - すべてのサブネットから、インターネットを経由せずにDynamoDBへセキュアにアクセスできるようにする。

## 3. サーバー・コンピューティング
- **EC2 Auto Scaling Group**:
  - パフォーマンス監視による自動スケールは行わない。
  - 指定したEC2インスタンス数が常に起動しているようにシステムを維持する。
  - EC2インスタンスは、各AZに均等に分散する。
  - ユーザーデータでOS環境変数を設定する。
- **AMI (Golden Image)**:
  - 運用によってメンテナンスされるAMI（ゴールデンイメージ）を利用する。
- **ECR Registry & Docker Image**:
  - EC2インスタンス起動時、ユーザーデータでECRからlatestバージョンをpullして起動する。

## 4. ロードバランサー・ドメイン
- **ALB (Application Load Balancer)**:
  - アプリケーションサーバーへのHTTP/HTTPSトラフィックの分散を行う。
  - スティッキーセッション（セッション維持）は使用しない。
- **Route 53**:
  - 同一AWSアカウントのホストゾーンでALBの名前解決をおこなう。
- **Certificate Authority**:
  - 同一AWSアカウントの独自ドメインでSSL証明書のDNS検証を行う。
```

```
## 5. データベース
- **DynamoDB**:
  - DynamoDB Gateway Endpoint経由でプライベートサブネットからアクセスする。
  - Query history table: アプリケーションがAthenaに実行したクエリ文の履歴テーブルとする。
  - Named query table: Athenaに実行したクエリ文を名前をつけて保存するテーブルとする。
- **DynamoDB Gateway Endpoint**:
  - VPC内からDynamoDBへのセキュアな通信を実現する。

## 6. 分析・ログ
- **Amazon Athena**:
  - S3上のデータに対してクエリを実行する。
- **Athena Log Bucket**:
  - Athenaのクエリ結果をS3バケットに保存する。
- **CUR Data Bucket**:
  - コスト・利用状況レポート（CUR）をS3バケットに保存する。
- **AWS Glue Data Catalog & Crawler**:
  - データカタログを管理し、S3データのスキーマ自動検出を行う。

## 7. 認証・認可
- **Amazon Cognito User Pool**:
  - アプリケーションサーバーへのユーザー認証を提供する。
  - ユーザー登録、ログイン、パスワードリセット機能を提供する。
  - JWTトークンベースの認証を実装する。
  - ユーザー名としてメールアドレスを使用する。
  - メールアドレスの自動検証を有効化する。
- **Cognito User Pool Client**:
  - SRPプロトコル（ALLOW_USER_SRP_AUTH）による安全な認証フローを提供する。
  - パスワード認証（ALLOW_USER_PASSWORD_AUTH）もサポートする。
  - リフレッシュトークン（ALLOW_REFRESH_TOKEN_AUTH）による長期セッション管理を提供する。
  - PKCE（Proof Key for Code Exchange）をサポートし、セキュリティを強化する。
- **Cognito User Pool Domain**:
  - Cognitoホスト型UIへのアクセス用ドメインを提供する。
  - OAuth 2.0フローをサポートする。
- **管理者ユーザー管理**:
  - システム初期化時に管理者ユーザーを自動作成する機能を提供する。
  - 管理者ユーザーは専用の「Admins」グループに所属する。
  - 管理者ユーザーの初期パスワードは自動生成し、セキュアに管理する。
- **認証フロー**:
  - ユーザーはCognito User Pool経由でアプリケーションにログインする。
  - 認証成功時にJWTトークン（ID Token、Access Token）を発行する。
  - アプリケーションサーバーはJWTトークンを検証してユーザー認証を行う。
- **認可制御**:
  - 認証されたユーザーのみがAthenaクエリ実行機能にアクセス可能とする。
  - DynamoDBのクエリ履歴・名前付きクエリはユーザー単位で分離せず、共用する。
  - 管理者グループのユーザーには追加の権限を付与可能とする。
- **セッション管理**:
  - JWTトークンの有効期限を適切に設定する（ID Token: 1時間、Access Token: 1時間、Refresh To
ken: 30日）。
  - ログアウト時にはクライアント側でトークンを破棄する。
- **ユーザー管理**:
  - ユーザー管理（ユーザー登録、削除、パスワードリセット）はAWSマネジメントコンソールまたは管理者機能から行
```

