GO の機械学習システムを支える MLOps 実践

鈴木 隆史

株式会社Mobility Technologies
開発本部 AI技術開発部 MLエンジニアリング 第一グループ
グループリーダー

自己紹介

鈴木 隆史 | Takashi Suzuki

AI技術開発部 MLエンジニアリング 第一グループ

- **2019年DeNA入社**

  機械学習の実験基盤やパイプラインの設計開発を担当

- 2020年Mobility Technologies転籍

TAXI GOes Next.

ABOUT GO

タクシーに未来をのせて

Taxi Goes Next

『GO』という名称には、行く、進む、
向かうといった言葉そのものの意味に加え、
タクシーに乗車されるお客様はもちろん、
運行を行う乗務員の方など、
サービスを使用する全ての人の笑顔や幸せなど、
未来をのせて走るという想いを込めました。
またロゴには、
人々の暮らしや未来を俯瞰で捉え集約した姿として、
地球をモチーフにデザインしました。
オンデマンド交通であるタクシーだからこそ実現できる、
百人百様の移動ニーズに寄り添ったサービスを
創造していきます。

JapanTaxi × MOV

JapanTaxiとMOVが１つのアプリに！

『GO』は、『MOV』をベースとしたアプリで、
『MOV』提携タクシー車両に加えて
『JapanTaxi』アプリ提携タクシー車両へも
配車注文することができます。

タクシー配車アプリ GO

1

一番近くのタクシーが
すぐ来る、すぐ乗れる

たくさんのタクシー会社の中から、一番近くのタクシーがスピーディーにあなたのもとへ配車されます。
アプリから簡単に呼べるので、寝坊してしまった朝や時間がない時の強い味方です。

2

目安到着時間がわかるから、
待ち時間も有効に使える

配車を依頼すると到着時間の目安がわかります。
到着通知も来るので、もう路上でタクシーを待つ必要はありません。
何かあった時は直接乗務員と連絡もとれるので安心してご利用いただけます。

3

ネット決済100%対応で
車内での支払いが不要

ネット決済に100%の車が対応しているため、決済はアプリで自動的におこなわれます。
目的地に着いたらそのまま降りるだけでOK。
お釣りやクレジットカードのやりとりに時間を取られません。

※後部座席に搭載されたタブレットに表示される二次元バーコードを読み込む決済方法も今後導入予定です。

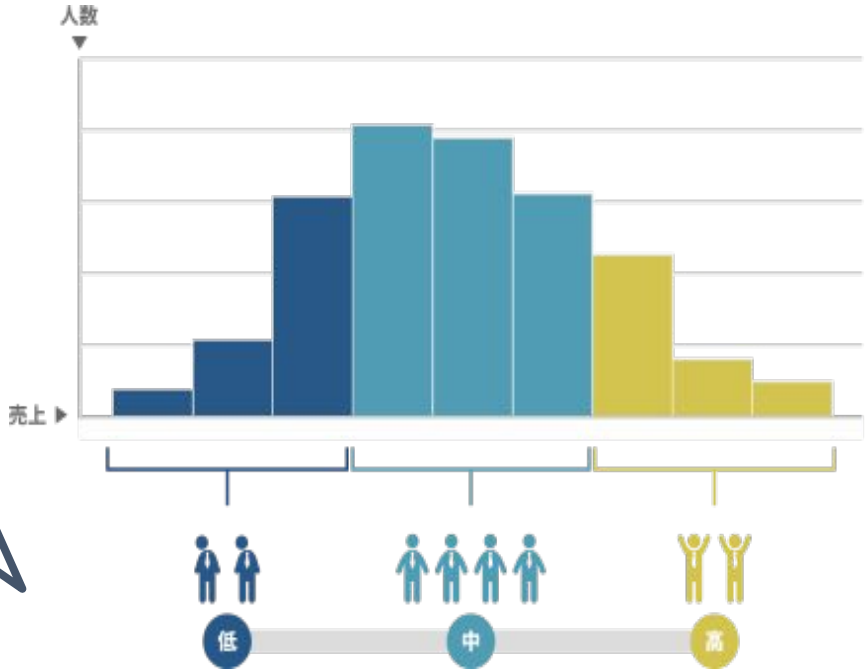
利用者ごとの主な機能

タクシー利用者	✔ 簡単にタクシーを呼ぶことができ、到着予測時間の確認や、支払いも簡単
タクシードライバー	✔ アプリを通じての配車で多くのお客様に使っていただける ✔ 経験の少ないドライバーに経路を推薦して効率よく働いてもらえる ⇒ お客様探索ナビ

