FINAL FANTASY XV POCKET EDITION を支える AWS サーバレス技術

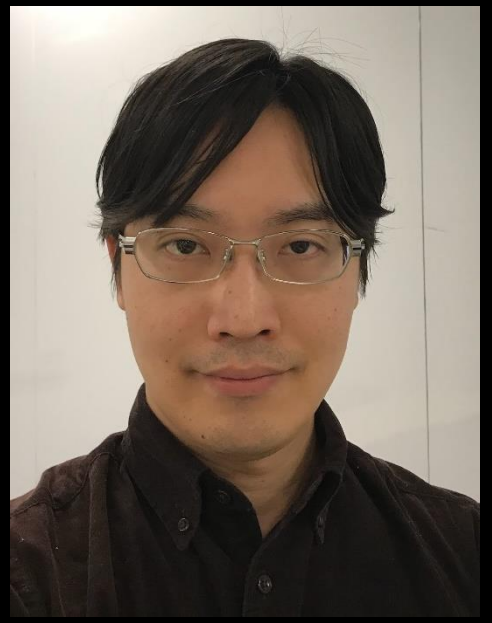
概要

- タイトルとシステムの紹介
  - タイトルの紹介
  - AWS の導入背景
  - システムの全体構成
- 技術詳細
  - トランザクションのない DynamoDB を使ったアプリケーション開発
  - DynamoDB のバックアップ
  - Lambda やサーバーレス技術を使う利点と欠点

（このスライドは公開予定です）

自己紹介

- 名前
  - 堀内 要介（ほりうち ようすけ）
- 所属
  - 株式会社Luminous Productions
    プログラマー



FINAL FANTASY XV POCKET EDITION(FFXV PE) について

- FINAL FANTASY XV のモバイル版
  - FFXV のストーリーをスマホで体験可能
  - iOS, Android
- 全世界150か国に配信
  - 2018年2月9日に全ての国へ配信開始
  - 累計ダウンロード数300万突破

講演では https://www.youtube.com/watch?v=TTxSIPqle3E と同じビデオを上映

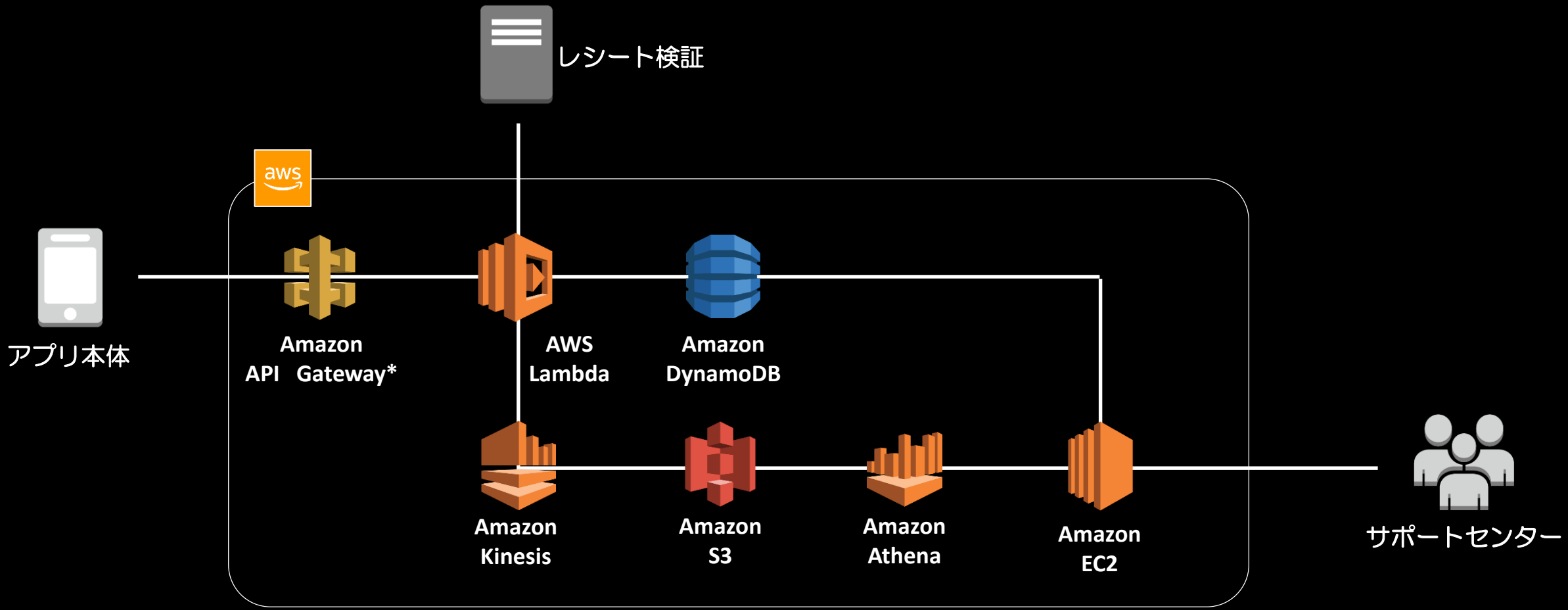
オンラインでの処理が必要になった背景

- アプリ内課金のレシート検証サーバが必要に
  - 有料版、無料体験版とアプリを分けたくなかった
    - お客様の離脱ポイントになることが懸念だった
  - 10章構成として、チャプター1を無料体験版、チャプター2以降を有料で販売

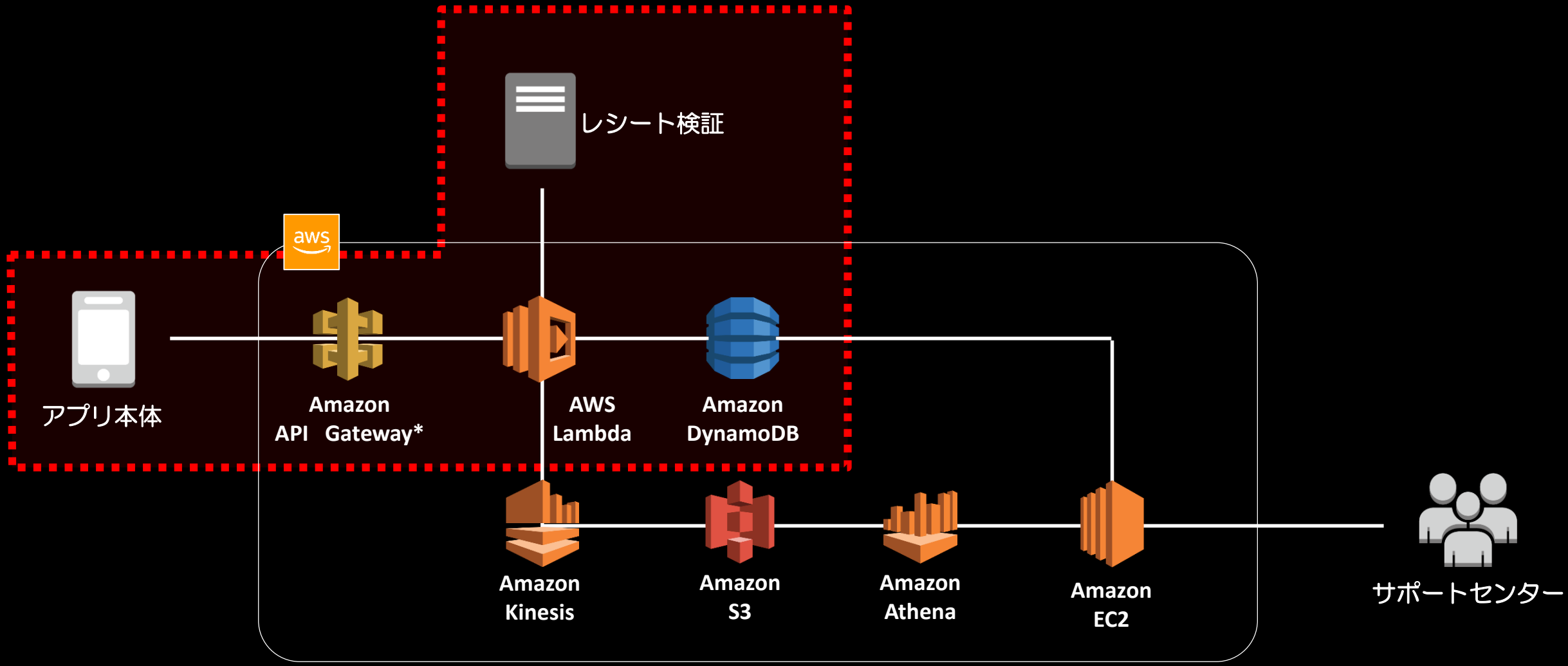
AWS 導入背景

- 150か国への同時配信
  - リリース時の高負荷に確実に耐えるようにしたい
  - リリース時やセール時に柔軟にスケールさせたい
- サーバ開発のコスト削減
  - 少人数、短期間での開発
- 必要なシステムの小ささ
  - 必要なのはレシート検証のみ

