AWS-51

アーキテクチャ道場！

内海 英一郎
技術統括本部
チーフテクノロジスト
アマゾン ウェブ サービス ジャパン合同会社

奥野 友哉
技術統括本部 インターネットメディアソリューション本部
ソリューションアーキテクト
アマゾン ウェブ サービス ジャパン合同会社

山﨑 翔太
技術統括本部 インターネットメディアソリューション本部 部長
シニアソリューションアーキテクト
アマゾン ウェブ サービス ジャパン合同会社

お題 1

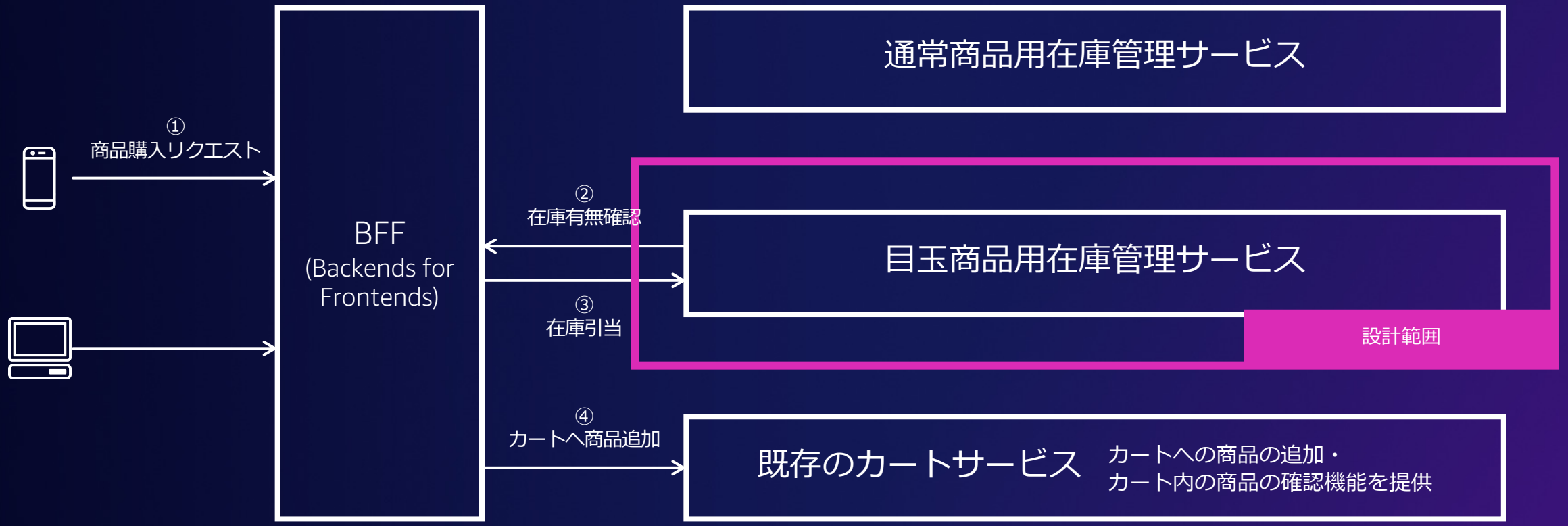
オンラインセール

アパレル EC サイトを運営する X 社のオンラインセールは人気ブランドのスニーカーが数量限定で出品されることで有名です。EC サイトへのアクセスは目玉商品の販売開始と同時にスパイクすることが分かっており、在庫管理サービスのスケーラビリティが課題になっています。

そこで X 社は新たに目玉商品専用の在庫管理サービスを構築して大量のアクセスを捌こうと考えました。あなたはこのサービスをどのように設計しますか？

お題 1
オンラインセール

⚠ 商品を抽選で販売したり在庫数を超える注文を受け付けてはいけません



💡 注文プロセスは商品がカートに追加された時点で完了とみなし、その後の
チェックアウト／カートからの商品の削除を考慮する必要はありません

自己紹介: 奥野 友哉 (おくの ともや)

会　　社：　アマゾン ウェブ サービス ジャパン 株式会社

所　　属：　技術統括本部

　　インターネットメディアソリューション部

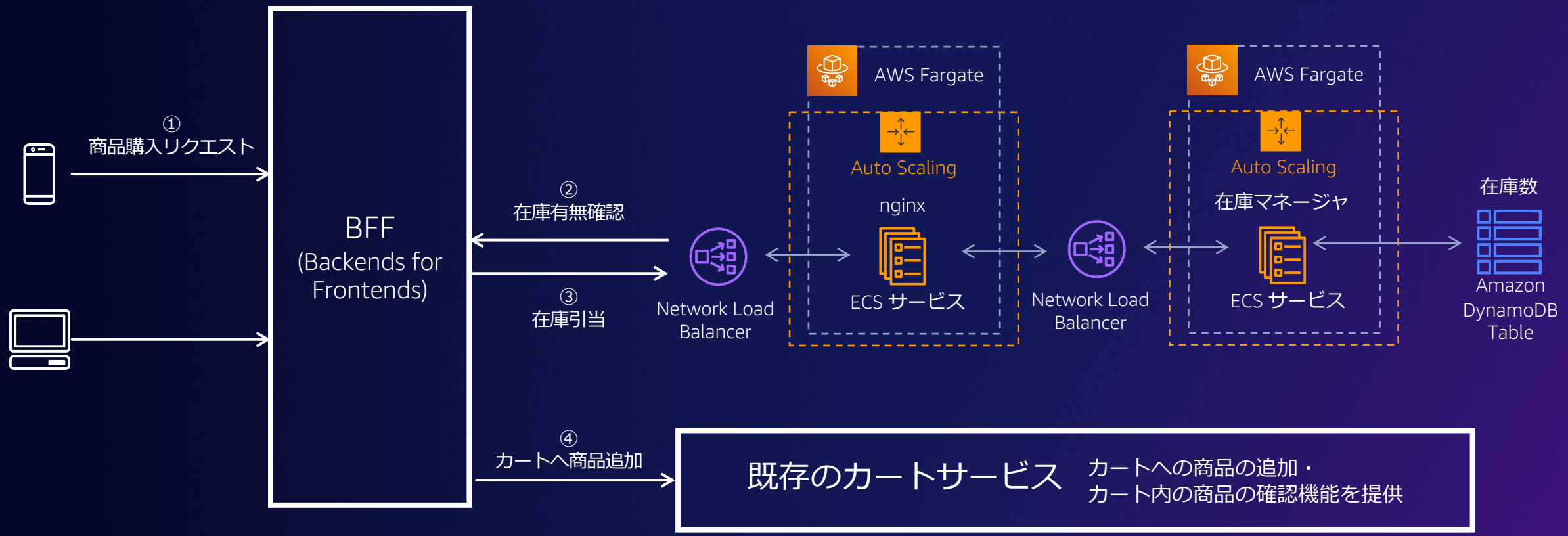
役　　職：　ソリューション アーキテクト

経　　歴：

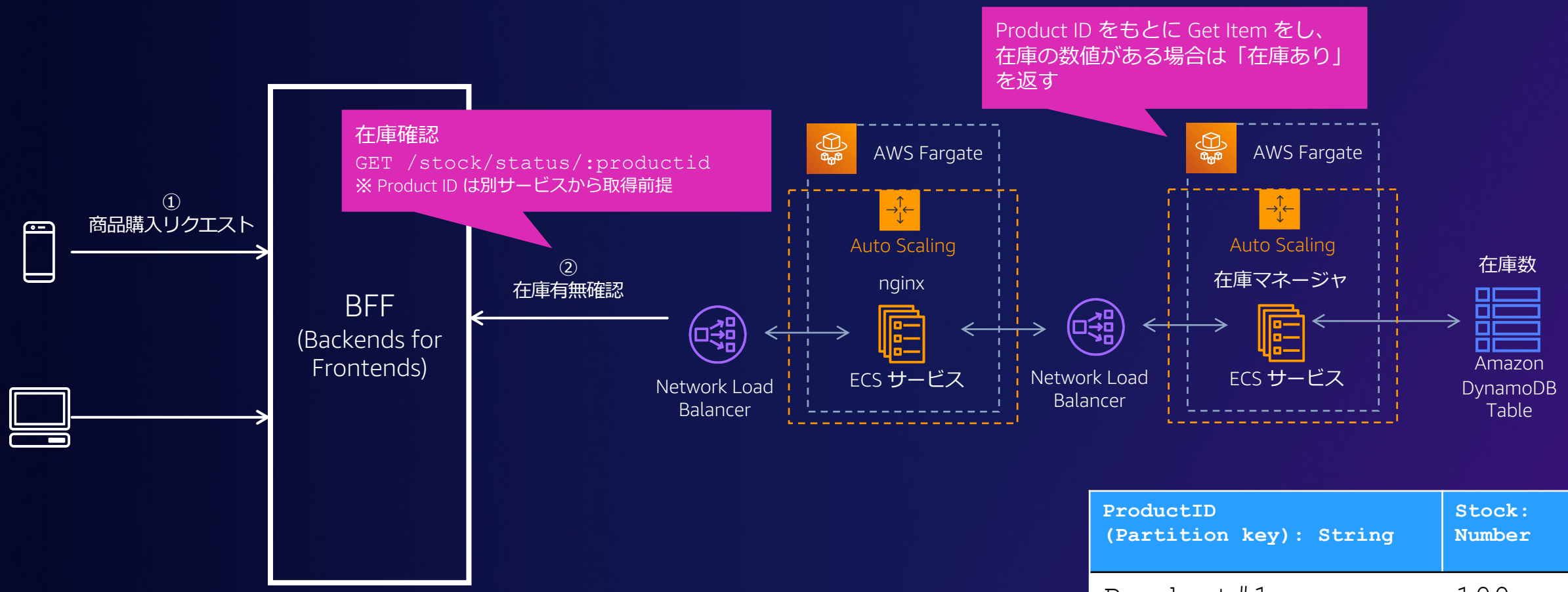
- 学生時代: 燃焼限界の研究 (CFD, データ分析), Ph.D.
- 2018 年: AWS へ新卒入社
- 2019-2021 年: 自動車、組立製造業界のお客様
- 2022 年: インターネットメディア業界のお客様