プロダクトの課題設定

乗務員は歩合制

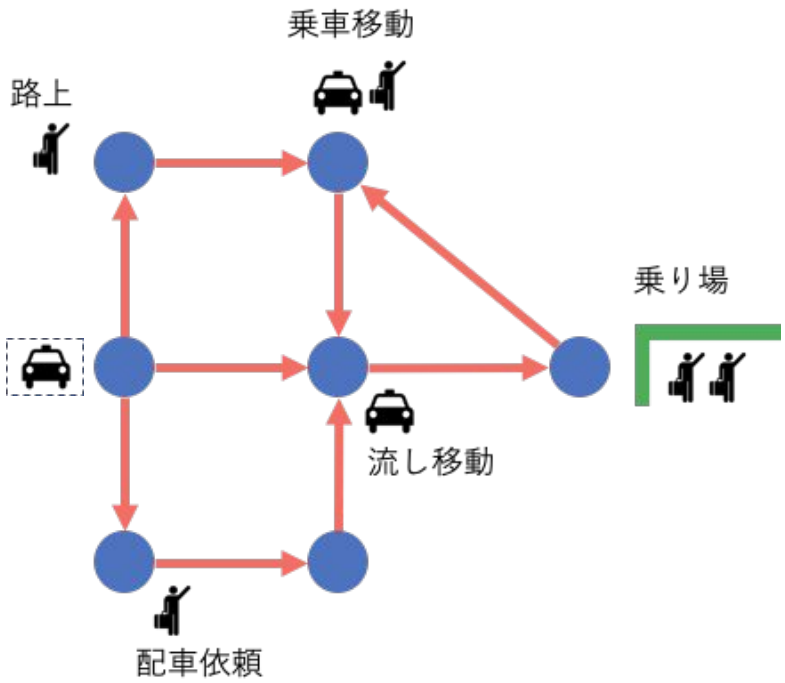
収入は
<u>探客スキル</u>に依存



生産性に
約2倍差
があります。

お客様探索ナビとは

需要を予測して最適な
営業経路を提案



お客様探索ナビとは

ベースとなる技術

この車

最適な

需要供給予測

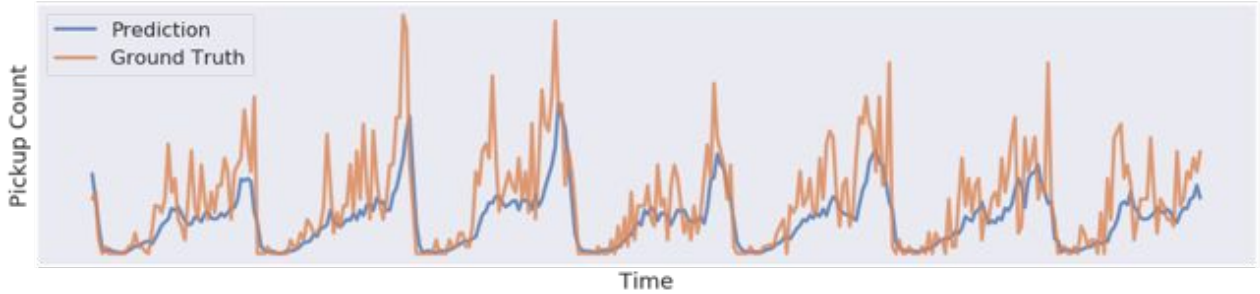
- 特徴量作成
- MLモデル推論

走行ルート推薦

- 最適方策の獲得
- 全体最適化

乗り場

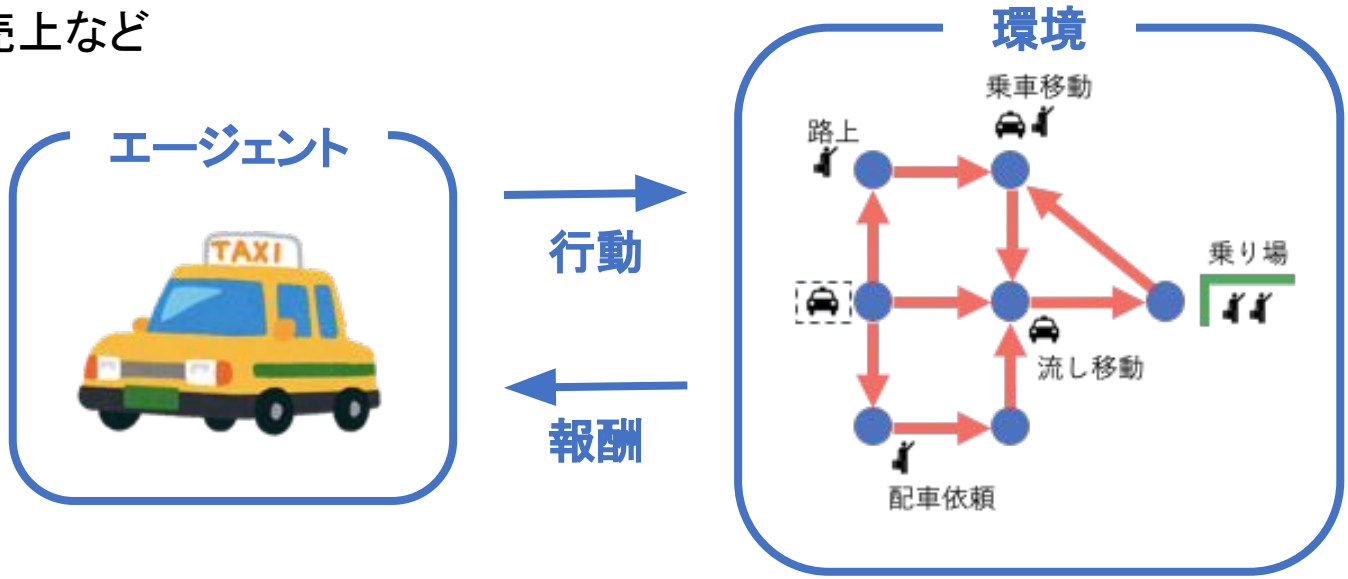
機械学習による需要供給予測



強化学習による走行ルート推薦

行動：進行方向

報酬：売上など



GOの
MLシステム紹介

01

MLシステムのリアルタイムデータの重要性

- 時間帯による需要変化、タクシーの供給変化など、刻一刻と状況は変化する
- 例: 突然の商業施設閉館による道路の需要減

高需要道路

5/26
施設閉館

低需要道路

