複雑なサービスメッシュはもう古い？
Cloud Run で手軽に始めるサービスメッシュ

伊藤 裕一

Google Cloud
アプリケーション モダナイゼーション コンサルタント

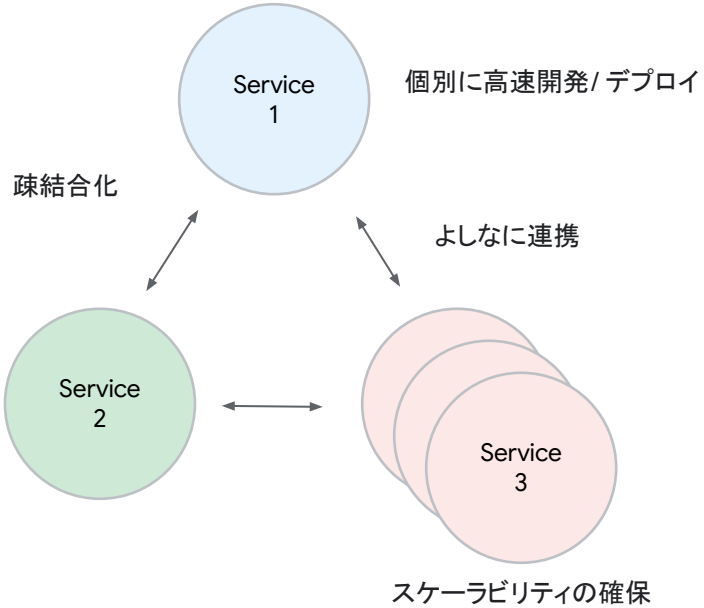
Google Cloud Next Tokyo

伊藤 裕一

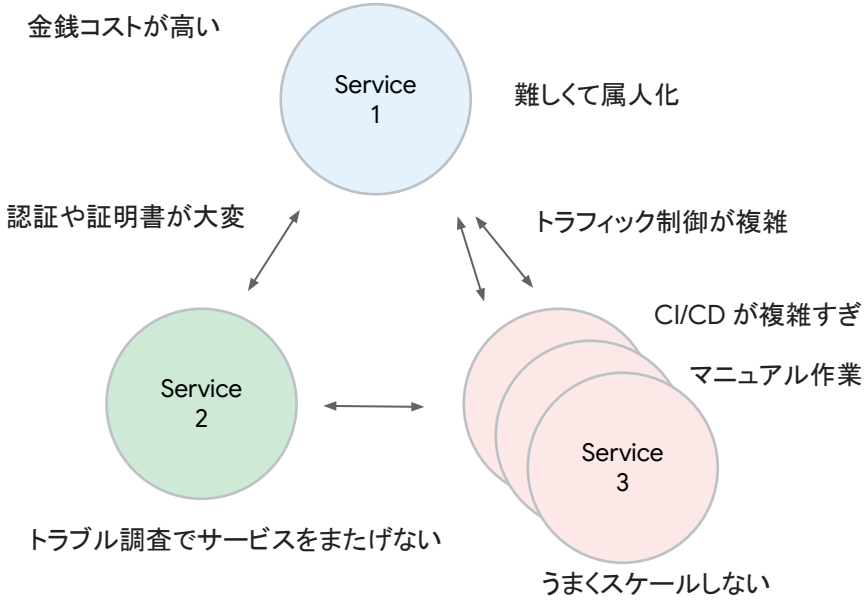
Google Cloud
アプリケーション モダナイゼーション
コンサルタント

マイクロサービスの理想と現実

理想



現実



Kubernetes と Istio の学習コストと運用コストが重い

マイクロサービスの理想と現実

高機能なものの難しくて手間のかかる K8s と Istio ではなく、

Cloud Service Mesh と Cloud Run でシンプルに解決できるかも?

目次

免責条項

2025 年 7 月 1 日時点では Cloud Service Mesh を Cloud Run に利用することは Pre-GA となります。

当資料の内容は将来的に変更される可能性があります。

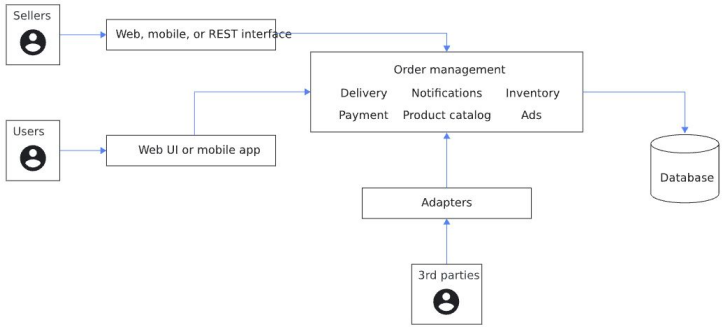
CSM と GKE の構成から、CSM と Cloud Run の構成へ移行を推奨するものではありません。