好きな AWS サービス: AWS CDK, AWS Ground Station

シンプルな設計

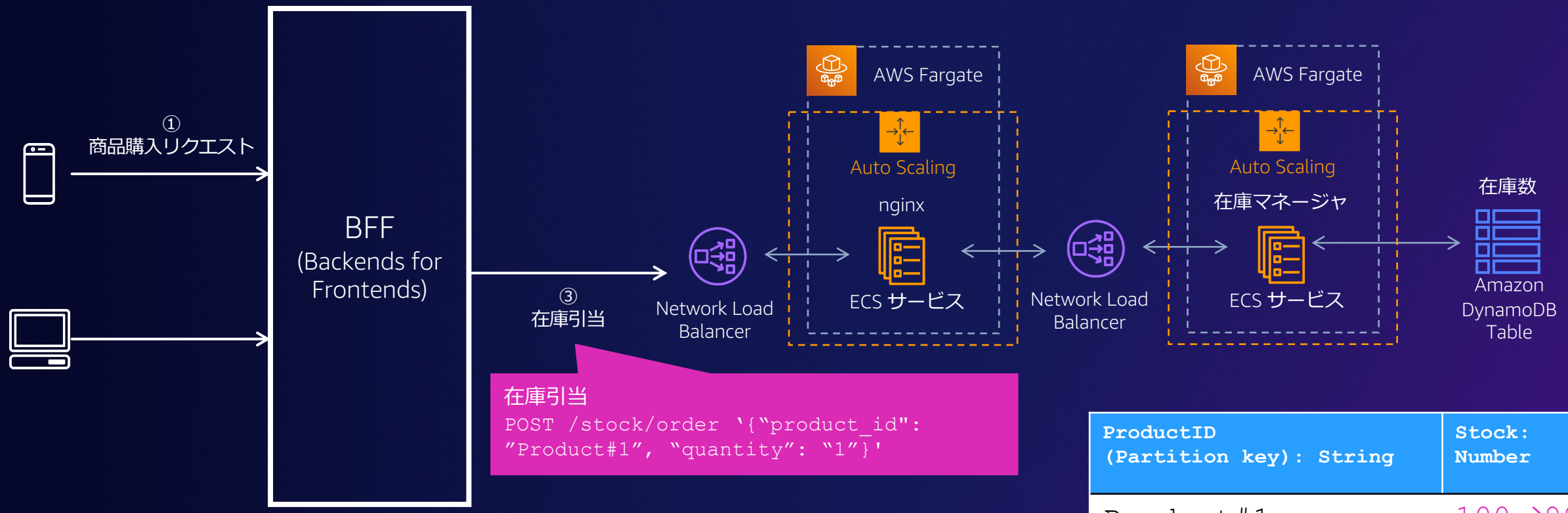


シンプルな設計



ProductID (Partition key): String	Stock: Number
Product#1	100
Product#2	98

シンプルな設計



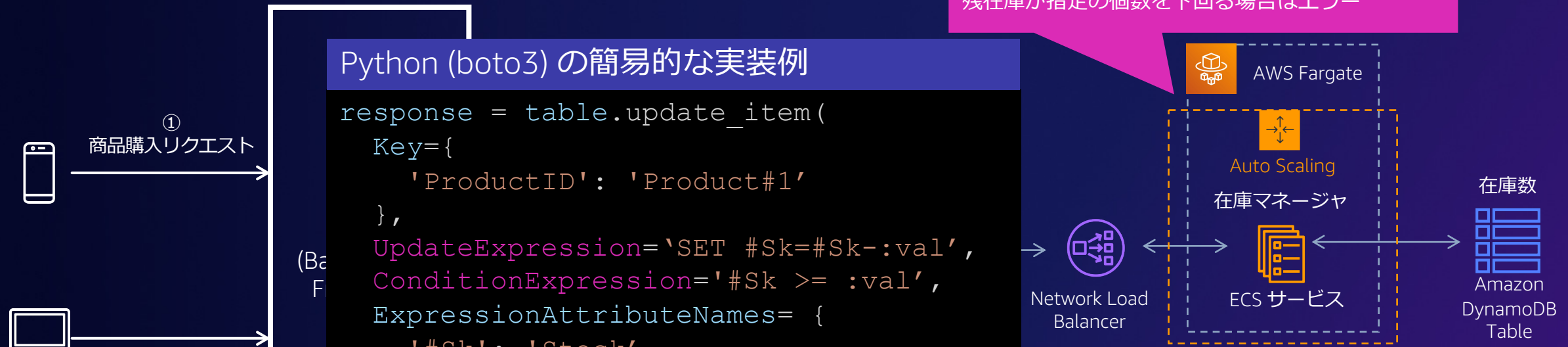
在庫引当

```
POST /stock/order '{"product_id":
"Product#1", "quantity": "1"}'
```

ProductID (Partition key): String	Stock: Number
Product#1	100→99
Product#2	98

シンプルな設計

在庫を指定の個数減らす
残在庫が指定の個数を下回る場合はエラー

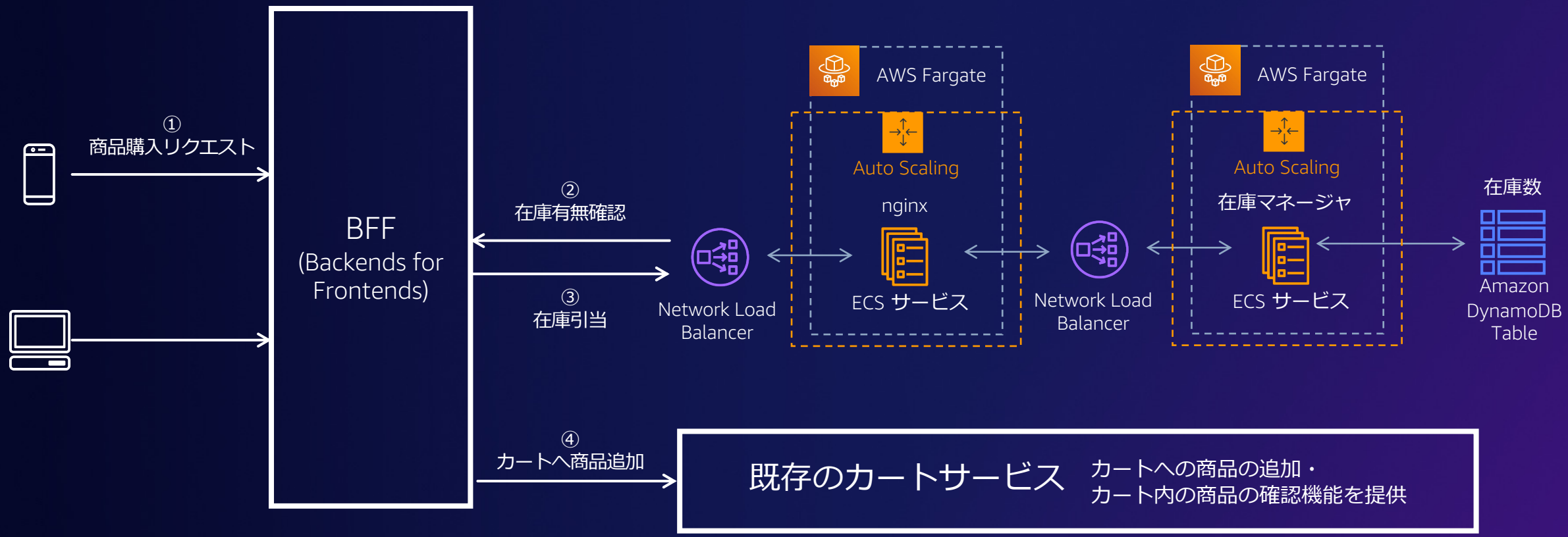


Python (boto3) の簡易的な実装例

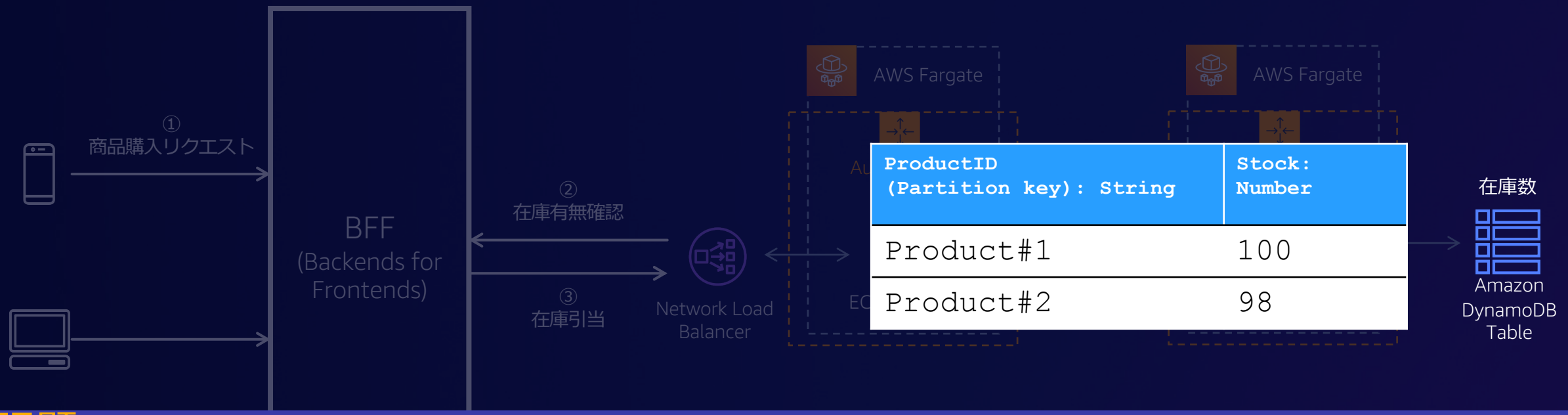
```
response = table.update_item(
  Key={
    'ProductID': 'Product#1'
  },
  UpdateExpression='SET #Sk=#Sk-:val',
  ConditionExpression='#Sk >= :val',
  ExpressionAttributeNames= {
    '#Sk': 'Stock'
  },
  ExpressionAttributeValues={
    ':val': -1
  }
)
```

ProductID (Partition key): String	Stock: Number
Product#1	100→99
Product#2	98

シンプルな設計



シンプルな設計



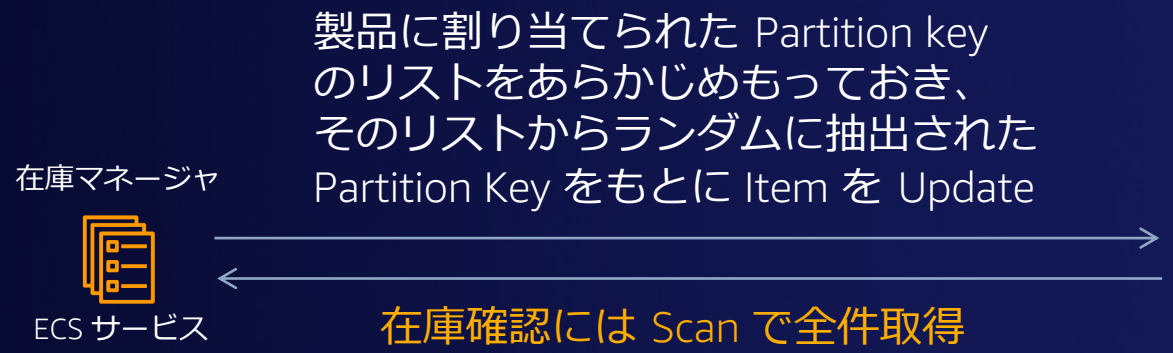
課題

特定の Item に Read/Update が集中するため、Partition ごとに存在する 3,000 RCU および 1,000 WCU の上限に当たる可能性がある