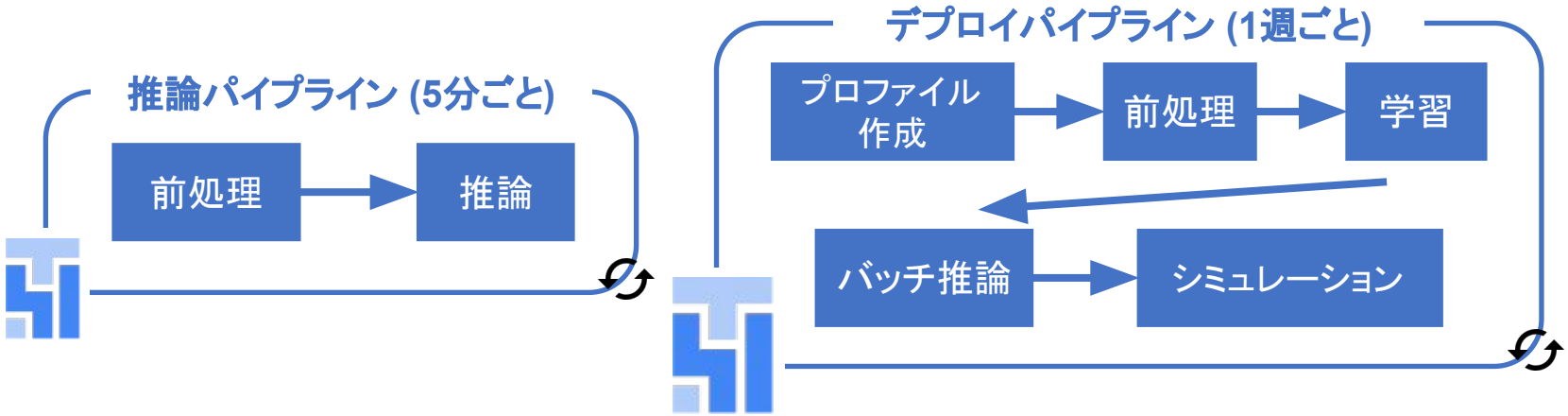
Map data ©2020 Google

MLシステムのワークフロー

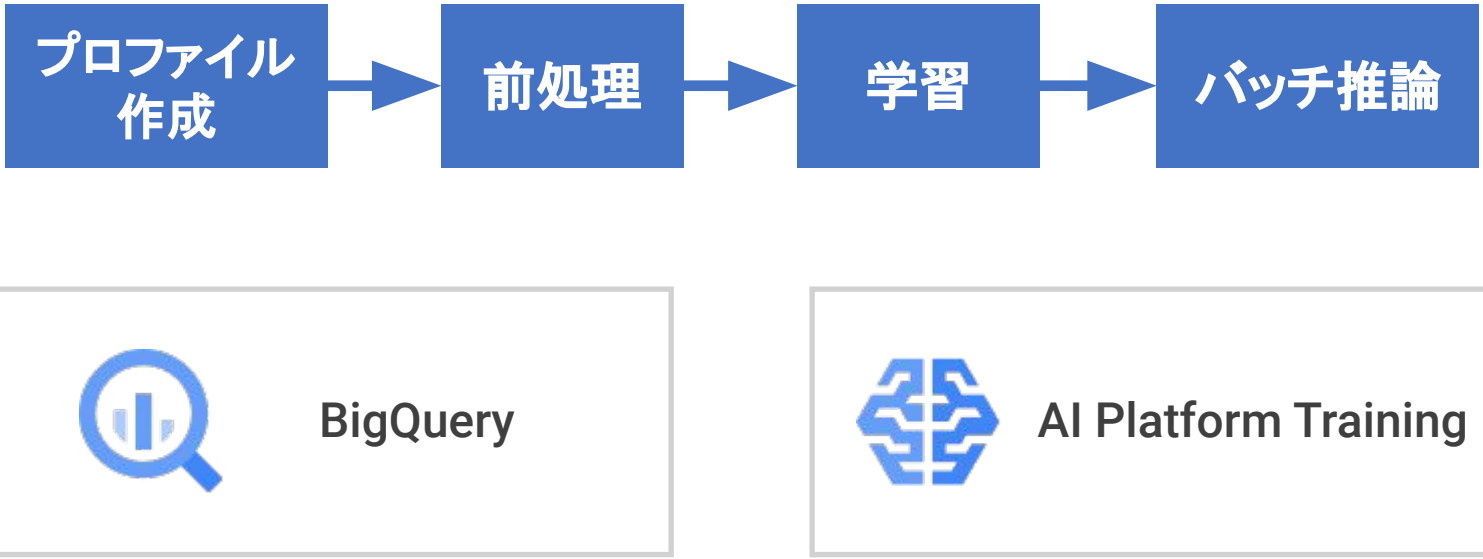
- 新鮮なデータで推論・学習（モデル更新）を行う必要がある

  現在のモデルでリアルタイムデータに対し推論（短期の変化）

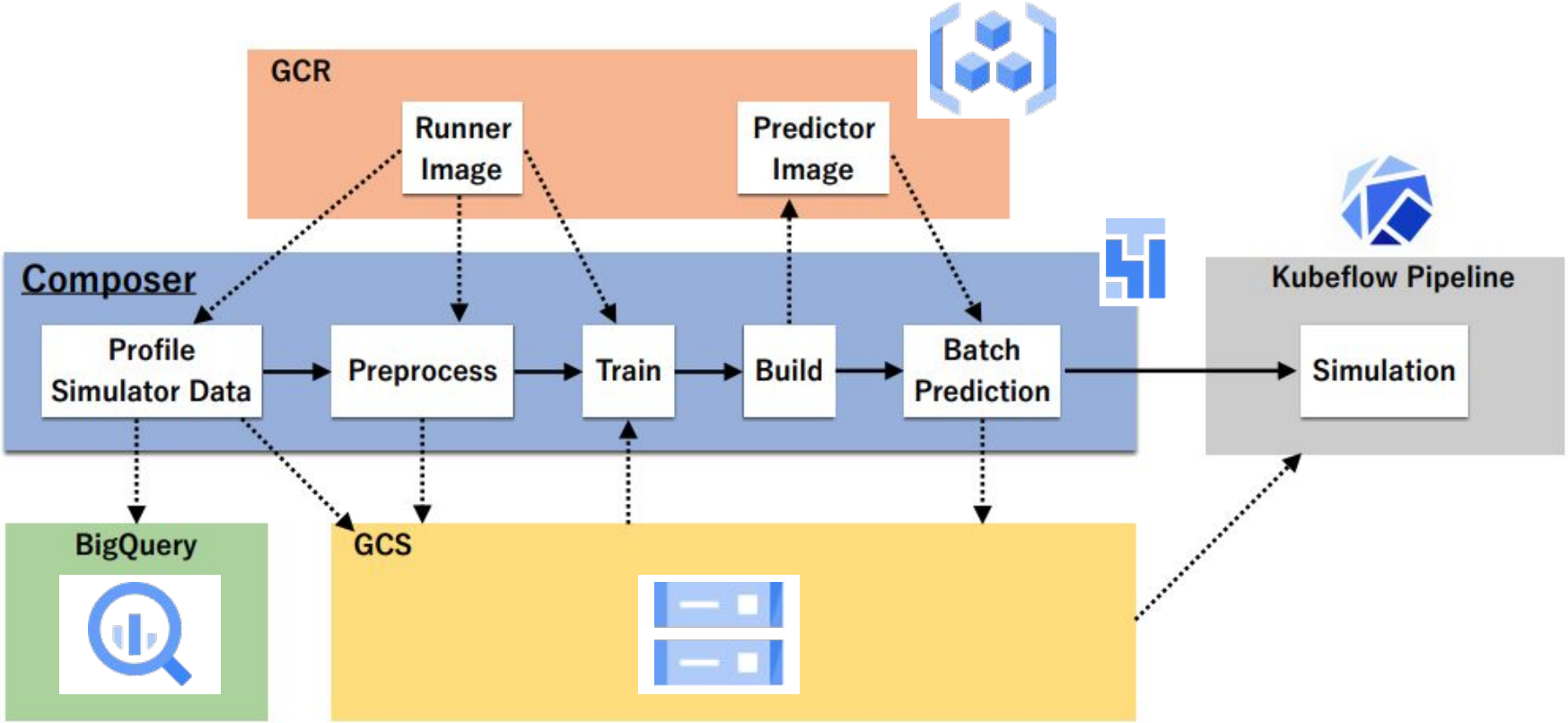
  直近のデータを用いて学習・モデルを更新（中長期の変化）



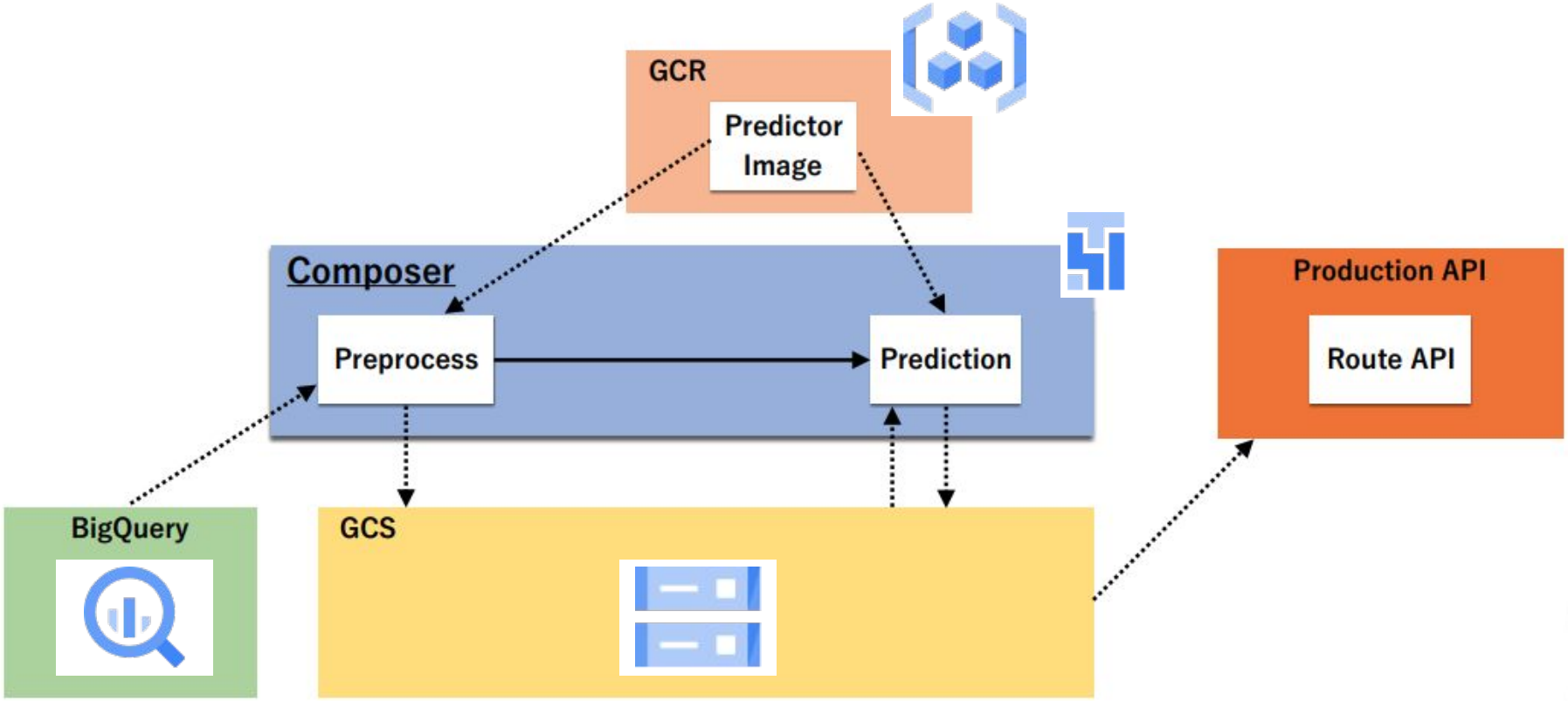
デプロイ用パイプライン（1週ごと）



デプロイ用パイプライン（1週ごと）



推論パイプライン(5分ごと)



MLシステム開発フロー

実験環境

- 分析/モデル開発
- 実験管理

CI/CD

本番環境

- 推論パイプライン
- モデル精度監視

MLシステム開発フロー

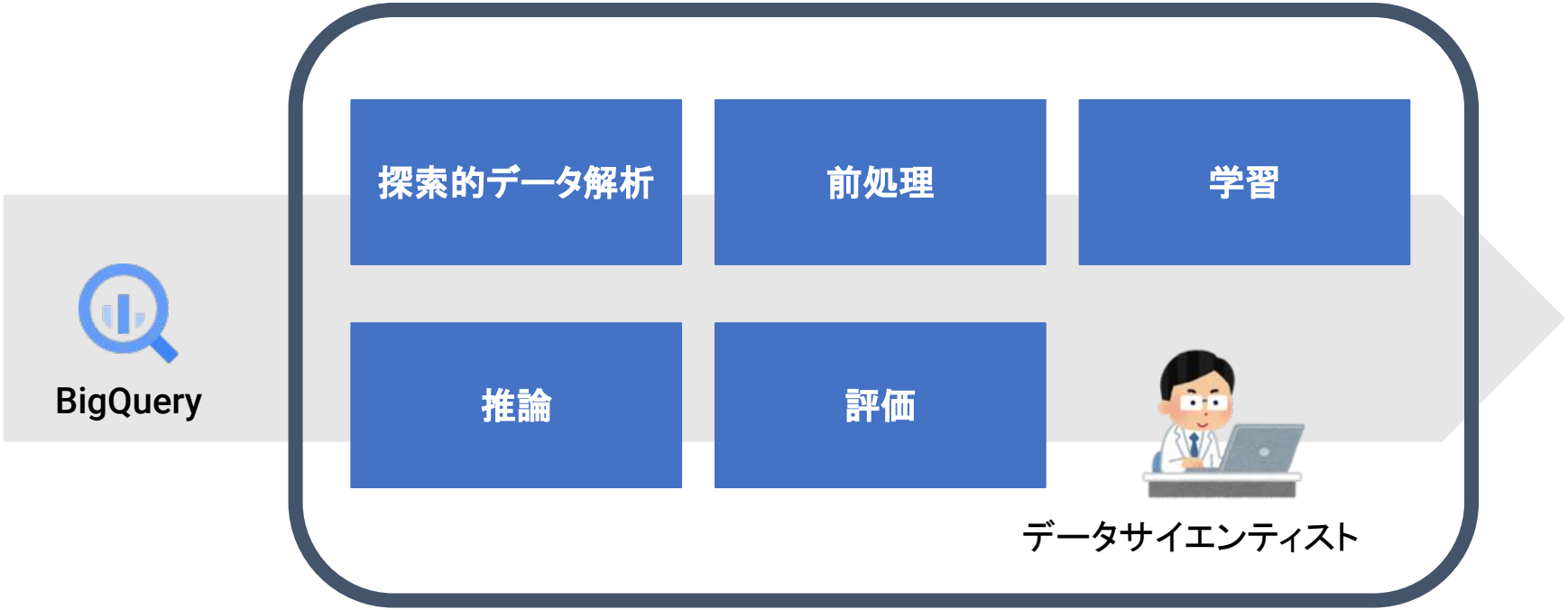
- 実験の再現性担保