全体構成図

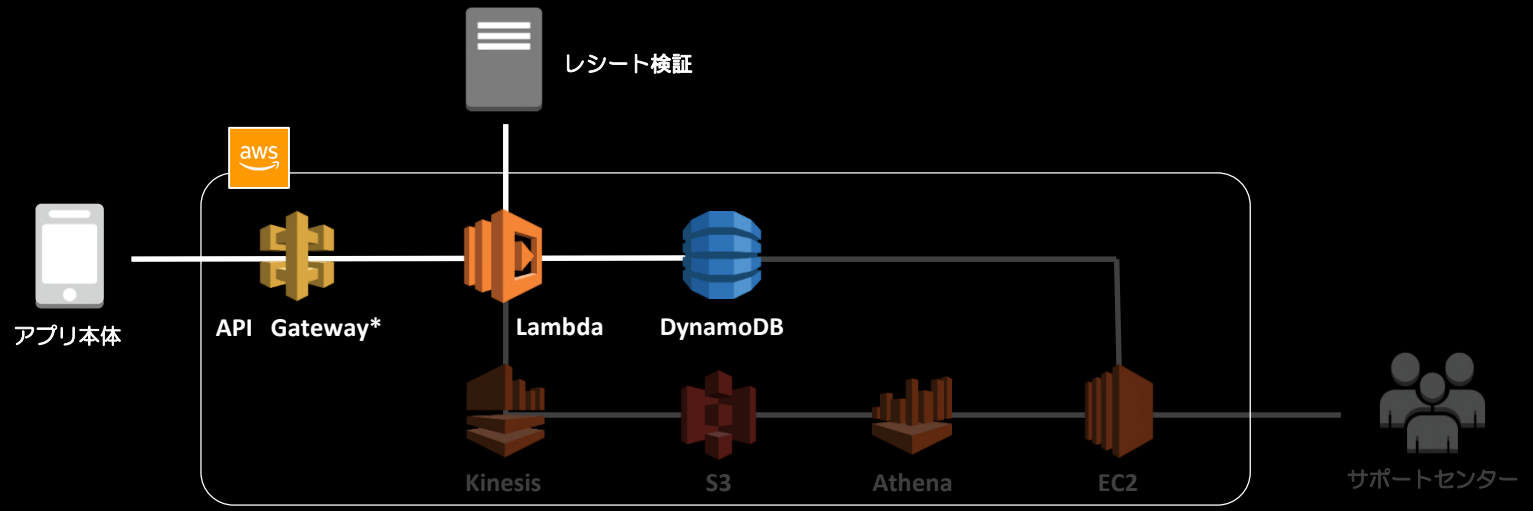


全体構成図



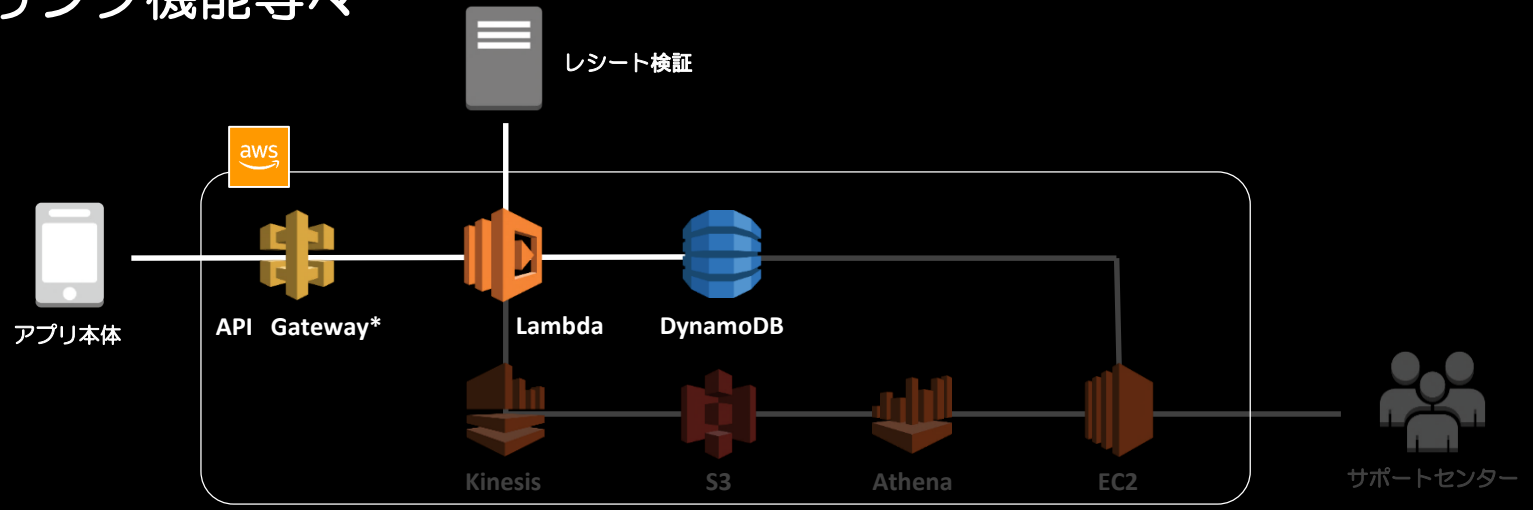
課金処理

- 使用したサービス
  - Amazon API Gateway、AWS Lambda、Amazon DynamoDB を使用
    - 高い可用性とスケーラビリティの実現が容易
  - いずれもフルマネージドサービス。インフラストラクチャの保守は不要



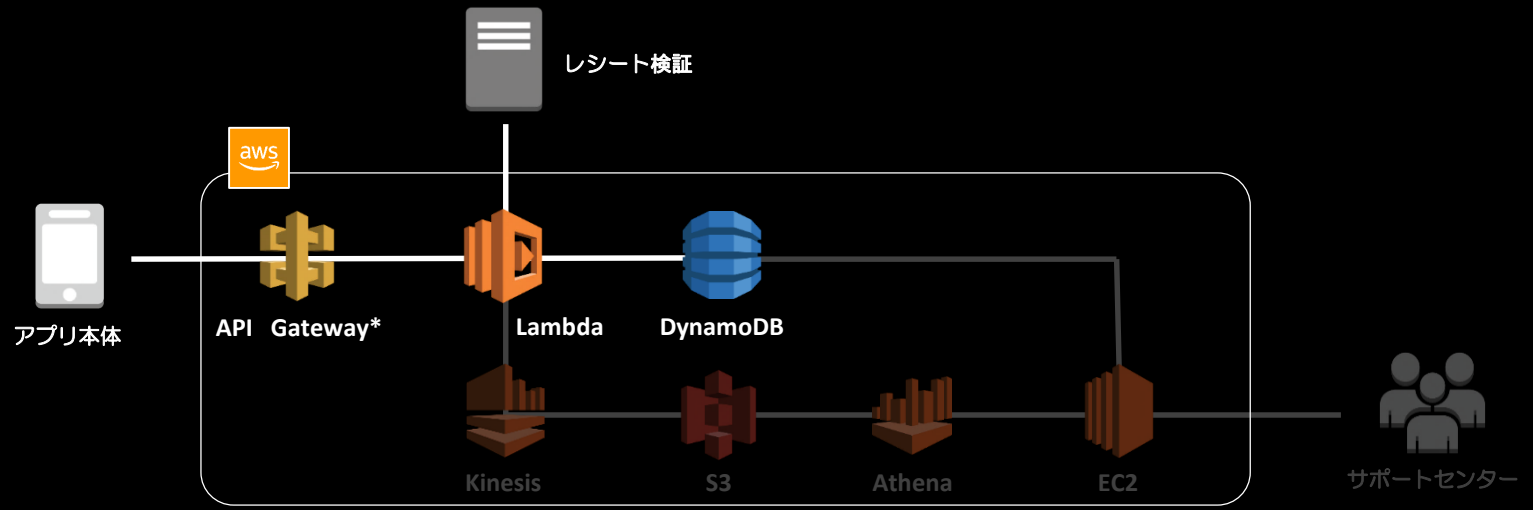
課金処理

- Amazon API Gateway
  - Web API レイヤーの構築が簡単。呼び出し先は柔軟に設定可能
    - AWS Lambda を呼ぶときは、Lambda のエイリアスでの指定がおすすめ
  - デフォルトで10000リクエスト/秒。引き上げの相談も可能
  - アクセスポリシーの設定で制御も簡単
  - API のライフサイクルの管理が可能。リリースステージ（α版、 β版、製品版など）やバージョンの違う API も同時に実行できる
  - モニタリング機能等々…