01. そのサービスメッシュ、複雑すぎませんか？

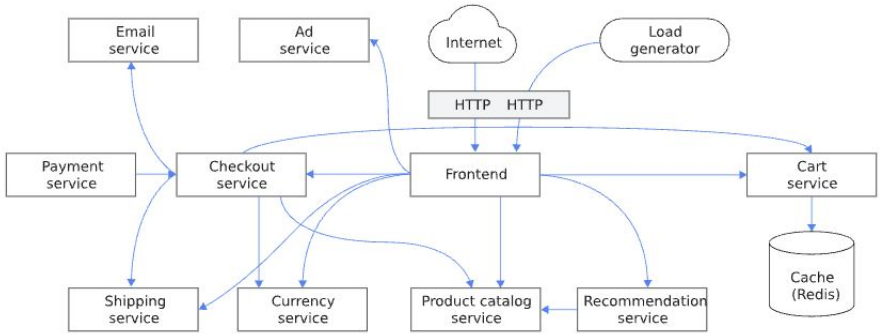
マイクロサービスの普及と、その課題

モノリスな設計の課題を解決するためにマイクロサービスな設計が普及

分散したアプリケーションを繋ぐ難しさが新たな課題となる



モノリシックｅコマース アプリケーションの例

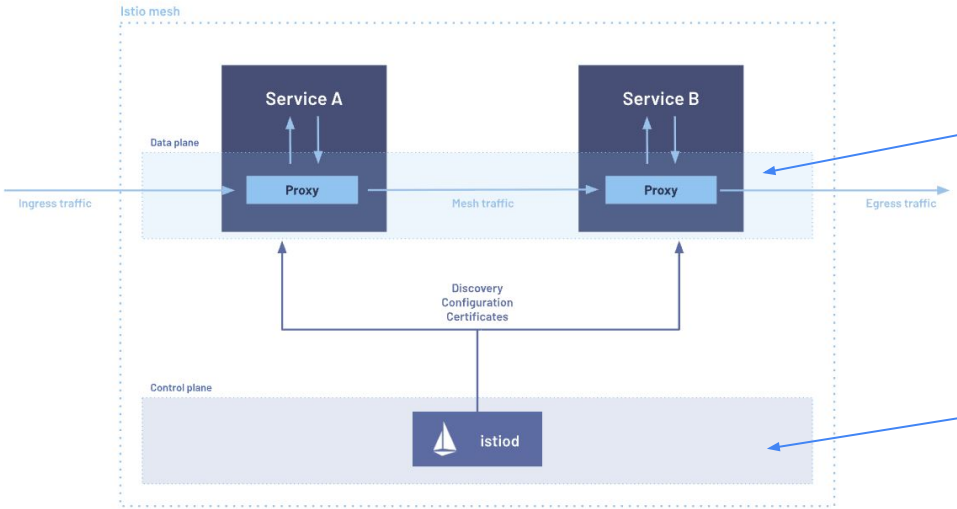


マイクロサービス化されたコマースアプリケーションの例

Kubernetes 上のサービスメッシュが抱える課題

サービスメッシュはマイクロサービスの課題を解決するソリューションの 1 つ

動けば便利だが、導入と運用の負担が大きいのが玉に傷



サイドカーモードの Istio のアーキテクチャ図

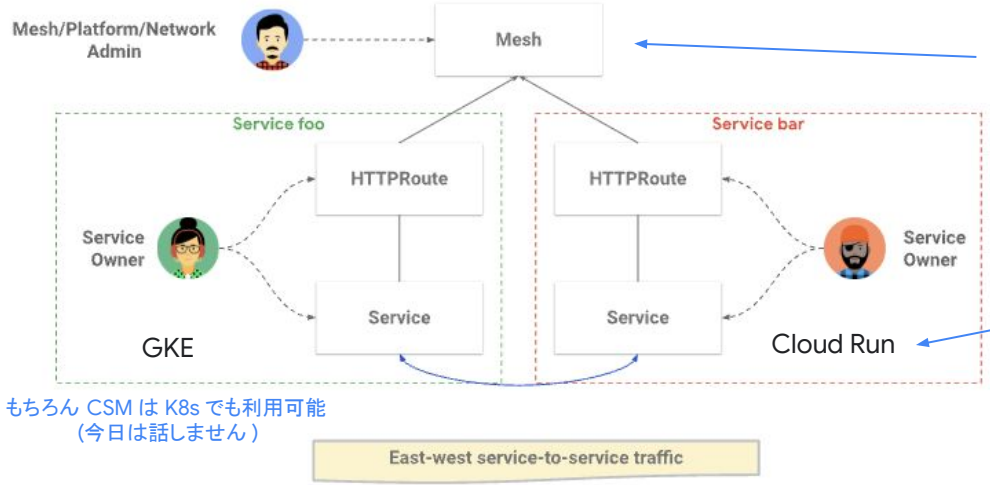
コンテナアプリの複雑性の増加
- サイドカーを追加する設定
- トラブルシューティングが難しい

アプリ基盤の複雑性の増加
- Kubernetes の導入と維持
- Istio (サービスメッシュ) の導入と維持

Cloud Service Mesh (CSM) と Cloud Run が課題解決

CSM は Google Cloud が提供するマネージドなサービスメッシュ

Cloud Run と組み合わせるとサービスメッシュの導入と維持コストが低減



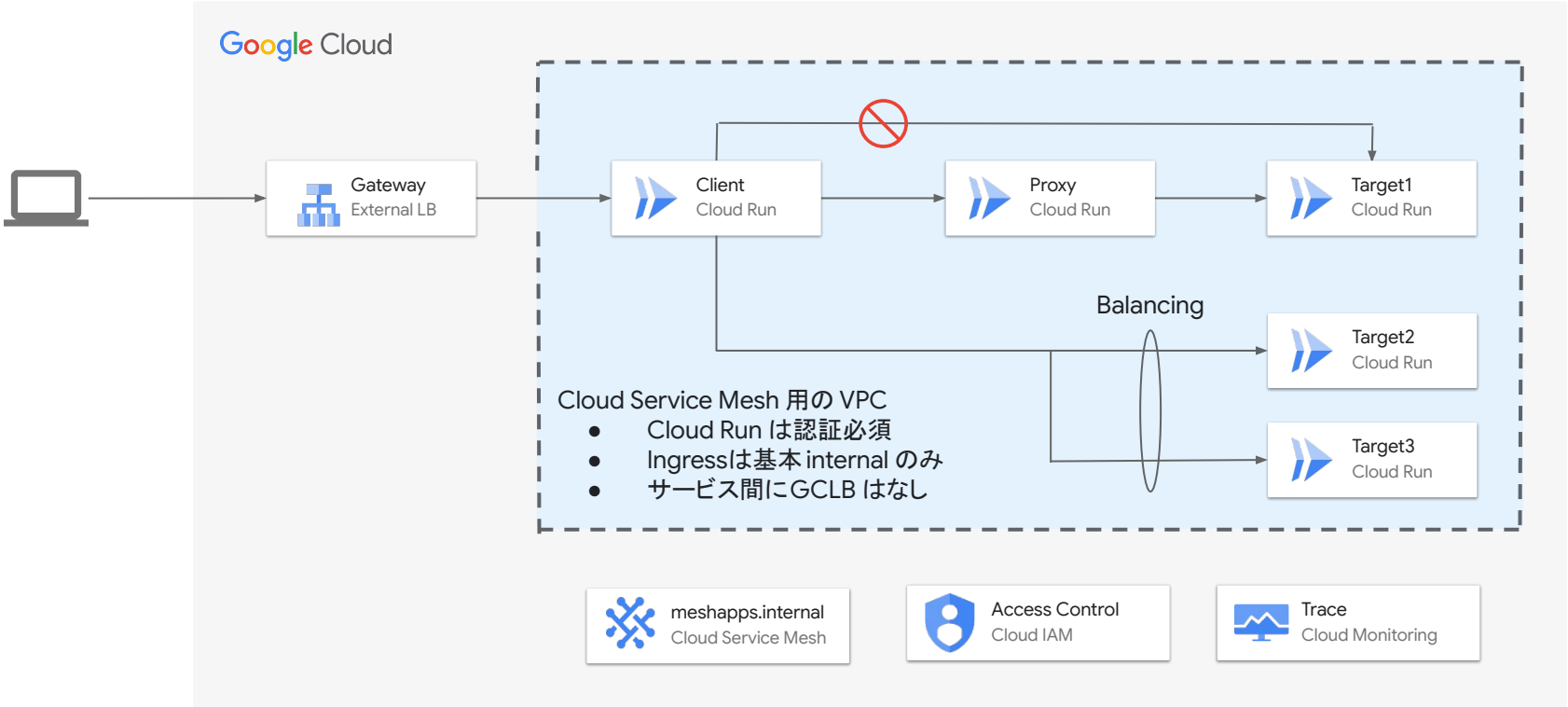
Cloud Service Mesh で管理するGKE/Cloud Run のサービスメッシュ環境