- 高速な開発イテレーションの確保
- テストの難しさ
- モデルの精度監視

などなど多くのML特有の課題

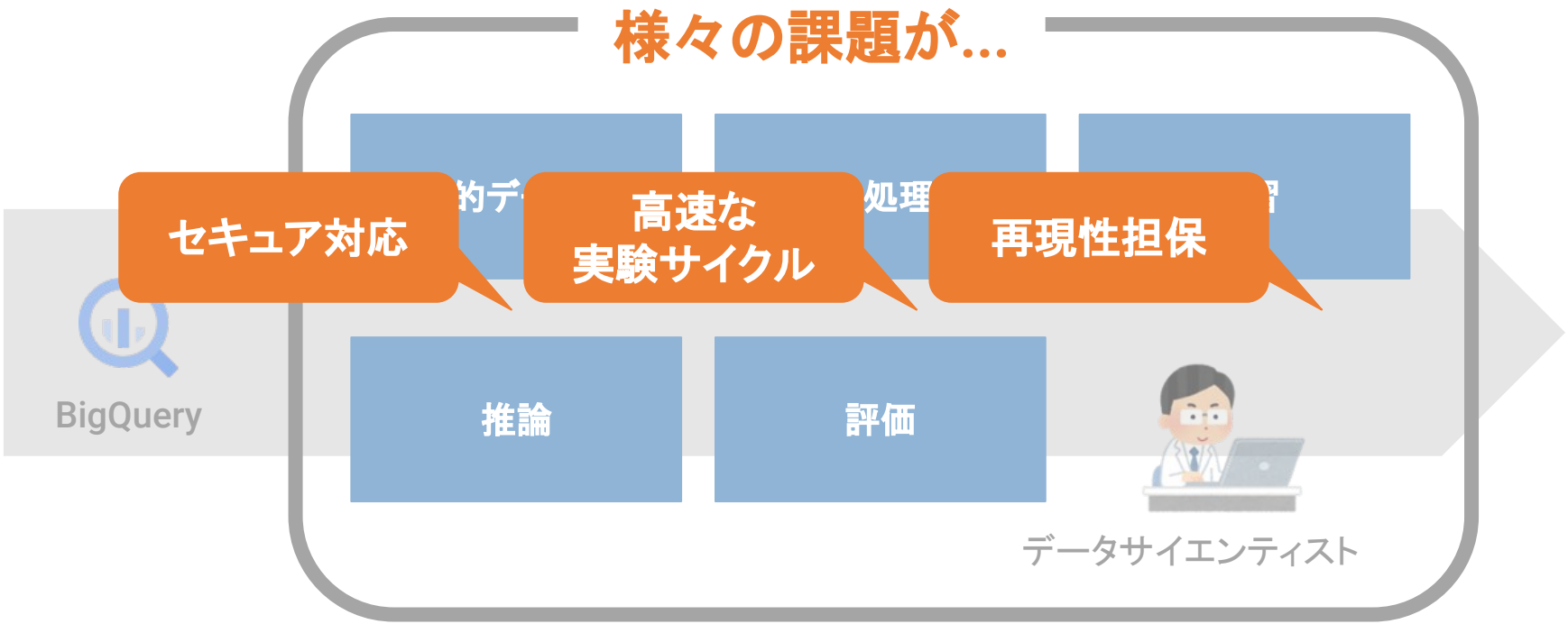
実験環境の
MLOps

02

MLシステムにおける実験環境の役割



MLシステムにおける実験環境の役割



セキュアなクラウド環境への課題

セキュア環境

- BigQueryのセキュアなデータへのアクセスのため、権限・証跡をきちんと管理したい

リッチなインスタンス

- もちろんローカルPC以上に快適に、できればスケールもさせたい

共有しやすさ

- htmlファイルを送り合うのではなく、Web上で閲覧権限者に共有したい

セキュアなクラウド環境への解決策

JupyterHub on GKE

- JupyterHubはユーザごとにJupyter環境を管理するツール
- HelmにJupyterHubのパッケージがありカスタム利用可能
  - OAuth認証
  - 共有ディレクトリ
  - python package + jupyterlab extension