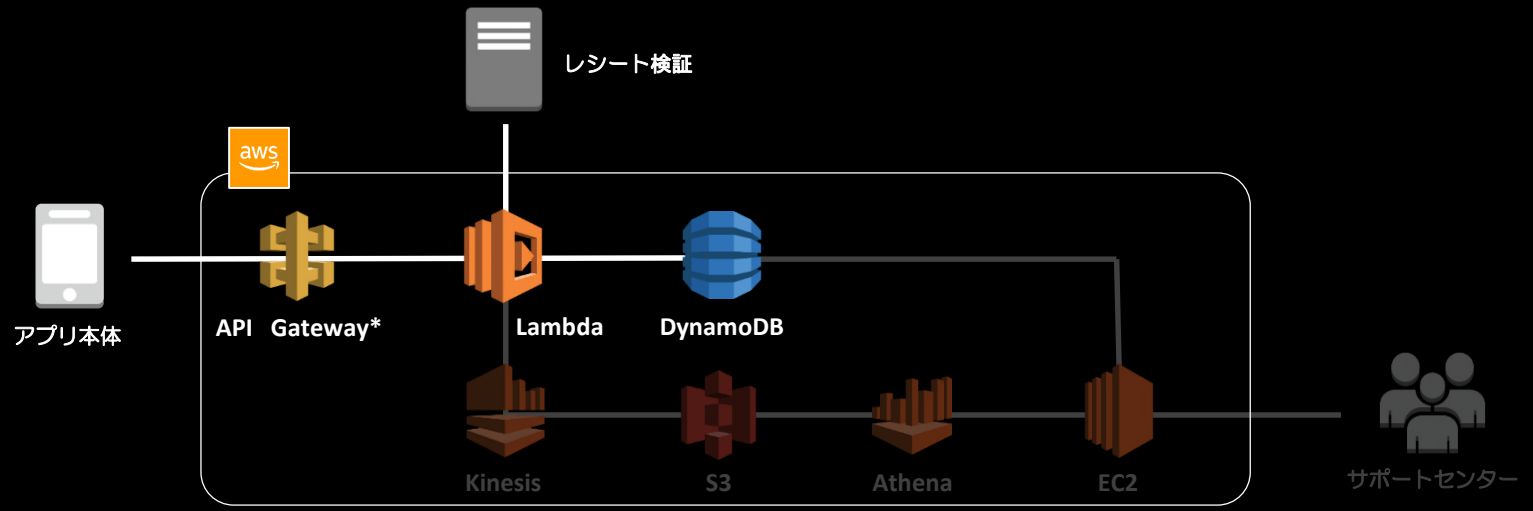
課金処理

- AWS Lambda
  - コードを書くだけで動作させることが出来る（サーバを用意する必要なし）
  - 自動でスケールする
    - デフォルトの同時実行数上限は1000
  - 言語は対応できるものから選ぶ必要がある。ステートレスであるといった制約はある
  - 簡単な操作で Web コンソール上からテストができるが…
    - 次は AWS SAM Local や LocalStack を試したい



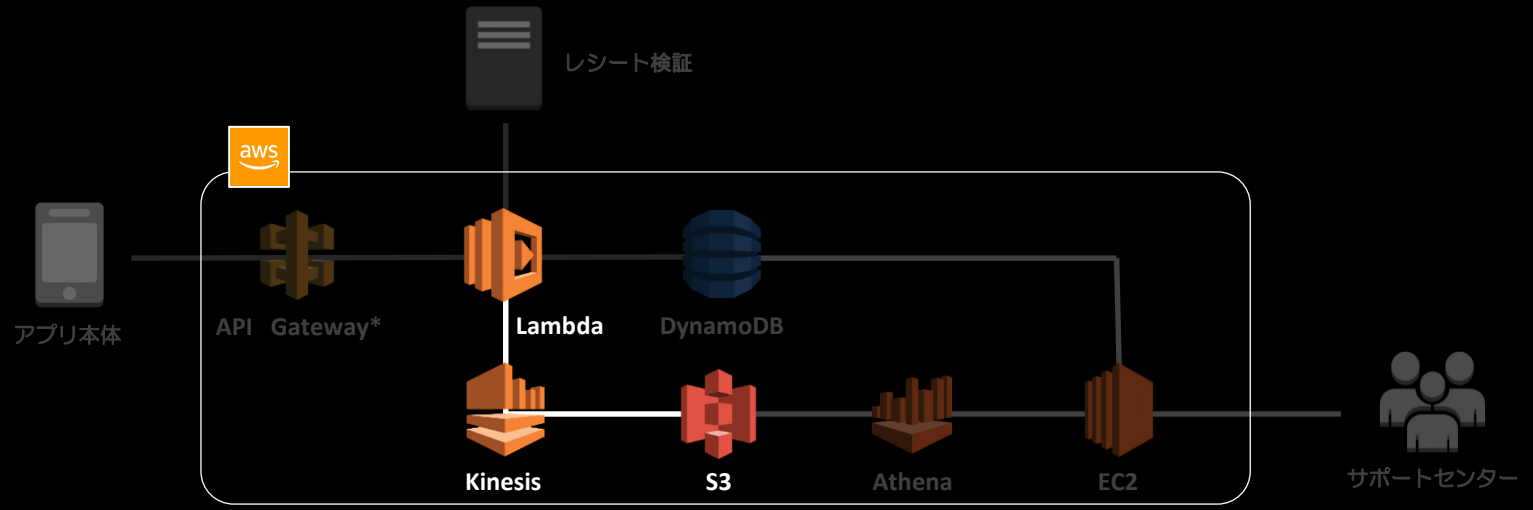
課金処理

- Amazon DynamoDB
  - NoSQL データベースサービス
  - キャパシティーユニットの数でパフォーマンスを制御する
    - デフォルトでは自動でスケールする
  - 今年 Point-In-Time Recovery が追加された
    - バックアップから、過去35日以内であれば任意の時点のテーブルを復元できる



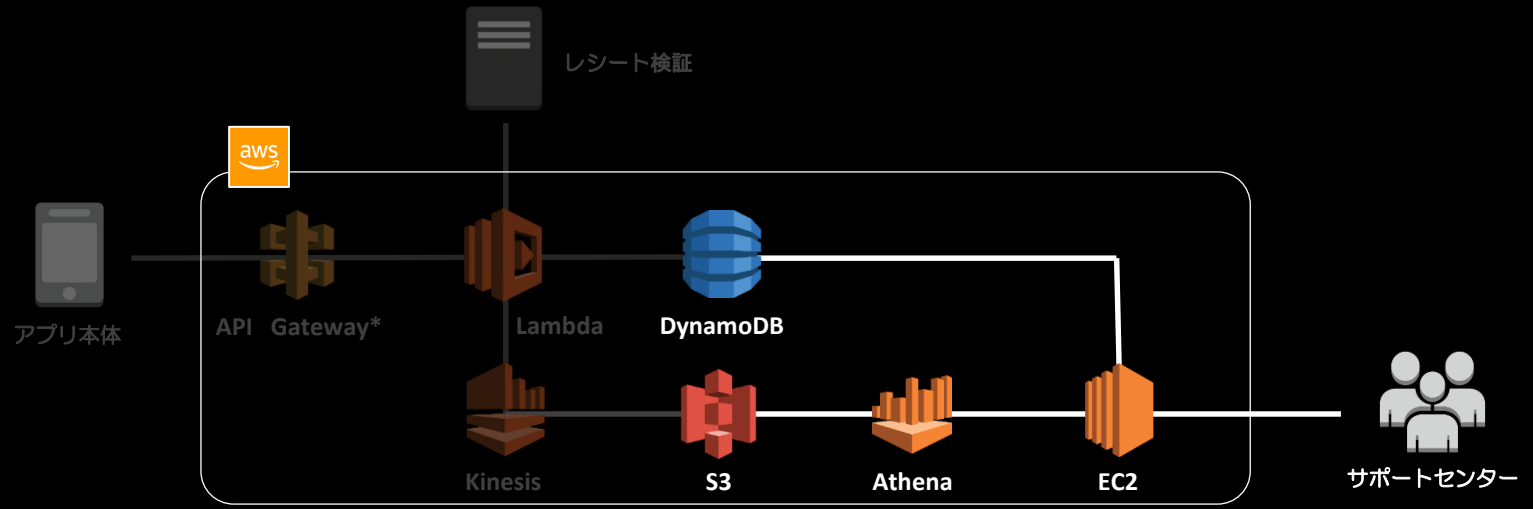
ログの収集

- Lambda から Amazon Kinesis 経由で Amazon S3 にデータを保存
  - データ集計や検索をするための社内ツールに利用
    - Amazon Kinesis Data Firehose は自動的に S3 bucket へ書き出しできるため便利
    - レシート情報は DynamoDB に入れるにはサイズが大きい
  - DynamoDB のデータ復元に利用
    - DynamoDB の定期バックアップと、S3 のデータから復元
    - Point-in-Time Recovery は開発当時にはなかった



データ集計・検索ツール

- データの集計や検索用に社内ツールを Amazon EC2 に設置
  - DynamoDB や S3 のデータを閲覧するための Web ツール
    - お客様からの問い合わせがあった場合の調査などに利用
  - Amazon Athena は S3 に蓄積した JSON データ等を SQL で読み出せる
    - データ抽出、変換、ロード（ETL）処理が不要



FFXV PE を無事にリリース

- AWS の導入により
  - バックエンド側に事故や障害は起きていない
  - 購入に関するお客様からの問い合わせもなし
  - 開発中も大きな障害は無く、少人数、短期間での開発ができた
- AWS 様からの親身なサポート
  - リクエスト数の上限の変更、負荷テストの相談など