=> 書き込み/読み込みをどのようにスケールさせるかを検討

書き込みのスケール方法

シャーディングによる書き込みのスケール



Primary key	Attributes
PartitionID (Partition key): String	Stock: Number
Product#1_#0	50
Product#1_#1	50

Partition 数が増加すると

- Scan を利用しているためコスト効率が悪い
- 在庫有無を調べるときにアプリケーション側の計算量が増えてしまう

読み込みのスケール方法

在庫マネージャ

ECS サービス

在庫がある Partition ID の List を別に用意
Attribute が存在していれば在庫あり

在庫テーブル

Primary key	Attributes
PartitionID (Partition key): String	Stock: Number
Product#1_#0	50
Product#1_#1	50

在庫状態テーブル

Primary key	Attributes
ProductID (Partition key): String	PartitionIDList: List
Product#1	[{ "S" : "#0" }, { "S" : "#1" }]

Scan は不要になり、アプリケーション側の計算量は減ったが、
特定の Item に対して読み込みが集中する問題は残る

読み込みのスケール方法

データ複製による読み込みのスケール

在庫マネージャ

ECS サービス

Amazon DynamoDB Accelerator (DAX)

在庫状態テーブルの複製

在庫テーブル

Primary key	Attributes
PartitionID (Partition key): String	Stock: Number
Product#1_#0	50
Product#1_#1	50

在庫状態テーブル

Primary key	Attributes
ProductID (Partition key): String	PartitionIDList: List
Product#1	[{ "S" : "#0" }, { "S" : "#1" }]

- Partition ID List の Update におけるレイテンシやスループット要件はシビアでないため、最新の情報を保持できる Write-through cache として DAX を利用
- Table の Update 時以外にも元のテーブルと同期をとるため、TTL を設定

書き込み/読み込みをスケールした構成

Primary key	Attributes
PartitionID (Partition key): String	Stock: Number
Product#1_#0	50
Product#1_#1	50

Primary key	Attributes
ProductID (Partition key): String	PartitionIDList: List
Product#1	[{ "S" : "#0" }, { "S" : "#1" }]

書き込み/読み込みをスケールした構成

Primary key	Attributes
PartitionID (Partition key): String	Stock: Number
	50
	50

在庫マネージャ

ECS サービス

Python (boto3) の簡易的な実装例

```
table1 =
amazondax.AmazonDaxClient.resource(endpoint_url=URL).Table(
'stockstatetable')
response = table1.query(
  KeyConditionExpression=Key('ProductID').eq('Product#1'))

partition_list = response['Items'][0]['PartitionIDList']
partititon = random.choice(partition_list)
quantity = 1
table2 = dynamodb.Table('stocktable')
response = table2.update_item(
    Key={'PartitionID': partition},
    UpdateExpression='SET #Sk=#Sk-:val',
    ConditionExpression='#Sk >= :val',
    ExpressionAttributeNames= {'#Sk': 'Stock'},
    ExpressionAttributeValues={':val': quantity})
```

butes

itionIDList:

{ "S" : "#0" }, {
" : "#1" }]

書き込み/読み込みをスケールした構成

在庫テーブル

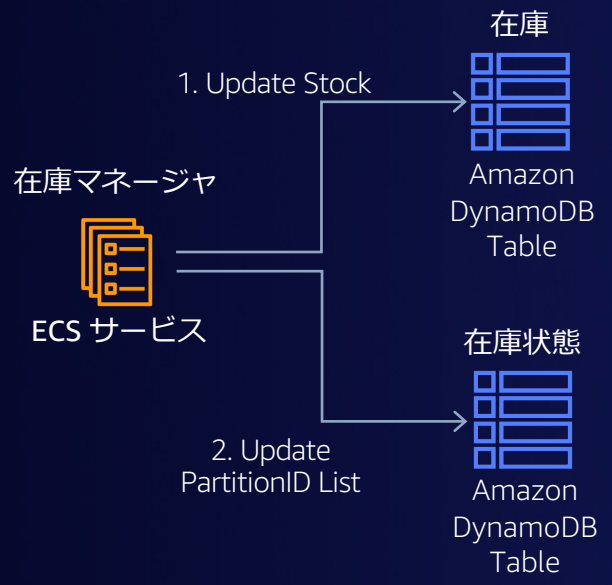
Primary key	Attributes
PartitionID (Partition key): String	Stock: Number
Product#1_#0	50
Product#1_#1	50

在庫状態テーブル

Primary key	Attributes
ProductID (Partition key): String	PartitionIDList: List
Product#1	[{ "S" : "#0" }, { "S" : "#1" }]

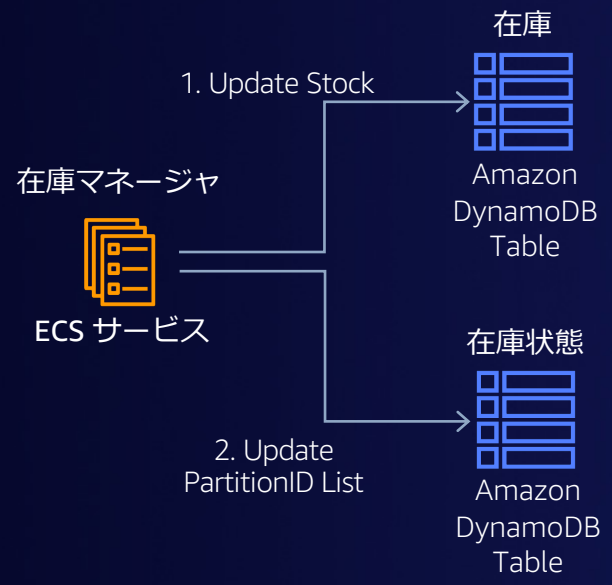
在庫が 0 になったとき、どのようにして在庫状態テーブルに反映を行うかが課題

在庫切れ対応に向けた選択肢



- 在庫状態の Update に失敗する場合は 不整合が発生

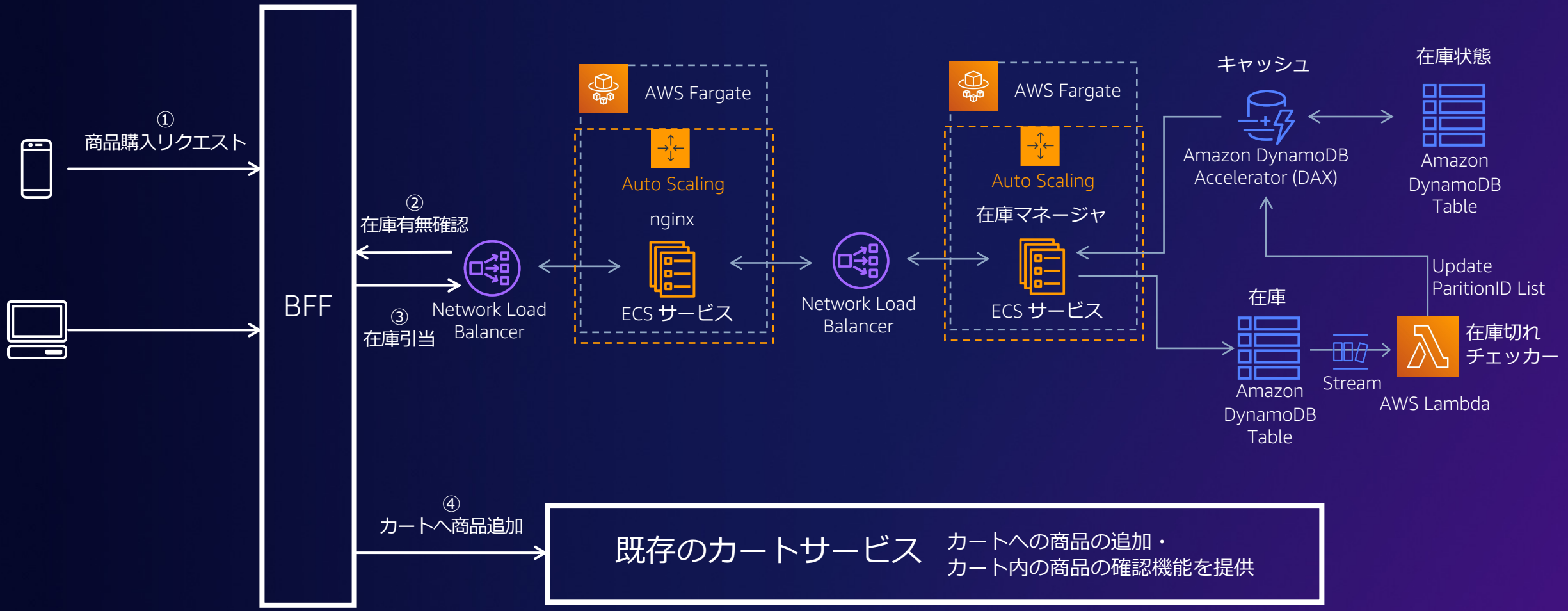
在庫切れ対応に向けた選択肢



- 在庫状態の Update に失敗する場合は 不整合が発生

- 在庫状態の Update に失敗した場合でも、リトライが実行でき、それでもダメな場合は DLQ に対応すべきイベントを残せる
- AWS Lambda の障害時でも DynamoDB Streams に対応すべきイベント履歴が残る
- 最終的に整合性が確保できる

