JAPAN | 2024

aws SUMMIT

AWS-12

データ基盤のコストを最適化するベストプラクティス

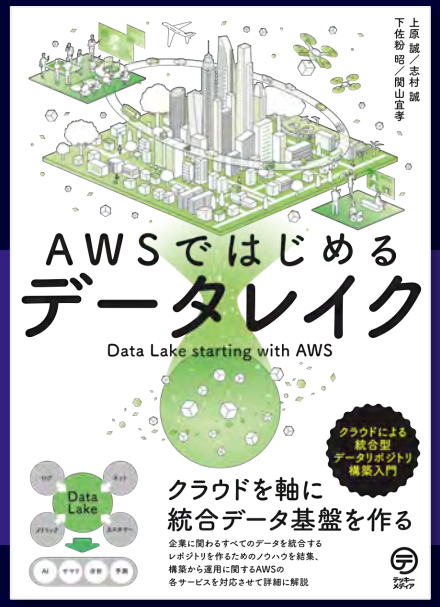
関山 宜孝
アマゾン ウェブ サービス ジャパン合同会社
Principal Big Data Architect, AWS Glue

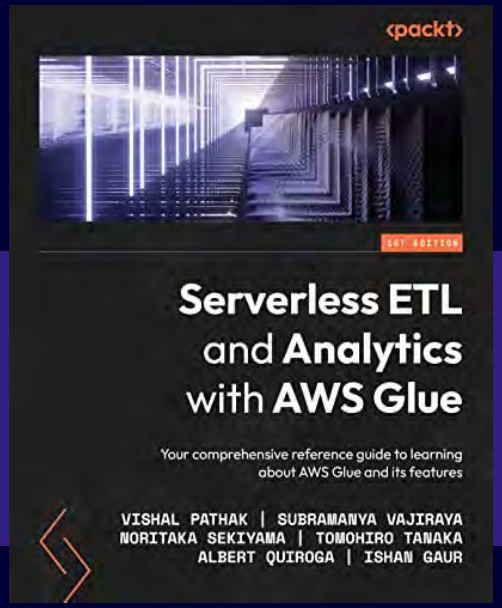
自己紹介

関山 宜孝

Principal Big Data Architect, AWS Glue

- 5年間 AWS サポートにて技術支援を担当
- 2019年にAWS Glue 開発チームにジョイン





@moomindani

moomindani

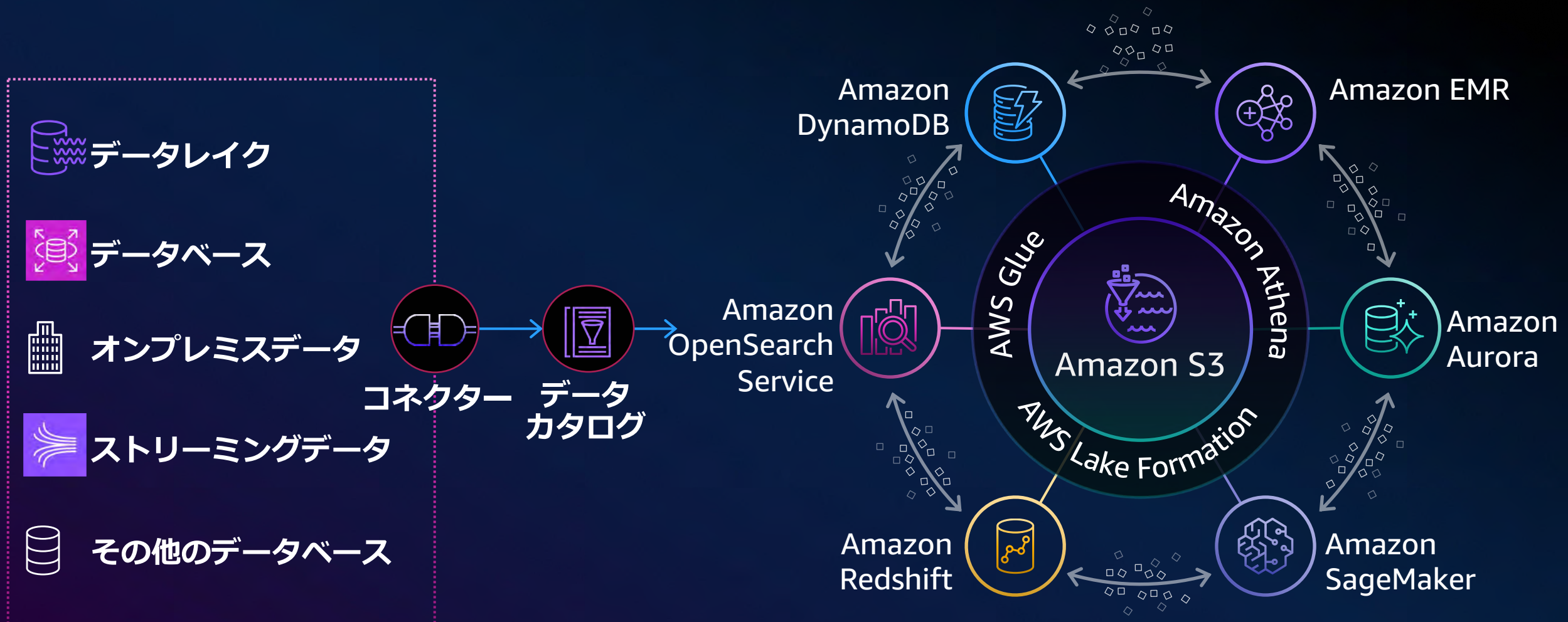
本セッションについて

本セッションでは、AWS で費用対効果の高いデータ基盤を構築・運用するベストプラクティスを解説します。

ワークロードに適したサービスやオプションを選択し、コストを犠牲にすることなくスケーラブルでパフォーマンスの高いアーキテクチャを設計します。

昨今、多くのお客様のデータ活用において
コストへの関心が高まっている

AWS を用いたデータ活用



AWS を用いたデータアーキテクチャの例

データ基盤のコストの課題

サービス横断で
コストが発生する

データ量と処理に
強く依存する

予測が難しい

適切な設計により大幅なコストダウンが可能

Cloud Financial Management (CFM) フレームワーク

AWS Well-Architected Framework で定義されたコスト最適化プロセス