自己紹介

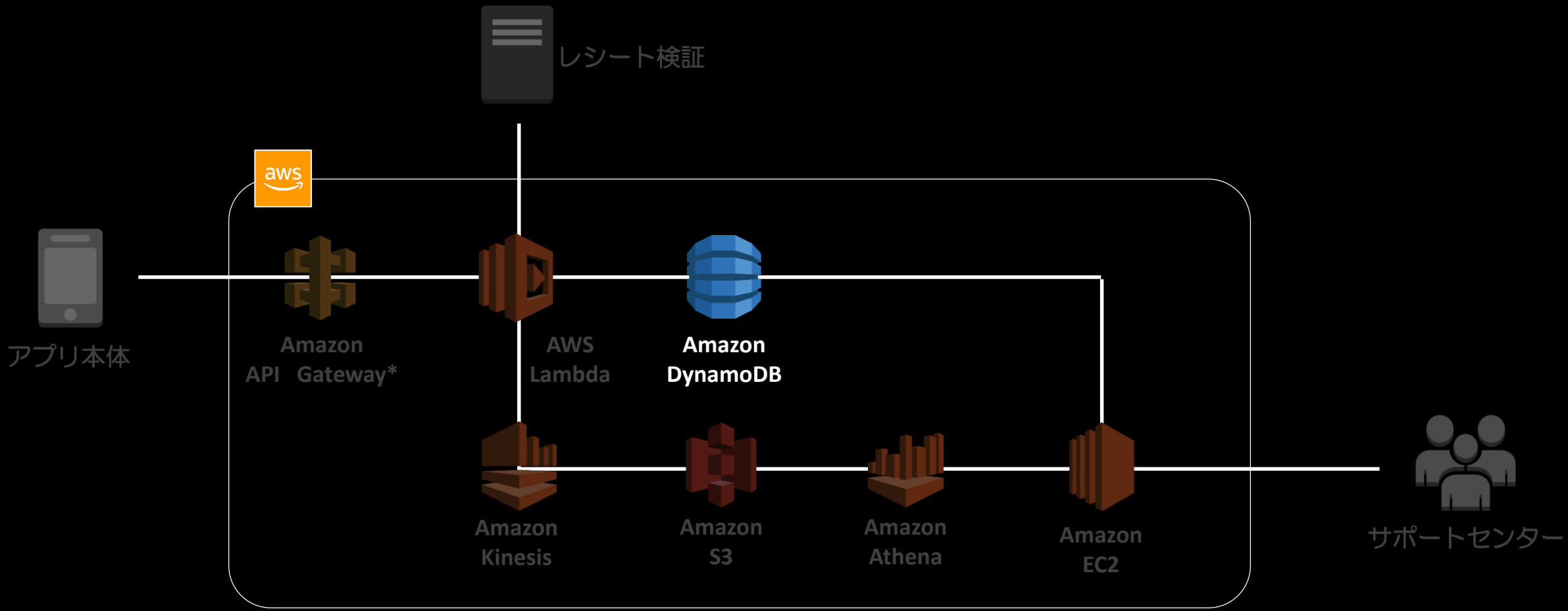
- 名前
  - 重国 和宏（しげくに かずひろ）
- 所属
  - 株式会社スクウェア・エニックス
    テクノロジー推進部
    - FFXV PE のテクニカルディレクター
    - サーバ、ネットワーク技術の R&D

概要（再掲）

- タイトルとシステムの紹介
  - タイトルの紹介
  - AWS の導入背景
  - システムの全体構成
- 技術詳細
  - トランザクションのない DynamoDB を使ったアプリケーション開発
  - DynamoDB のバックアップ
  - Lambda やサーバーレス技術を使う利点と欠点

（このスライドは公開予定です）

Amazon DynamoDB 技術詳細



DynamoDB のパフォーマンスチューニング

- Capacity Auto Scaling にお任せ
  - リリース直後は minimum provisioned capacity を上げておく
- 特定の Partition Key にアクセスを集中させない
  - Partition Key に応じて DynamoDB のノードにデータ割り振られるので、特定の Partition Key にアクセスが集中するとスケールしない
  - アプリからサーバの生存チェックをする Web API が、レコードが 1 個だけの DynamoDB テーブルにアクセスしていたことがあったのを修正
- Query と Scan の違い
  - Primary Key を使う Query とテーブルを全て読み込む Scan の違いに注意

DynamoDB にトランザクションはない

- トランザクションを使うために検討した代替手段
  - Transaction Library for DynamoDB
    - Java で書かれているので、Lambda コードを Node.js で書くという方針と合わない
    - パフォーマンス低下
  - Amazon Aurora
    - Lambda のスケーラビリティを十分に生かせない
    - マネージドだが DynamoDB より運用が大変
- 代替手段は FFXV PE の要件を満たさない

トランザクションなしでアプリケーションを開発

- トランザクションのない DynamoDB をそのまま使うことに決定
  - 複数レコードの一貫性が保たれないのを許容
  - 一貫性が保たれなくても、本当に問題ないか？

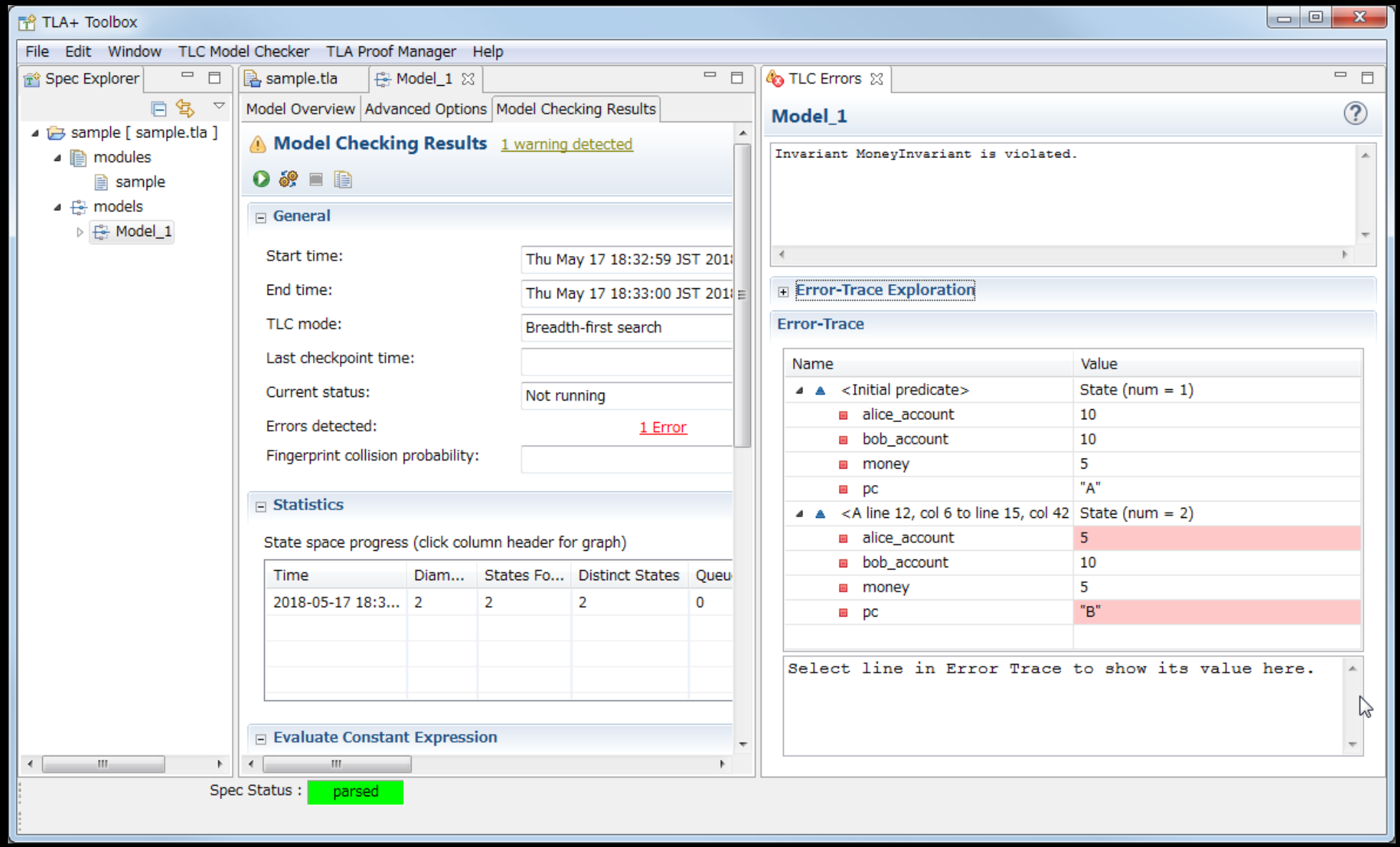
- TLA+ でロジックを書いて想定通りかチェック
  - TLA+ は AWS サービスの内部動作のチェックに使われた
    - How Amazon web services uses formal methods
    - Formal Method の専門家でなくとも使える、と書かれていて興味を持っていた
  - 実装は別なので、チェックが通っても実際のシステムの正常動作は保障されない
    - 全てのロジックを書くのではなく、コアとなる部分を適切な粒度で記述する
  - クライアントは FFXV PE アプリと仮定せずにチェック
    - チート目的のアクセスも考慮

TLA+ のコード例

```
VARIABLES alice_account, bob_account, money, pc

Init == /\ alice_account = 10
        /\ bob_account = 10
        /\ money = 5
        /\ pc = "A"

A == /\ pc = "A"
     /\ alice_account' = alice_account - money
     /\ pc' = "B"
     /\ UNCHANGED << bob_account, money >>

B == /\ pc = "B"
     /\ bob_account' = bob_account + money
     /\ pc' = "Done"
     /\ UNCHANGED << alice_account, money >>

Next == A \/ B

MoneyInvariant == alice_account + bob_account = 20
```

https://learntla.com/introduction/example/ 掲載のサンプルコードを一部抜粋して修正

TLA+ 実行例



複数の手法でアプリケーションをチェック

- TLA+ でチェック
  - 保たれるべき性質（保たれないと実害のある性質）が保たれていることを確認した
    - 何が保たれるべき性質かの判断が重要
- 仕様を複数人でレビュー
  - サーバだけでなく、クライアント（アプリ）の仕様も含めてレビューした
    - TLA+ でクライアントのロジックは書いていない
- サーバ実装をテスト
  - TLA+ で確認済みの性質が満たされているかテストした
    - 実装の正しさは保障できないが、実装の間違い（バグ）の発見はできる