改善したアーキテクチャの全体像



お題 2

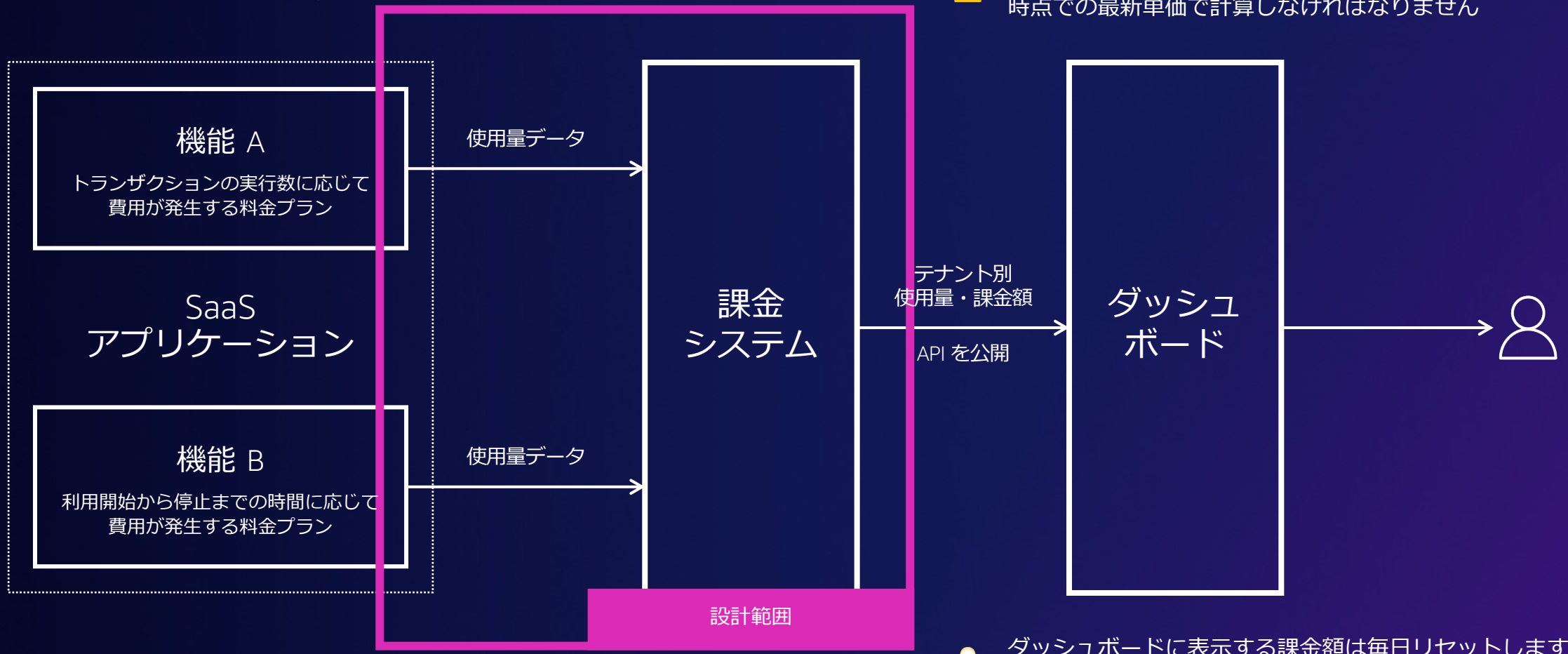
SaaS メータリング

経費精算 SaaS プロダクトのローンチに向けて準備を進めている Y 社は課金システムの開発に着手することにしました。

テナント別に集計した使用量や課金額はテナント向けダッシュボードに表示する予定です。テナント管理者がリアルタイムに利用状況を確認できるよう、テナント向けダッシュボードにはできるだけ最新のデータを反映しなければなりません。あなたはこのシステムをどのように設計しますか？

お題 2

SaaS メータリング



⚠ 課金額は機能利用時点での単価ではなくダッシュボード表示時点での最新単価で計算しなければなりません

💡 ダッシュボードに表示する課金額は毎日リセットします。請求金額の確定／請求データの作成を考慮する必要はありません

自己紹介: 山崎 翔太（やまざき しょうた)

会　　社：　アマゾン ウェブ サービス ジャパン 株式会社

所　　属：　技術統括本部

　　インターネットメディアソリューション本部

役　　職：　部長

　　シニアソリューションアーキテクト

担　　当：

- 大規模インターネット企業グループを担当する SA チームをリード
- メディアやコマースから金融まで幅広い業界のクラウド活用を支援
- ビッグデータ分析の技術専門チームに所属

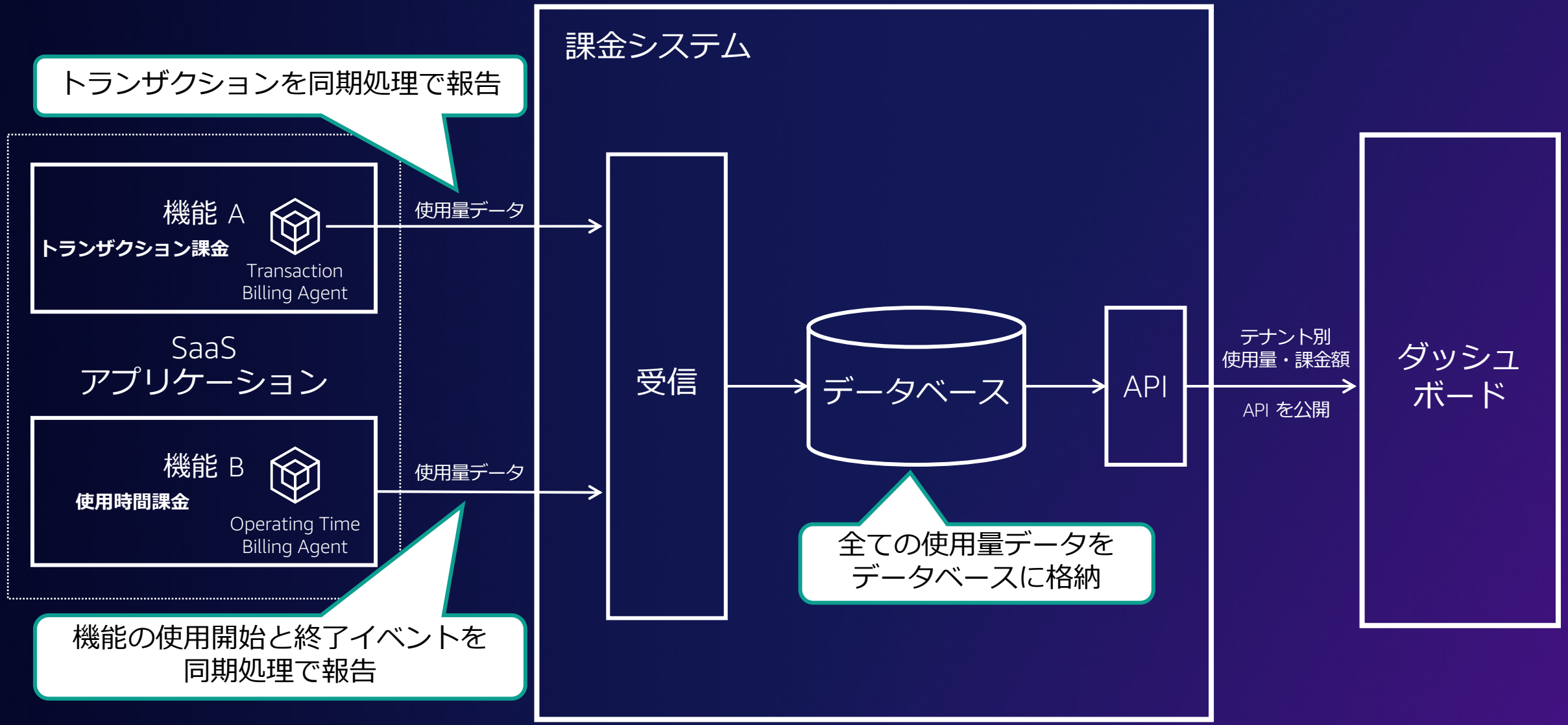
好きな AWS サービス： Amazon Kinesis, AWS Lambda

アーキテクチャ設計のポリシー

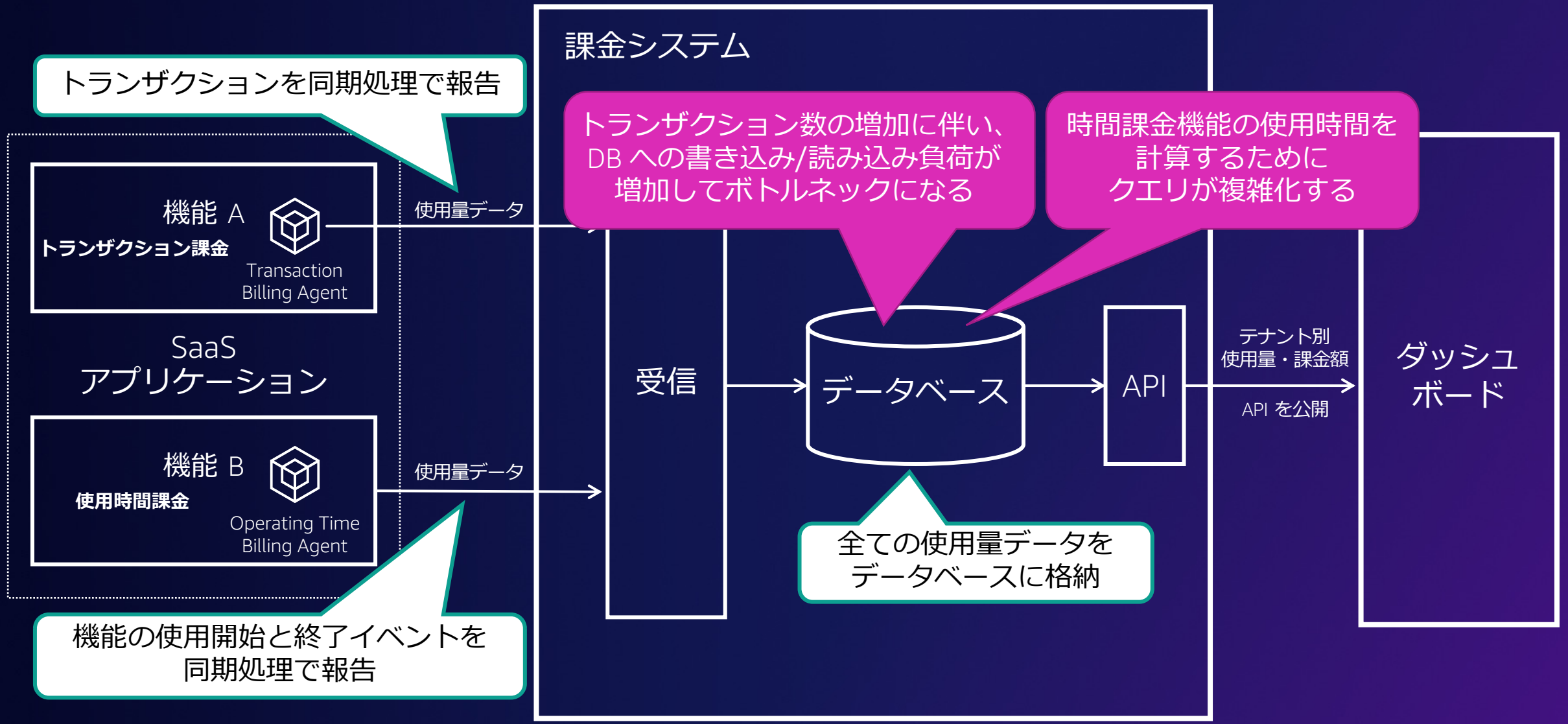
責任範囲の定義

SaaS アプリケーション	課金システム
• 使用量データの報告 （報告しなければ課金できない）	• 使用量データの受信と計上 （報告された使用量を課金する）
• ユーザーへのサービス提供を優先 （使用量報告の障害時には遅れて報告）	• 遅れて到着した使用量データの計上
• 使用量データへの一意 ID の採番	• 二重計上の防止