Cloud Financial Management (CFM) フレームワーク

AWS Well-Architected Framework で定義されたコスト最適化プロセス

データ基盤に適用

コストの可視化

If you can’t see it,
you can’t change it

コストの可視化

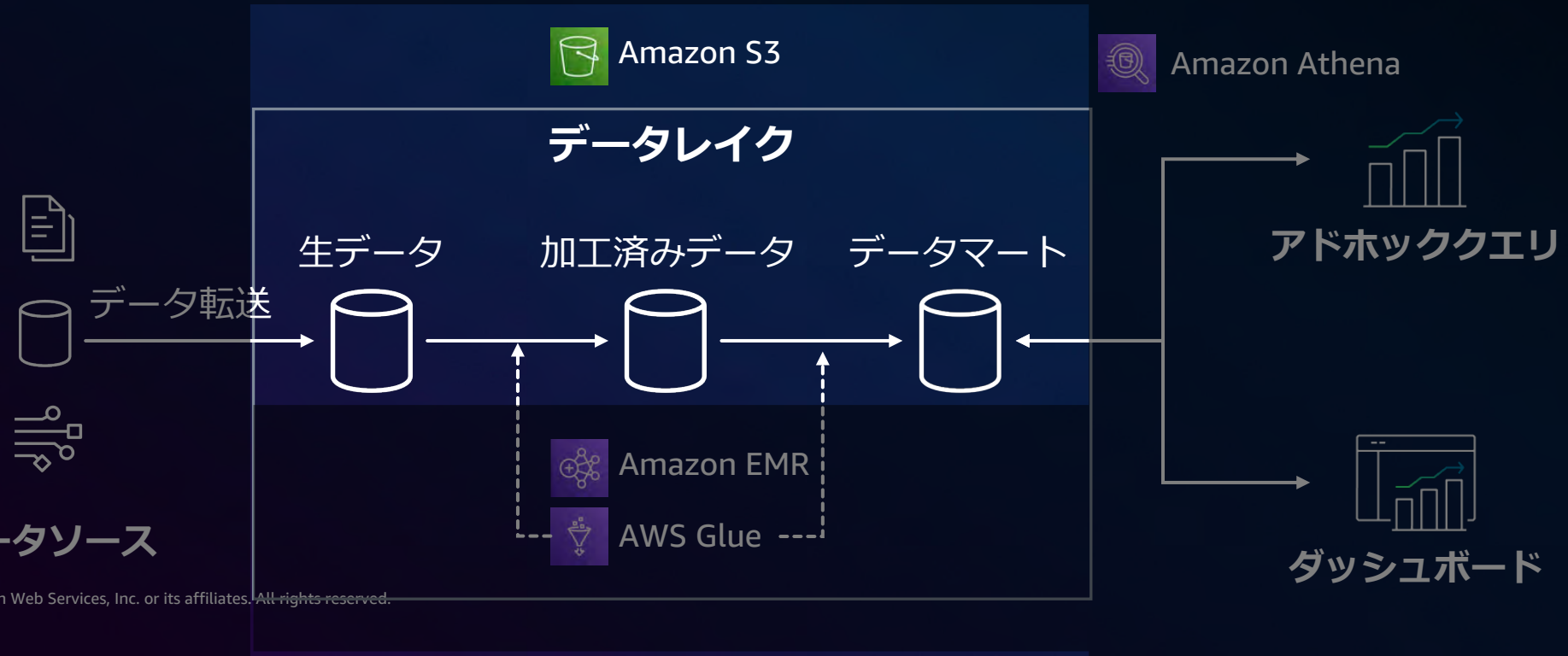
ストレージ

データ転送

データ
パイプライン

クエリ

コストの可視化

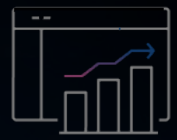


コストの可視化

ストレージ

データ転送

データ
パイプライン



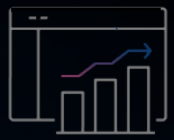
クエリ

コストの可視化

ストレージ

データ転送

データ
パイプライン



クエリ

Amazon S3

Amazon Athena

データレイク

データ転送

データソース

生データ

加工済みデータ

データマート

Amazon EMR

AWS Glue

アドホッククエリ

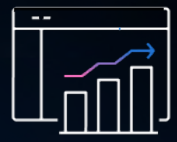
ダッシュボード

コストの可視化

ストレージ

データ転送

データ
パイプライン



クエリ

Amazon S3

データレイク

データソース

データ転送

生データ

加工済みデータ

データマート

Amazon EMR

AWS Glue

Amazon Athena

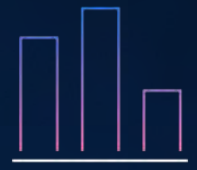
アドホッククエリ

ダッシュボード