DynamoDB のバックアップ（概要）

- Data Pipeline -> On-Demand Backup -> Point-in-Time Recovery
  - 開発・運用している間にバックアップがどんどん簡単に
- 「差分ログ」を自作
  - 今から作るなら、Point-in-Time Recovery を使う前提で、必要性を再検討すべき

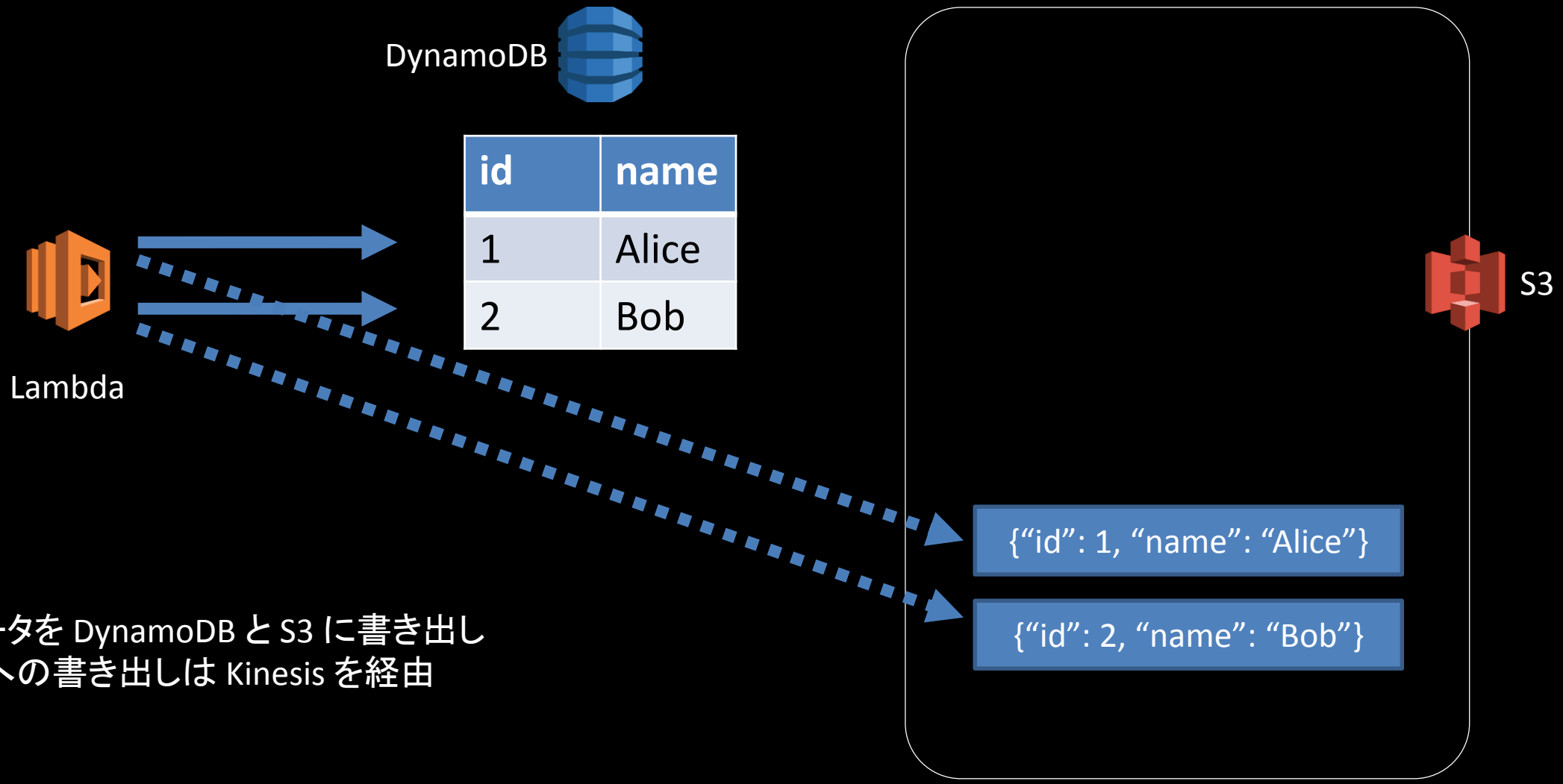
Data Pipeline による DynamoDB のバックアップ（2017年秋）

- AWS Data Pipeline で DynamoDB テーブルを S3 に export
  - バックアップされる DynamoDB テーブルの read capacity を消費する
    - デフォルトでは 25% の read capacity を消費する
    - Export のためだけに capacity を上げられないので、export に時間がかかる
  - Data Pipeline は Amazon EMR (Elastic MapReduce) クラスタを動かす
    - パラメタ調整に手間がかかる
    - Data Pipeline のジョブが終了した後でも、EMR クラスタを構成するインスタンスが残っていたことがある
    - DynamoDB のバックアップには Data Pipeline は大がかり過ぎる
  - 静止点をバックアップするものではない

「差分ログ」の自作（2017年秋）

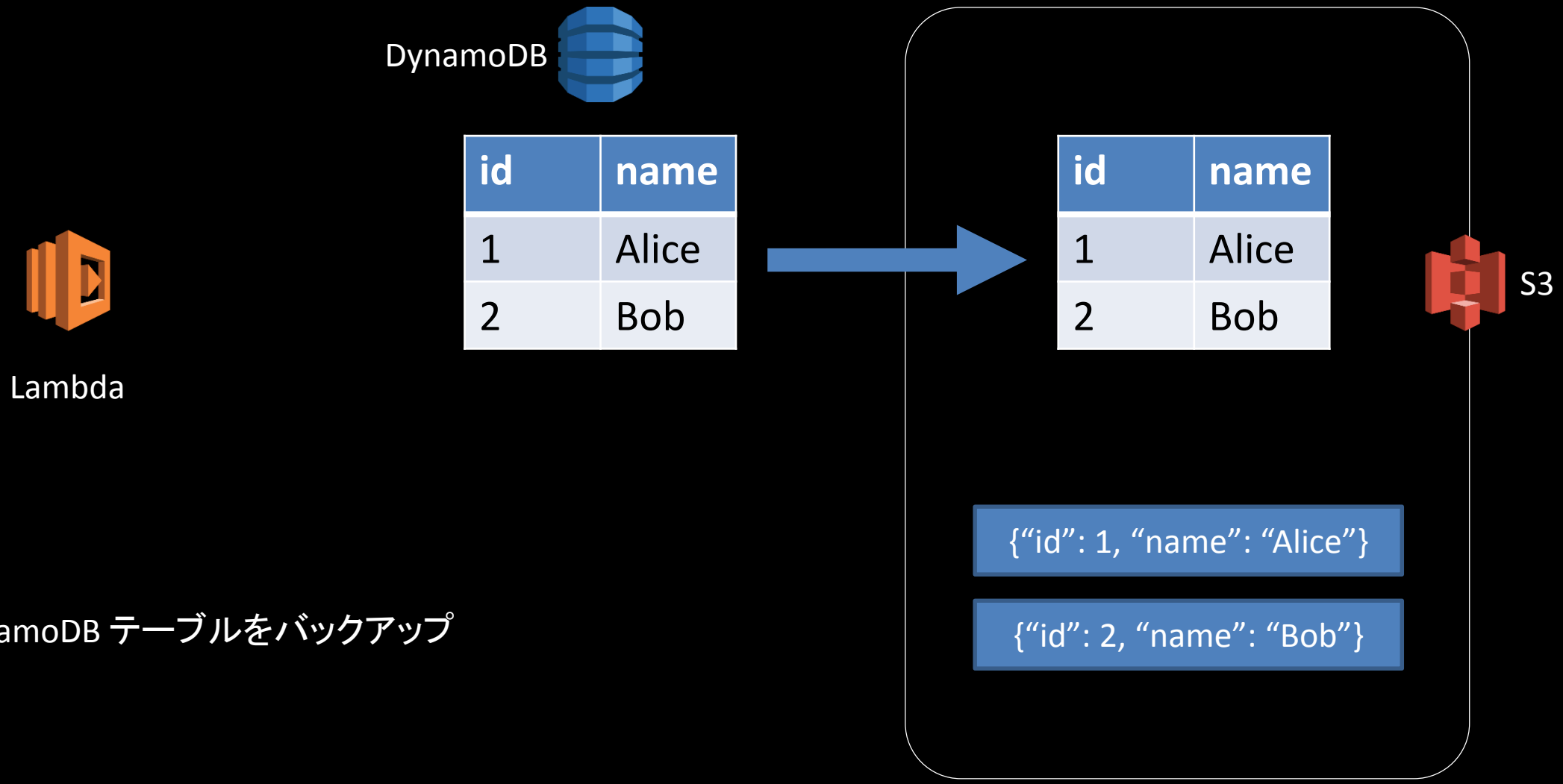
- バックアップ時からデータ破損直前までのデータも復旧したい
  - 仕組みを自作する必要がある
- S3 にあるデータを「差分ログ」として使う
  - MySQL の行ベースの binlog のようなもの

バックアップと「差分ログ」からテーブルを復旧 (1/6)