アプリ基盤の複雑性の緩和
- コンテナ基盤は Google 任せ
- サービスメッシュ基盤も Google 任せ

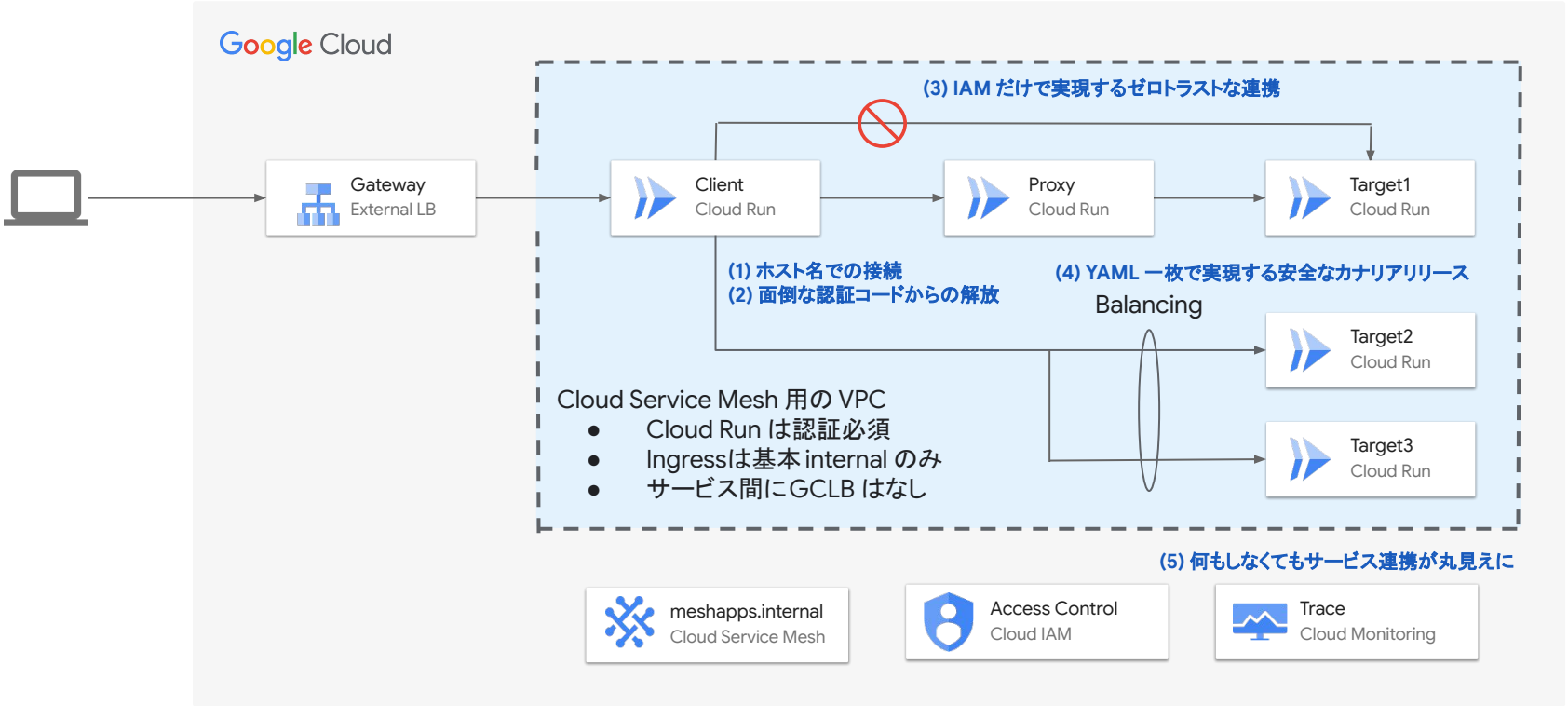
コンテナアプリの複雑性の緩和
- Cloud Run は K8s より簡単
- サイドカーは勝手に挿入される
- アクセス制御は IAM 任せ