AWS Cost Explorer

利用量に基づく
AWSコストを確認
する無料ツール

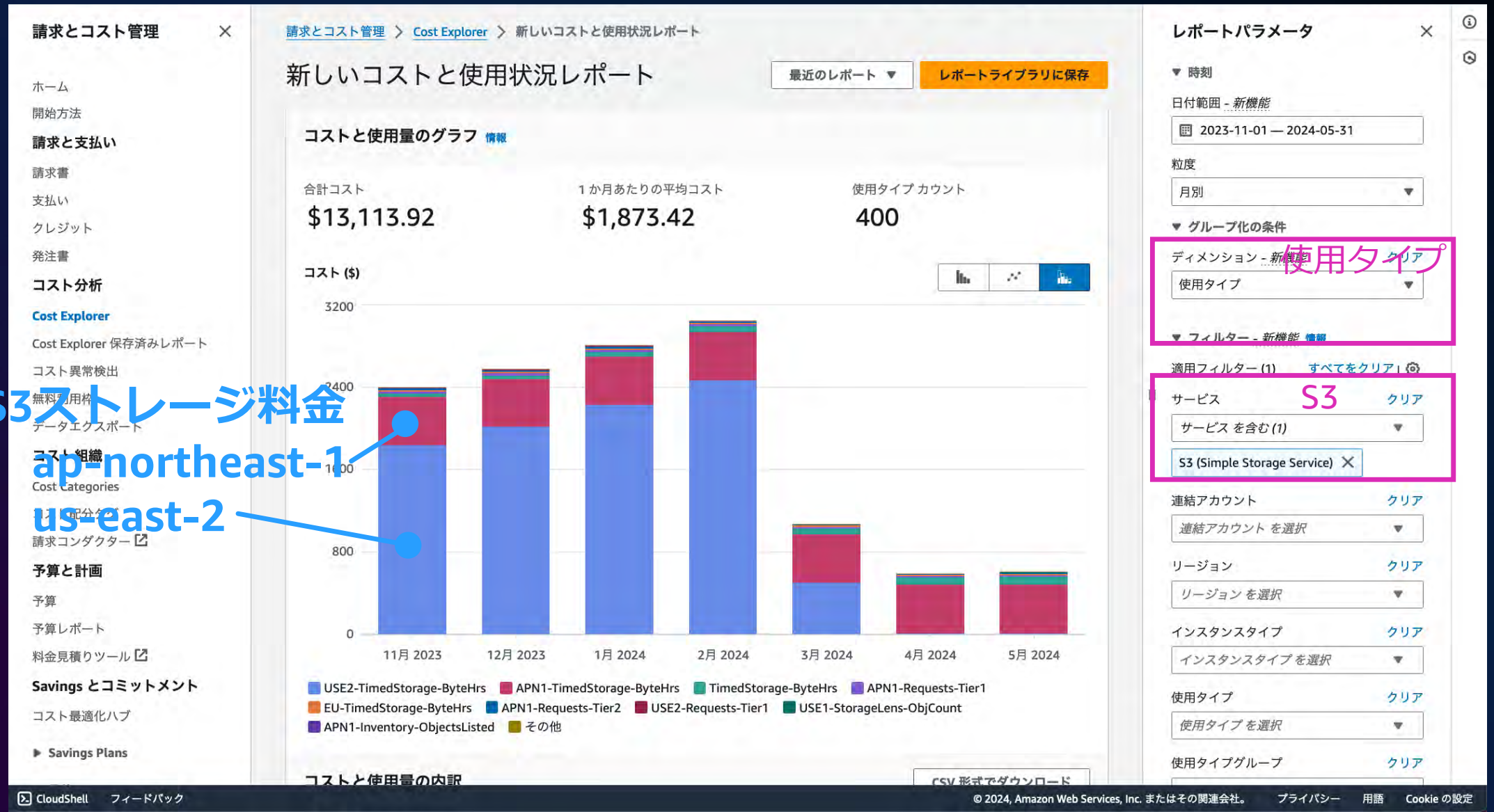
AWS費用を監視・
最適化



デフォルトで直近
14ヶ月分のコスト
を遡って分析

将来の12ヶ月分の
コストを予測

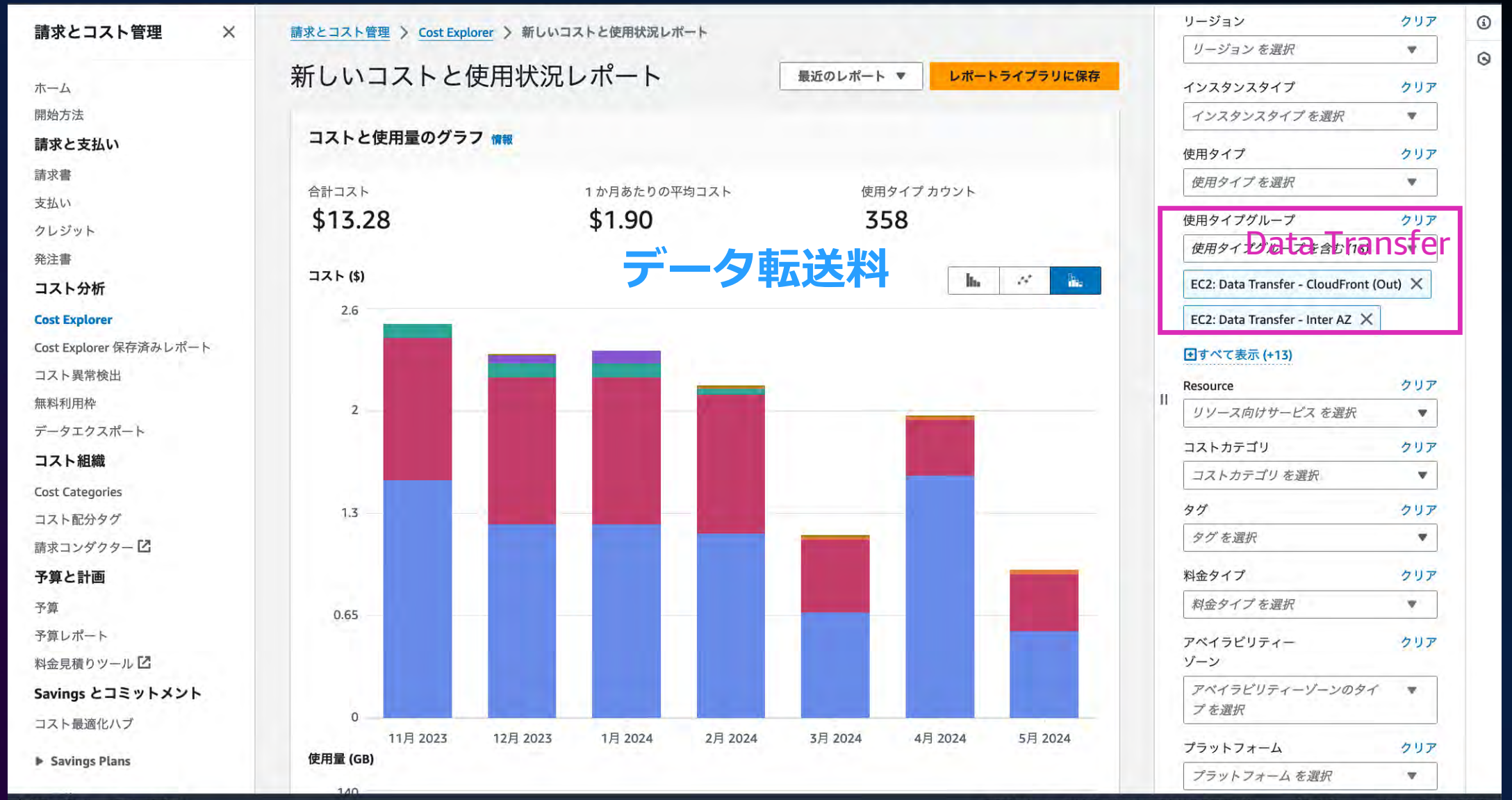
AWS Cost Explorer：ストレージ



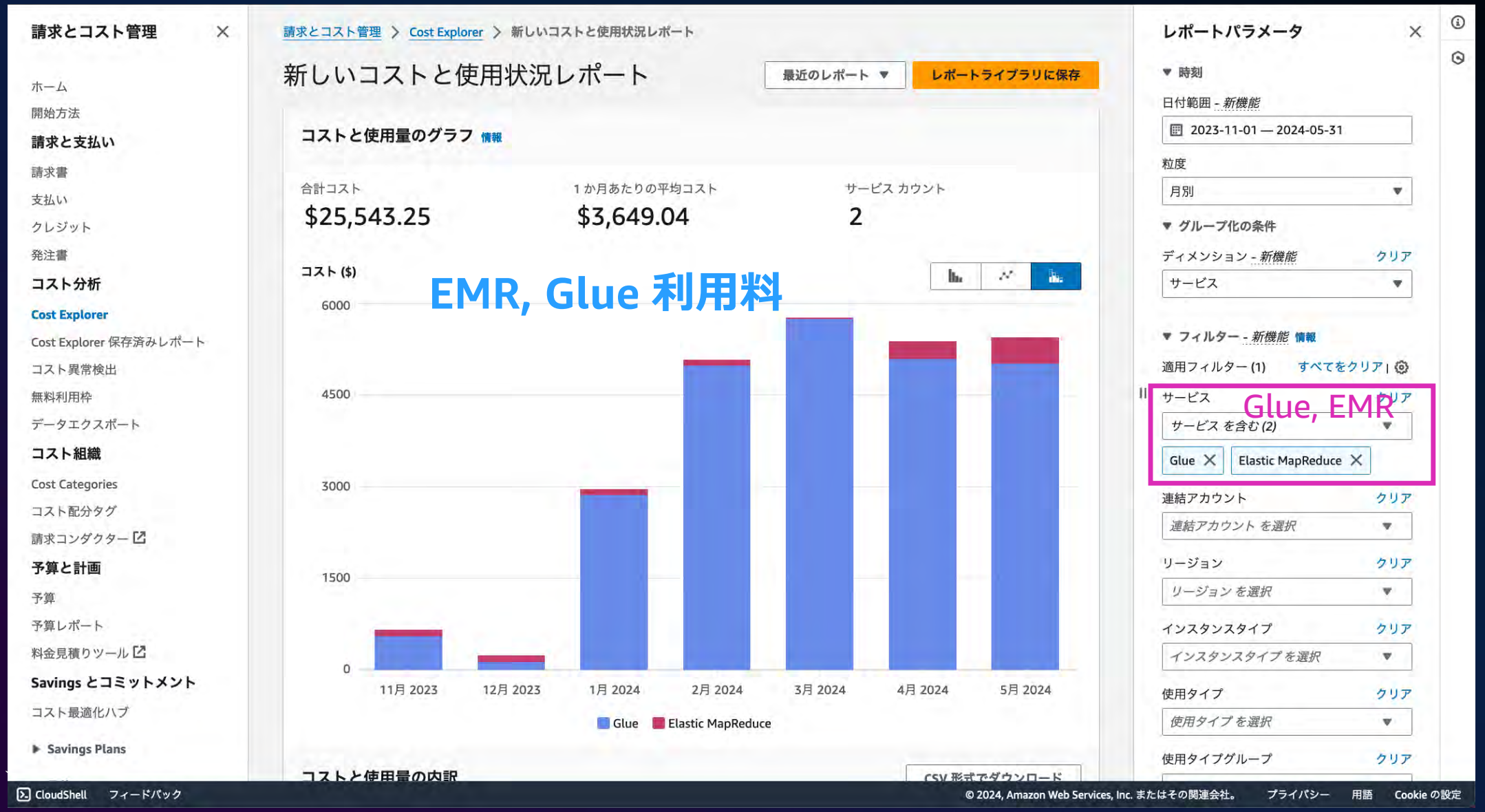
S3ストレージ料金

- ap-northeast-1
- us-east-2

AWS Cost Explorer：データ転送



AWS Cost Explorer：データパイプライン



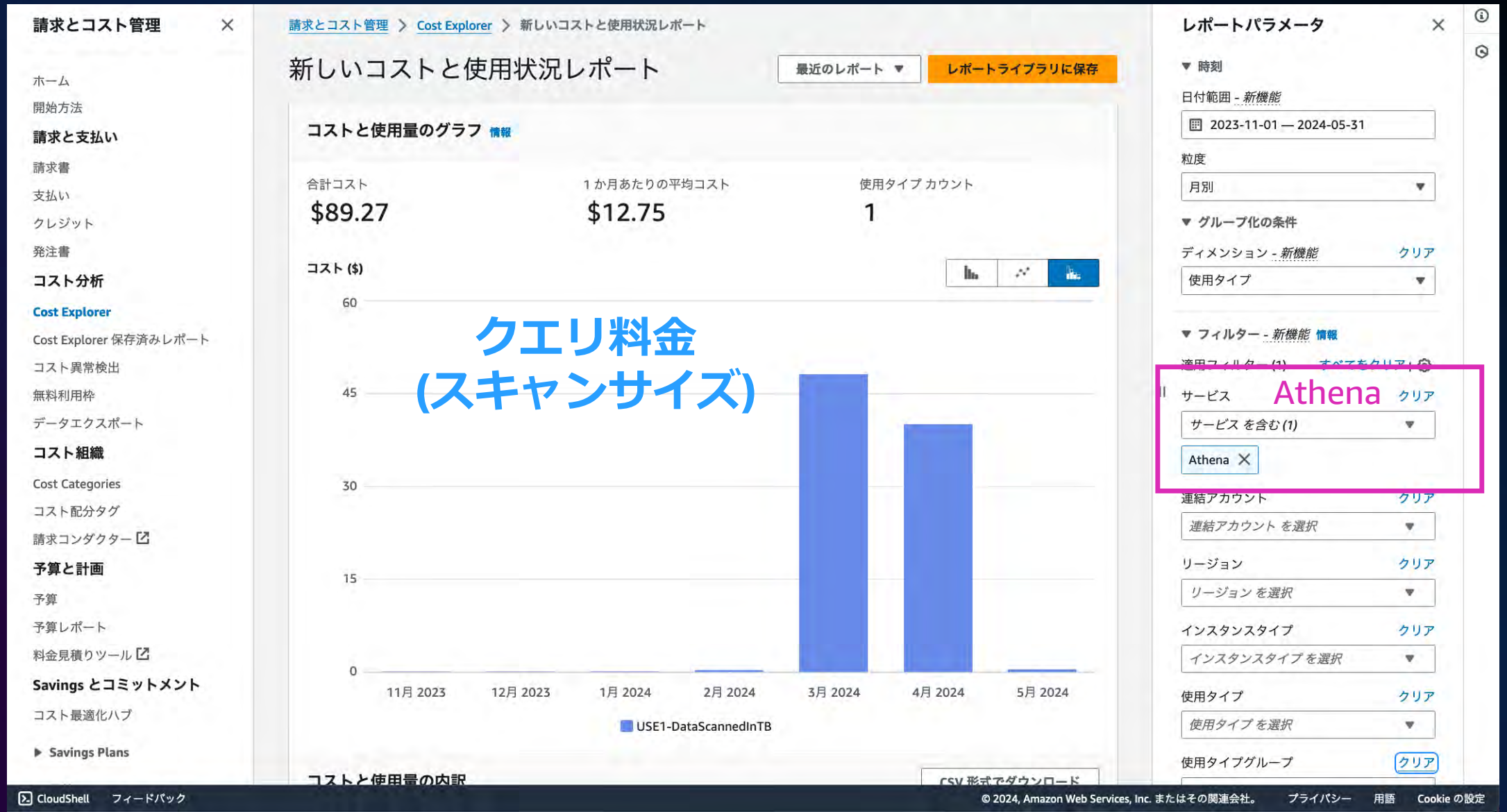
ストレージ

データ転送

データパイプライン

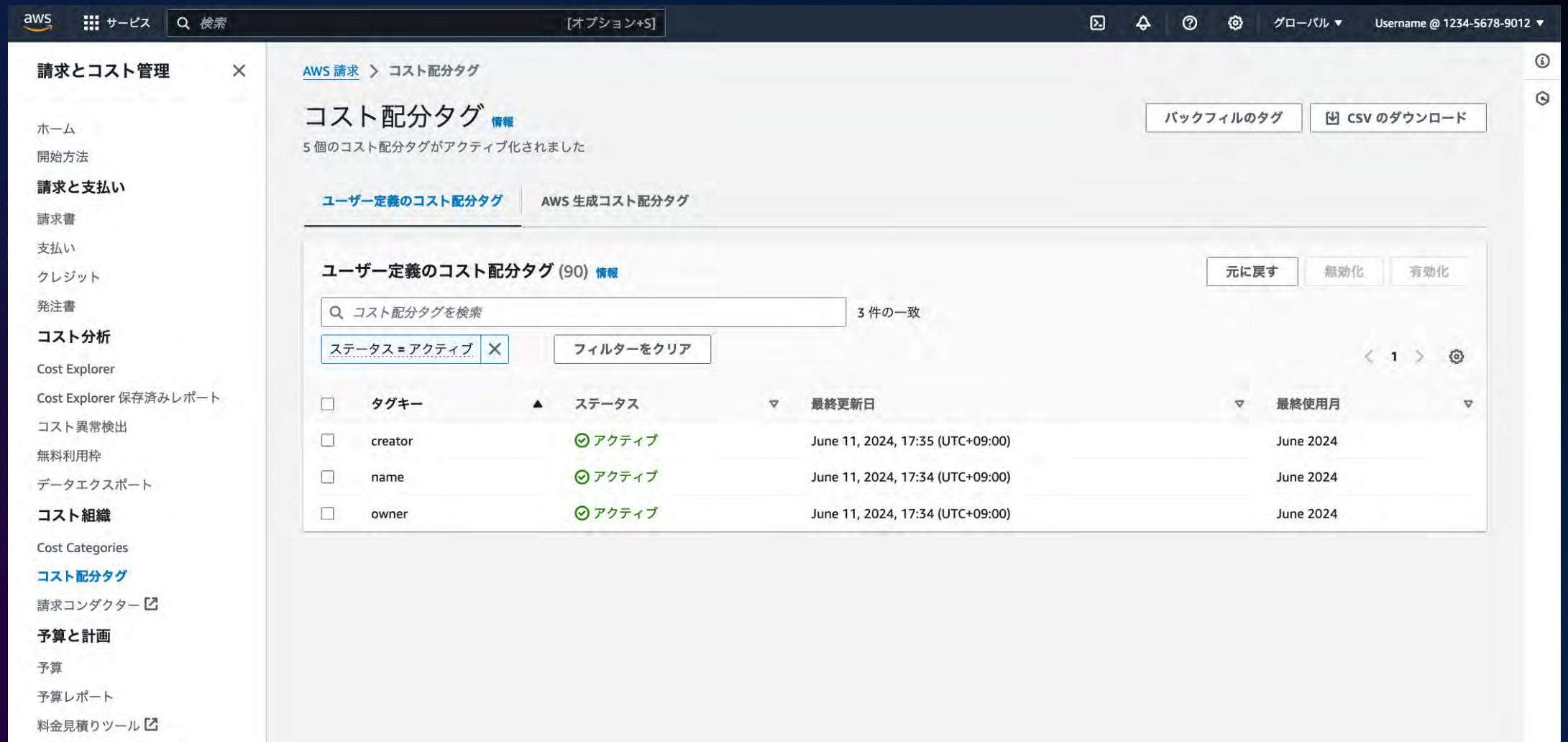
クエリ

AWS Cost Explorer：クエリ



コスト配分タグ

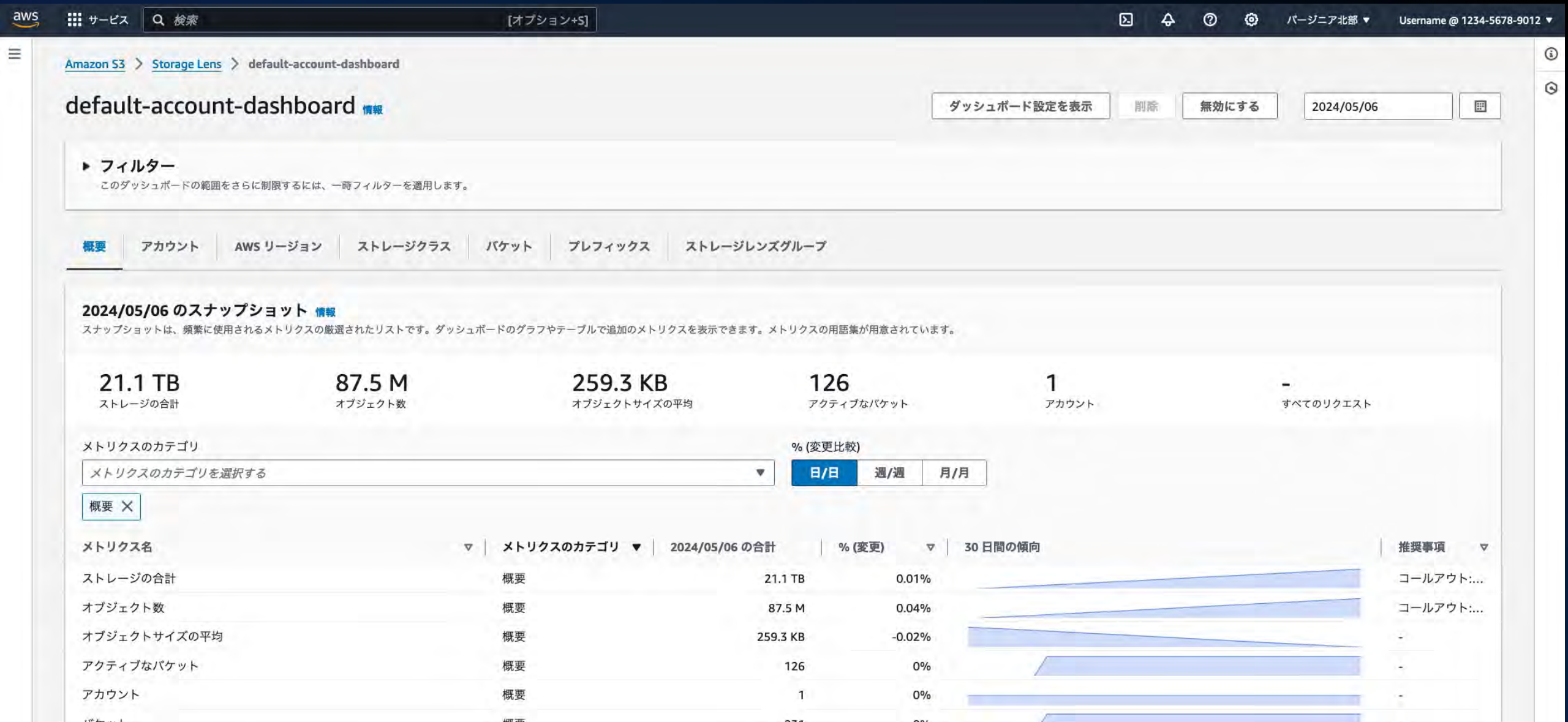
AWS リソースに設定されたタグごとにコスト配分を管理可能



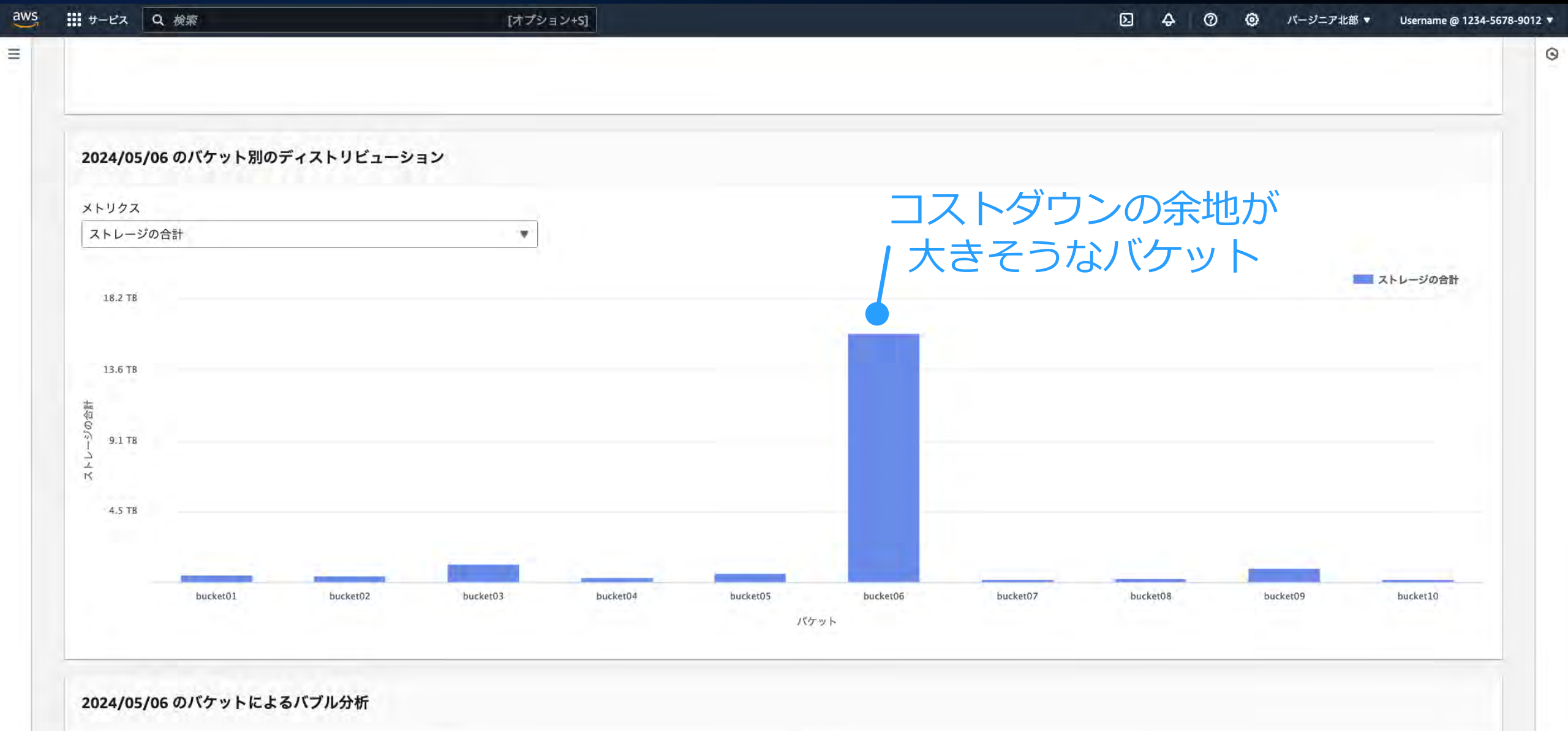
Amazon S3 Storage Lens

- Amazon S3 コンソール上のインタラクティブなダッシュボード
- オブジェクトストレージの利用状況を組織全体で可視化
- リージョンや、ストレージクラス、バケット、プレフィックスでドリルダウン
- 利用量とアクティビティに関する詳細なメトリクス