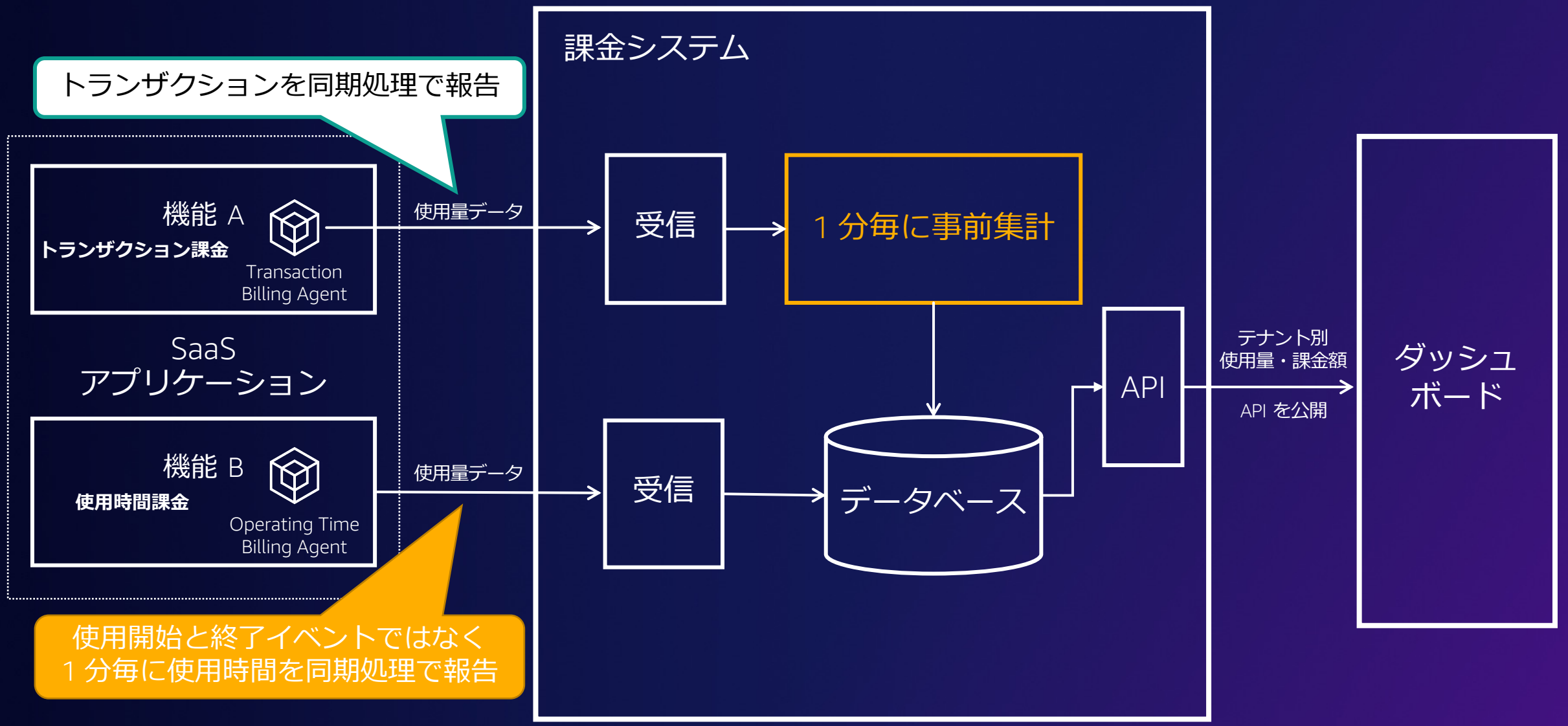
シンプルな設計



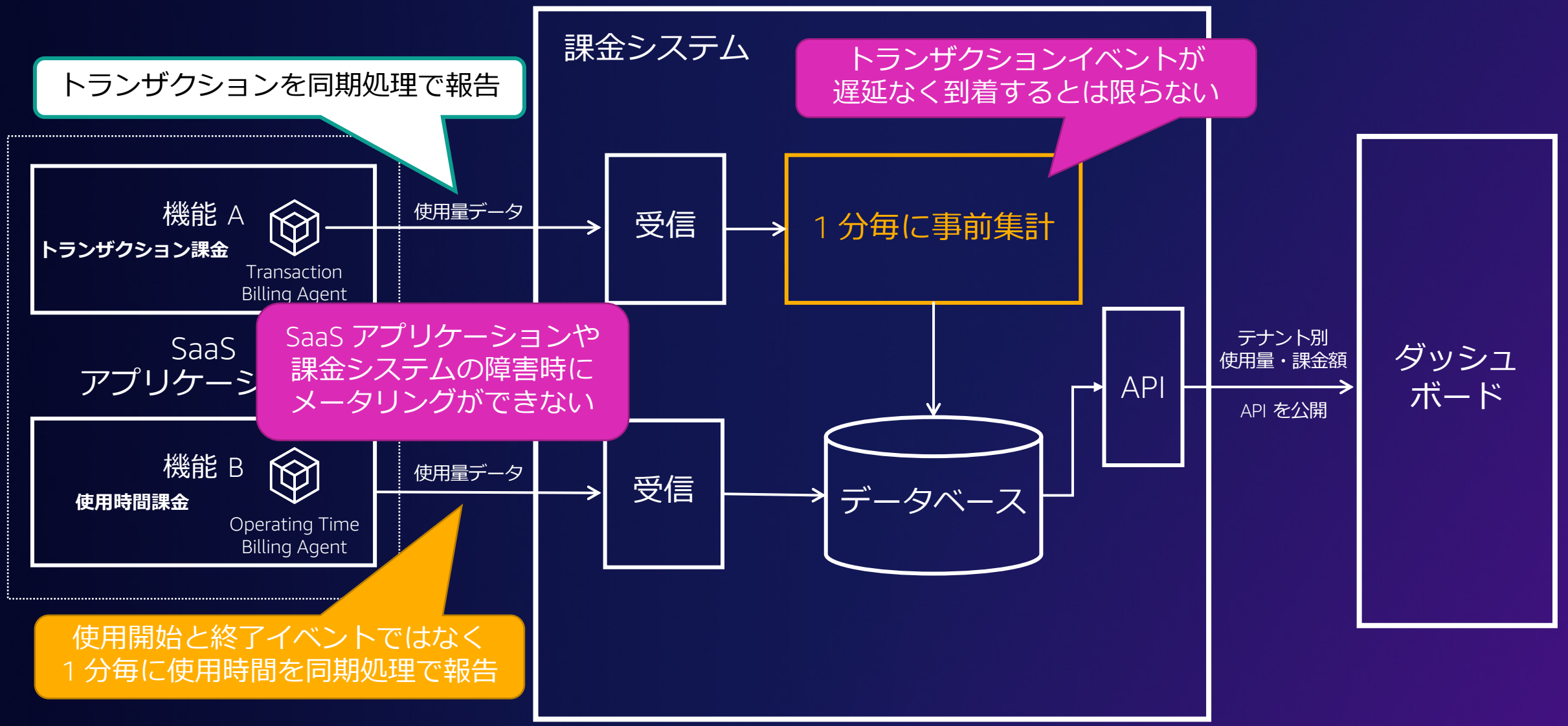
想定される課題（１）



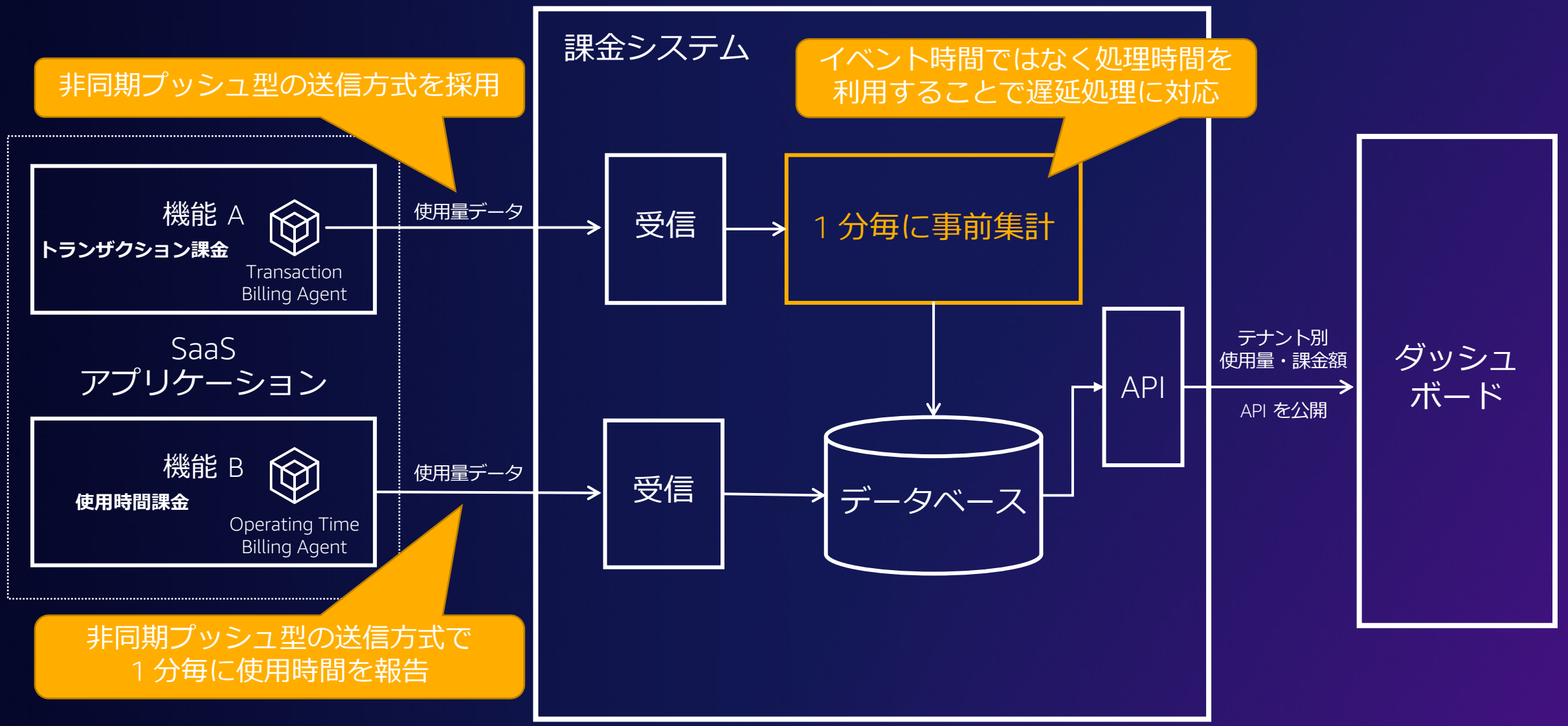
アーキテクチャによる解決案（1）



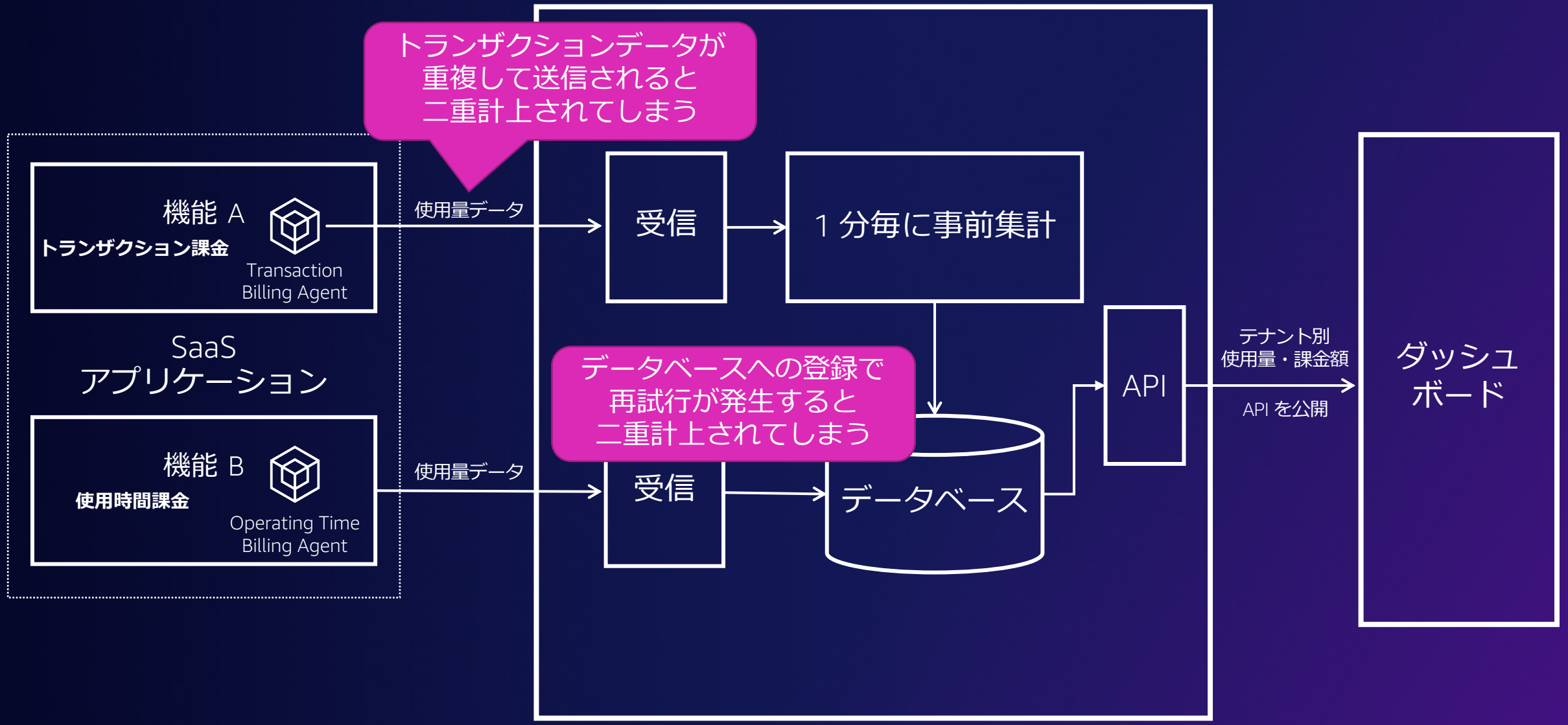
想定される課題（2）



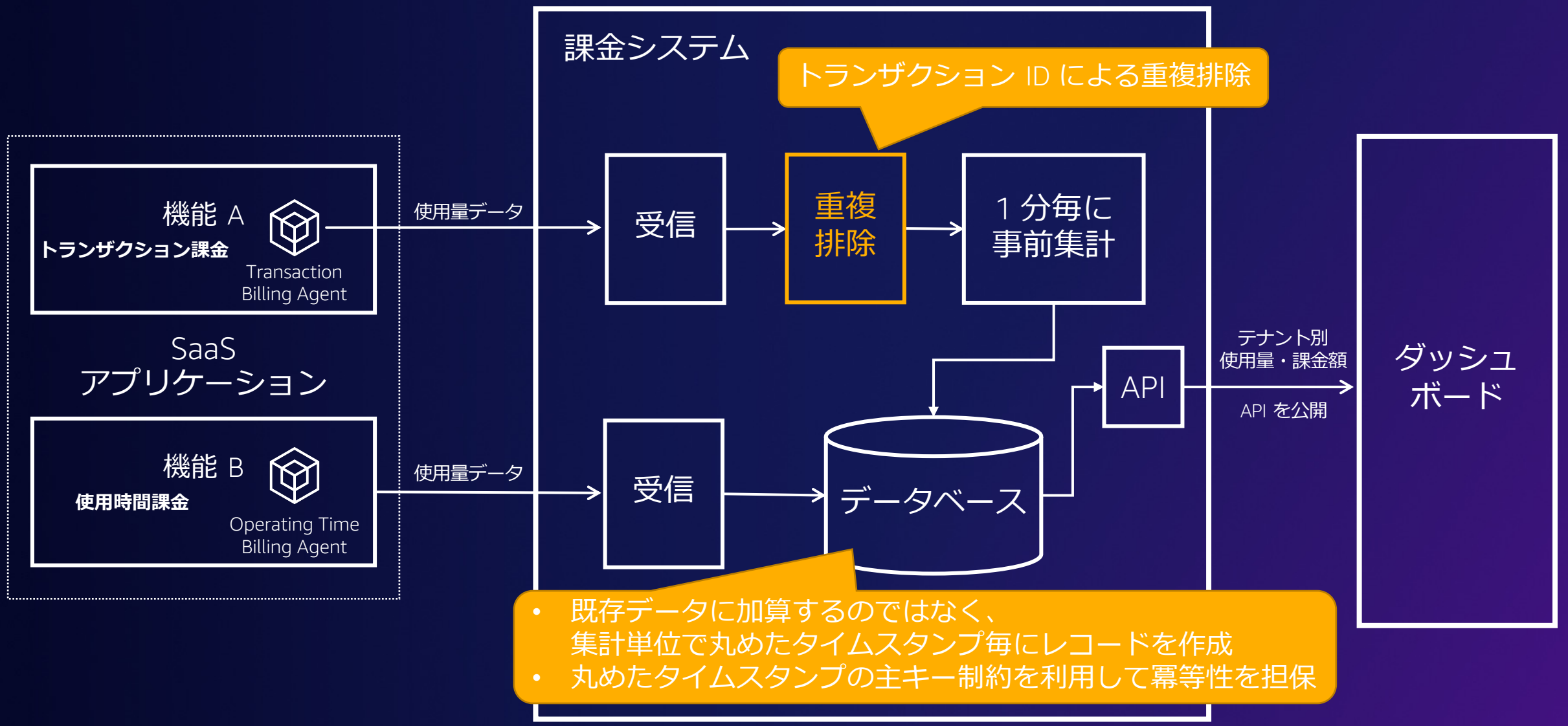
アーキテクチャによる解決案（2）



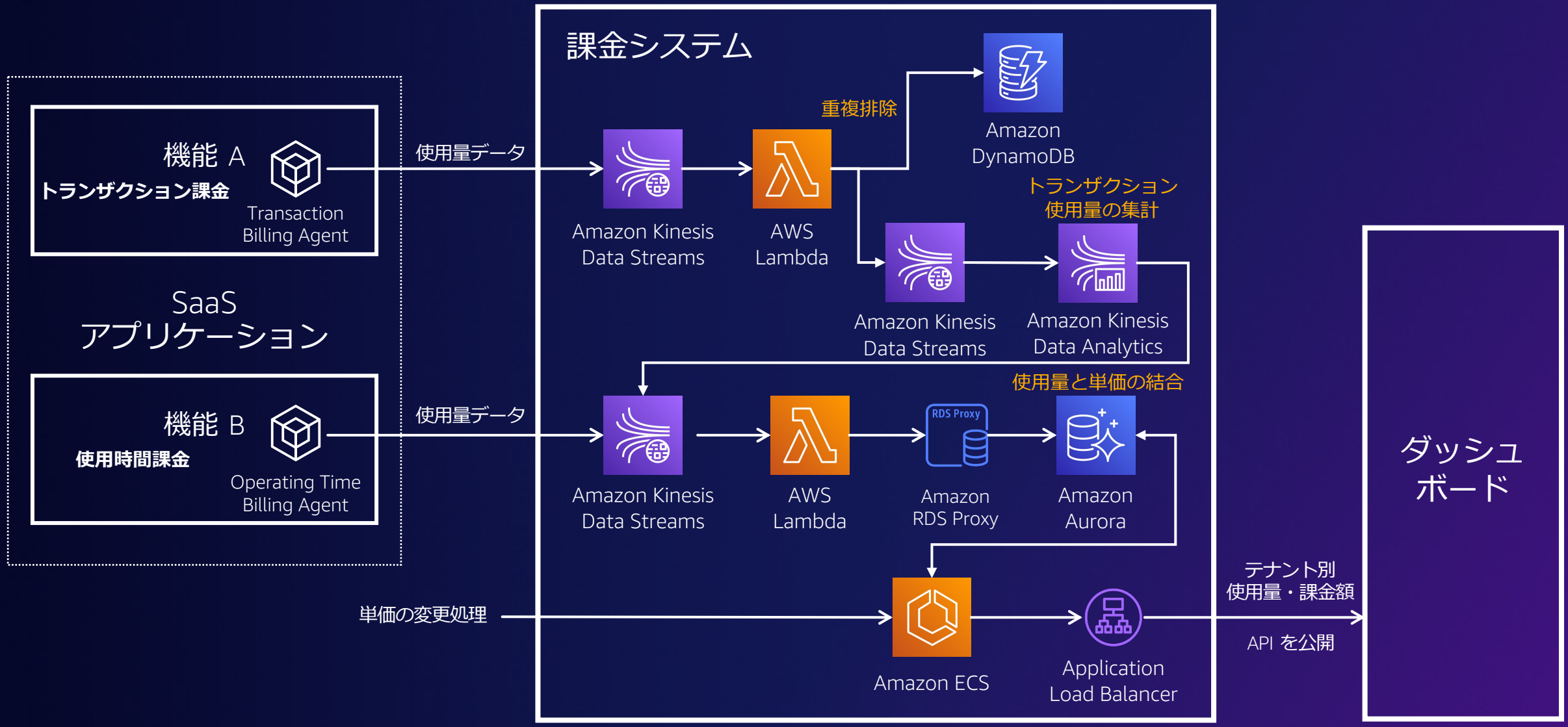
想定される課題（3）



アーキテクチャによる解決案（3）



全体アーキテクチャ



① トランザクションの収集

使用量データ

Amazon Kinesis
Data Streams

入力イベントの例

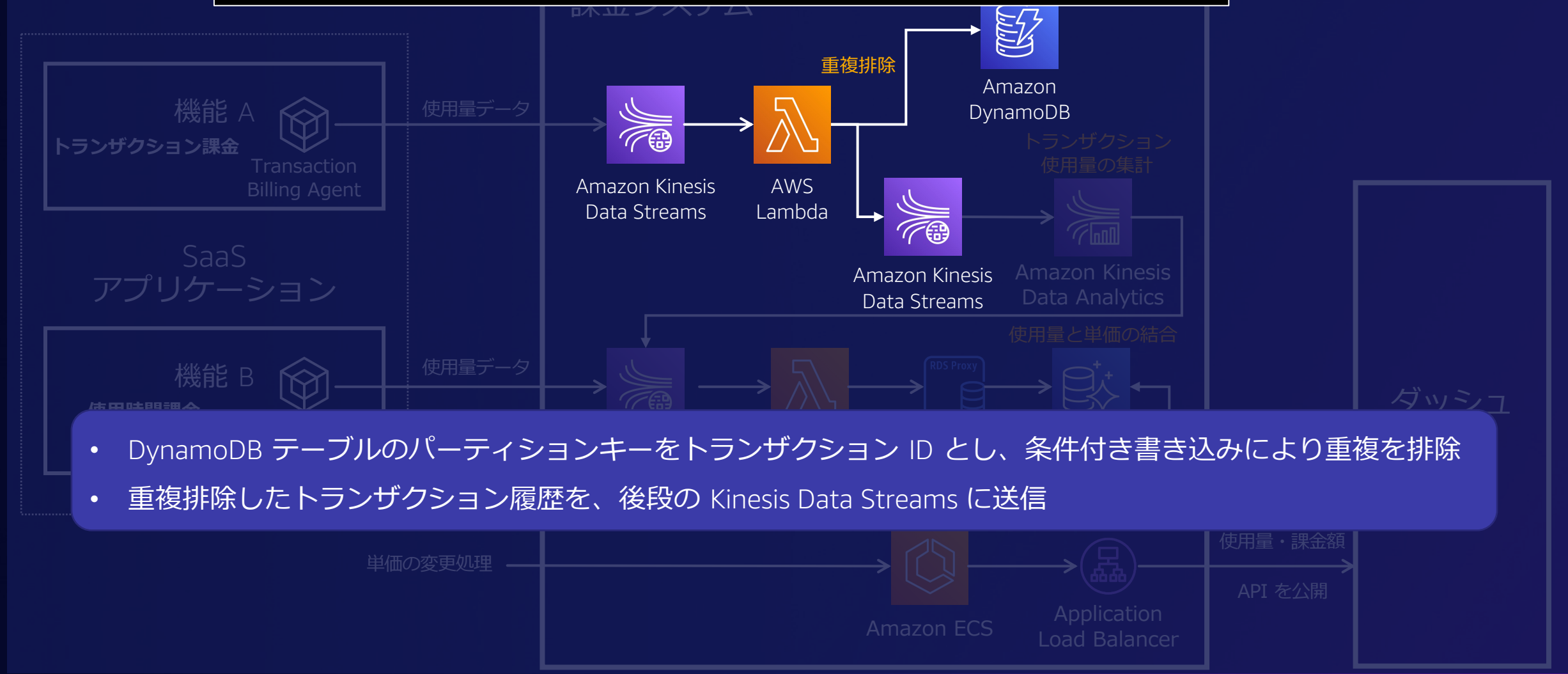
EVENT_TIME	TENANT_ID	FEATURE_ID	TRANSACTION_ID
2022-05-25 10:01:21.9	ABC	EXPENCE	ABC_EX_11254
2022-05-25 10:01:22.1	ABC	EXPENCE	ABC_EX_11255
2022-05-25 10:01:22.4	ABC	EXPENCE	ABC_EX_11256
2022-05-25 10:01:23.6	ABC	EXPENCE	ABC_EX_11257

- SaaS アプリ側にエージェントを配置
- ログをローカルでバッファし、トランザクション履歴を非同期で送信