```
う。
  - 管理者ユーザーは自動作成され、初期パスワードは安全に生成・管理される。

## 8. セキュリティ
- **パブリック/プライベートサブネット分離**:
  - 外部公開リソースと内部リソースを分離する。
- **ALB経由の通信制御**:
  - 直接EC2にアクセスできないようにする。
- **DynamoDB Gateway Endpoint**:
  - インターネット経由せずにDynamoDBへアクセスする。
- **証明書管理**:
  - Route 53と連携してドメイン認証を行う。
- **EC2へのSSH接続制御**:
  - EC2インスタンスはSSMマネージドノードとし、SSMフリーとマネージャで管理接続する。
- **認証セキュリティ**:
  - Cognito User Poolでパスワードポリシーを強制する（最小8文字、大文字・小文字・数字・特殊文字を含む）。
  - MFA（多要素認証）をオプションで有効化可能とする。
  - JWTトークンの署名検証を必須とする。
  - ユーザー存在エラーの防止機能を有効化する。
  - 管理者ユーザーの初期パスワードは強力なランダムパスワードを自動生成する。

## 9. データフロー
- ユーザーはRoute 53経由でALBにアクセスする。
- ALBがリクエストをEC2 Auto Scaling Groupのアプリケーションサーバーに転送する。
- アプリケーションサーバーは認証が必要な機能に対してCognito User PoolでJWTトークンを検証する。
- 認証されたユーザーのリクエストに対して、アプリケーションサーバーは必要に応じてDynamoDBへデータアクセスを行う（Gateway Endpoint経由）。
- アプリケーションのログやデータはS3バケットに保存し、Athenaで分析する。
- DockerイメージはECRからpullする。

## 10. 本設計外で構築するリソース
- **Route 53**
  - 独自ドメインの管理。
  - 証明書の検証、ALBのレコード登録は可能とする。
- **Athena Log Bucket**:
  - Athenaのクエリ結果を保存するS3バケット。
- **CUR Data Bucket**:
  - コスト・利用状況レポート（CUR）を保存するS3バケット。
- **AWS Glue Data Catalog**:
  - S3データのカタログ情報。
- **Crawler**:
  - S3バケット内のデータを自動でクロールし、Glue Data Catalogにスキーマ情報を登録する。
- **DynamoDBテーブル**:
  - アプリケーションマイグレーション時に、アプリケーションがデプロイする。

## 11. その他
- **冗長化・可用性**:
  - 複数AZ構成、Auto Scalingを実施する。
- **拡張性**:
  - Auto Scaling、マネージドサービスを活用する。
- **運用**:
  - AMIによる標準化、ECRによるイメージ管理を行う。
```

```
---

以上、本システムのAWS基本設計とする。
```

(3) AWS詳細設計

```
# AWS詳細設計

## 1. 概要
- AWSシステムを、Terraformでデプロイするための詳細設計を行う。
- 以下はTerraformのデプロイ対象外とする。
  - EC2インスタンスのAMI（AMIはゴールデンマスタ方式とし、人的な運用で維持管理する）
  - アプリケーションのビルド（GitHub ActionsでDockerコンテナをビルドし、ECRリポジトリにアップロードする）
  - Athenaの結果を出力するS3バケット
  - CURデータを保存するS3バケット
  - CURデータをクロールするGlue Crawlerと、作成されるGlue Data Catalog
  - DynamoDBテーブル（アプリケーション実行時にGitHub Actionsで動的に作成され、Systems Manager Parameter Storeでテーブル名を管理する）