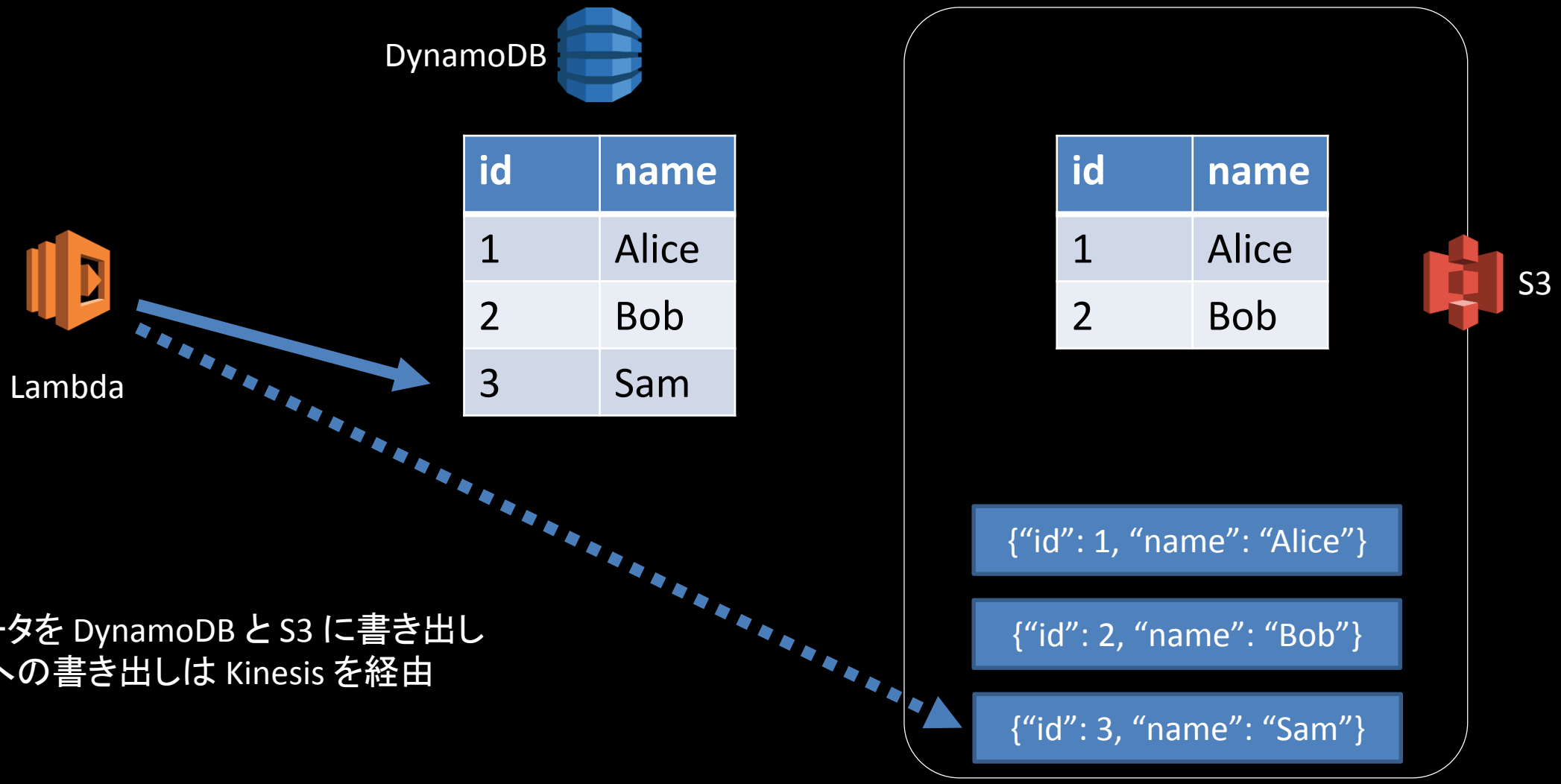


データを DynamoDB と S3 に書き出し
S3 への書き出しは Kinesis を経由

バックアップと「差分ログ」からテーブルを復旧 (2/6)

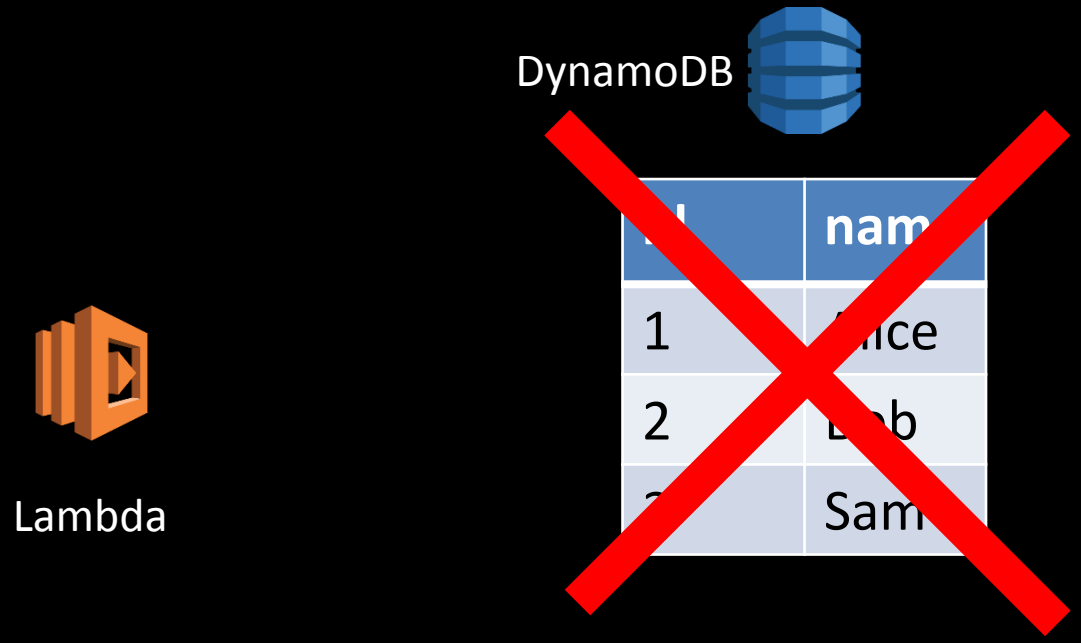


バックアップと「差分ログ」からテーブルを復旧 (3/6)