JupyterHub on GKE

セキュア環境

- SSL・IP制限（Cloud Armor）・個人認証（Cloud IAP）
- group account連携でメンバーの追加/削除の棚卸しが楽
- Datalabと比較してセキュア要件向き
- package + extensionのバグも一括対応可

リッチなインスタンスと共有しやすさ

- GKEのためスケールが容易
- 大人数での作業時やモデル開発時にはスケールアウト
- 共有ディレクトリでのNotebook共有の手軽さ

高速な開発イテレーションサイクルの課題

細かいパラメータ調整

- 本番環境ではAirflowやKubeflowなどのワークフローエンジンで決まり
- 実験環境のような細かいパラメータ調整や素早いイテレーションサイクルが必要な場合だとワークフローシステムは仰々しい

依存関係の担保

- ヒューマンエラーないように、依存関係を意識することなくジョブを実行できてほしい
- ワークフローと同様に依存関係がないジョブは並列実行してほしい

高速な開発イテレーションサイクルの解決策

実験用に最適化したカスタムランナー

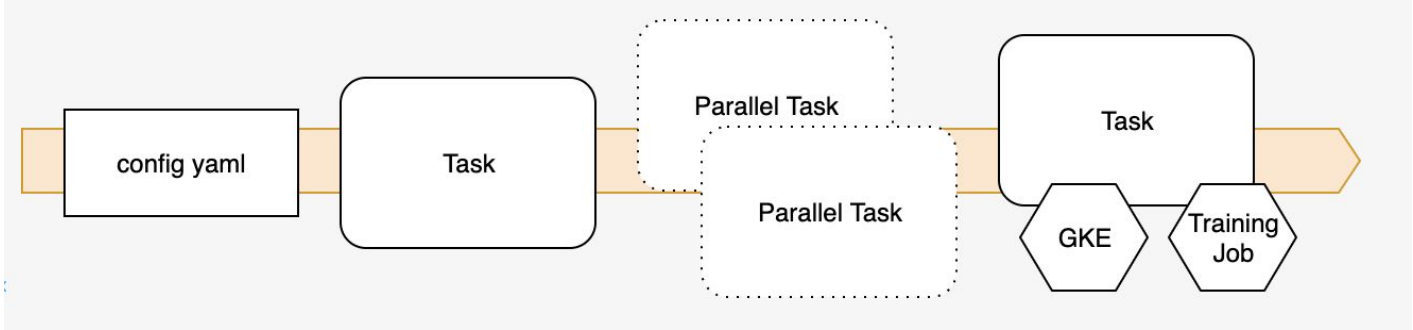
- タスクランナーInvokeを利用
- ローカルでシェルスクリプトを実行するPythonパッケージ

Fabric (Python)

実験用に最適化したカスタムランナー

細かいパラメータ管理と依存担保

- 実験環境の依存ジョブを個別に定義
- 細かいパラメータや定数はyamlファイルで定義
- 並列処理はconcurrent.featuresモジュールでカスタマイズ
- 重い処理はAI Platform Training JobやGKEに外出しして軽量化



AI Platform Training Job 同期処理

ワークフローなどで同期処理したい際にはState確認が有効

```
try:
    job = ml_jobs.create(body=job_spec, parent=project_id).execute()
    while True:
        result = ml_jobs.get(name=job_name).execute()
        if result["state"] == "SUCCEEDED":
            return result
        elif result["state"] in ["FAILED", "CANCELLED"]:
            raise RuntimeError("job failed: {}".format(result["errorMessage"]))
        sleep(10)
except (TimeoutError, KeyboardInterrupt) as e:
    ml_jobs.cancel(name=job_name).execute()
    raise e
```

実験の再現性の課題

前提

- モデルの実験データにはBigQueryを利用している
- 手動更新から、ワークフローからwrite truncateする自動運用へ移行
- 単純にappend tableするとjob再実行時にデータ重複が発生してしまい、冪等性を担保できない

実験データの冪等性担保

- 元データが都度更新されるため、実験を再実行すると結果を再現できず、モデルを正確に評価できなかった
- write truncateでtable作り直しによるコスト増大も発生

実験の再現性の解決策

Merge Query Statement

- 既存レコードと新規レコードの追加・更新・削除を組み合わせで実行できる
- つまり、既存のレコードへ重複・上書きの影響なく新規レコード生成することも可能
- 作り直しにはレコード削除やtable削除が必要になる
- 分析クエリ費用とQuotasには注意が必要

BigQuery

Merge Query Statement

既存レコード

新規レコード

不一致条件

```
MERGE dataset.NewArrivals AS target
USING (
  SELECT * FROM UNNEST(
    [('microwave', 10, 'warehouse #1'),
    ('dryer', 30, 'warehouse #1'),
    ('oven', 20, 'warehouse #2')])
) AS source
ON target.product = source.product
WHEN NOT MATCHED THEN
  INSERT ROW
```

実験環境MLOpsのまとめ(1)

1. <u>セキュアなクラウド環境</u>
   - GKE構成でセキュアでスケールしやすく
2. <u>実験時の高速なイテレーションサイクル</u>
   - タスクランナーで手軽にカスタムできるJobへ
   - 依存関係を担保しつつ、ワークフローと良い所取り

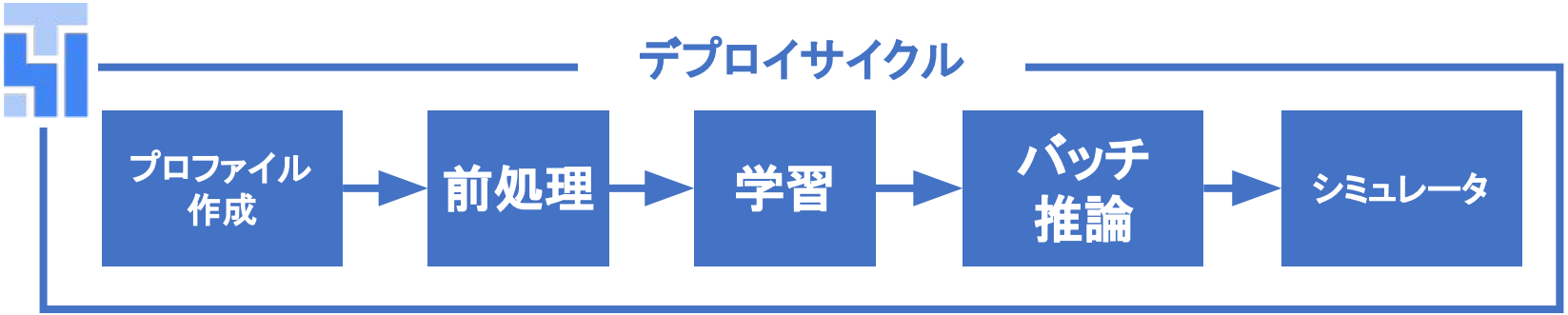
実験環境MLOpsのまとめ(2)