## 2. ネットワーク構成
- **VPC**:
  - CIDRブロックは/24限定とする。
- **AZ（アベイラビリティゾーン）**:
  - 最低2つのAZで構成し、デプロイパラメータ指定で3つ以上に拡張可能とする。
- **サブネット**:
  - パブリックサブネット:
    - NAT Gateway
      - 1つ目のAZには必ず配置する。
      - 2つ目以降のAZにNAT Gatewayをデプロイするマルチ NAT構成はオプショナルとし、デプロイパラメータで設定可能とする。
  - プライベートサブネット:
    - マルチNAT構成の場合、同一AZのNAT Gatewayをデフォルトルートとする。
- **DynamoDB VPCエンドポイント**:
  - Gatewayタイプでデプロイする。
  - すべてのサブネットにルーティングを追加する。

## 3. サーバー・コンピューティング
- **EC2 Auto Scaling Group**:
  - 最低数をデプロイパラメータで指定可能とする。
  - 最低数と最大数は同じとする。
  - AMI IDはデプロイパラメータで指定可能とする。
- **ECR Repository**:
  - リポジトリ名をデプロイパラメータで指定可能とする。
- **EC2**:
  - ユーザーデータ:
    AMIに含まれていない以下の処理だけを実行する。
    1. タイムゾーンを日本に設定する。
    1. OS環境変数を設定する。
       - Systems Manager Parameter StoreからDynamoDBテーブル名を動的に取得する。
       - 取得できない場合はデフォルト値を使用する。
```

```
        | OS環境変数名 | 用途 | 設定方法 |
        |----|----|----|
        | AWS_REGION | AWSリージョン | Terraformから動的設定 |
        | AWS_DEFAULT_REGION | AWSデフォルトリージョン | Terraformから動的設定 |
        | ATHENA_DATABASE | Athenaデータベース名 | Terraformから動的設定 |
        | ATHENA_WORKGROUP | Athenaワークグループ名 | Terraformから動的設定 |
        | ATHENA_OUTPUT_LOCATION | Athenaクエリ結果出力S3バケット（URL） | Terraformから動的設定 |
        | DYNAMODB_QUERIES_TABLE | クエリ保存用DynamoDBテーブル名 | Parameter Storeから動的取得 |
        | DYNAMODB_EXECUTIONS_TABLE | 実行履歴用DynamoDBテーブル名 | Parameter Storeから動的取得 |
        | DYNAMODB_TAGS_TABLE | タグ管理用DynamoDBテーブル名 | Parameter Storeから動的取得 |
        | COGNITO_USER_POOL_ID | Cognito User Pool ID | Terraformから動的設定 |
        | COGNITO_CLIENT_ID | Cognito User Pool Client ID | Terraformから動的設定 |
        | COGNITO_REGION | Cognito User Poolのリージョン | Terraformから動的設定 |
    1. ECRリポジトリからlatestバージョンのDockerイメージをpullする。
    1. .env.productionファイルを作成し、環境変数を設定する。
    1. Dockerイメージを起動する。
       - 作成した.env.productionファイルを環境変数ファイルとしてコンテナに渡す。
       - ポートフォワーディングは80:80。
       - コンテナ名は「app-container」とし、再起動ポリシーを「unless-stopped」に設定する。
  - インスタンスプロファイル:
    - インラインでカスタムポリシーを作成する。
      | 対象リソース | アクション | 許可 | 備考 |
      |----|----|----|----|
      | DynamoDBテーブル | フルアクセス | 許可 | 本プロジェクトで作成したテーブルのみ |
      | Systems Manager Parameter Store | 読み取り専用 | 許可 | DynamoDBテーブル名取得用 |
      | Athena | フルアクセス | 許可 | 本AWSアカウントのAthenaのみ |
      | S3バケット | フルアクセス | 許可 | 本AWSアカウントのS3バケットのみ |
      | Cognito | 読み取り専用 | 許可 | JWKSエンドポイントアクセス用 |
      | ECR | 読み取り専用 | 許可 | Dockerイメージpull用 |
    - その他、要件を満たすために必要なマネージドポリシーを適宜アタッチする。

## 4. ロードバランサー・ドメイン
- **ALB (Application Load Balancer)**:
  - セキュリティグループ
    - インバウンドルール
      - デプロイパラメータでソースIPアドレスを複数指定可能とする。
      - ソースIPアドレスに対してHTTP、HTTPSを許可する。
  - ヘルスチェック
      - ヘルスチェックパスはデプロイパラメータで指定可能とする。
  - ドメイン名
     - デプロイパラメータで指定可能とする。
  - FQDN
     - デプロイパラメータで指定可能とする。
  - 証明書
      - Route53でドメイン検証を行う。
```