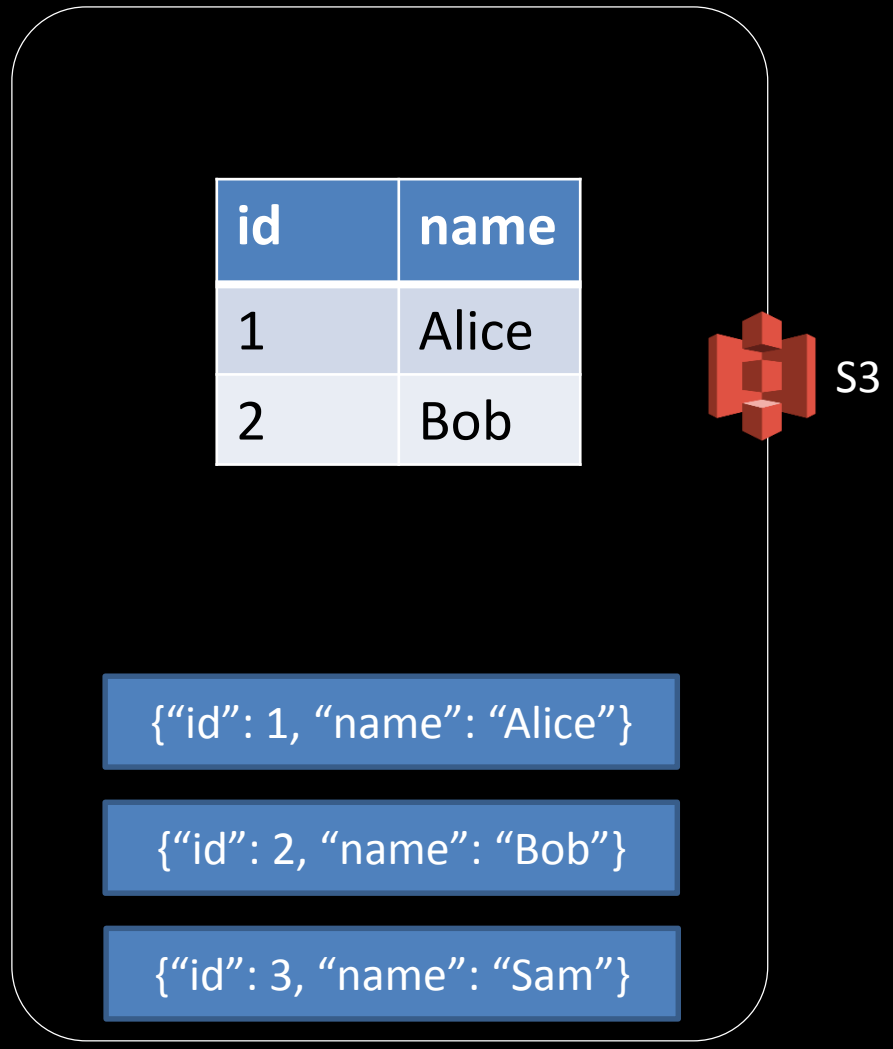


データを DynamoDB と S3 に書き出し
S3 への書き出しは Kinesis を経由

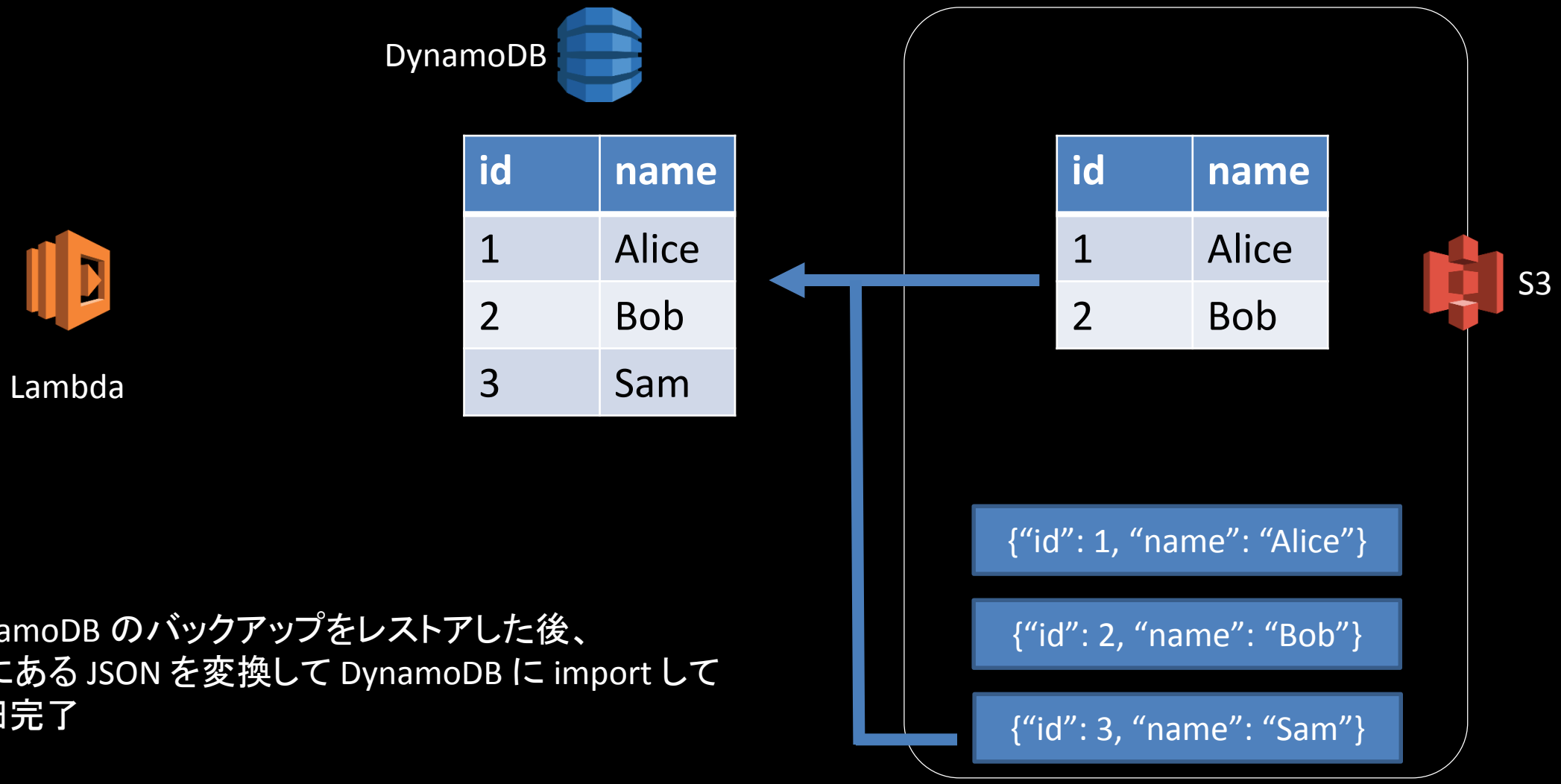
バックアップと「差分ログ」からテーブルを復旧（4/6)



DynamoDB テーブルのデータが破損

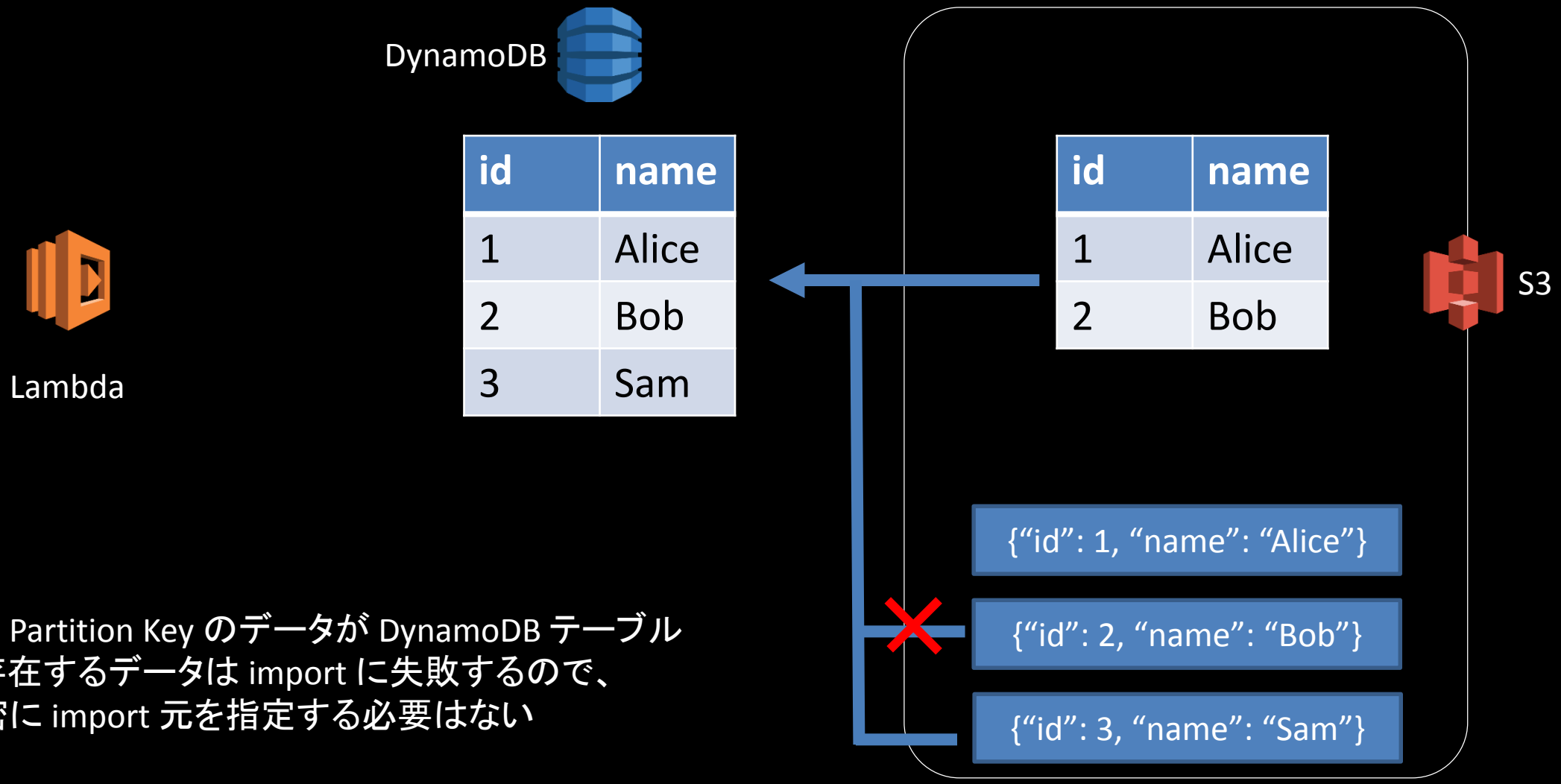


バックアップと「差分ログ」からテーブルを復旧 (5/6)



DynamoDB のバックアップをレストアした後、
S3 にある JSON を変換して DynamoDB に import して
復旧完了

バックアップと「差分ログ」からテーブルを復旧 (6/6)



同じ Partition Key のデータが DynamoDB テーブルに存在するデータは import に失敗するので、厳密に import 元を指定する必要はない

DynamoDB のバックアップ（2017年秋）

- 「差分ログ」から復旧するテーブルの設計
  - レコードを追加するだけで、更新・削除しないようにする
    - レコードが更新・削除されるテーブルを「差分ログ」から復旧するプログラムを書くのは大変
  - データ更新の厳密な前後関係の保証も難しい
    - Lambda でデータにタイムスタンプを付与しても、Lambda のインスタンス間で時計が一致することは保障されない
  - DynamoDB なら、レコードは追加のみのテーブル設計でもパフォーマンスの心配をしなくてよい