  - SaaS アプリケーションの可用性を優先
  - データの重複、遅延、順序逆転への対応は課金システム側で実装
- トランザクション ID をパーティションキーとして送信
  - テナント毎の利用量に依存せずスループットをスケール

② トランザクションの重複排除

実装例

```
PutItem with 'ConditionExpression' => 'attribute_not_exists( TransactionId )'
```



- DynamoDB テーブルのパーティションキーをトランザクション ID とし、条件付き書き込みにより重複を排除
- 重複排除したトランザクション履歴を、後段の Kinesis Data Streams に送信

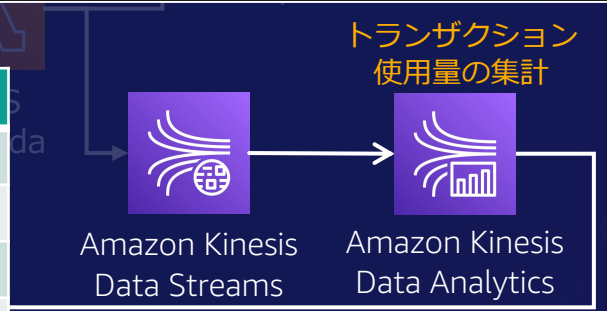
③ トランザクションの事前集計

SQL での実装例

```
INSERT INTO “DESTINATION_SQL_STREAM”
  SELECT STREAM
    tenant_id, feature_id, "transaction" AS billing_type
    COUNT(transaction_id) AS transaction_count
  FROM "SOURCE_SQL_STREAM_001"
  GROUP BY tenant_id, feature_id, STEP("SOURCE_SQL_STREAM_001".ROWTIME BY INTERVAL '1' MINUTE);
```

出力イベントの例

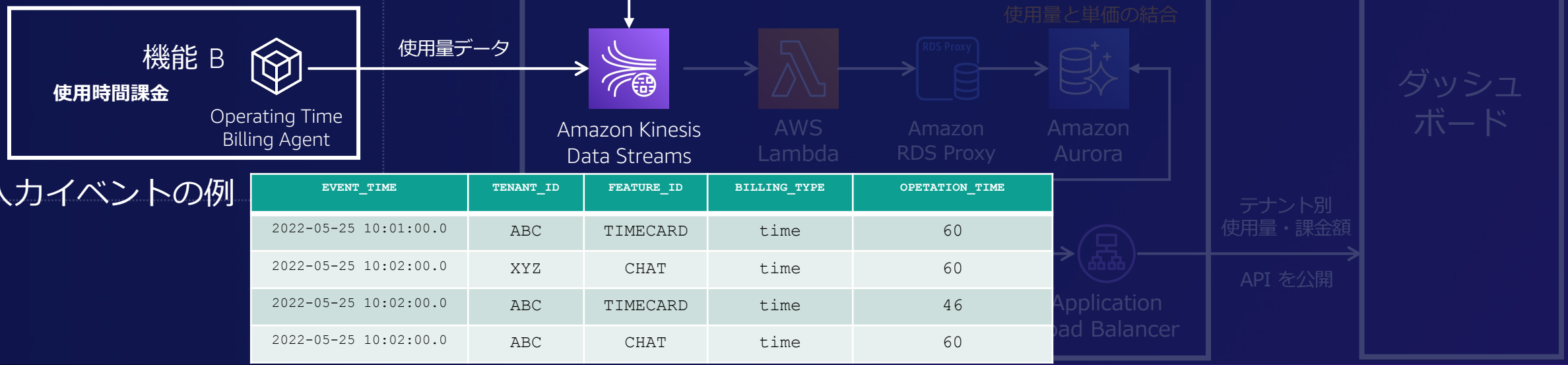