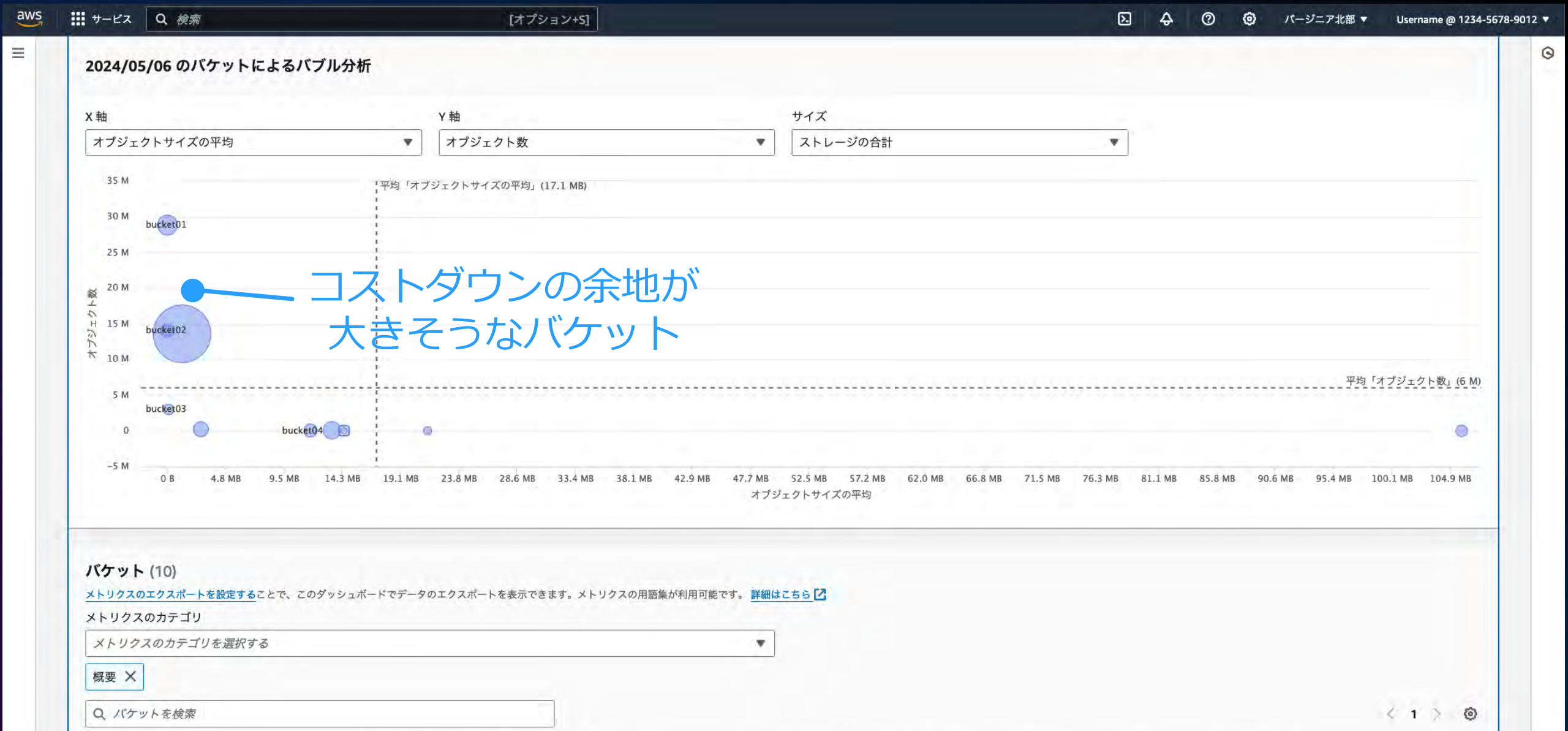
S3 Storage Lens



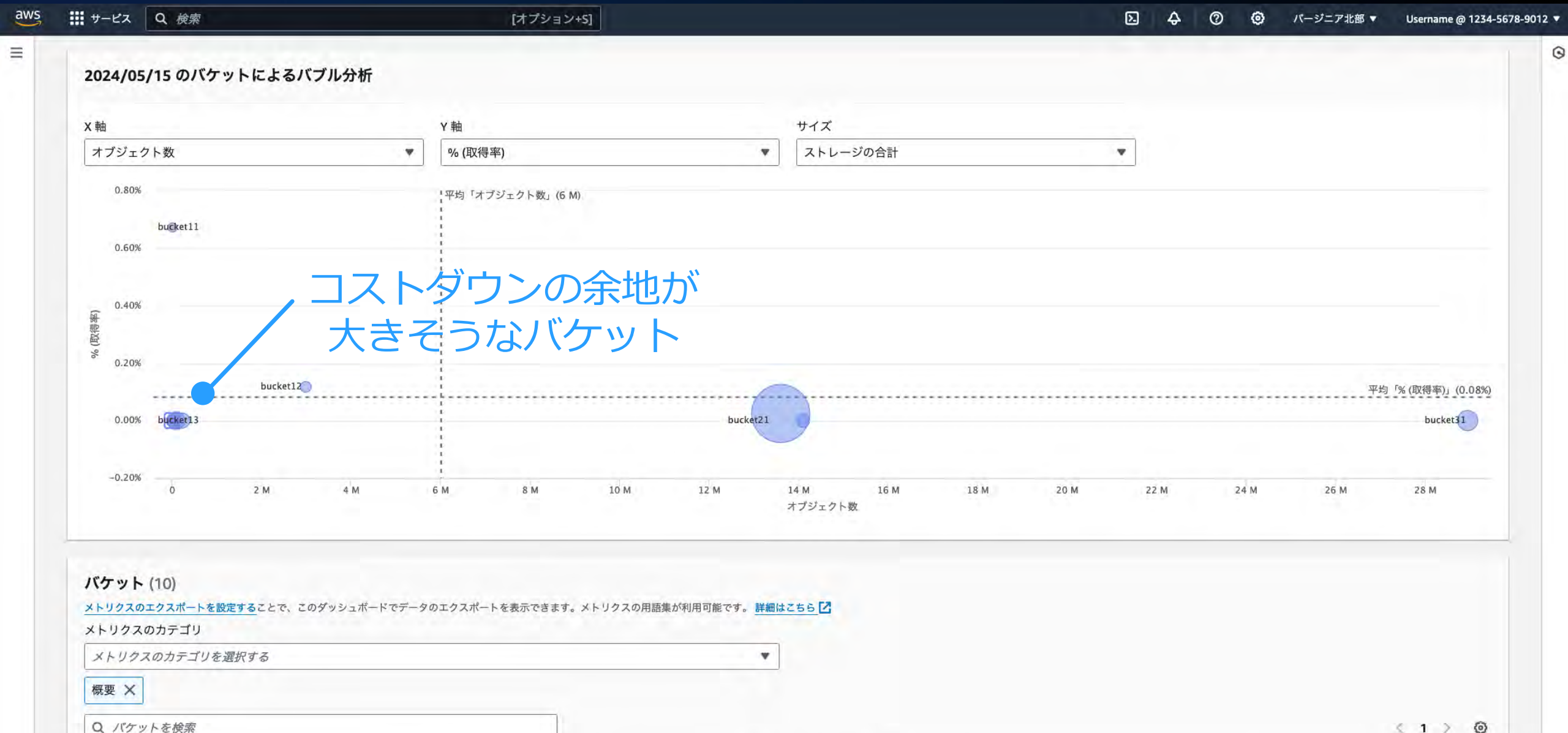
データサイズの大きいバケットを特定



平均オブジェクトサイズの小さいバケットの特定



アクセス頻度の低いデータの特定



コストの最適化

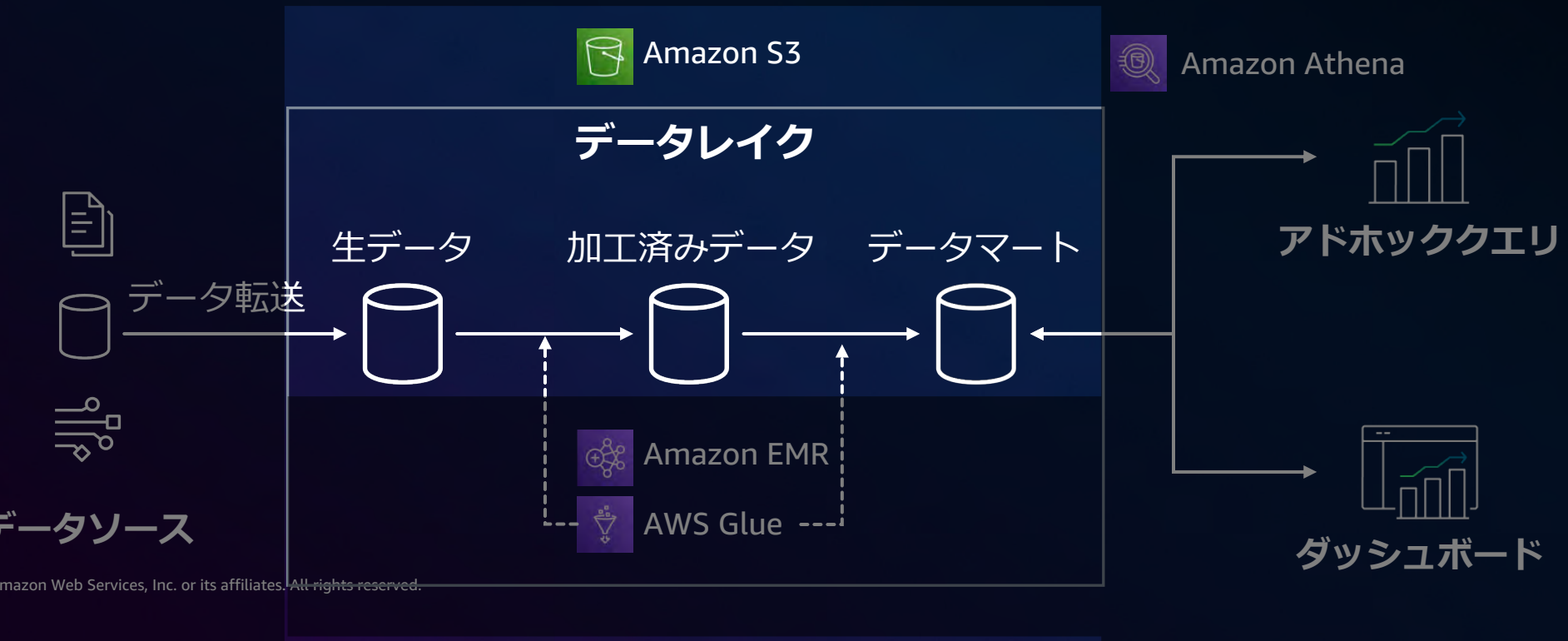
ストレージコストの最適化

ストレージ

データ転送

データ
パイプライン

クエリ



ストレージコストの最適化

課題

- ✗ 不適切なストレージクラス
- ✗ ライフサイクル管理の欠如
- ✗ 効率の悪いストレージ利用
- ✗ 不要な/一時的なデータの保持

最適化

- ✓ S3 ストレージクラスの活用
- ✓ S3 ライフサイクルの活用
- ✓ データの圧縮
- ✓ 不要なデータの自動削除

ストレージコストの最適化

課題	最適化
✗ 不適切なストレージクラス	✓ S3 ストレージクラスの活用
✗ ライフサイクル管理の欠如	✓ S3 ライフサイクルの活用
✗ 効率の悪いストレージ利用	✓ データの圧縮
✗ 不要な/一時的なデータの保持	✓ 不要なデータの自動削除

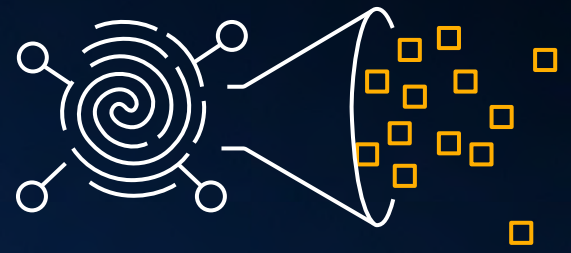
データ要件に適した S3 ストレージクラスの選択

アクセス頻度

例：書き込みは一度
読み取りは多数
読み書きともに直近のデータ
へのアクセスが多い

保存期間

例：データレイクで直近5年分
のデータを保管・分析



性能

例：直近のデータはすぐに取
得できる必要あり
1年以上前のデータの取得には
時間がかかってもよい

S3 ストレージクラスの選択

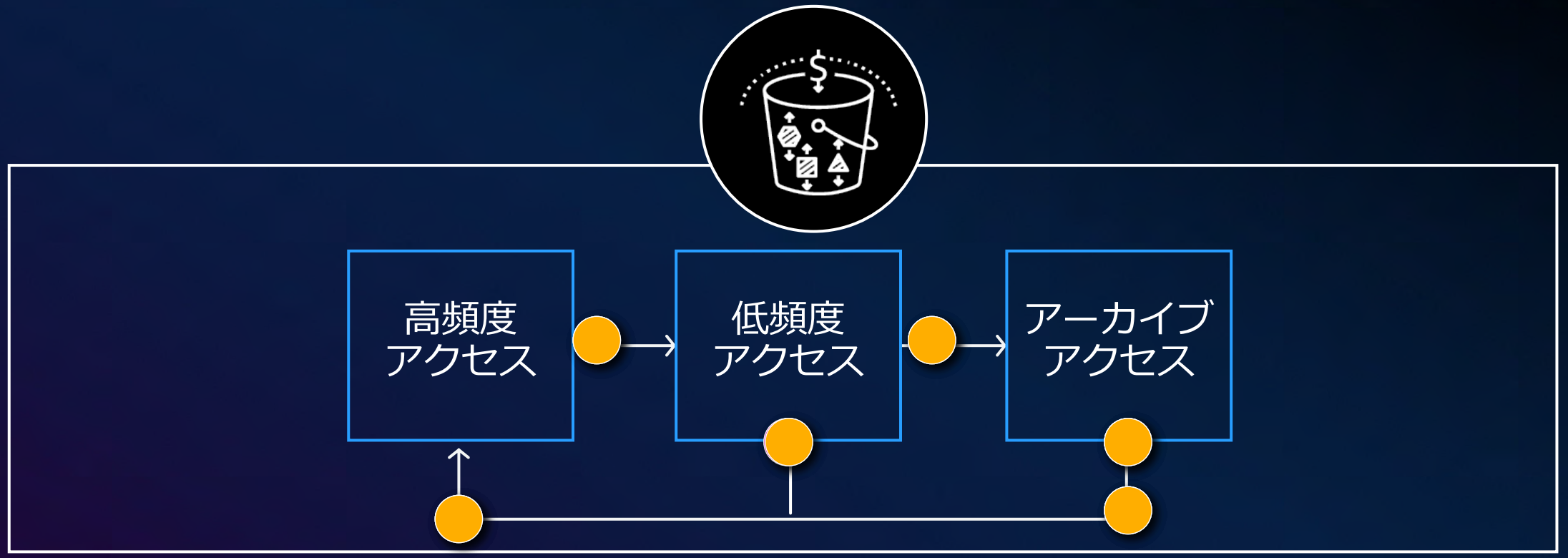
S3 Standard	S3 Standard-IA	S3 Glacier Instant Retrieval	S3 Glacier Flexible Retrieval	S3 Glacier Deep Archive
				
頻繁にアクセスするデータ	たまにアクセスするデータ	ごく稀にアクセスするデータ	アーカイブするデータ	長期間アーカイブするデータ

アクセス時間：ミリ秒

アクセス時間：分・時間

ホット → コールド

Intelligent-Tiering – ストレージクラスの自動選択



ミリ秒アクセス(自動)

ストレージコストの最適化

課題	最適化
✗ 不適切なストレージクラス	✓ S3 ストレージクラスの活用
✗ ライフサイクル管理の欠如	✓ S3 ライフサイクルの活用
✗ 効率の悪いストレージ利用	✓ データの圧縮
✗ 不要な/一時的なデータの保持	✓ 不要なデータの自動削除

データ要件に適したライフサイクル設定の例

S3 Standard

1年

S3 Glacier
Instant Retrieval

2年

S3 Glacier
Deep Archive

ストレージコストの最適化

課題	最適化
✗ 不適切なストレージクラス	✓ S3 ストレージクラスの活用
✗ ライフサイクル管理の欠如	✓ S3 ライフサイクルの活用
✗ 効率の悪いストレージ利用	✓ データの圧縮
✗ 不要な/一時的なデータの保持	✓ 不要なデータの自動削除

ストレージコストの最適化

課題	最適化
✗ 不適切なストレージクラス	✓ S3 ストレージクラスの活用
✗ ライフサイクル管理の欠如	✓ S3 ライフサイクルの活用
✗ 効率の悪いストレージ利用	✓ データの圧縮
✗ 不要な/一時的なデータの保持	✓ 不要なデータの自動削除

ストレージコストの最適化

課題

- ✗ 不適切なストレージクラス
- ✗ ライフサイクル管理の欠如
- ✗ 効率の悪いストレージ利用
- ✗ 不要な/一時的なデータの保持

最適化

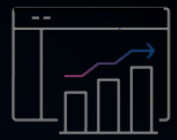
- ✓ S3 ストレージクラスの活用
- ✓ S3 ライフサイクルの活用
- ✓ データの圧縮
- ✓ 不要なデータの自動削除