02. CSM で Cloud Run 開発 / 運用はこう変わる

本日のお題：サンプルのマイクロサービスの構成図



本日のお題：サンプルのマイクロサービスの構成図

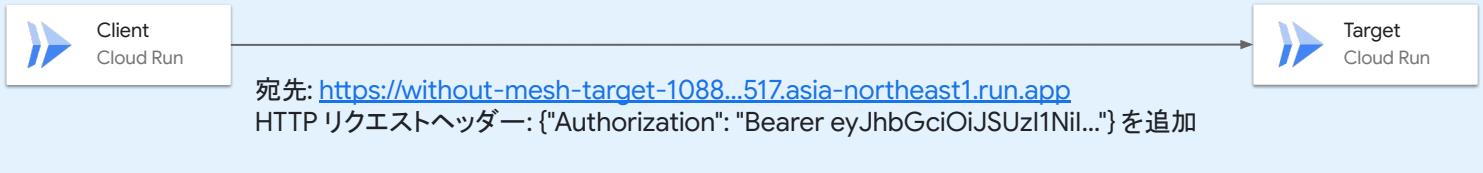


機能 1-2: サービス間の通信作法がシンプルに

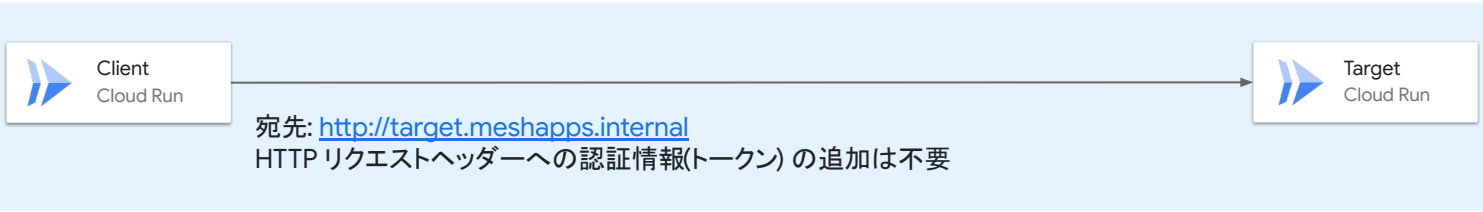
通信先の指定をサービスメッシュのホスト名で可能となる

Cloud Run 間の通信に認証ヘッダー (トークン) 付与が不要となる

CSM なし

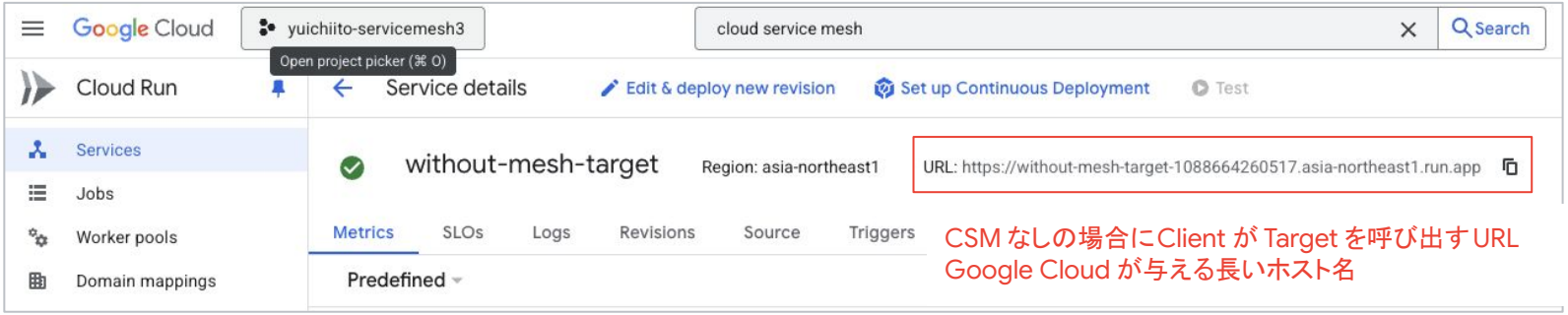


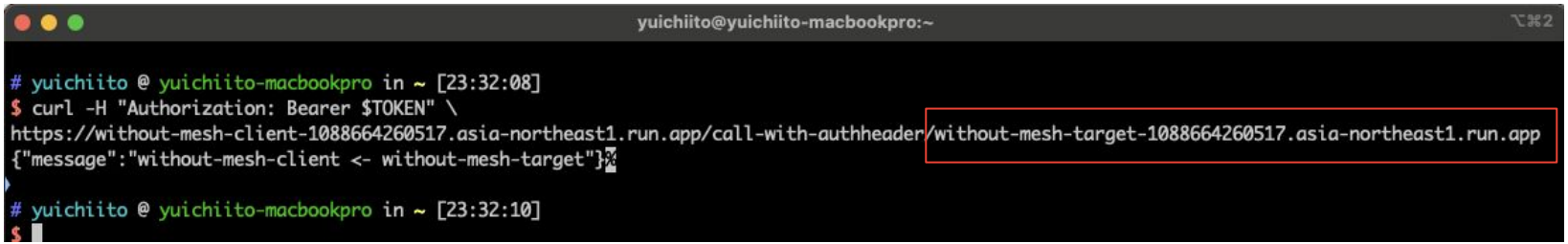
CSM あり



機能 1: アプリ内の宛先名が CSM 内部ホスト名に

CSM を使わない場合





機能 1: アプリ内の宛先名が CSM 内部ホスト名に

CSM を使う場合

```
1  name: "route-proxy"
2
3  hostnames:
4  - "proxy.meshapps.internal"
5
6  meshes:
7  - "projects/yuichiito-servicemesh3/locations/global/meshes/meshapps"
8
9  rules:
10 - action:
11     destinations:
```

宛先に利用されるCSM 内だけでのホスト名
後述するHTTP Route の定義でBackend Service とマッピングの宣言をする

```
yuichiito@yuichiito-macbookpro:~

# yuichiito @ yuichiito-macbookpro in ~ [23:36:05]
$ curl -H "Authorization: Bearer $TOKEN" http://${ELB_IP}/call/proxy.meshapps.internal
{"message":"client <- proxy"}%
```

CSM ありの場合に、Client が Proxy を呼び出す URL (http:// は内部で付与)
上記の HTTP Route 定義で与えたCSM の内部ホスト名を使える

機能 2: CSM ではこの認証コード、もう要りません

```
auth_req = google.auth.transport.requests.Request()
id_token = google.oauth2.id_token.fetch_id_token(auth_req, target_url)
headers = {"Authorization": f"Bearer {id_token}"}

try:
    async with httpx.AsyncClient() as client:
        response = await client.get(
            target_url, headers=headers, timeout=5
        )
        response.raise_for_status()  # Raise an exception for 4xx/5xx
```

Google API を使って
トークンを取得

トークンをヘッダのデータに追
加

宛先への HTTP リクエストに
トークンを含むカスタムヘッダを指
定。
これをやらないと403 Errorに

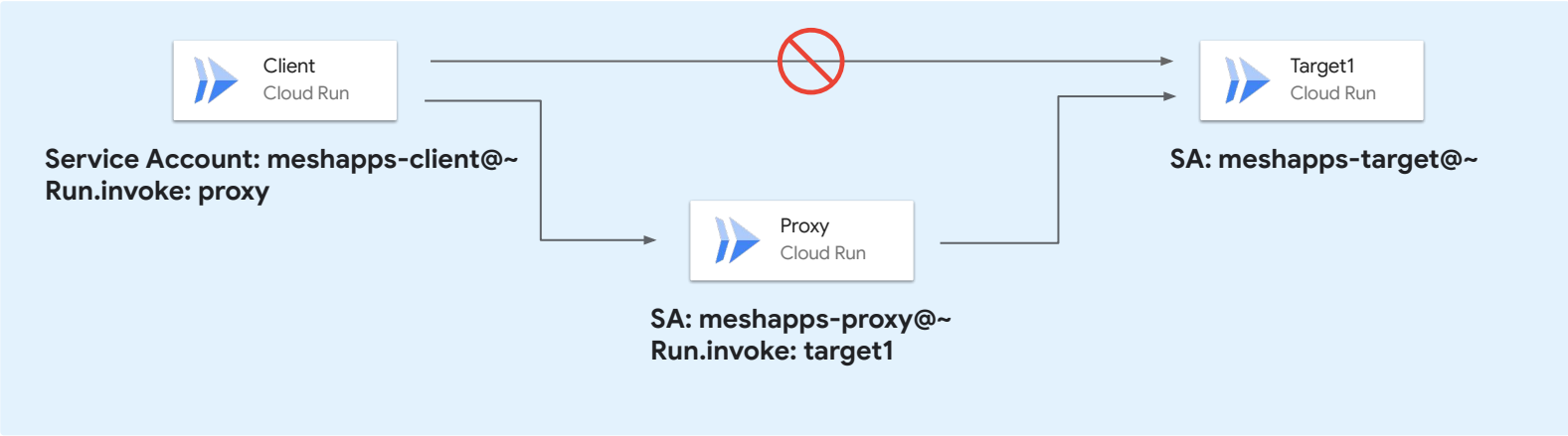
CSM を**利用しない場合** のコード

※ 分かりやすさ優先のコードです。本来はToken のキャッシュなどをしてください

機能 3: mTLS 相互認証とアクセス制御の実行例

誰と誰が連携できるかを IAM で定義 (CSMなしでも可能)