```
## 5. データベース
- **DynamoDB**:
  - **テーブル作成方針**:
    - DynamoDBテーブルはTerraformではなく、GitHub Actionsでアプリケーション実行時に動的に作成される。
    - 作成されたテーブル名はSystems Manager Parameter Storeに保存される。
    - EC2インスタンスのユーザーデータでParameter Storeからテーブル名を取得し、環境変数として設定する。
  - **Systems Manager Parameter Store設定**:
    - パラメータ名: `/{project}/{env}/dynamodb/table-names`
    - 形式: StringList（カンマ区切りのテーブル名リスト）
    - 初期値: "placeholder"（GitHub Actionsで実際の値に更新）
    - ライフサイクル管理: 値の変更を無視（ignore_changes = [value]）
  - **テーブル命名規則**:
    - クエリ保存用: `{project}-{env}-queries`
    - 実行履歴用: `{project}-{env}-executions`
    - タグ管理用: `{project}-{env}-tags`
  - **フォールバック設定**:
    - Parameter Storeからテーブル名が取得できない場合、上記命名規則に従ったデフォルト値を使用する。

## 6. 認証・認可
- **Amazon Cognito User Pool**:
  - User Pool名をデプロイパラメータで指定可能とする。
  - ユーザー名設定:
    - ユーザー名属性としてメールアドレスを使用する（username_attributes = ["email"]）。
  - 自動検証設定:
    - メールアドレスの自動検証を有効化する（auto_verified_attributes = ["email"]）。
  - パスワードポリシー:
    - 最小文字数: 8文字以上
    - 大文字、小文字、数字、特殊文字を含む
    - 一時パスワードの有効期限: 7日
  - アカウント復旧設定:
    - パスワードリセット機能を有効化する。
    - 検証方法はEmailとする（verified_email優先度1）。
  - メール設定:
    - Cognito標準のメール送信機能を使用する（COGNITO_DEFAULT）。
  - セキュリティ設定:
    - ユーザー存在エラーの防止機能を有効化する（prevent_user_existence_errors = "ENABLED"）。
  - 削除保護:
    - 削除保護を無効化する（deletion_protection = "INACTIVE"）。
  - MFA（多要素認証）:
    - オプショナル設定とし、デプロイパラメータで有効/無効を指定可能とする。
    - SMS、TOTPアプリケーションの両方をサポートする。
  - ユーザー属性:
    - 必須属性: email
    - オプション属性: name, family_name, given_name
    - email属性の検証を必須とする。
- **Cognito User Pool Client**:
  - Client名をデプロイパラメータで指定可能とする。
  - 認証フロー:
    - ALLOW_USER_PASSWORD_AUTH（ユーザー名・パスワード認証）を有効化する。
    - ALLOW_USER_SRP_AUTH（SRPプロトコル）を有効化する。
    - ALLOW_REFRESH_TOKEN_AUTH（リフレッシュトークン）を有効化する。
  - トークン有効期限:
```