データ転送コストの最適化

ストレージ

データ転送

データ
パイプライン



クエリ

データ転送コストの最適化

課題

- ✗ 非効率なデータ転送経路
- ✗ ローカリティの考慮不足
- ✗ 転送効率の悪いデータ転送
- ✗ 不要なデータレプリケーション

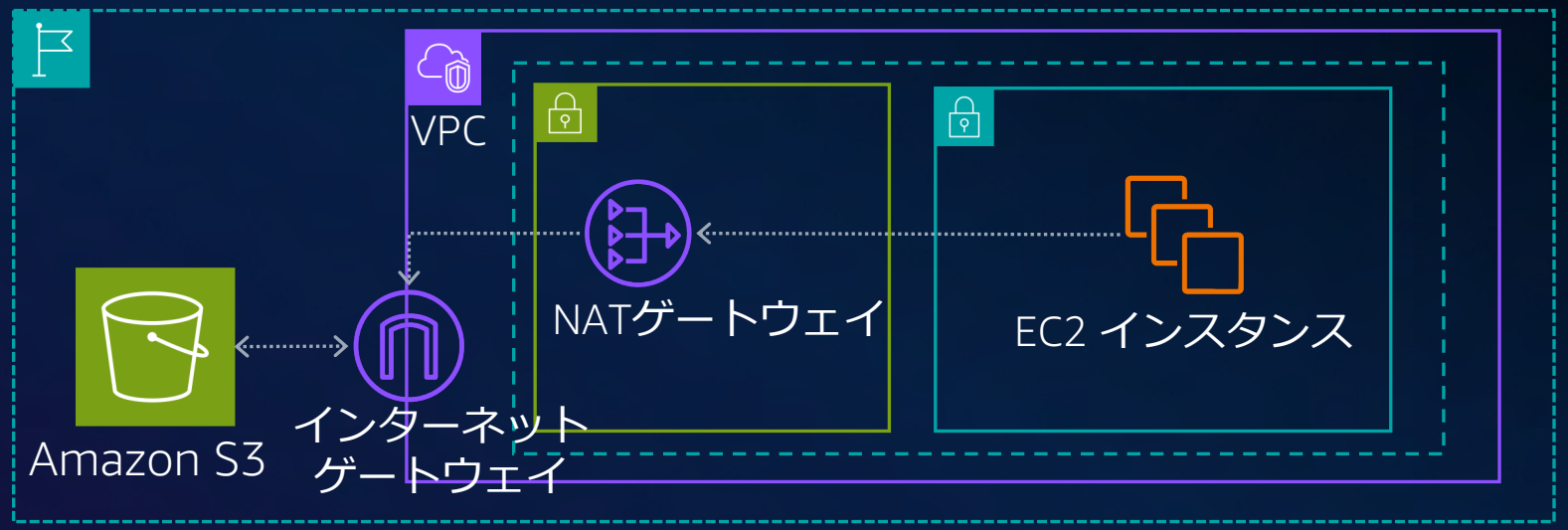
最適化

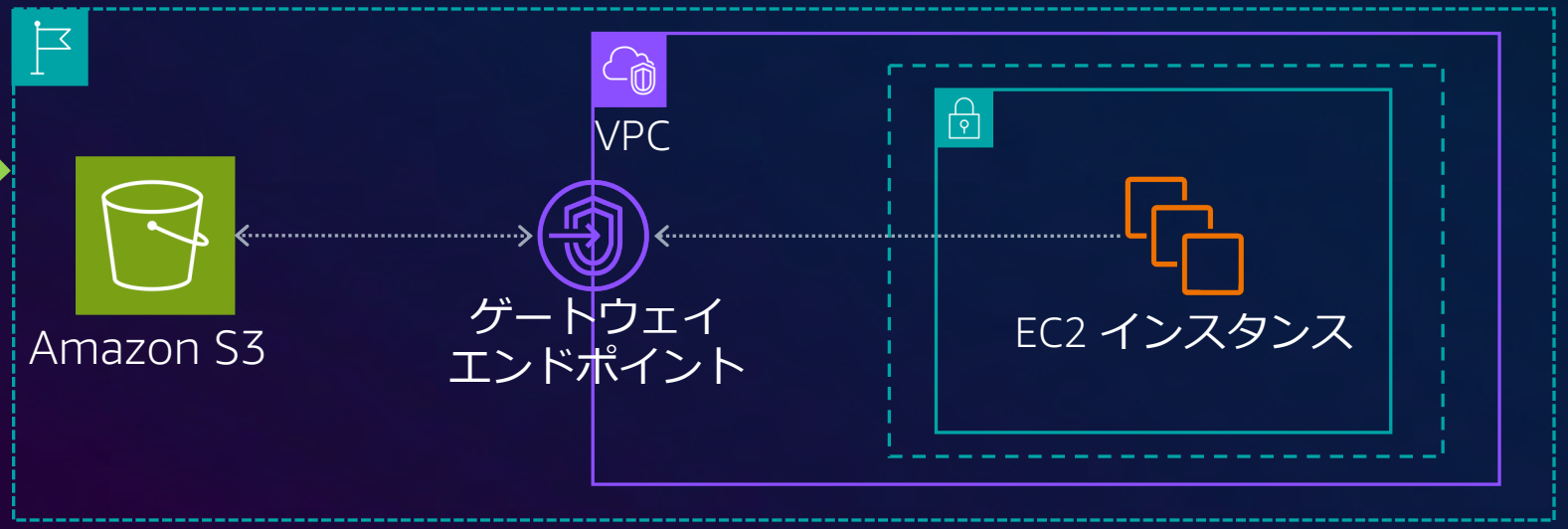
- ✓ データ転送経路の最適化
- ✓ ローカリティを考慮した設計
- ✓ 圧縮による転送効率の向上
- ✓ 必要最小限のレプリケーション

データ転送コストの最適化

課題	最適化
✗ 非効率なデータ転送経路	✓ データ転送経路の最適化
✗ ローカリティの考慮不足	✓ ローカリティを考慮した設計
✗ 転送効率の悪いデータ転送	✓ 圧縮による転送効率の向上
✗ 不要なデータレプリケーション	✓ 必要最小限のレプリケーション

データ転送経路の最適化





不要なデータ転送を廃止

NATゲートウェイを削減

データ転送コストの最適化

課題	最適化
✗ 非効率なデータ転送経路	✓ データ転送経路の最適化
✗ ローカリティの考慮不足	✓ ローカリティを考慮した設計
✗ 転送効率の悪いデータ転送	✓ 圧縮による転送効率の向上
✗ 不要なデータレプリケーション	✓ 必要最小限のレプリケーション

データ転送コストの最適化

課題	最適化
✗ 非効率なデータ転送経路	✓ データ転送経路の最適化
✗ ローカリティの考慮不足	✓ ローカリティを考慮した設計
✗ 転送効率の悪いデータ転送	✓ 圧縮による転送効率の向上
✗ 不要なデータレプリケーション	✓ 必要最小限のレプリケーション

データ転送コストの最適化

課題	最適化
✗ 非効率なデータ転送経路	✓ データ転送経路の最適化
✗ ローカリティの考慮不足	✓ ローカリティを考慮した設計
✗ 転送効率の悪いデータ転送	✓ 圧縮による転送効率の向上
✗ 不要なデータレプリケーション	✓ 必要最小限のレプリケーション

データ転送コストの最適化

課題

- ✗ 非効率なデータ転送経路
- ✗ ローカリティの考慮不足
- ✗ 転送効率の悪いデータ転送
- ✗ 不要なデータレプリケーション

最適化

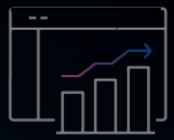
- ✓ データ転送経路の最適化
- ✓ ローカリティを考慮した設計
- ✓ 圧縮による転送効率の向上
- ✓ 必要最小限のレプリケーション

データパイプラインコストの最適化

ストレージ

データ転送

データ
パイプライン



クエリ

Amazon S3

Amazon Athena

データソース

データ転送

データレイク

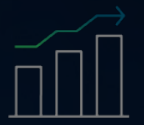
生データ

加工済みデータ

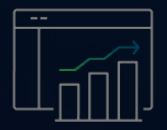
データマート

Amazon EMR

AWS Glue



アドホッククエリ



ダッシュボード

データパイプラインコストの最適化

課題

- ✗ デフォルトのままの料金プラン
- ✗ 過剰な/過小なリソース
- ✗ アイドルリソース
- ✗ ボトルネックのあるジョブ
- ✗ 必要以上のデータの処理
- ✗ 重い変換や不要な集合演算
- ✗ 不正なデータへの無駄な処理

最適化

- ✓ 適切な料金プランの選択
- ✓ オートスケーリングの活用
- ✓ サーバーレスの活用
- ✓ ジョブのチューニング
- ✓ 処理対象データの削減
- ✓ 差分データ反映の最適化
- ✓ データクオリティ管理の導入

データパイプラインコストの最適化

課題	最適化
✗ デフォルトのままの料金プラン	✓ 適切な料金プランの選択
✗ 過剰な/過小なリソース	✓ オートスケーリングの活用
✗ アイドルリソース	✓ サーバーレスの活用
✗ ボトルネックのあるジョブ	✓ ジョブのチューニング
✗ 必要以上のデータの処理	✓ 処理対象データの削減
✗ 重い変換や不要な集合演算	✓ 差分データ反映の最適化
✗ 不正なデータへの無駄な処理	✓ データクオリティ管理の導入

適切な料金プランの選択

オンデマンド

長期コミットメントなしに
使った分だけの支払い

ステートフル・スパイク
のあるワークロード

Savings plans

1年間または3年間の
コミットメントにより
最大 **72%** のコスト削減

予測可能で安定した
ワークロード

スポット

余剰キャパシティにより
オンデマンドに比べて
最大 **90%** のコスト削減

ステートレスなワークロード

AWS Glue Flex Jobs

Standard
実行クラス

1分以内で起動
予測可能なレイテンシー

マイクロバッチや
低レイテンシワークロード

Flex
実行クラス

最大 **34%** **コスト削減**

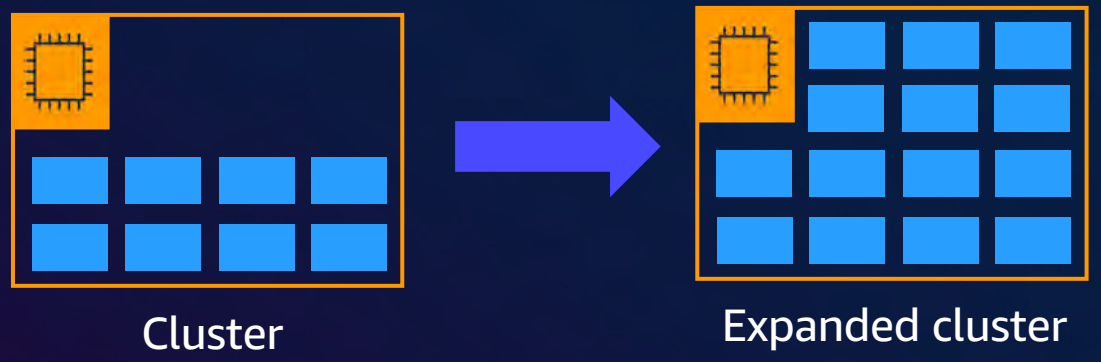
一度きりのデータロード

データパイプラインコストの最適化

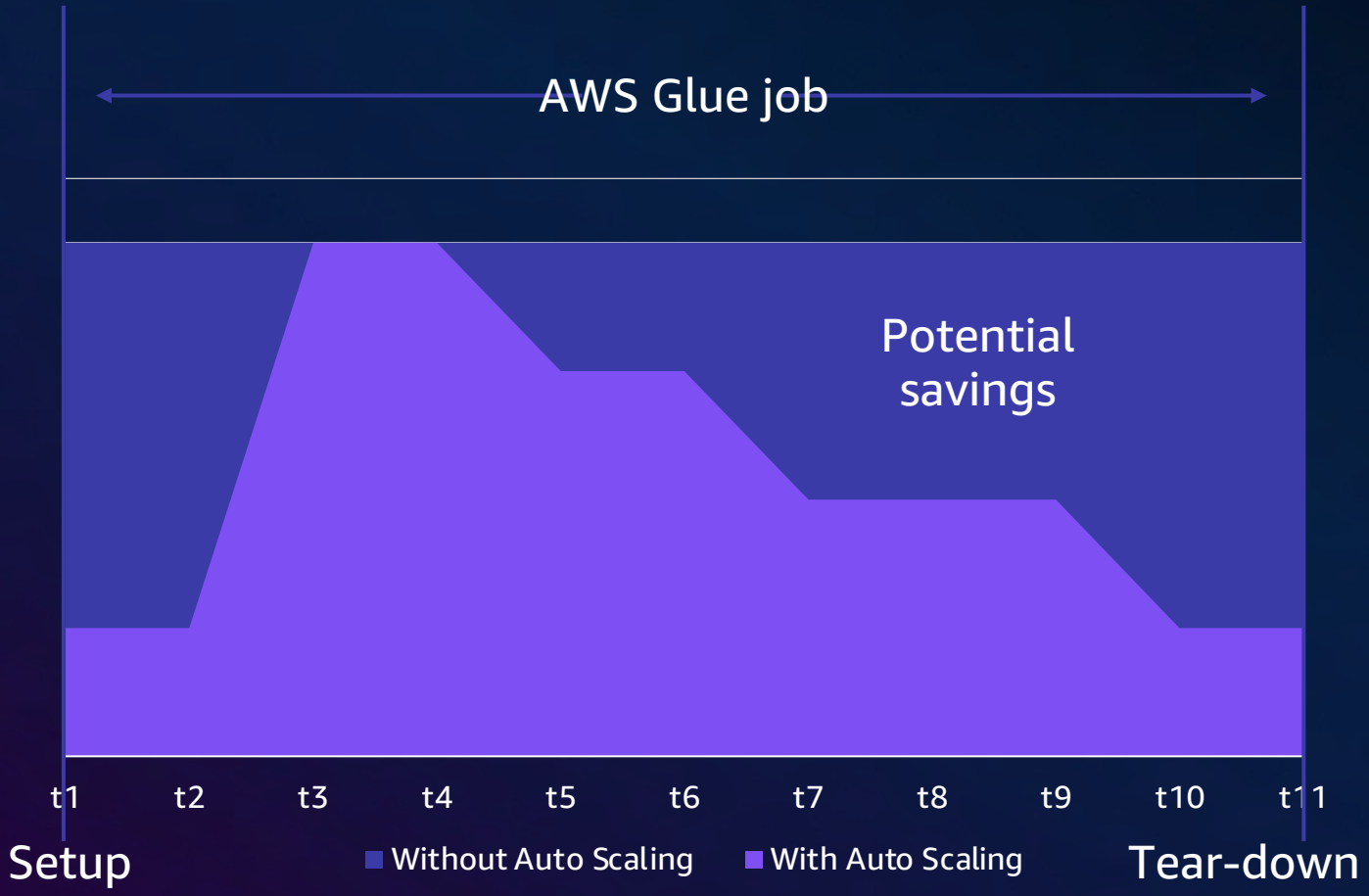
課題	最適化
✗ デフォルトのままの料金プラン	✓ 適切な料金プランの選択
✗ 過剰な/過小なリソース	✓ オートスケーリングの活用
✗ アイドルリソース	✓ サーバーレスの活用
✗ ボトルネックのあるジョブ	✓ ジョブのチューニング
✗ 必要以上のデータの処理	✓ 処理対象データの削減
✗ 重い変換や不要な集合演算	✓ 差分データ反映の最適化
✗ 不正なデータへの無駄な処理	✓ データクオリティ管理の導入

Amazon EMR Managed Scaling

- クラスタ内のインスタンス数をワークロードに合わせて自動調整
- スポットインスタンスと併用
- 20%–60% のコスト削減



AWS Glue Auto Scaling



Cost = f(compute)

- **コストを削減**
- **自動で動作**
- **シンプルなキャパシティ計画**

AWS Glue usage profiles

Glue リソースのコストをより柔軟に管理

- 開発者用、テスター用、分析者用など、ユーザーやロールに設定したプロファイルごとに Glue リソースのキャパシティを管理
- ワーカー数、ワーカータイプ、タイムアウト値のデフォルト値、最小値、最大値を設定
- Glue ジョブ、Glue Interactive Sessions に対応