mTLS によりクライアントとサーバーが互いに相手を認証。通信の HTTPS 化

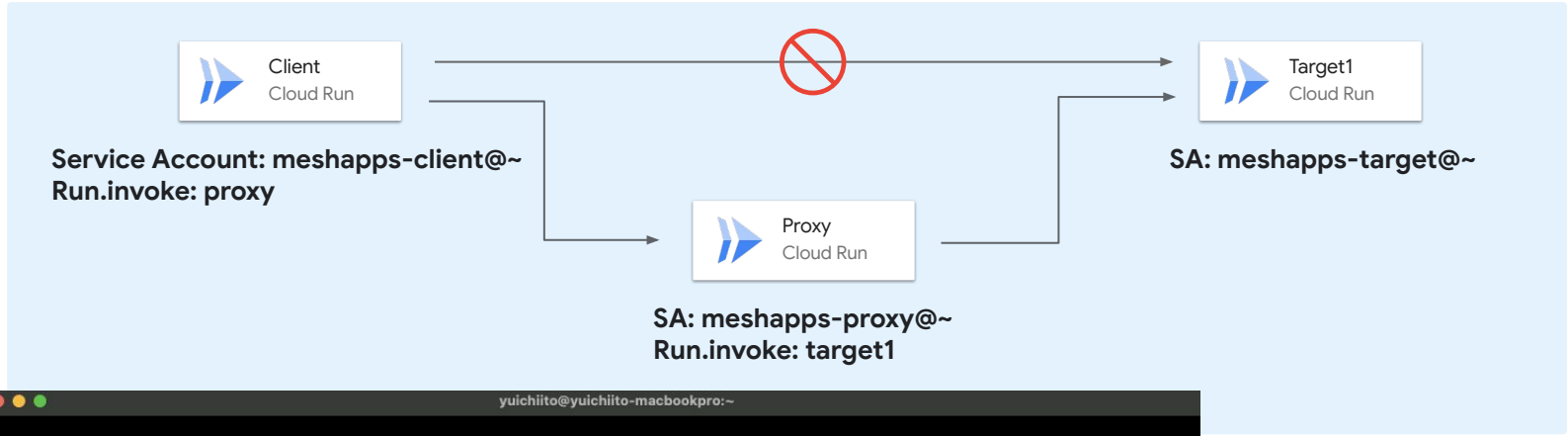


機能 3: mTLS 相互認証とアクセス制御の実行例

```
$ curl -H "Authorization: Bearer $TOKEN" http://${ELB_IP}/call/target1.meshapps.internal

> 403 Forbidden
> Your client does not have permission...
```

Client は Target1 を呼ぶ権限がないので、
アクセスすると403 Error Forbidden で失敗
(ログが多いため、主要箇所のみ抜粋




```
yuichiito@yuichiito-macbookpro:~

# yuichiito @ yuichiito-macbookpro in ~ [16:11:02]
$ curl -H "Authorization: Bearer $TOKEN" http://${ELB_IP}/call/proxy.meshapps.internal
{"message":"client <- proxy"}%

# yuichiito @ yuichiito-macbookpro in ~ [16:11:27]
$ curl -H "Authorization: Bearer $TOKEN" http://${ELB_IP}/call/proxy.meshapps.internal/target1.meshapps.internal
{"message":"client <- proxy <- target1"}%
```

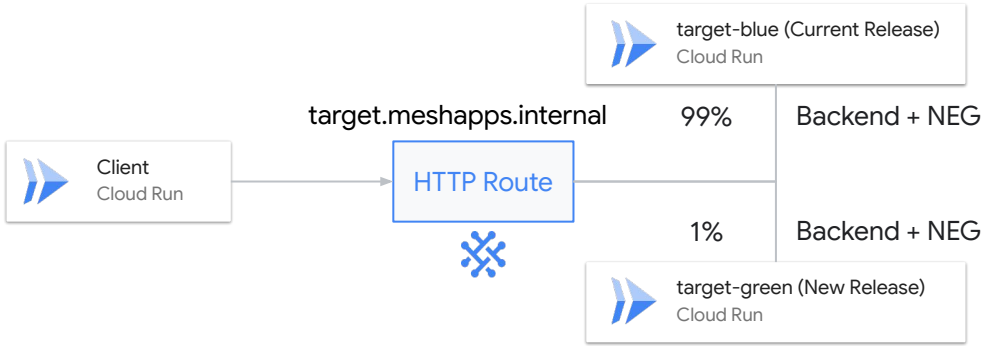
Client -> Proxy と Proxy -> Target1 は
アクセス権限があるので通信が成功

機能 4: HTTP Route による内部のトラフィック管理

Cloud Load Balancer ではなく CSM の HTTP Route 機能でトラフィック管理

HTTP Route で利用できる機能の例

- カナリアリリース
- A/B テスト
- リトライ処理
- サーキットブレーカー
- フォールトインジェクション
- その他



カナリアリリースの例(新サービスに1% だけ流す)

機能 4: Client から Target2, 3 へのバランシング例

target > ! http_route_target23.yml > abc name

```yaml
name: "route-target23"

hostnames:
- "target23.meshapps.internal"

meshes:
- "projects/yuichiito-servicemesh3/locations/global/meshes/meshapps"

rules:
- action:
    destinations:
    - serviceName: "projects/yuichiito-servicemesh3/locations/global/bac
      weight: 50
    - serviceName: "projects/yuichiito-servicemesh3/locations/global/bac
      weight: 50
```

HTTP Route 定義にて宛先 target23.meshapps.internal を作成
target2, target3 に 50:50 でバランシング