```
      - ID Token: 1時間（60分）
      - Access Token: 1時間（60分）
      - Refresh Token: 30日
    - トークン有効期限単位:
      - Access Token、ID Token: 分単位（minutes）
      - Refresh Token: 日単位（days）
    - セキュリティ設定:
      - ユーザー存在エラーの防止機能を有効化する（prevent_user_existence_errors = "ENABLE
D"）。
      - クライアントシークレットは生成しない（generate_secret = false）。
    - OAuth設定:
      - サポートするアイデンティティプロバイダー: COGNITO
      - 認可コードフローを有効化する。
      - スコープ: openid, email, profile
    - コールバックURL:
      - デプロイパラメータで指定可能とする。
    - ログアウトURL:
      - デプロイパラメータで指定可能とする。
      - デフォルト: ["http://localhost:3000/"]
  - **Cognito User Pool Domain**:
    - ドメイン名をデプロイパラメータで指定可能とする。
    - Cognitoホスト型UIへのアクセス用ドメインを提供する。
    - OAuth 2.0フローをサポートする。
  - **管理者ユーザー自動作成機能**:
    - 管理者ユーザー作成フラグをデプロイパラメータで指定可能とする（create_admin_user）。
    - 管理者メールアドレスをデプロイパラメータで指定可能とする。
    - メールアドレス形式のバリデーションを実装する。
    - 管理者ユーザーの初期パスワード生成:
      - 16文字のランダムパスワードを自動生成する。
      - パスワードポリシーに準拠（大文字・小文字・数字・特殊文字を各1文字以上含む）。
      - 初回ログイン時のパスワード変更を強制しない（message_action = "SUPPRESS"）。
    - 管理者ユーザー属性:
      - username: 管理者メールアドレス
      - email: 管理者メールアドレス
      - email_verified: true（検証済み）
  - **ユーザーグループ管理**:
    - 管理者グループ「Admins」を自動作成する。
    - グループ設定:
      - グループ名: "Admins"
      - 説明: "管理者グループ"
      - 優先度: 1
    - 管理者ユーザーを自動的にAdminsグループに追加する。
  - **セキュリティ設定**:
    - User Pool Domain:
      - Cognitoドメインまたはカスタムドメインをデプロイパラメータで選択可能とする。
    - 高度なセキュリティ機能:
      - 異常なサインイン試行の検出を有効化する。
      - アダプティブ認証を有効化する。
    - デバイス追跡:
      - オプショナル設定とし、デプロイパラメータで有効/無効を指定可能とする。
  - **出力値**:
    - User Pool ID: アプリケーションの環境変数として使用
    - User Pool ARN: IAMポリシーでの参照用
    - User Pool Client ID: アプリケーションの環境変数として使用
```

```
  - User Pool Domain: OAuth フロー用
  - User Pool Endpoint: JWKSエンドポイントアクセス用
  - 管理者ユーザー情報: メールアドレス、ユーザー名、初期パスワード（機密情報として管理）

## 7. タグ戦略
- タグを付与できるリソース全てにタグを付ける。
- 以下のタグをデプロイパラメータで指定可能とする。
   - **project**
   - **env**
   - **owner**
   - **department**
- Nameタグには、以下のタグ値を組み合わせたプレフィックスを付与する。
   -  `project-env-`

---

以上、本システムのAWS詳細設計とする。
```

(4) Terraform設計書

````
# Terraform設計書

## 1. 概要
- TerraformのIaCコード設計方針を記載する。

## 2. Terraform要件
- Terraformバージョン: 1.9.8以上
- AWSプロバイダー: hashicorp/aws（5.74.0以上）
- バックエンド: S3

## 構成ファイル
- プロジェクトツリー
    ```
    project/infrastructure/core_infra2
    ├── iac_code
    │   ├── env
    │   │   ├── sample.tfvars
    │   │   └── sample_backend.tfvars
    │   ├── variables.tf
    │   ├── main.tf
    │   ├── provider.tf
    │   └── modules/
    ├── override.tf.local
    ├── tf_init.sh
    ├── tf_plan.sh
    ├── tf_apply.sh
    └── tf_destroy.sh
    ```

- ディレクトリ
  - env
     - Github ActionsのTerraformデプロイ時に参照されるtfversファイルを格納する。
     - sampleファイルを作成する。
  - iac_code
     - terraformコードを格納する。
     - 最低限、`main.tf`、`variables.tf`、`provider.tf`に分割する。
     - 可読性向上のため、`main.tf`の内容を`modules`ディレクトリ配下に分割して良い。

  - ローカルデプロイ用ファイル
     - override.tf.local
       ```
       terraform {
         # tfstateファイルをローカルに保存する
         backend "local" {
           path = "terraform.tfstate"
         }
       }
       ```
````