データパイプラインコストの最適化

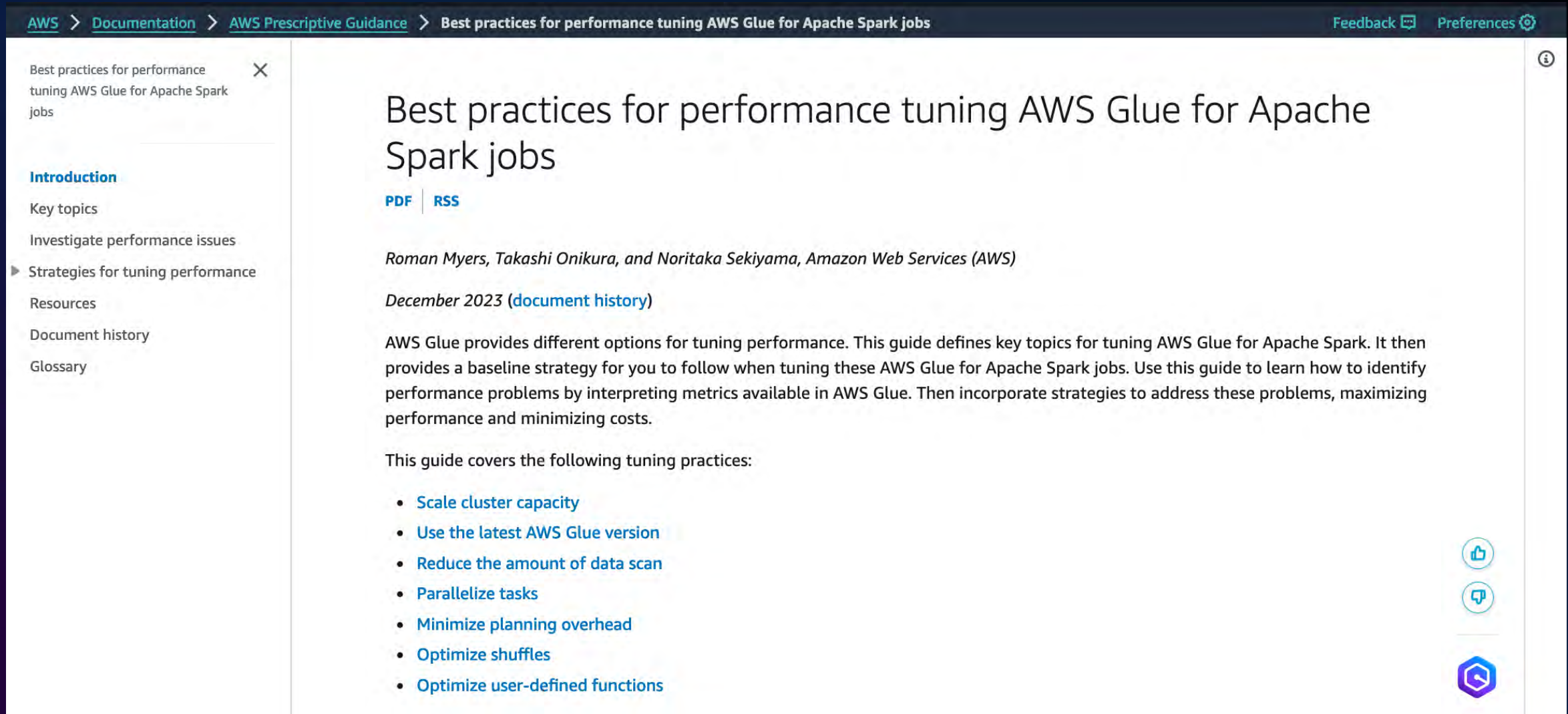
課題	最適化
✗ デフォルトのままの料金プラン	✓ 適切な料金プランの選択
✗ 過剰な/過小なリソース	✓ オートスケーリングの活用
✗ アイドルリソース	✓ サーバーレスの活用
✗ ボトルネックのあるジョブ	✓ ジョブのチューニング
✗ 必要以上のデータの処理	✓ 処理対象データの削減
✗ 重い変換や不要な集合演算	✓ 差分データ反映の最適化
✗ 不正なデータへの無駄な処理	✓ データクオリティ管理の導入

データパイプラインコストの最適化

課題	最適化
✗ デフォルトのままの料金プラン	✓ 適切な料金プランの選択
✗ 過剰な/過小なリソース	✓ オートスケーリングの活用
✗ アイドルリソース	✓ サーバーレスの活用
✗ ボトルネックのあるジョブ	✓ ジョブのチューニング
✗ 必要以上のデータの処理	✓ 処理対象データの削減
✗ 重い変換や不要な集合演算	✓ 差分データ反映の最適化
✗ 不正なデータへの無駄な処理	✓ データクオリティ管理の導入

Best practices for performance tuning AWS Glue for Apache Spark jobs

https://docs.aws.amazon.com/prescriptive-guidance/latest/tuning-aws-glue-for-apache-spark/introduction.html

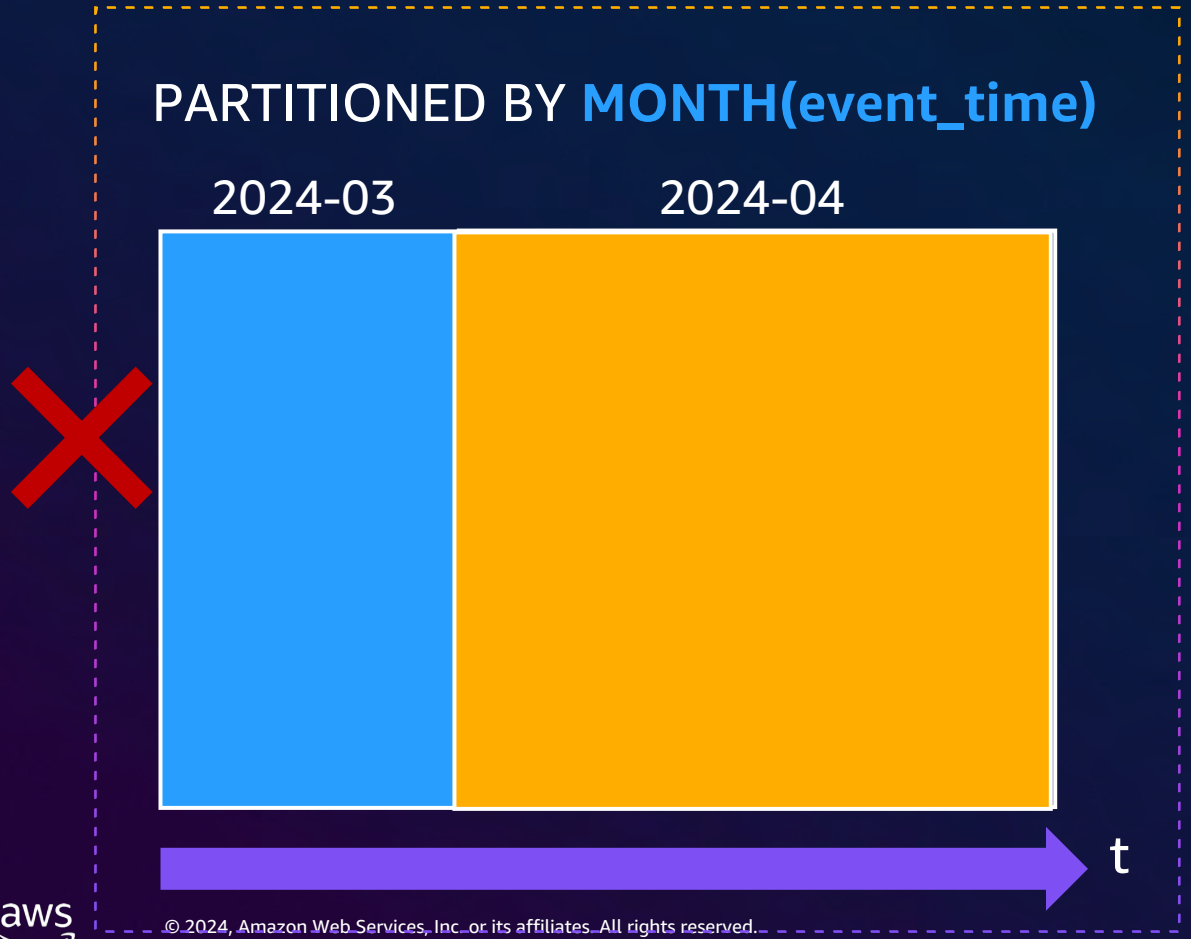


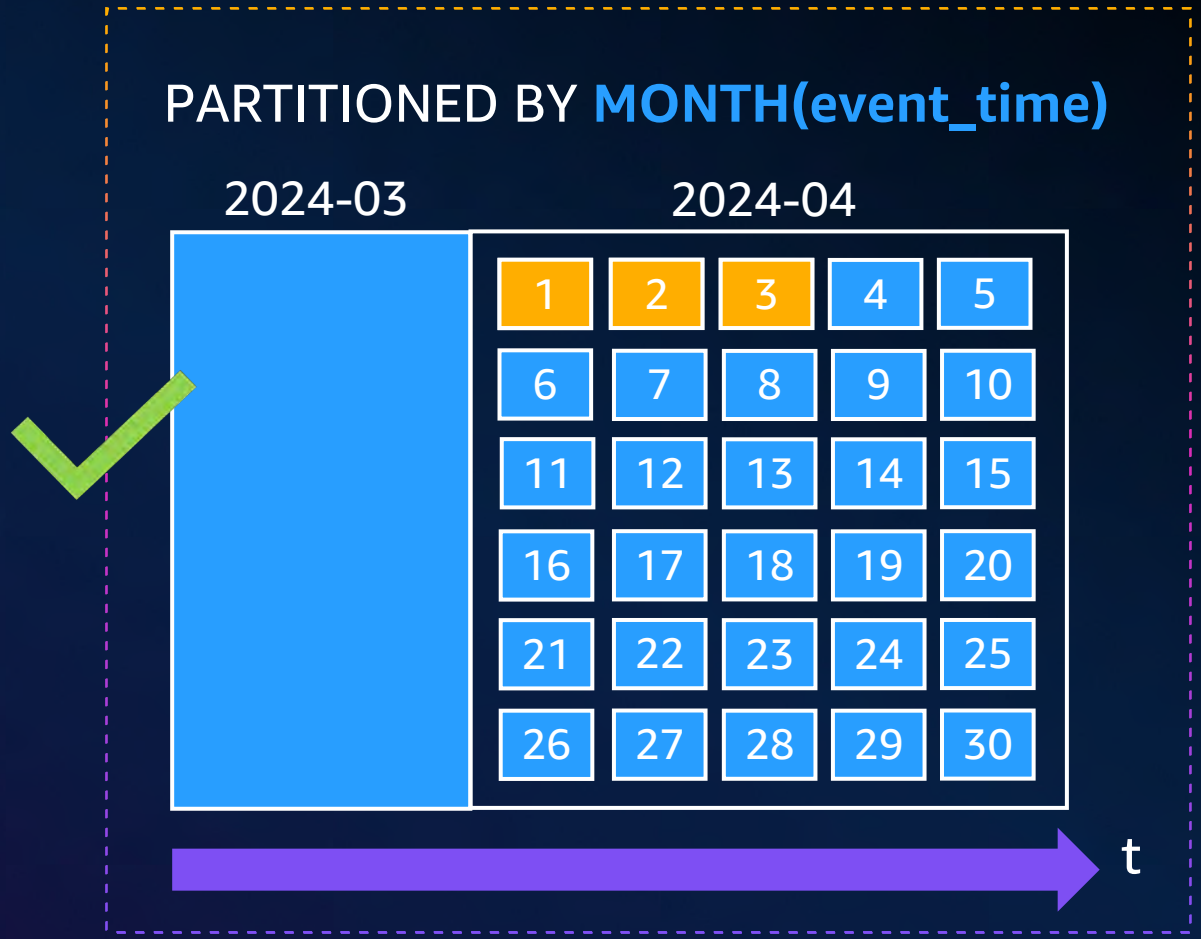
データパイプラインコストの最適化

課題	最適化
✗ デフォルトのままの料金プラン	✓ 適切な料金プランの選択
✗ 過剰な/過小なリソース	✓ オートスケーリングの活用
✗ アイドルリソース	✓ サーバーレスの活用
✗ ボトルネックのあるジョブ	✓ ジョブのチューニング
✗ 必要以上のデータの処理	✓ 処理対象データの削減
✗ 重い変換や不要な集合演算	✓ 差分データ反映の最適化
✗ 不正なデータへの無駄な処理	✓ データクオリティ管理の導入

処理対象データの削減

- データの洗い替えによる更新対象範囲を必要最小限に



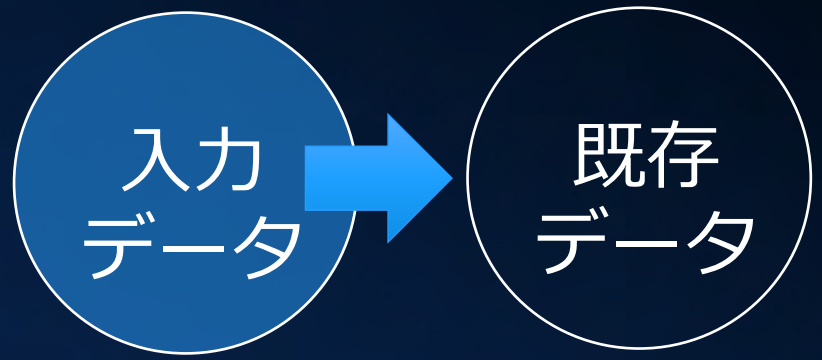


データパイプラインコストの最適化

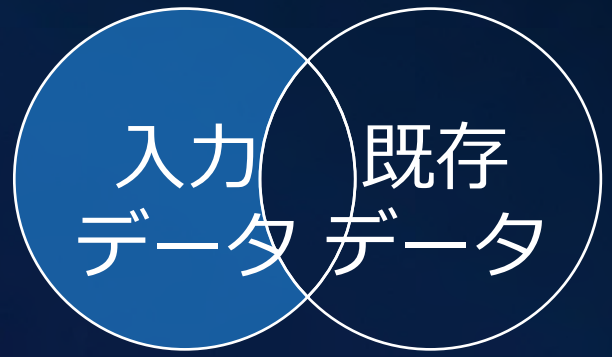
課題	最適化
✗ デフォルトのままの料金プラン	✓ 適切な料金プランの選択
✗ 過剰な/過小なリソース	✓ オートスケーリングの活用
✗ アイドルリソース	✓ サーバーレスの活用
✗ ボトルネックのあるジョブ	✓ ジョブのチューニング
✗ 必要以上のデータの処理	✓ 処理対象データの削減
✗ 重い変換や不要な集合演算	✓ 差分データ反映の最適化
✗ 不正なデータへの無駄な処理	✓ データクオリティ管理の導入