yuichiito@yuichiito-macbookpro:~

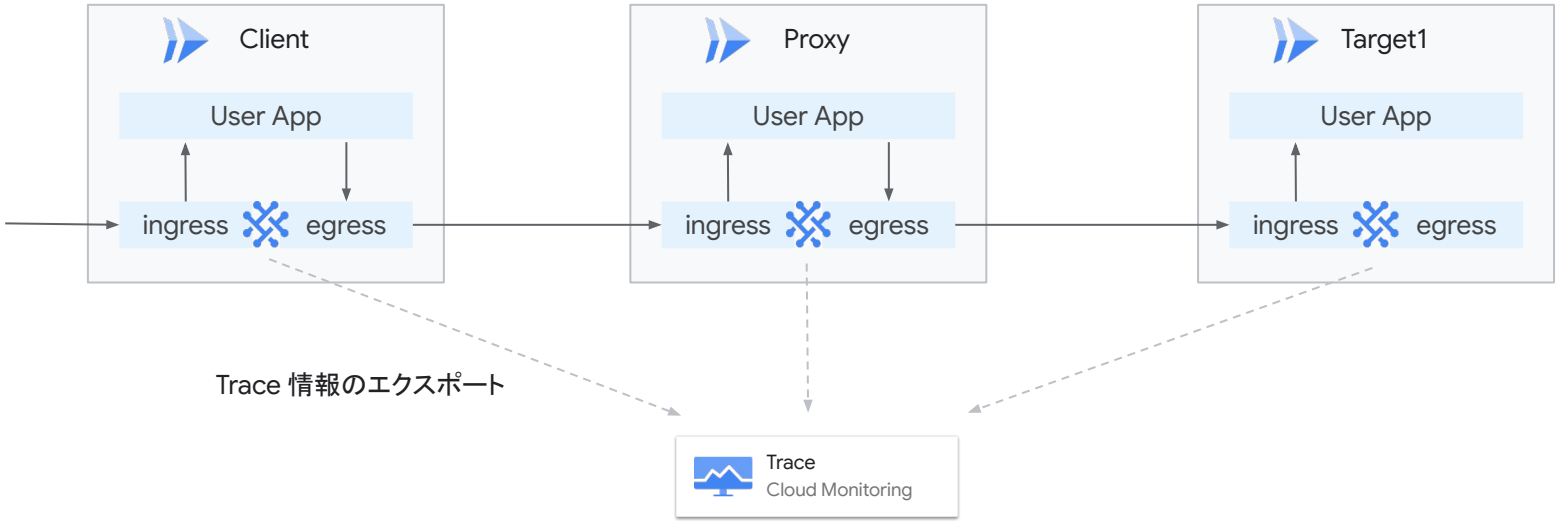
```
# yuichiito @ yuichiito-macbookpro in ~ [16:20:10]
$ for i in {1..10}
do
  curl -H "Authorization: Bearer $TOKEN" "http://${ELB_IP}/call/target23.meshapps.internal"
  echo
done
{"message":"client <- target3"}
{"message":"client <- target2"}
{"message":"client <- target3"}
{"message":"client <- target3"}
{"message":"client <- target2"}
{"message":"client <- target2"}
{"message":"client <- target2"}
{"message":"client <- target2"}
{"message":"client <- target3"}
{"message":"client <- target2"}
```

Client への応答が Target2, Target3 でほぼ 1:1 にバランシングされている

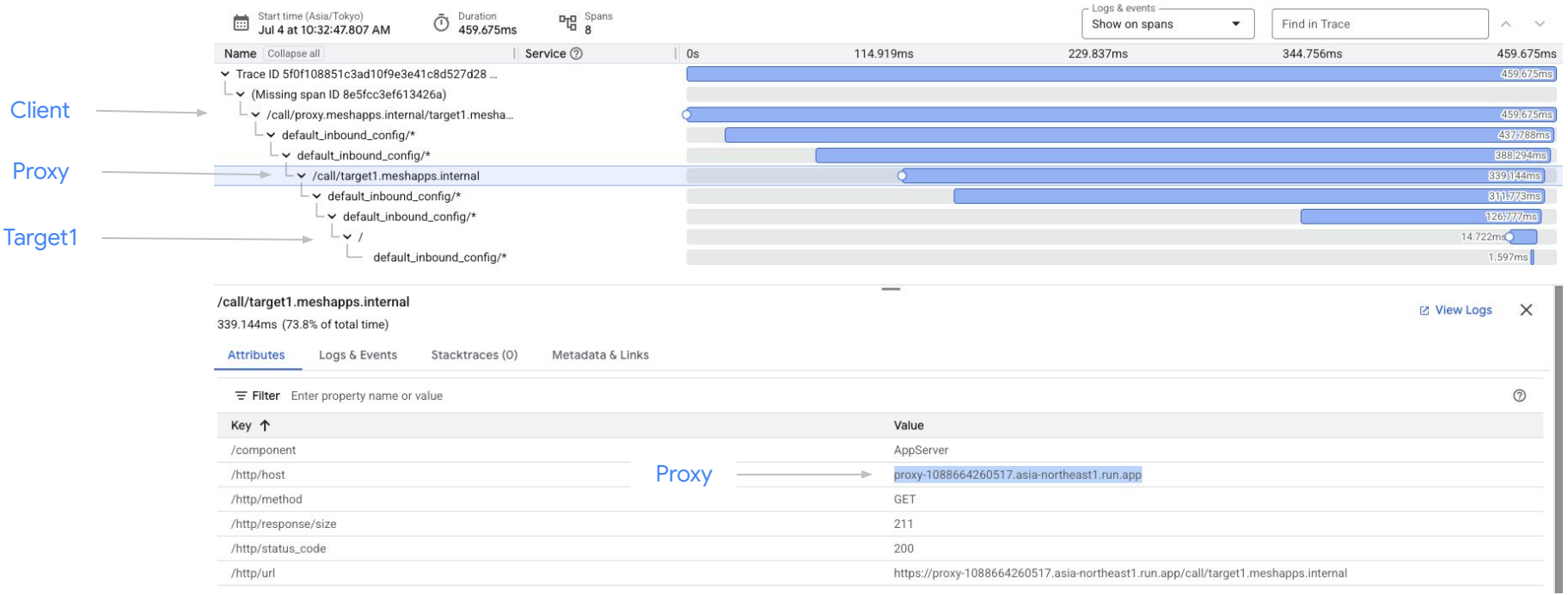
機能 5: CSM と Cloud Trace の自動連携

サービス間の通信では HTTP ヘッダでトレース ID を伝播

サイドカーが自動で Cloud Trace に送信し、サービス間の連携をトレース



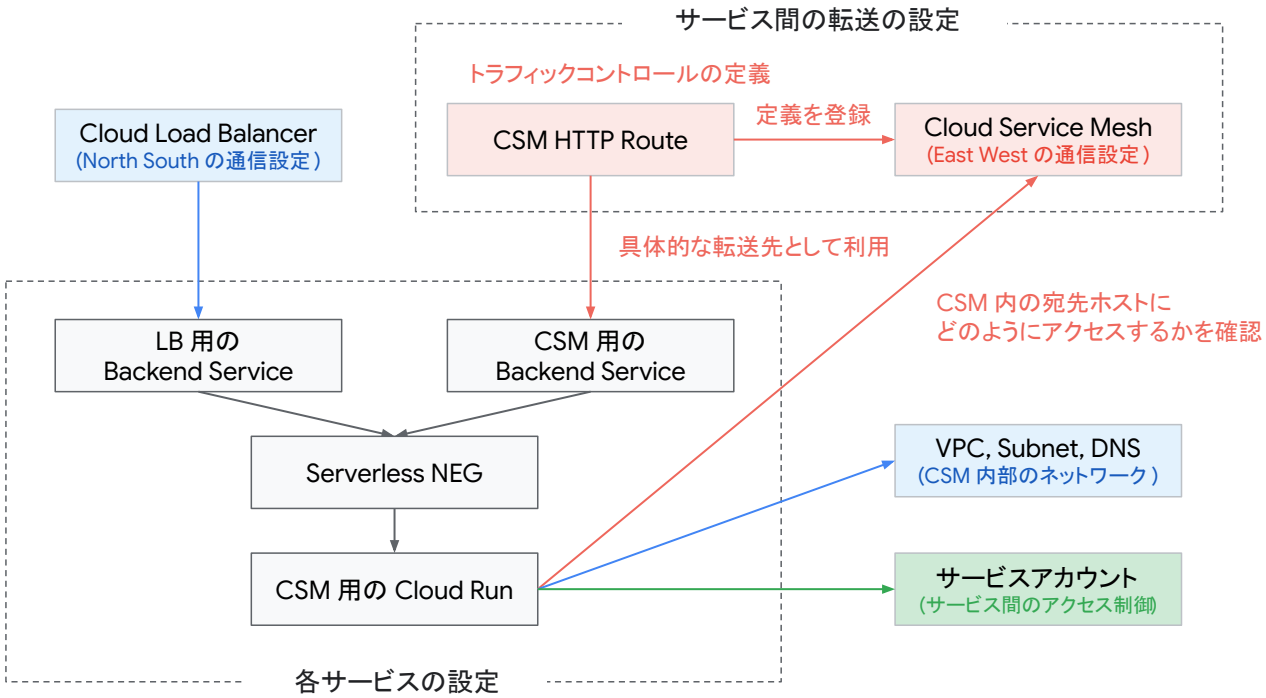
機能 5: Client -> Proxy -> Target の自動トレースの例