```
      - デプロイシェル（共通処理）
        - 第一引数にtfvarsファイルを要求する（必須）
        - 第二引数にbackend_tfvarsファイルを要求する（オプション）
        - 第一引数、第二引数はcore_infraディレクトリからの相対パスで指定する
        - バックエンド設定の判定
          - 第二引数が指定された場合：S3バックエンドを使用
          - 第二引数が未指定の場合：ローカルバックエンドを使用（`override.tf.local`を`iac_cod
e/override.tf`にコピー）
        - AWSプロファイルを「NCP-TF-HOMEWORK03」に変更
        - 処理の最後に`iac_code/override.tf`を削除する
      - デプロイシェル（個別処理）
        - tf_init.sh
          - S3バックエンド使用時：`terraform init -upgrade -backend-config=第二引数 -var
-file=第一引数`
          - ローカルバックエンド使用時：`terraform init -upgrade -var-file=第一引数`
        - tf_plan.sh
          - `terraform plan -var-file=第一引数`コマンドを実行する
        - tf_apply.sh
          - 第二引数が指定された場合：`terraform apply -var-file=第一引数 -auto-approve
`コマンドを実行する
          - 第二引数が未指定の場合：`terraform apply -var-file=第一引数`コマンドを実行する
        - tf_destroy.sh
          - 第二引数が指定された場合：`terraform destroy -var-file=第一引数 -auto-appro
ve`コマンドを実行する
          - 第二引数が未指定の場合：`terraform destroy -var-file=第一引数`コマンドを実行す
る

## 3. 使用例

### ローカルバックエンドでの実行
    ```bash
    # 初期化
    ./tf_init.sh sample.tfvars

    # プラン確認
    ./tf_plan.sh sample.tfvars

    # 適用
    ./tf_apply.sh sample.tfvars

    # 削除
    ./tf_destroy.sh sample.tfvars
    ```

S3バックエンドでの実行
    ```bash
    # 初期化
    ./tf_init.sh sample.tfvars sample_backend.tfvars

    # プラン確認
    ./tf_plan.sh sample.tfvars sample_backend.tfvars
```
```

```
    # 適用
    ./tf_apply.sh sample.tfvars sample_backend.tfvars

    # 削除
    ./tf_destroy.sh sample.tfvars sample_backend.tfvars
    ```

以上、本システムのTerraform設計とする。

付録C. ケース③で作成したドキュメント

(1) システム要件 ～作業者が作成～

```
## 概要

AWSのコストレポートをデータソースとしたBIツール

### 対象データ

- AWSの「エクスポート」機能を使用してS3にエクスポートされたコストレポートファイル
- AWS Organizationの管理アカウントから出力
  - 対象の組織アカウント: 約150アカウント

### 機能要件

- クエリエディタでSQLを入力して実行
  - 自然言語からクエリを生成する機能あり
- 出力は表形式とグラフ形式の 2 種類の出力があり
  - 生成AIを使用してグラフ出力
    - グラフ表示に使用するツールは選定中
  - 表は画面表示とエクスポートの2種類あり
    - 画面は1画面最大100行までのページングを実施
    - エクスポートはCSV形式
- 実行したクエリを記憶する機能あり
  - 記憶したクエリに対してタグをつけることができる。
  - nameタグや、その他のタグで検索可能

### システム構成

#### 運用環境

- AWS上に構築
- 構築にはterraformを使用する。
##### 使用するサービス

- EC2
- ECR
- DynamoDB
- S3
- Glue
- Athena
- ALB
- Cognito
```

```
##### AWSサービスの用途

###### IAM

- EC2のロールを管理
- terraform実行用のIAMユーザーを作成
- GitHubAction用にIDプロバイダを提供

###### EC2

- Amazonlinux2023
- 内部でdockerを起動
- ECRのアプリケーションリポジトリのlatestバージョンをpullして実行する
- ALBの背後に配置