差分データ反映の最適化：旧来の手法

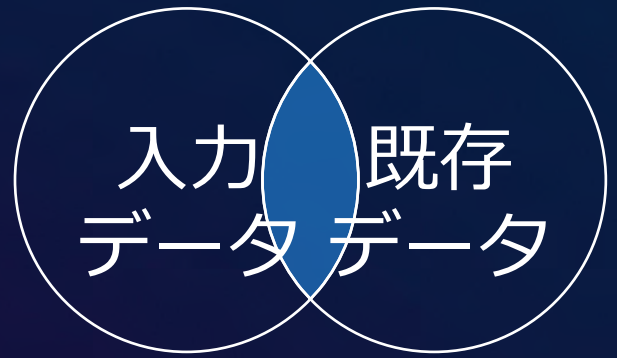
0. 初期状態



1. INSERT

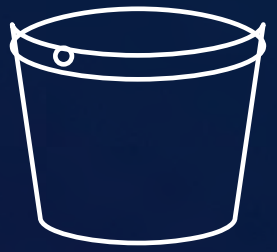


2. UPDATE



差分データ反映の最適化：レコード単位の更新

データレイク



Apache Iceberg

Apache Hudi

Delta Lake

データウェアハウス

Amazon Redshift

差分データ反映の最適化：MERGE INTO クエリ

Apache Spark / Apache Iceberg

```
MERGE INTO {catalog_name}.{database_name}.{table_name} AS t
USING (SELECT * FROM tmp_{table_name}_updates) AS u
ON t.product_id = u.product_id
WHEN MATCHED THEN UPDATE SET
     t.product_name = u.product_name,
     t.price = u.price,
     t.category = u.category,
     t.updated_at = u.updated_at
WHEN NOT MATCHED THEN INSERT *
```

UPDATE

INSERT

データパイプラインコストの最適化

課題	最適化
✗ デフォルトのままの料金プラン	✓ 適切な料金プランの選択
✗ 過剰な/過小なリソース	✓ オートスケーリングの活用
✗ アイドルリソース	✓ サーバーレスの活用
✗ ボトルネックのあるジョブ	✓ ジョブのチューニング
✗ 必要以上のデータの処理	✓ 処理対象データの削減
✗ 重い変換や不要な集合演算	✓ 差分データ反映の最適化
✗ 不正なデータへの無駄な処理	✓ データクオリティ管理の導入

不正なデータへの無駄な処理

データクオリティ管理の導入

AWS Glue Data Quality

データレイクとパイプライン全体で
高品質なデータを保つ仕組み

- サーバーレス、スケーラブル、高速
- データ格納前（伝送路上）、データ格納後のデータ品質を検査
- 機械学習ベースで、データ品質の隠れた問題や以上を検知
- 様々なスキルセットのユーザーから利用可能

データパイプラインコストの最適化

課題

- ✗ デフォルトのままの料金プラン
- ✗ 過剰な/過小なリソース
- ✗ アイドルリソース
- ✗ ボトルネックのあるジョブ
- ✗ 必要以上のデータの処理
- ✗ 重い変換や不要な集合演算
- ✗ 不正なデータへの無駄な処理

最適化

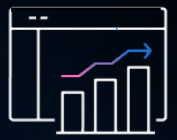
- ✓ 適切な料金プランの選択
- ✓ オートスケーリングの活用
- ✓ サーバーレスの活用
- ✓ ジョブのチューニング
- ✓ 処理対象データの削減
- ✓ 差分データ反映の最適化
- ✓ データクオリティ管理の導入

クエリコストの可視化

ストレージ

データ転送

データ
パイプライン



クエリ

クエリコストの最適化

課題

- ✗ 不要なデータスキャン
- ✗ 非効率的なデータ形式
- ✗ クエリ最適化の欠如
- ✗ 同一クエリの繰り返し実行
- ✗ 不要なデータ移動

最適化

- ✓ データレイアウトの最適化
- ✓ データ形式や圧縮形式の最適化
- ✓ クエリのチューニング
- ✓ クエリ結果の再利用
- ✓ フェデレーテッドクエリの活用

Athena 料金プラン

Pay per query

$5/TB

- 簡単に利用可能
- 実行したクエリのみに対して課金

Pay for compute

$0.30/DPU-Hour

- 事前にキャパシティを予約
- 予約されたキャパシティに対して課金

1DPU: 4 vCPU & 16 GB RAM

ⓘ 両方の料金プランを同時に、同一アカウント内で併用可能

Athena キャパシティ予約

関連付けたワークグループに対して
コンピュート能力を確保・利用

- スケールと同時実行数を管理
- 高優先度のクエリを予約されたコンピュートで実行
- スキャン量ベースではなく、フラットレートで予測可能な料金

キャパシティ予約の作成

DPU 数を設定

ワークグループを選択

キャパシティを割り当て

クエリコストの最適化

課題	最適化
✗ 不要なデータスキャン	✓ データレイアウトの最適化
✗ 非効率的なデータ形式	✓ データ形式や圧縮形式の最適化
✗ クエリ最適化の欠如	✓ クエリのチューニング
✗ 同一クエリの繰り返し実行	✓ クエリ結果の再利用
✗ 不要なデータ移動	✓ フェデレーテッドクエリの活用

クエリコストの最適化

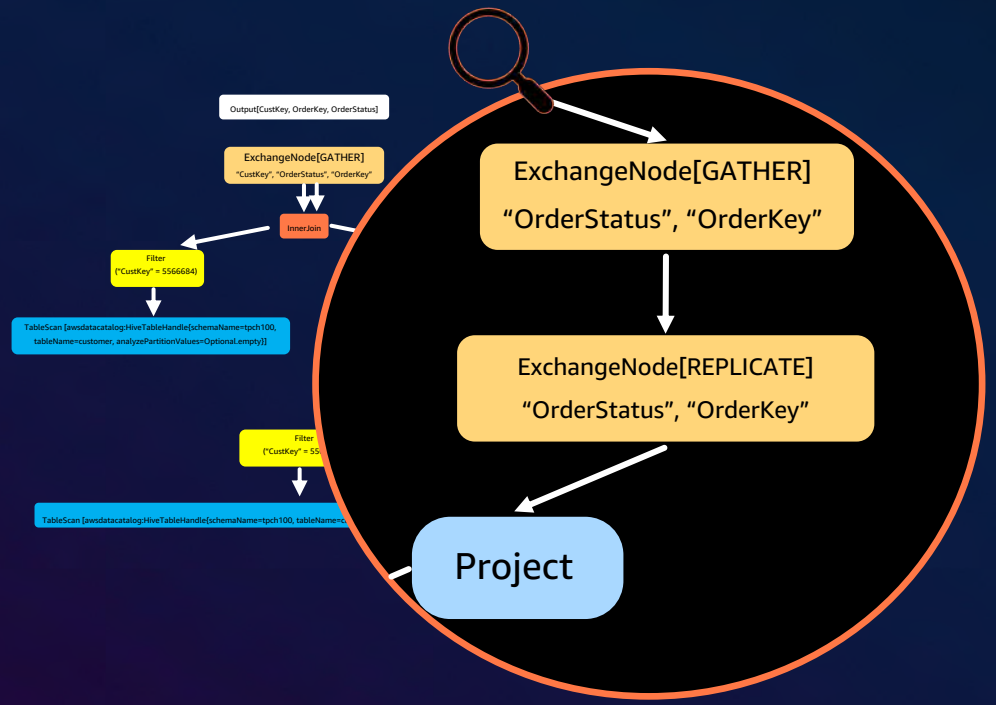
課題	最適化
✗ 不要なデータスキャン	✓ データレイアウトの最適化
✗ 非効率的なデータ形式	✓ データ形式や圧縮形式の最適化
✗ クエリ最適化の欠如	✓ クエリのチューニング
✗ 同一クエリの繰り返し実行	✓ クエリ結果の再利用
✗ 不要なデータ移動	✓ フェデレーテッドクエリの活用

クエリコストの最適化

課題	最適化
✗ 不要なデータスキャン	✓ データレイアウトの最適化
✗ 非効率的なデータ形式	✓ データ形式や圧縮形式の最適化
✗ クエリ最適化の欠如	✓ クエリのチューニング
✗ 同一クエリの繰り返し実行	✓ クエリ結果の再利用
✗ 不要なデータ移動	✓ フェデレーテッドクエリの活用

Athena クエリプラン

EXPLAIN [(option [, ...])] statement



EXPLAIN ANALYZE [(option [, ...])] statement

```
Fragment 1
     CPU: 24.60ms, Input: 10 rows (1.48kB); per task: std.dev.: 0
     Output layout: [date, time, location, bytes, requestip, meth
     - Limit[10] => [[date, time, location, bytes, requestip, met
             CPU: 1.00ms (0.03%), Output: 10 rows (1.48kB)
             Input avg.: 10.00 rows, Input std.dev.: 0.00%
         - LocalExchange[SINGLE] () => [[date, time, location, by
                 CPU: 0.00ns (0.00%), Output: 10 rows (1.48kB)
                 Input avg.: 0.63 rows, Input std.dev.: 387.30%
             - RemoteSource[2] => [[date, time, location, bytes,
                     CPU: 1.00ms (0.03%), Output: 10 rows (1.48kB
                     Input avg.: 0.63 rows, Input std.dev.: 387.3
```

JSON、テキスト、ヴィジュアルによるクエリプラン

JSON またはテキストによるクエリ詳細

Athena コストベース オプティマイザ

クエリプランを最適化しデータクエリをブースト

- AWS Glue Data Catalog のテーブル・カラム統計情報を活用
- クエリ高速化のために最も効率の良いクエリプランを選択
- デフォルトで有効

Top 10 Performance Tuning Tips for Amazon Athena

https://aws.amazon.com/blogs/big-data/top-10-performance-tuning-tips-for-amazon-athena/

AWS Big Data Blog

Top 10 Performance Tuning Tips for Amazon Athena

by Mert Hocanin and Pathik Shah | on 24 MAR 2017 | in Amazon Athena, AWS Big Data | Permalink | Comments | Share

February 2024: This post was reviewed and updated to reflect changes in Amazon Athena engine version 3, including cost-based optimization and query result reuse.

Amazon Athena is an interactive analytics service built on open source frameworks that make it straightforward to analyze data stored using open table and file formats in Amazon Simple Storage Service (Amazon S3) using standard SQL. Athena is serverless, so there is no infrastructure to manage, and you pay only for the queries that you run. Athena is easy to use, simply point to your data in Amazon S3, define the schema, and start querying using standard SQL.

In this post, we review the top 10 tips that can improve query performance. We focus on aspects related to storing data in Amazon S3 and tuning specific to queries.

Storage

This section discusses how to structure your data so that you can get the most out of Athena. You can apply the same practices to Amazon EMR data processing applications such as Spark, Trino, Presto, and Hive when your data is stored in Amazon S3. We discuss the following best practices:

1. Partition your data
2. Bucket your data
3. Use compression
4. Optimize file size
5. Use columnar file formats

1. Partition your data

Partitioning divides your table into parts and keeps the related data together based on column values such as date, country, and region. Partitions act as virtual columns. You define them at table creation, and they can help reduce the amount of data scanned per query, thereby improving performance. You can restrict the amount of data scanned by a

Query tuning

The Athena SQL engine is built on the open source distributed query engines Trino and Presto. Understanding how it works provides insight into how you can optimize queries when running them. This section details the following best practices:

1. Optimize `ORDER BY`
2. Optimize joins
3. Optimize `GROUP BY`
4. Use approximate functions
5. Only include the columns that you need

6. Optimize ORDER BY

The ORDER BY clause returns the results of a query in sort order. Athena uses distributed sort to run the sort operation in parallel on multiple nodes. If you're using the `ORDER BY` clause to look at the top or bottom N values, use a `LIMIT` clause to reduce the cost of the sort, which results in a faster query runtime.

For example, the following table summarizes the runtime for a dataset with a 7.25 GB table, uncompressed in text format, with approximately 60 million rows.

Query	Runtime

`SELECT *`
`FROM lineitem`
`ORDER BY l_shipdate`
	274 seconds

`SELECT *`
`FROM lineitem`
`ORDER BY l_shipdate`
`LIMIT 10000`
	4.6 seconds

クエリコストの最適化

課題	最適化
✗ 不要なデータスキャン	✓ データレイアウトの最適化
✗ 非効率的なデータ形式	✓ データ形式や圧縮形式の最適化
✗ クエリ最適化の欠如	✓ クエリのチューニング
✗ 同一クエリの繰り返し実行	✓ クエリ結果の再利用
✗ 不要なデータ移動	✓ フェデレーテッドクエリの活用