ROWTIME	TENANT_ID	FEATURE_ID	BILLING_TYPE	TRANSACTION_COUNT
2022-05-25 10:01:00.0	ABC	EXPENCE	transaction	12,364
2022-05-25 10:02:00.0	XYZ	TRAVEL	transaction	3
2022-05-25 10:02:00.0	ABC	EXPENCE	transaction	14,531
2022-05-25 10:02:00.0	ABC	TRAVEL	transaction	324



- Kinesis Data Analytics によるストリーム処理で、1 分間のトランザクション数を集計
  - 大量のトランザクションを事前集計することで、後段のデータベース書き込み負荷を抑制
- タンブリングウィンドウと集計関数を利用して、 テナント/機能毎に集計結果のイベントを発報
- 集計のウィンドウ関数で利用するタイムスタンプには、処理時間 (ROWTIME) を使用
  - エージェント側のイベント時間を利用しないことで、データ到着の遅延と順序逆転に対応

④ 使用時間の収集

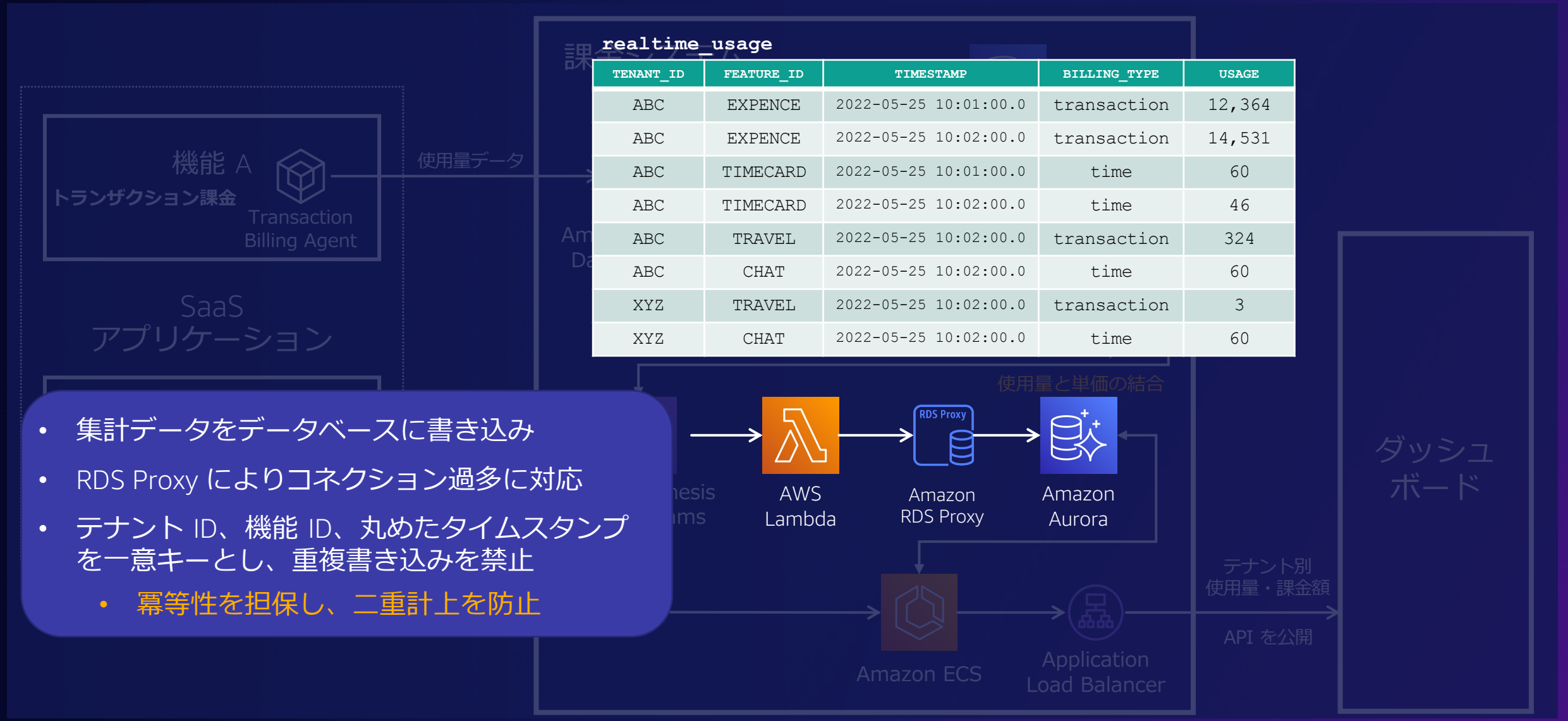
- SaaS アプリ側にエージェントを配置
- 1 分毎にエージェントが機能の使用時間を報告
  - 逐次処理によりメータリングの可用性とリアルタイム性を確保
  - データのキー設計をトランザクション課金と合わせ後段の処理を共通化
- キーとなるタイムスタンプには、エージェント側で丸めたイベント時間を使用
- テナント ID + 機能 ID をパーティションキーとして送信 (1req/min)