###### ALB

- WEBサービスを公開
- ターゲットはEC2のオートスケーリンググループ

###### Route53

- システムのホストゾーンを作成
- ドメインは別アカウントで取得済みドメインのサブドメインを使用する

###### ECR

- アプリケーションコンテナイメージを格納

###### DynamoDB

- システムが記録する主要なデータを格納
  - クエリの実行履歴
  - クエリに紐づけたタグの情報

###### S3

- AWSのコストレポートファイル
- Athenaの実行結果
- terraformのtfstate

###### Glue

- AWSのコストレポートファイルのデータカタログを管理

###### Athena

- AWSのコストレポートファイルのデータカタログに対してクエリを実行

###### Cognito

- ユーザー認証を実行
```

```
  - Hosted UIを使用して認証、認証成功後、WEBシステムに認証ユーザー情報を送信する。

#### CD/CI

- CI
  - アプリケーション
    - アプリケーションのテストを実装
    - git push、pullrequestのタイミングで実装したテストを実行
  - インフラストラクチャ
    - terraformで実装
    - git push、pullrequestのタイミングでCI実行
      - tflint
      - tfvars
      - terraformのテストコード
- CD
  - GitHubアクションで実行
    - GitHubのUI「Action」タブ上で実行
    - タグをつけたコミットを選択してデプロイ
      - デプロイ先もUI上で選択する
  - CDプロセス
    - [GitHub Action] アプリケーションのコンテナをビルドしてECRにpushする
    - [GitHub Action] EC2のオートスケーリンググループに更新を指示
    - [AWS] EC2のオートスケーリンググループ内のEC2インスタンスを再起動
    - [AWS] EC2起動時にECRのlatestバージョンのイメージをpullしてdockerを起動
```

(2) AWSシステム構成 ～「(1) システム要件」からAIが作成～

```
# AWSシステム構成書

## 1. システム概要
AWSコストレポートをデータソースとしたBIツールで、AWS Organizationの約150アカウントのコストデータを可視化・分析します。

## 2. アーキテクチャ図
    ```
    +------------------+   +------------------+   +------------------+
    | | | | | |
    | AWS コスト | ---> | S3 | ---> | Glue |
    | (150アカウント) | | (コストレポート) | | (データカタログ) |
    | | | | | |
    +------------------+   +------------------+   +------------------+
                                                                |
                                                                v
    +------------------+   +------------------+   +------------------+
    | | | | | |
    | Cognito認証 | <--- | ALB | <--- | Route53 |
    | (ユーザー認証) | | (ロードバランサ) | | (DNS/ドメイン) |
    | | | | | |
    +------------------+   +------------------+   +------------------+
                                 |
                                 v
    +------------------+   +------------------+   +------------------+
    | | | | | |
    | DynamoDB | <--- | EC2 (ASG) | <--- | ECR |
    | (クエリ履歴) | | (アプリケーション) | | (コンテナレジストリ) |
    | | | | | |
    +------------------+   +------------------+   +------------------+
                                 |
                                 v
                        +------------------+
                        | |
                        | Athena |
                        | (クエリ実行) |
                        | |
                        +------------------+
    ```

## 3. コンポーネント構成

### 3.1 ネットワーク構成
- **Route53**
  - ドメイン管理（既存ドメインのサブドメインを使用）
  - ALBへのルーティング設定
```

```
### 3.2 ロードバランサ
- **Application Load Balancer (ALB)**
  - インターネット向け
  - HTTPSリスナー（443ポート）
  - ターゲットグループ: EC2オートスケーリンググループ
  - アクセス制御: ソースIP制限（IPv4）

### 3.3 コンピューティング
- **EC2 オートスケーリンググループ**
  - インスタンスタイプ: t3a.medium（最新世代のバースト可能インスタンスタイプ）
    - vCPU: 2
    - メモリ: 4 GiB
    - ネットワークパフォーマンス: 最大5Gbps
  - 最小/最大/希望サイズ: 2/4/2
  - 起動テンプレート: Amazon Linux 2023（LTSバージョン）
  - ヘルスチェック: HTTP 80ポート
  - スケーリングポリシー:
    - スケールアウト: CPU使用率70%を3分間超えた場合
    - スケールイン: CPU使用率30%を5分間下回った場合
  - インスタンス更新ポリシー: ローリングアップデートを有効化