3. <u>**実験の再現性**</u>
   - 実験の信頼性を上げるために、Merge Queryでデータの冪等性担保
4. <u>**学習データの拡大**</u>
   - AI Platform Training Jobで学習環境を手軽にスケールアップ可能
   - 不要なGKEランニングコスト削減に有効
   - Training Jobに拘らずに、タスクによってはモデル分離も検討

CI/CDのMLOps

03

MLシステムにおけるテストの難しさ



- 開始から終了まで8時間ほど要する
- 学習など、ステップ単位でも1時間以上要するものも
- BigQuery や GCS を含む複雑な依存関係
- 動作の保証だけでなく、精度も考慮する必要

MLシステムにおけるテストの難しさ

開発サイクルにおけるテスト

- ローカルもしくは CI で実行するテスト
- 30分ほど要する
- 最小限のダミーデータを用いる
- 精度やデータノイズは一切見ない
- パスすれば開発環境まで自動デプロイ

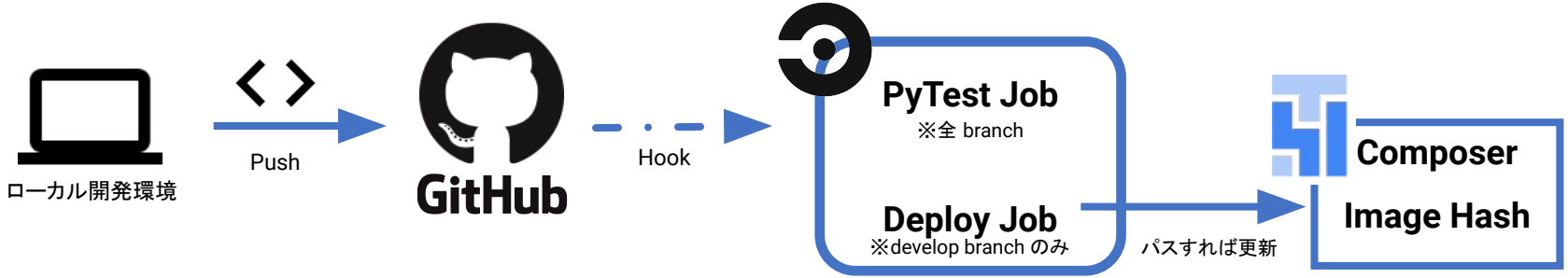
開発環境パイプラインの実行

- 最新 develop image で定期実行
- 8時間ほど要する
- プロダクションと同条件のデータ
- 精度の面は監視・通知によってカバー
- 問題なければ本番環境にデプロイ

開発サイクルにおけるテスト

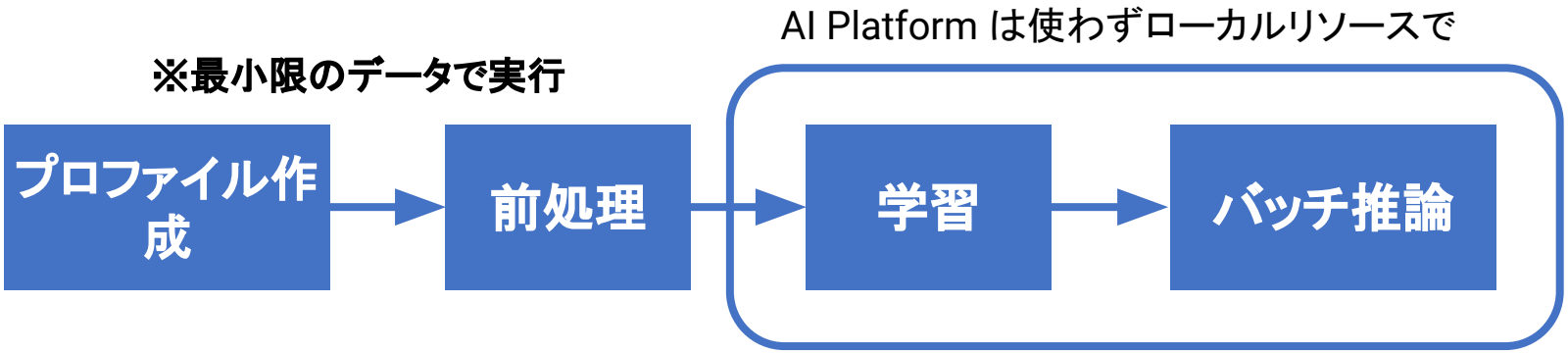
Github + CircleCI + PyTest によるテスト運用

- 運用ルール: PR を出す時は最低限テストをパスさせる
- develop ブランチにマージされたら、CI で最新イメージをビルド
  - 開発環境 Airflow に自動デプロイ
  - 精度などの問題はパイプラインの定期実行で検知



PyTestの設計

- 基本的にパイプラインの流れを最低限のデータで再現
- CircleCI 上での実行時間は30分弱



PyTestの設計

- Setup/Teardown で一時データセットやバケットを用意

  テスト実行ごとにハッシュ値を用意し、BQ や GCS 上で被りがないように

```
@pytest.fixture(scope="session", autouse=True)
def setup_teardown(request, test_config):
    # setup
    suffix = uuid.uuid4()
    create_dataset(suffix)

    # teardown
    delete_blobs(suffix)
    delete_dataset(suffix)
```

PyTestの設計

- 依存関係の定義は PyTest Dependencies を用いる

  DAG では書けないが、必要十分

```
@pytest.mark.dependency(depends=["test_preprocess_a", "test_preprocess_b"])
def test_trainer(test_config):
    trainer = Trainer(
        test_config["xxx"].format(test_config["hash"]),
        test_config["yyy"].format(test_config["hash"]),
    )
    trainer.train()
    trainer.upload_model()
```

CI/CD MLOpsのまとめ

1. **<u>開発サイクルにおけるテスト</u>**
   - 動作確認のためのテスト
   - Github + CircleCI + PyTestで最小限データと実行時間で動作確認
2. **<u>開発パイプラインにおけるテスト</u>**
   - 精度確認を含めた完全なテスト
   - 最新 develop branch に自動追従させ、定期実行
   - 精度面は監視運用で検証