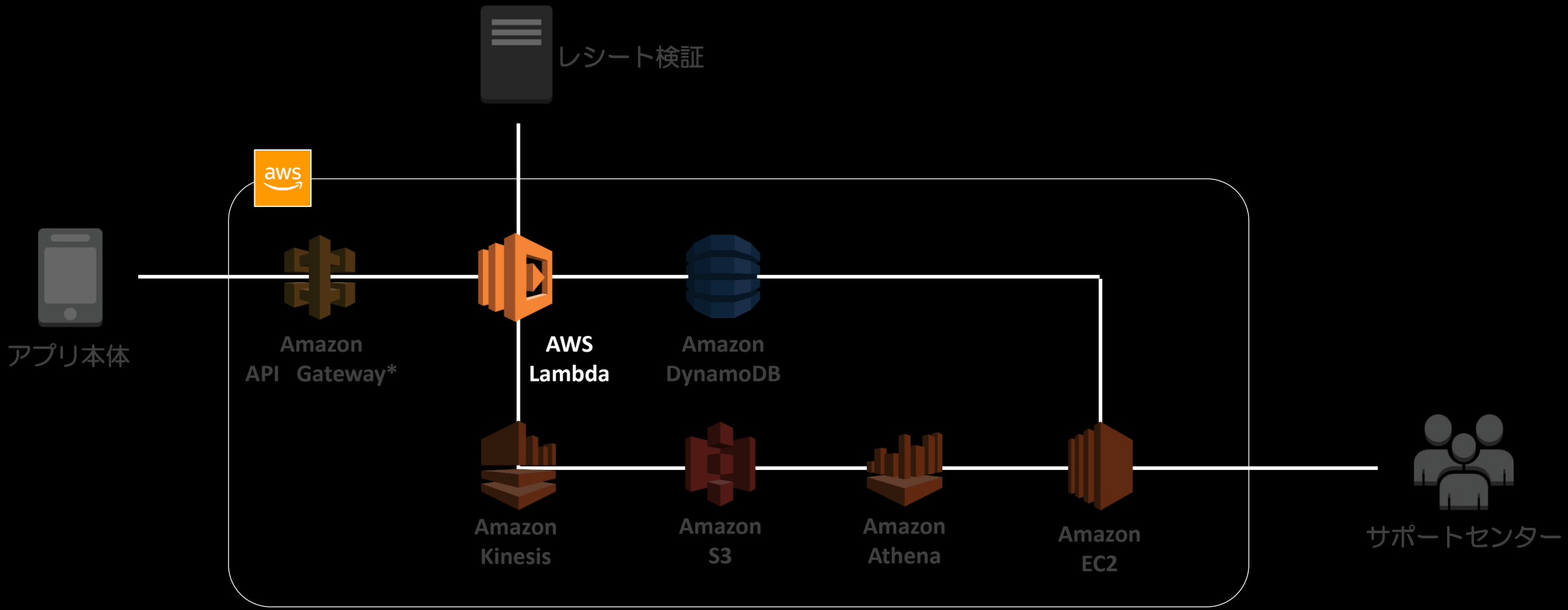
DynamoDB の On-Demand Backup（2017年冬）

- On-Demand Backup でバックアップが簡単に
  - バックアップされる DynamoDB の read capacity を消費しない
  - 数千万レコードのバックアップもすぐに終わる
    - O(1) ではないとのことだが、テストした範囲では、レコード数の増加に伴うバックアップ時間の増加はなかった
  - Point-in-Time Recovery ではない
  - レストアにはそれなりの時間がかかる
- Data Pipeline によるバックアップを On-Demand Backup に変更
  - 運用コストが大幅に削減
  - 「差分ログ」の保存は継続

DynamoDB の Point-in-Time Recovery（2018年春）

- 秒単位で Point-in-Time Recovery (PITR) できるようになった
  - On-Demand Backup の定期実行を PITR のできる自動バックアップに変更
    - 「差分ログ」の保存は継続
- アプリケーション要件によっては、PITR だけで十分と思われる

AWS Lambda 技術詳細



Lambda 開発言語は Node.js を採用

- Node.js v6.10 を採用した理由
  - ブラウザで書き換えてすぐ試せる
    - 2017年秋にエディターが AWS Cloud9 になって更に便利に
  - https://github.com/voltrue2/in-app-purchase
    - レシート検証の Node.js モジュール
  - ストアのレシート検証サーバのレスポンス待ちが支配的なので、実行時間の短縮をしなくてもよい
    - Lambda は実行時間に応じてお金がかかるので、Lambda から外部プロセスを呼び出して実行完了を待つのは、Lambda のベストプラクティスから外れるので注意
    - レシート検証サーバのレスポンス待ち時間にお金がかかっても、Lambda を使うことで節約になる

Lambda の開発フローの課題（ブラウザ内で作業が完結しない）

- AWS Cloud9 が2017年冬にリリース
  - ブラウザでコードを書いて、実行、デバッグできる
  - ブラウザだけで開発できるようになったか？
- Cloud9 から VCS と連携できない
  - VCS と連携できれば、ほぼブラウザ内で作業を完結できそう
  - それまではローカル PC でコードを書く前提で作業を効率化する

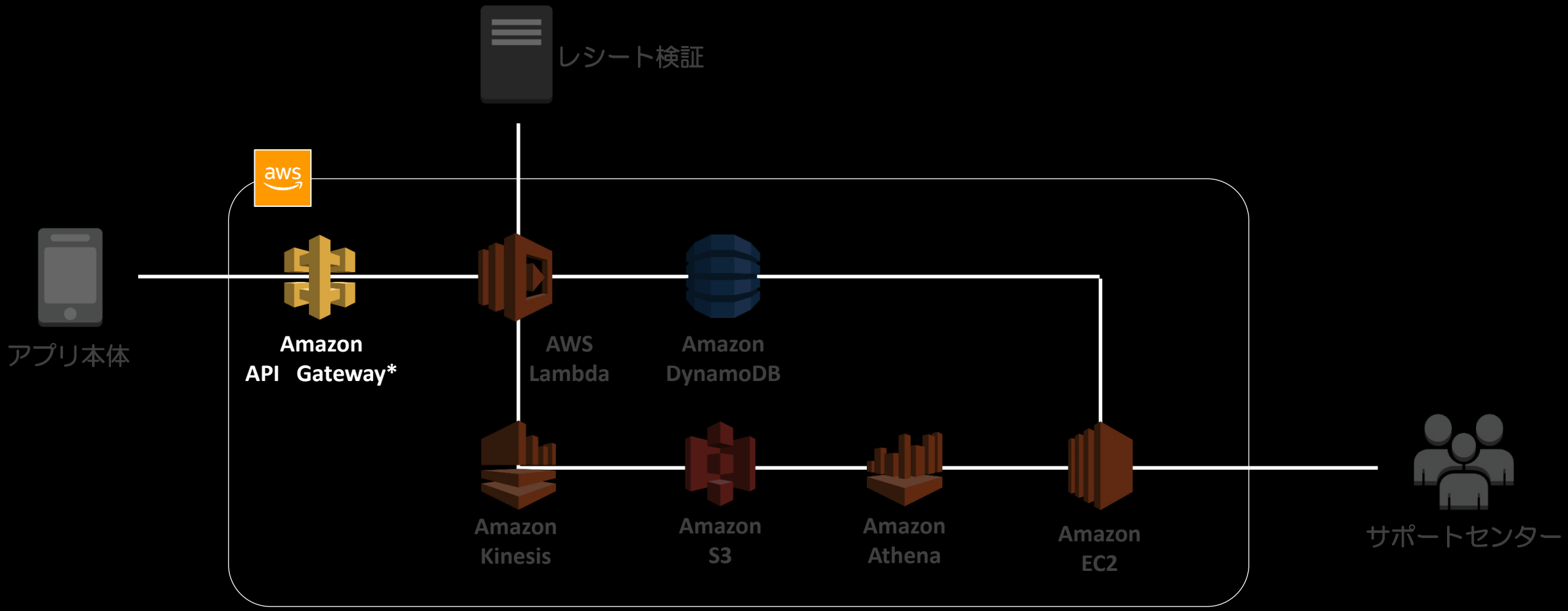
Lambda の開発フローの課題（複数の Lambda コードの管理）

- 複数 Lambda の共通部分の管理が煩雑
  - 複数の Lambda コードに、同じ環境変数、同じライブラリが重複して入ってしまう
    - コードが独立しているメリットの裏返し
  - ブラウザの外で開発フローを構築すれば一元管理は可能

Lambda のアプリ外での利用

- 負荷テストのクライアントとして使う
  - JMeter master/salve で slave の数や JMeter のパラメータを調整する手間がなくなる
    - 実際、この調整がうまくいかず、Lambda がスケールしないと誤解していた時期があった
  - 負荷テストのクライアントがステートレスでよいなら検討する価値あり
  - Goad
    - Go で書かれた負荷テストのクライアントを複数リージョンの AWS Lambda にデプロイして負荷テストを実行するツール
- FFXV PE アプリが使うアセットがダウンロードできるかチェック
  - アセット１個をダウンロードする Lambda をアセットの数だけ実行して、チェックを短時間で終えることができる

Amazon API Gateway 技術詳細



API Gateway を使う際の注意点

- Stage への Deployment History
  - 履歴には、デプロイ時の API Gateway のパラメタは記録されない
    - デプロイ時の API Gateway が使用する Lambda コードのバージョンも記録されない
  - 昔のデプロイを再現できるようにするには、デプロイ時のパラメタを独自に保存する必要がある

サーバレスアプリケーションの管理コスト

- サーバーレスサービスの組み合わせ、ログ、モニタリングの管理
  - CloudWatch Logs を CloudTrail で集約
  - CloudWatch Metrics / CloudWatch Alarms
  - サービス同士の接続は手動で管理
    - 開発、テスト、本番と複数環境があるので、管理する組み合わせは更に増える
- FFXV PE では何とかなったが…
  - より大規模な構成になれば、適切なソリューションや方法論が必要になると思われる
  - 小規模な構成に向いた手法はないか？
    - AWS X-Ray や Epsagon などのソリューションがいくつかあるが、本格的な（サーバーレス）マイクロサービスアーキテクチャ向け、という印象

まとめ

- API Gateway + Lambda + DynamoDB
  - 運用コストを大幅に下げつつサーバの安定運用を実現できた
- DynamoDB
  - トランザクションなしでアプリケーションの要件を満たせるか十分な検討が必要
  - On-Demand Backup や Point-in-Time Recovery でバックアップは非常に楽になった
- サーバーレス開発
  - サーバーレスアプリケーション開発に課題はあるが、小規模なアプリケーションであれば大きな問題とはなりにくい

ご清聴ありがとうございました