入力イベントの例

EVENT_TIME	TENANT_ID	FEATURE_ID	BILLING_TYPE	OPETATION_TIME
2022-05-25 10:01:00.0	ABC	TIMECARD	time	60
2022-05-25 10:02:00.0	XYZ	CHAT	time	60
2022-05-25 10:02:00.0	ABC	TIMECARD	time	46
2022-05-25 10:02:00.0	ABC	CHAT	time	60

⑤ データベース書き込み



- 集計データをデータベースに書き込み
- RDS Proxy によりコネクション過多に対応
- テナント ID、機能 ID、丸めたタイムスタンプを一意キーとし、重複書き込みを禁止
  - 冪等性を担保し、二重計上を防止

⑥ 課金額の計算と API の公開

realtime_usage

TENANT_ID	FEATURE_ID	TIMESTAMP	BILLING_TYPE	AMOUNT
ABC	EXPENCE	2022-05-25 10:01:00.0	transaction	12,364
ABC	EXPENCE	2022-05-25 10:02:00.0	transaction	14,531
ABC	TIMECARD	2022-05-25 10:01:00.0	time	60
ABC	TIMECARD	2022-05-25 10:02:00.0	time	46
ABC	TRAVEL	2022-05-25 10:02:00.0	transaction	324
ABC	CHAT	2022-05-25 10:02:00.0	time	60
XYZ	TRAVEL	2022-05-25 10:02:00.0	transaction	3
XYZ	CHAT	2022-05-25 10:02:00.0	time	60

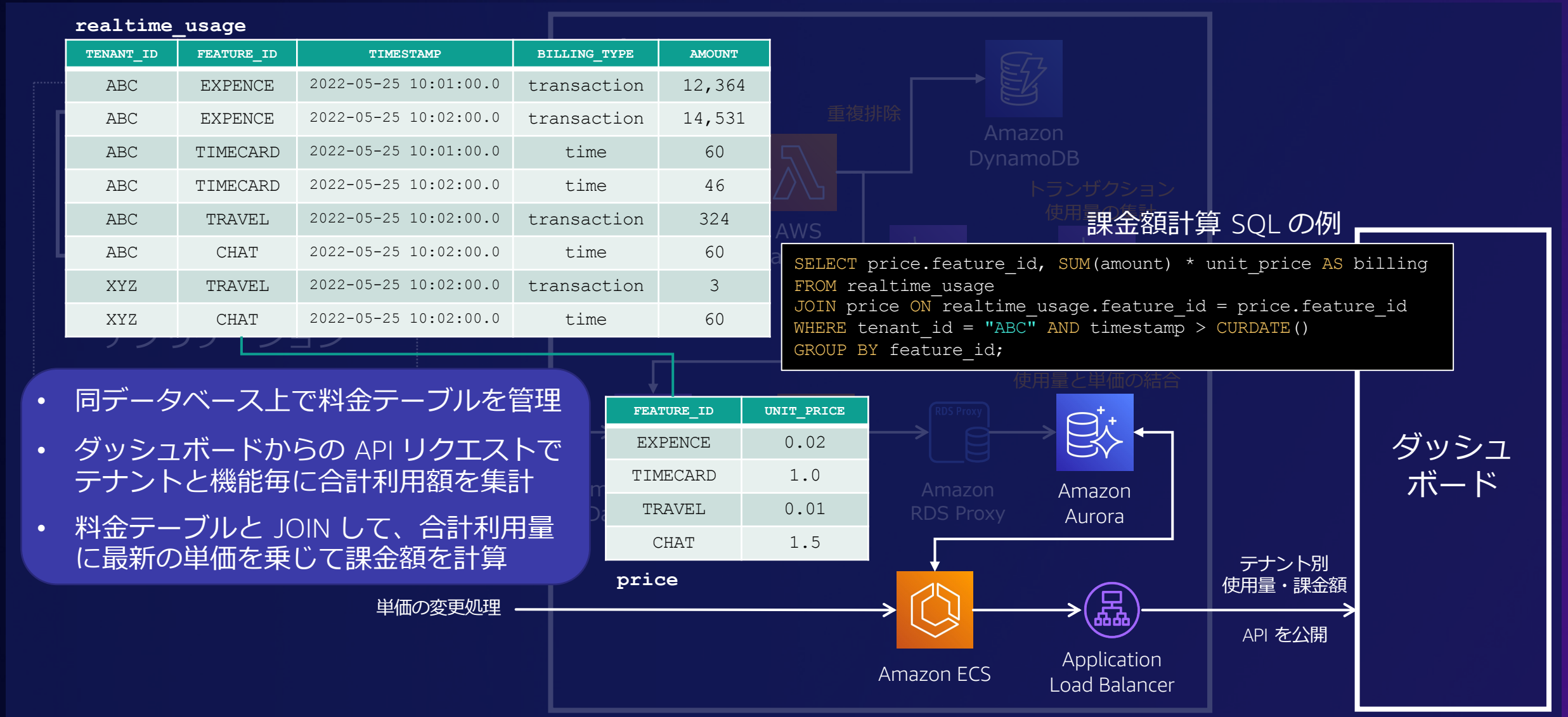
- 同データベース上で料金テーブルを管理
- ダッシュボードからの API リクエストでテナントと機能毎に合計利用額を集計
- 料金テーブルと JOIN して、合計利用量に最新の単価を乗じて課金額を計算

price

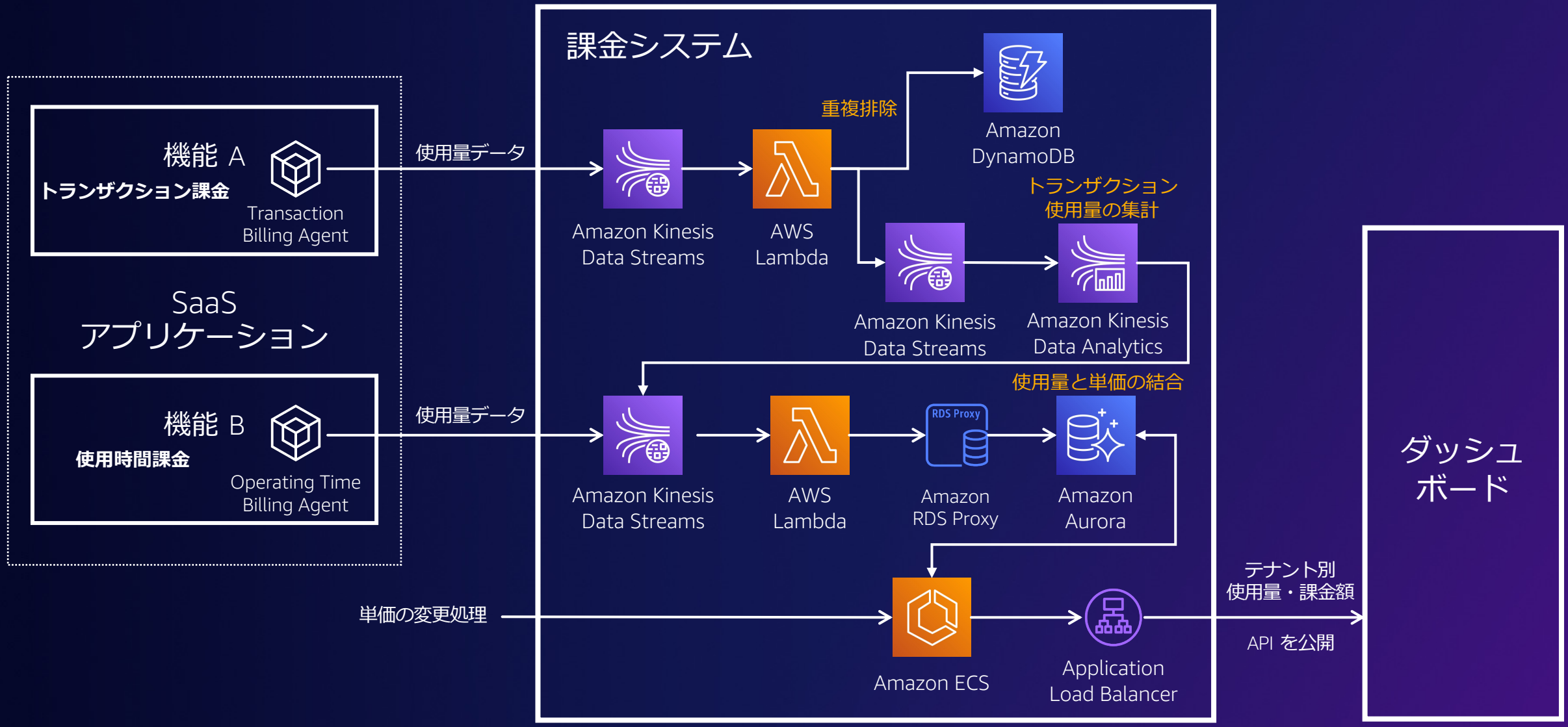
FEATURE_ID	UNIT_PRICE
EXPENCE	0.02
TIMECARD	1.0
TRAVEL	0.01
CHAT	1.5

課金額計算 SQL の例

```
SELECT price.feature_id, SUM(amount) * unit_price AS billing
FROM realtime_usage
JOIN price ON realtime_usage.feature_id = price.feature_id
WHERE tenant_id = "ABC" AND timestamp > CURDATE()
GROUP BY feature_id;
```



全体アーキテクチャ



Thank you!

内海 英一郎

@eiichirouchiumi

奥野 友哉

tomoya-okuno

山﨑 翔太

shota-yamazaki