本番環境の
MLOps

04

MLシステムにおける監視

監視項目

推論パイプラインがエラーで停止

↓

最新データで経路を引けない

学習データのノイズや時期性

↓

モデルの精度悪化

テストデータのノイズ等

↓

正しい精度評価ができない

精度が悪いモデルを本番デプロイ

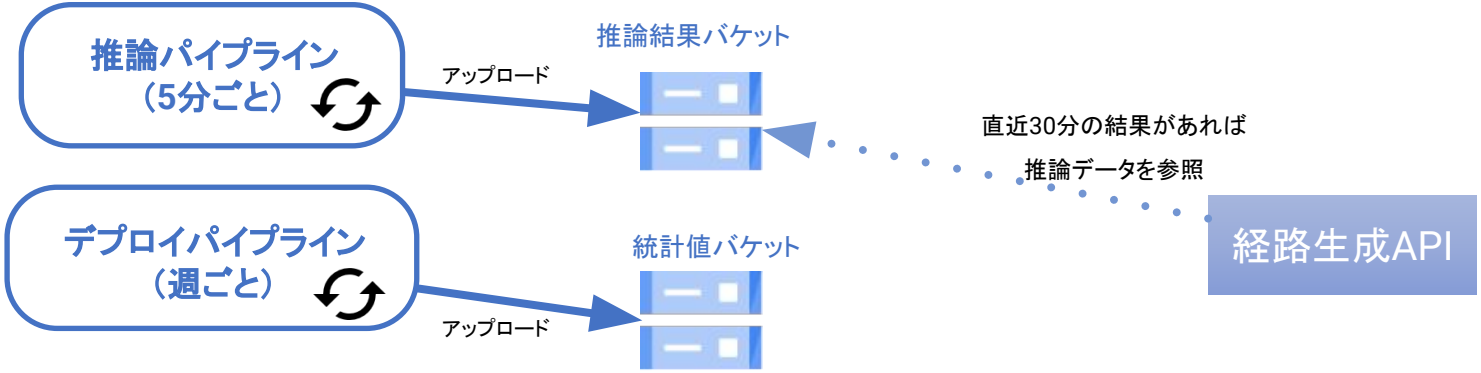
↓

経路、UXに悪影響

システムエラー監視

前提

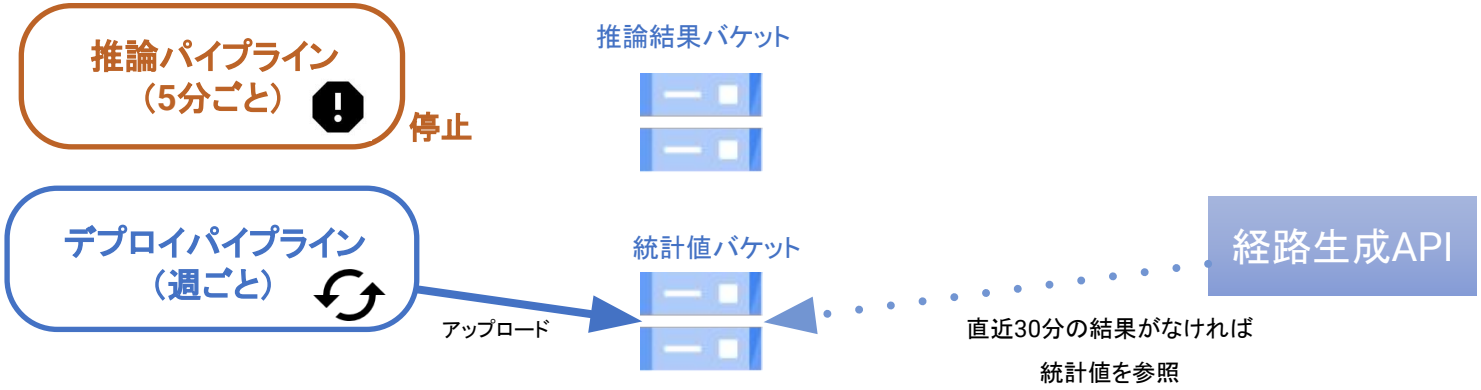
- 推論結果が一定時間更新されない際、統計値プロファイルを基にした経路が引かれる
- つまり推論パイプラインが止まっても、ナビは機能する（重要）
- しかし推論を基にした経路のほうがシミュレーション営収が良い 場合がほとんど



システムエラー監視

前提

- 推論結果が一定時間更新されない際、統計値プロファイルを基にした経路が引かれる
- つまり推論パイプラインが止まっても、ナビは機能する（重要）
- しかし推論を基にした経路のほうがシミュレーション営収が良い 場合がほとんど



システムエラー監視

推論パイプライン停止の通知

- Airflow の Slack Webhook Operator を使う
- DAG の on_failure_callback でタスク失敗時に実行する Operator を指定可能

```
def task_fail_slack_alert(context):
    slack_msg = "Task Failed."
    failed_alert = SlackWebhookOperator(
        task_id="task_id",
        http_conn_id="slack",
        webhook_token=Variable.get("webhook.token"),
        message=slack_msg,
        username="airflow",
    )
    return failed_alert.execute(context=context)
```

デプロイフローでの精度エラー監視

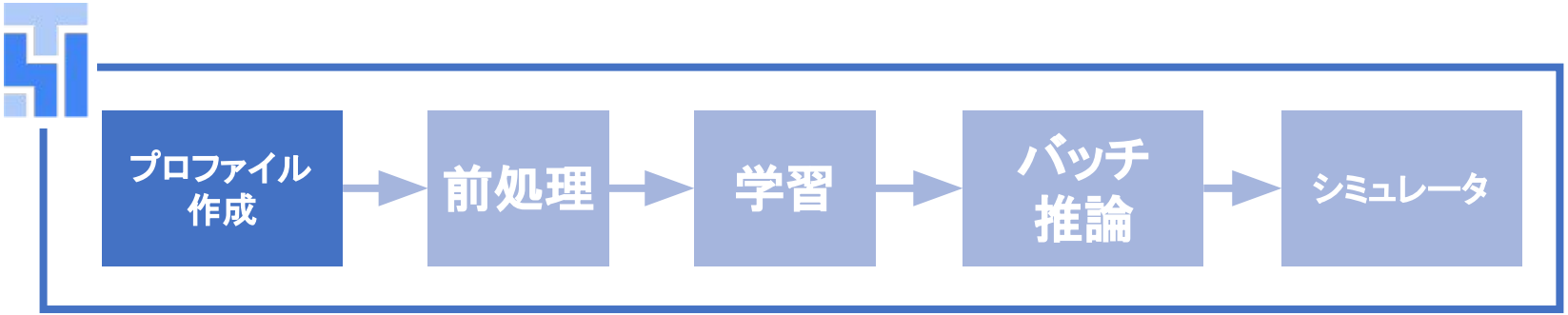
RMSE (Root Mean Squared Error)

- 道路ごとに、ある時間枠で何台のタクシーが通るか → 供給
- 道路ごとに、ある時間枠で何回の乗車が発生するか → 需要
- これらを2つのモデルで予測しており、その誤差を見る