03. Cloud Run on CSM の構築ウォークスルー

※ コマンドサンプルや YAML サンプルは配布資料にあるため、ハイレベルな説明のみを実施します

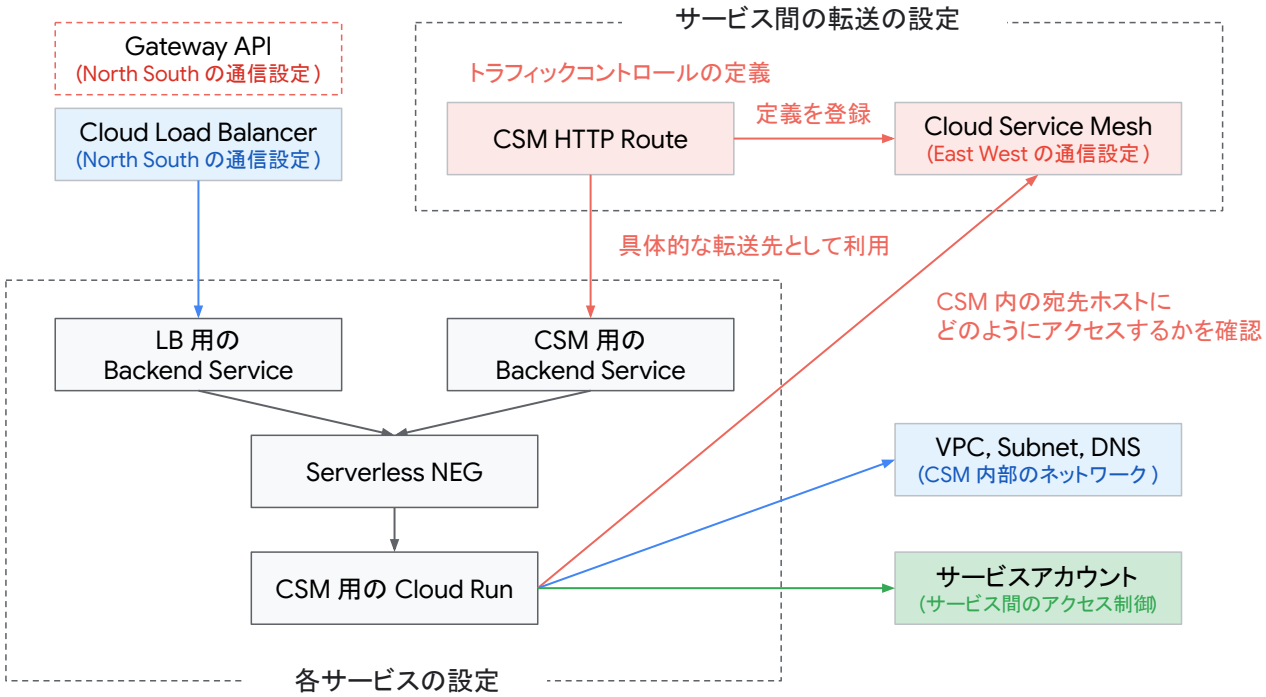
設定の大分類（CSM, Cloud Run, Network, SA）



設定の大分類 (CSM, Cloud Run, Network, SA)

現時点では Gateway API の利用には GKE や GCE(Envoy) などが必要です。Cloud Run 単体では機能提供していません。
従来ながらの Application Load Balancer の利用を推奨します。

CSM 内部利用のみのサービスにはこれら CLB と LB 用の バックエンドサービスの設定は不要です。

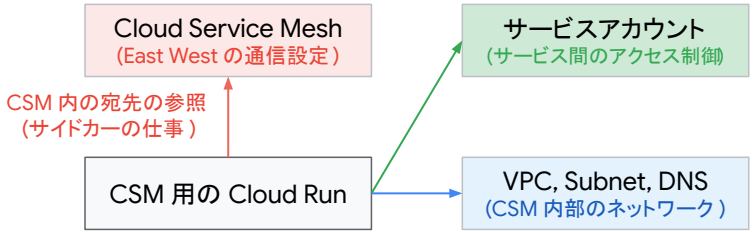


手順 1: CSM 内部ネットワークと、CSM 基盤の構築

1. CSM 内部**ネットワーク** の構築と、**サービスアカウント** の作成 (IaC の手動適用が一般的)
   a. CSM や Cloud Run に必要な Google Cloud API の有効化
   b. CSM 専用の VPC や Subnet の作成。外と隔絶することを推奨。必要に応じて外接用の Cloud NAT や PSC (Cloud SQL などに接続)などを作成
   c. Subnet で Private IP Google Access を有効化(隔絶環境でも Google API を呼ぶため)
   d. Cloud Run のサービスごとの SA 作成
      - CSM 利用に必要な権限を設定
      - SA では run.invoker は設定しない (全ての Cloud Run を呼べるようになってしまうので、アクセス制御ができなくなる。
      - (Run.invoker は Cloud Run の設定で個別に設定する。次ページで解説)

2. **Cloud Service Mesh** の基盤構築の流れ (IaC の手動適用を推奨)
   a. CSM 構成の YAML ファイルを作成。gcloud コマンドで展開
   b. CSM 内部用の DNS ゾーンとレコードの作成