### 3.4 ストレージ
- **Amazon S3**
  - バケット1: コストレポート用
    - ライフサイクルルール: 3年後にGlacier Deep Archiveへ移行
    - 暗号化: SSE-S3
  - バケット2: Athenaクエリ結果用
  - バケット3: Terraform状態管理用

- **DynamoDB**
  - テーブル1: クエリ履歴
    - パーティションキー: user_id
    - ソートキー: timestamp
  - テーブル2: クエリタグ
    - パーティションキー: query_id
    - ソートキー: tag_name
  - オンデマンドキャパシティーモード

### 3.5 データベース
- **Amazon Athena**
  - クエリ実行タイムアウト: 1分
  - 同時実行数: 10
  - ワークグループ: cost_analysis_workgroup

### 3.6 認証・認可
- **Amazon Cognito**
  - ユーザープール: ユーザー認証用
  - アプリクライアント: Webアプリケーション用
  - ホスト型UIを使用
  - グループ: 初期は全ユーザーにフルアクセス
```

```
## 4. データフロー

1. **データ収集**
    - AWSコストレポートが1日1回S3にエクスポート
    - Glueクローラーが S3のデータを定期的にスキャン
    - Glueデータカタログが更新

2. **クエリ実行**
    ```
    ユーザー
    → ALB (HTTPS 443)
    → EC2インスタンス (Dockerコンテナ)
    → Athena (クエリ実行)
    → S3 (結果保存)
    → ユーザーに結果表示
    ```

3. **クエリ保存**
    - 実行されたクエリはDynamoDBに保存
    - タグ付け機能で後から検索可能

## 5. セキュリティ

### 5.1 ネットワークセキュリティ
- VPC構成
  - パブリックサブネット: ALB配置
  - プライベートサブネット: EC2インスタンス配置
- セキュリティグループ
  - ALB: 443ポートのみ許可
  - EC2: ALBからのみ80ポート許可

### 5.2 データ保護
- データ暗号化
  - 保存時: S3バケットポリシーで暗号化を強制
  - 転送時: TLS 1.2以上を強制
- アクセス制御
  - IAMロールを使用した最小権限の原則
  - ソースIP制限（IPv4）

## 6. 監視・運用

### 6.1 監視項目
- **CloudWatch アラーム**
  - EC2 CPU使用率 > 80% (5分間)
  - ALB ターゲット応答時間 > 5秒 (5分間)
  - 5xxエラー率 > 1% (5分間)

### 6.2 ログ管理
- **CloudWatch Logs**
  - ALBアクセスログ
  - EC2システムログ
  - アプリケーションログ
```

```
## 7. バックアップ・ディザスタリカバリ

### 7.1 バックアップ
- **S3**
  - バージョニングを有効化
  - マルチリージョンレプリケーション（バックアップ用）
- **DynamoDB**
  - ポイントインタイムリカバリ (PITR) を有効化

### 7.2 ディザスタリカバリ
- マルチAZ構成で高可用性を確保
- 別リージョンへのフェイルオーバー手順をドキュメント化

## 8. コスト管理

### 8.1 コスト見積もり
- EC2: t3.medium x 2台 (常時稼働)
- EBS: gp3 50GB x 2台
- ALB: 1台
- S3: 500GB (月間増加量: 10GB)
- データ転送: 100GB/月

## 9. デプロイメント

### 9.1 インフラストラクチャ
- **Terraform**
  - モジュール化された構成
  - 環境ごとの変数管理
  - リモートステート管理 (S3 + DynamoDB)

### 9.2 CI/CDパイプライン
- **GitHub Actions**
  - プルリクエスト時: テスト実行
  - マージ時: ステージング環境へデプロイ
  - タグプッシュ時: 本番環境へデプロイ

## 10. 今後の拡張性

### 10.1 マルチテナント化
- CognitoグループとDynamoDBのパーティションキーを活用したデータ分離
- LakeFormationを使用したきめ細かいアクセス制御

### 10.2 パフォーマンス最適化
- クエリキャッシュの導入
- コストデータの集計テーブルの事前計算

### 10.3 機能拡張
- カスタムレポートの定期実行とメール配信
- コスト予測機能の追加
- コスト最適化の推奨事項の自動生成
```