シミュレーション結果

- バッチ推論の結果を用いた経路に従うと、どれだけの営収となるか

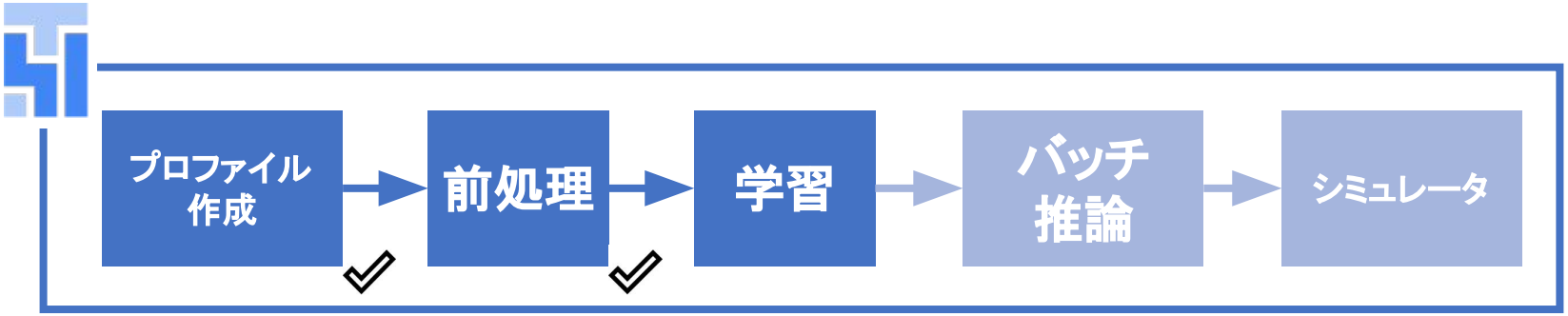
デプロイフローでの精度エラー監視



プロファイルの Validation

- ✓ BQの一時テーブルを利用して平均値がおかしな値でないかを確認
- ≡ 閾値はエイヤで決めており少し緩めだが、多段なので許容

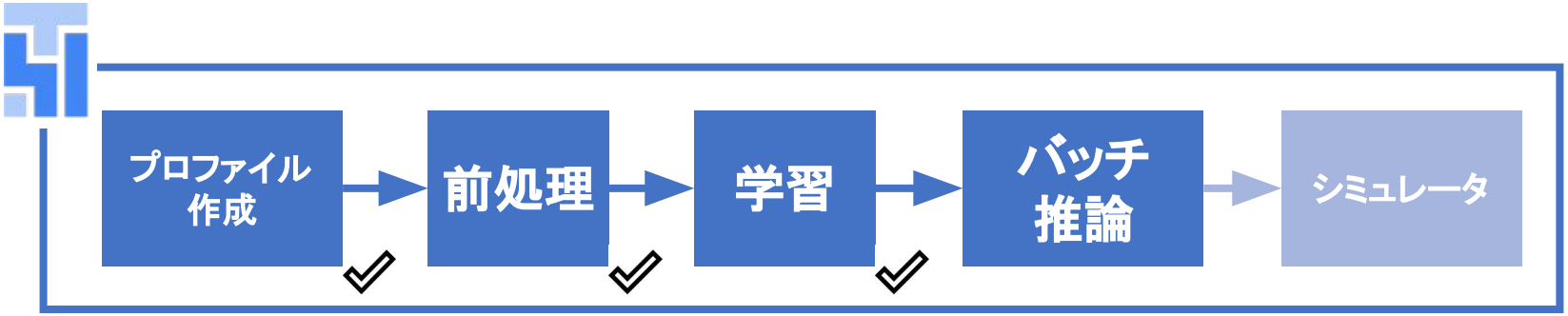
デプロイフローでの精度エラー監視



学習時の Validation

✓ 検証データに対する RMSE がプロファイルよりも悪ければ弾く

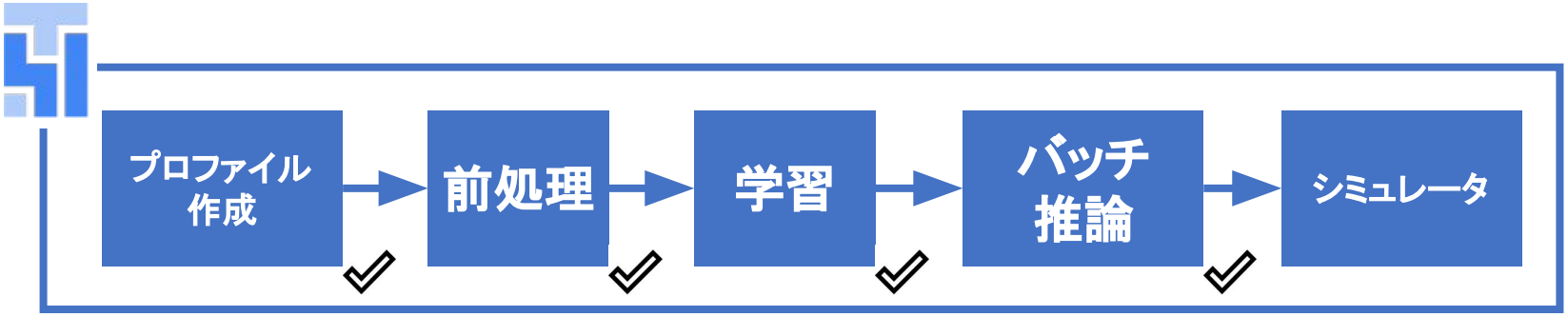
デプロイフローでの精度エラー監視



バッチ推論時の Validation

- ✓ テストデータに対する RMSE がプロファイルよりも悪ければ弾く
- ≡ Monitoring で監視し、ここで弾かれたらアラートを飛ばす

デプロイフローでの精度エラー監視



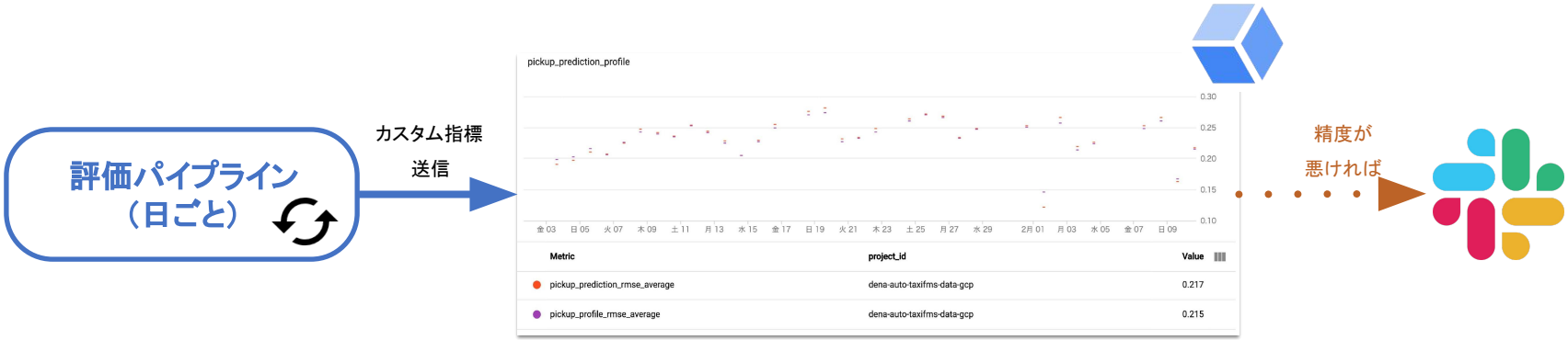
シミュレータの Validation

✓シミュレーション結果が現行モデルを超えていなければ弾く

デプロイ済みモデルの精度劣化監視

評価パイプライン（1日1回）

- 直近1日の推論結果を Ground Truth（実際のログ）を用いて評価
- 推論 RMSE とプロファイル RMSE をカスタム指標として Monitoring に送信
- 推論結果がプロファイルよりも悪かった場合にアラート発火



Monitoring

本番環境MLOpsのまとめ

1. **<u>統計値でシステム担保</u>**
   - MLリアルタイム推論の停止前提に統計値でフォールバック
2. **<u>モデルデプロイ時には多段Validation</u>**
   - データ異常値によるモデルと統計値のズレを想定する
   - デプロイ時には多段Validationで精度を担保

Thank you