手順 2: CSM 用に Cloud Run を構成

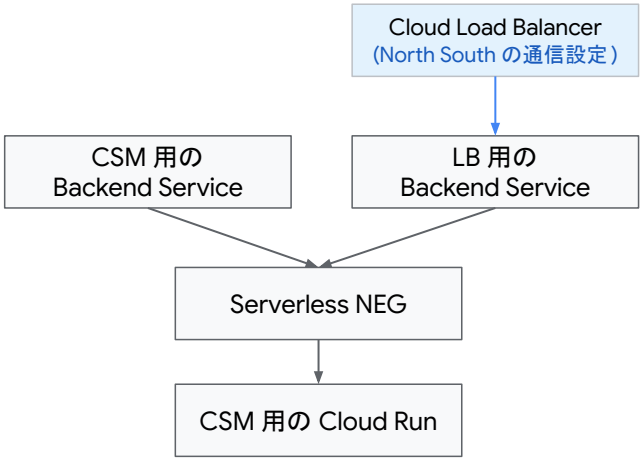


- **Cloud Service Mesh** に参加
- Cloud Service Mesh 用の**ネットワークに参加**
  - VPC を指定
  - Subnet を指定
  - VPC Direct Egress の All 設定で CSM 内の通信に VPC の利用を強制
- CSM **ネットワーク** 外からのアクセスを可能な限り遮断（Ingress の設定）
  - Internal + Load Balancer (CSM 外部から ELB/ILB でアクセスされるもののみ)
  - Internal (基本はこれ)
- 外部から直接接続しない内部サービスは必ず **Cloud Run の認証を強制** 。外部接続のフロントなどは認証がなくてもよい場合がある
- サービスごとに**個別のサービスアカウントを指定** （デフォルト SA の使用は非推奨）
- **IAM ではなく Cloud Run の権限設定** で、どのサービスアカウントがどの Cloud Run サービスを呼べるかを個別に定義。ここでメッシュ内のアクセス制御を設定
- これらは全て CI で設定することを推奨。ソースコードのビルド機能も検討可能

手順 3: CSM 用 Cloud Run へのバックエンドの作成

- HTTP Route 用のバックエンドの作成（CI での設定を推奨）
  a. Serverless NEG の作成
  b. CSM 用の Global な Backend Service の作成 (ロードバランシング スキーマが CSM 専用)
  c. Backend Service に NEG を結びつけ

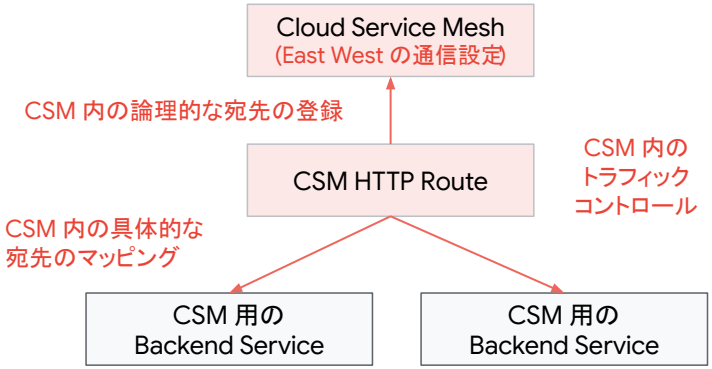
- CSM 外から CSM 内に接続するための従来ながらの ELB / ILB の作成
  - 設定は一般的なものなので割愛
  - 外部接続の細かなトラフィック調整が必要なら CD の定義を推奨



手順 4: HTTP Route で CSM 内のホスト名の登録

1. HTTP Route の YAML ファイルを作成
   - 参加するサービスメッシュの宣言
   - 内部での接続に使うホスト名の定義
   - CSM 用バックエンドを指定してトラフィックコントロールの定義

2. HTTP Route の YAML をインポートして、CSM 用の転送設定をする
   - YAML ファイルの Git Push をトリガーとした CD での適用を推奨




```
name: "route-target23"

hostnames:
- "target23.meshapps.internal"

meshes:
- "projects/yuichiito-servicemesh3/locations/global/meshes/meshapps"

rules:
- action:
    destinations:
    - serviceName: "projects/yuichiito-servicemesh3/locations/global/ba
      weight: 50
    - serviceName: "projects/yuichiito-servicemesh3/locations/global/ba
      weight: 50
```

論理的な宛先 (CSM 内のホスト名)

参加する CSM

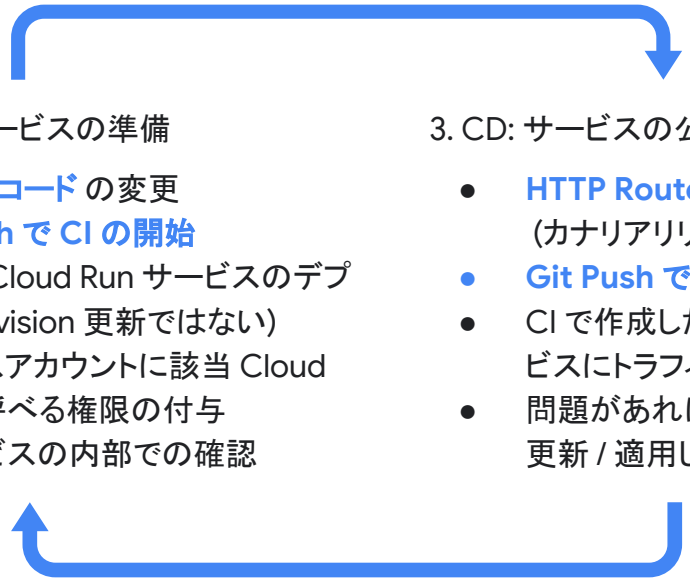
トラフィック制御のルールと具体的な転送先

HTTP Route の定義ファイルの例

開発 (CI) と運用 (CD) の「関心事の分離」の実現例

CSM は規則的な設定項目が多いので、手動での運用はトラブルを生む可能性が高い

IaC (Terraform など)や CI/CD (Cloud Build, GitHub Actions など) の利用を推奨



1. CSM 環境の構築

- CSM ネットワークの構築
- CSM の構築
- サービスアカウントの作成

2. CI: 開発とサービスの準備

- **アプリのコード** の変更
- **Git Push で CI の開始**
- 新規で Cloud Run サービスのデプロイ (Revision 更新ではない)
- サービスアカウントに該当 Cloud Run を呼べる権限の付与
- 新サービスの内部での確認

3. CD: サービスの公開

- **HTTP Route ファイル** の更新 (カナリアリリースなどから開始)
- **Git Push で CD の開始**
- CI で作成した新規 Cloud Run サービスにトラフィックを流す
- 問題があれば再度 HTTP Route を更新 / 適用し、切り戻す

今日のまとめ

Before: Kubernetes + Istio 構成では、インフラ（K8s, Istio）とアプリ（サイドカー）の両方が複雑だった

After: Cloud Run + CSM 構成を使えば、インフラはGoogle マネージドになり、開発者はシンプルなアプリ開発と、宣言的なトラフィック管理（HTTP Route）に集中できる

結論:「インフラの複雑さ」と「開発の関心事」を綺麗に分離できる。それがCloud Run + CSM の最大の価値

※「GKE Autopilot + Managed CSM (クラスタ内コントロールプレーンでない)」の構成も運用負担の軽減にはおすすめです。
まずは「コンテナ基盤に Kubernetes を使いたいか Cloud Run を使いたいか」を軸に比較するのがよいと思います。