Athena クエリ結果の再利用

同一クエリの処理時間の改善

データスキャンコストの削減

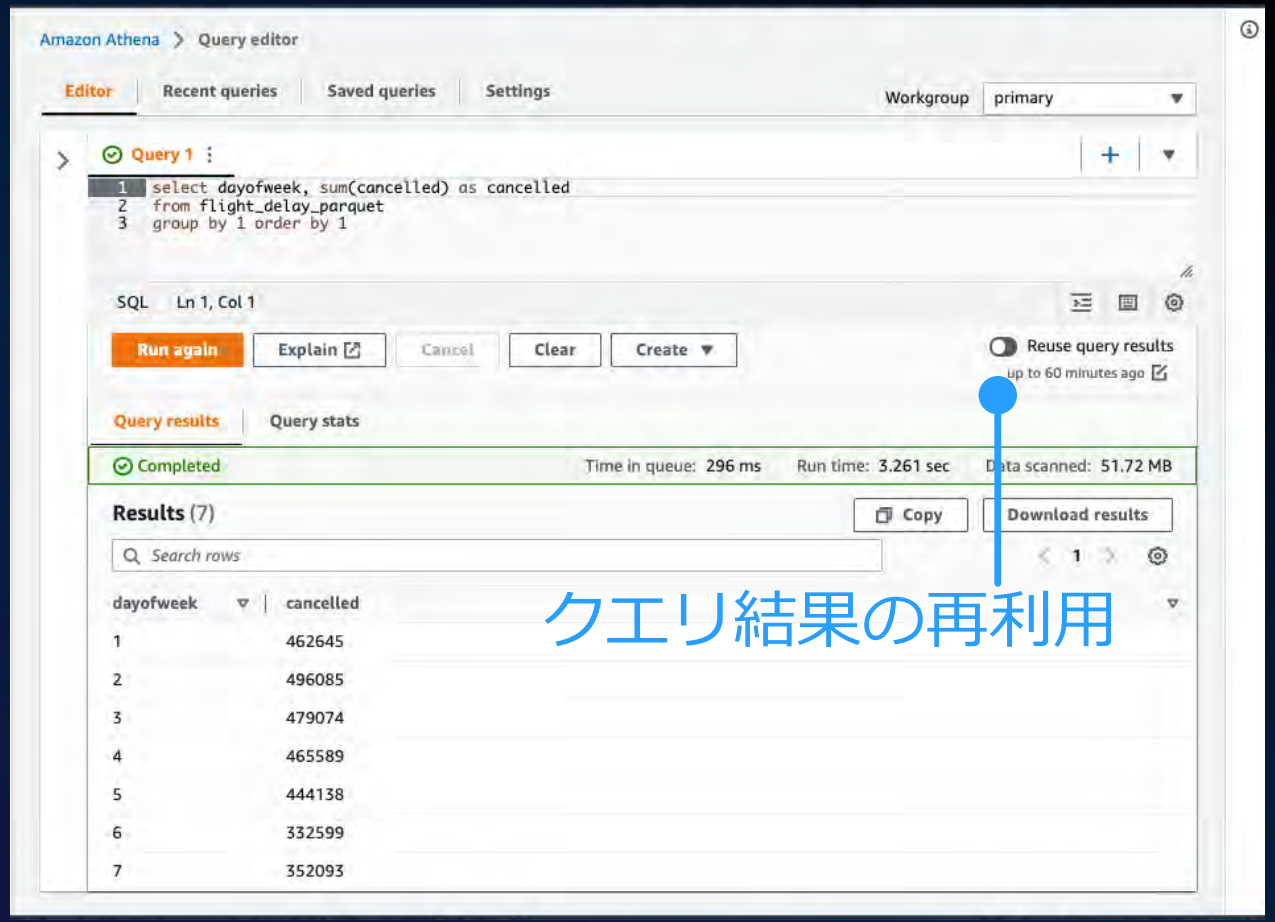
ワークグループ単位の結果共有

5x

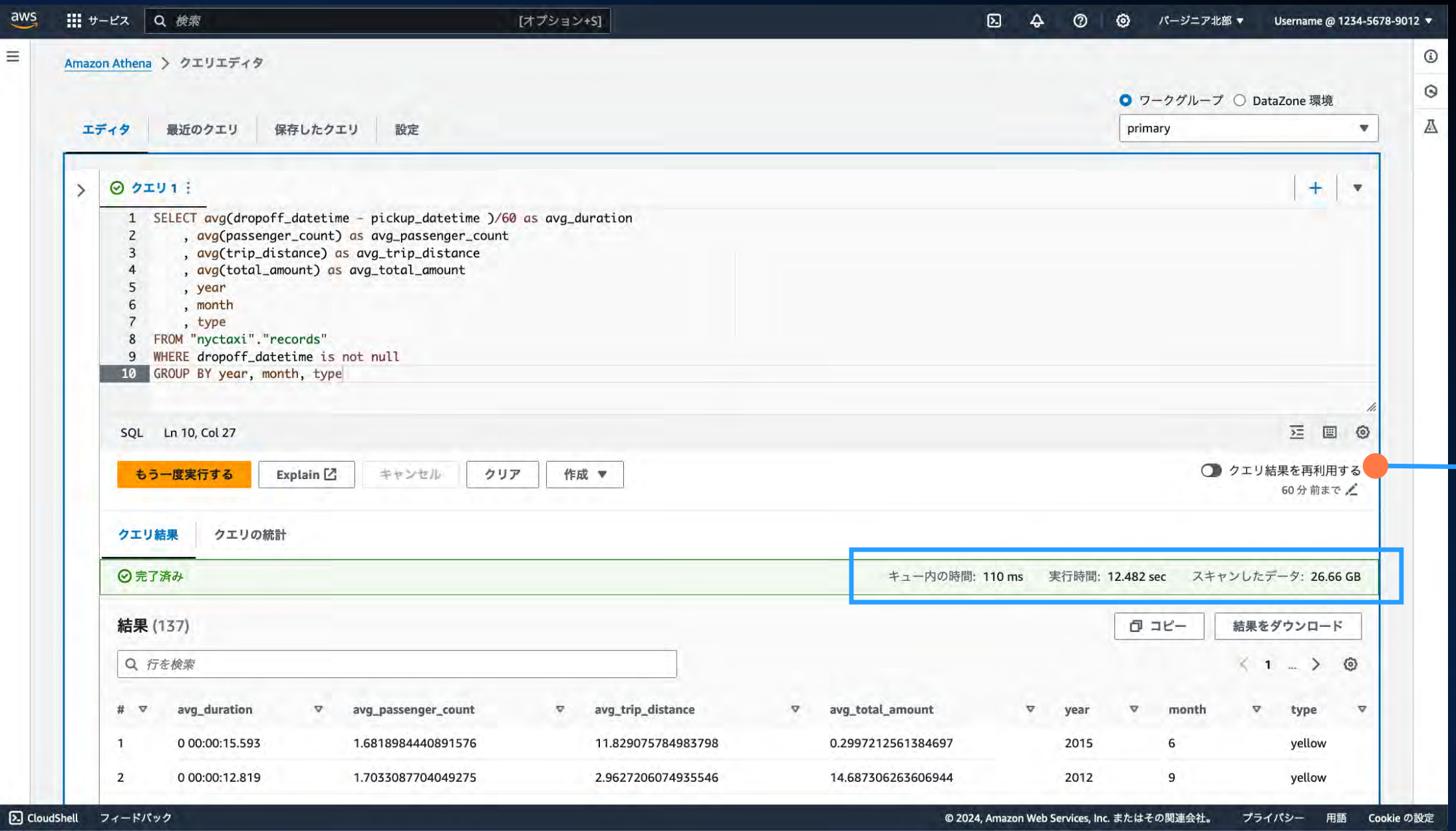
繰り返し実行されるクエリを最大5倍高速に実行

$0

データスキャンなしで無料

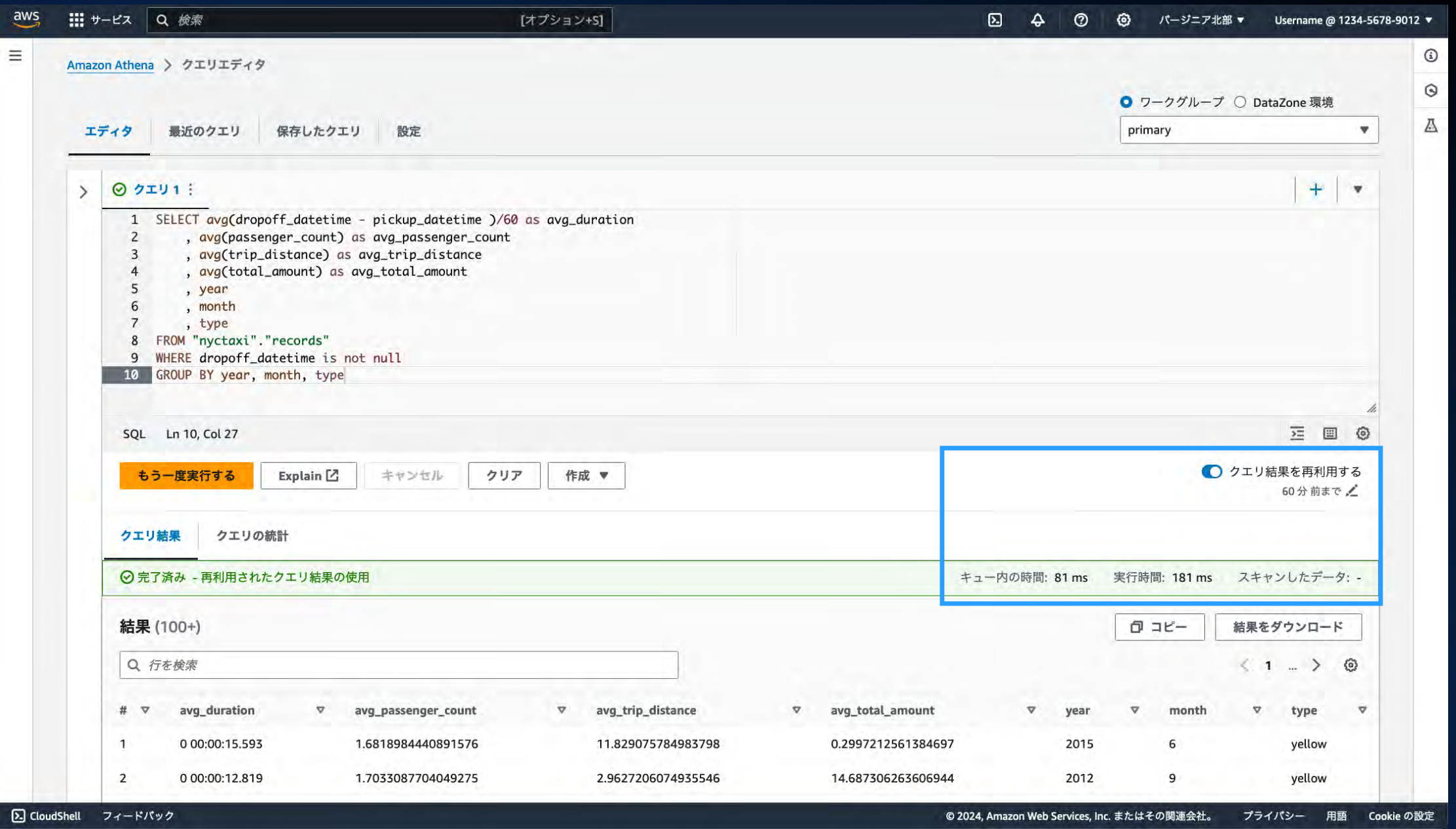


Athena クエリ結果の再利用

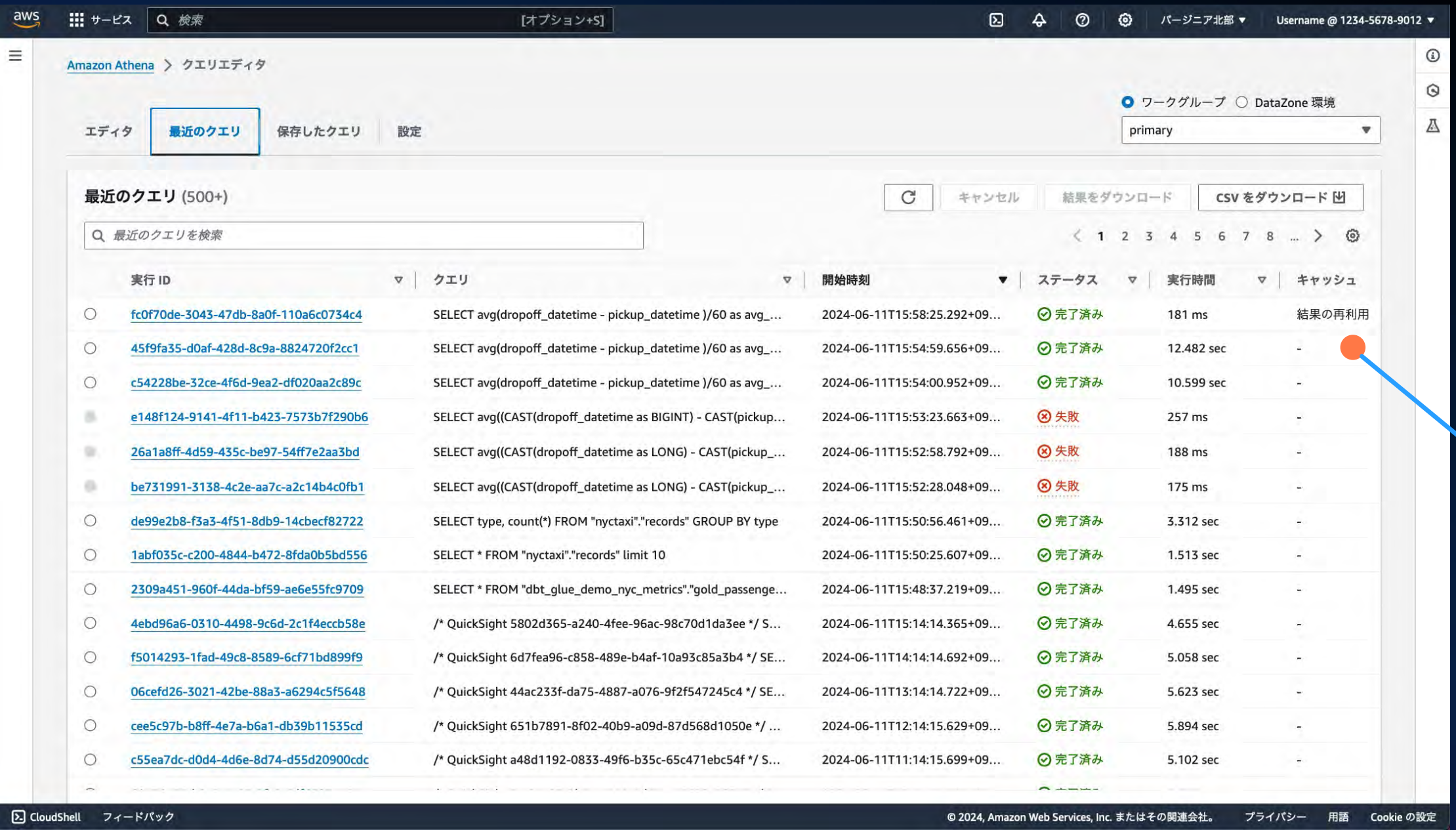


クエリ結果
再利用
の有効化

Athena クエリ結果の再利用



Athena クエリ結果の再利用



キャッシュ状態

クエリコストの最適化

課題	最適化
✗ 不要なデータスキャン	✓ データレイアウトの最適化
✗ 非効率的なデータ形式	✓ データ形式や圧縮形式の最適化
✗ クエリ最適化の欠如	✓ クエリのチューニング
✗ 同一クエリの繰り返し実行	✓ クエリ結果の再利用
✗ 不要なデータ移動	✓ フェデレーテッドクエリの活用

クエリコストの最適化

課題

- ✗ 不要なデータスキャン
- ✗ 非効率的なデータ形式
- ✗ クエリ最適化の欠如
- ✗ 同一クエリの繰り返し実行
- ✗ 不要なデータ移動

最適化

- ✓ データレイアウトの最適化
- ✓ データ形式や圧縮形式の最適化
- ✓ クエリのチューニング
- ✓ クエリ結果の再利用
- ✓ フェデレーテッドクエリの活用

おわりに

おわりに

ストレージ

データ転送

データ
パイプライン

クエリ

コスト最適化のジャーニー

Thank you!

関山宜孝

sekiyama@amazon.com