# データ収集スケジューラ ドキュメント

このディレクトリには、データ収集スケジューラシステムの技術ドキュメントが含まれています。

# 📚 ドキュメント一覧

## 1. アーキテクチャ図

システムの静的構成とコンポーネント詳細

**内容:**

- システム構成図（Mermaid）
- データフロー図
- タイミングチャート
- 各コンポーネントの詳細仕様

## 2. シーケンス図

AI 社員 API とのやり取りを含む動的な処理フロー

**内容:**

- 全体フロー（1 分間のサイクル）
- Sender Lambda 詳細シーケンス
- Collector Lambda 詳細シーケンス
- エラーハンドリングフロー
- データフロー詳細

## 3. OpenAPI 仕様書

モック AI 社員 API の完全な API 仕様

**内容:**

- エンドポイント定義
- リクエスト/レスポンススキーマ
- エラーレスポンス
- 使用例

# 🔍 クイックリファレンス

## システム概要

5 秒間隔で AI 社員 API をポーリングし、レスポンス数を DynamoDB に保存する完全サーバーレスシステム。

## 主要コンポーネント

1. **EventBridge Rule** - 1 分ごとのトリガー
2. **Sender Lambda** - 12 個の遅延メッセージ生成
3. **SQS Queue** - サブミニッツスケジューリング
4. **Collector Lambda** - API 呼び出し + データ保存
5. **API Gateway + Mock API Lambda** - AI 社員 API モック
6. **DynamoDB** - メトリクスデータ保存

## データフロー

```
EventBridge (1分)
  → Sender Lambda
  → SQS (遅延: 0,5,10...55秒)
  → Collector Lambda (5秒ごと)
```

```
  → Mock API
  → DynamoDB
```

# API エンドポイント

# レスポンス数取得

```
GET /response_count?from={ISO8601}&to={ISO8601}
```

### レスポンス例:

```
{
  "from": "2025-12-03T10:23:00Z",
  "to": "2025-12-03T10:23:05Z",
  "count": 65
}
```

# ヘルスチェック

```
GET /health
```

### レスポンス例:

```
{
  "status": "ok"
}
```

# 📊 データモデル

## DynamoDB テーブル: AiResponseMetrics

| 属性名 | 型 | キー | 説明 |
| --- | --- | --- | --- |
| metricName | String | PK | メトリクス識 |
| slotTime | String | SK | 5<br>秒境界のタイ |
| count | Number | - | レスポンス数 |

## SQS メッセージ形式

```
{
  "from": "2025-12-03T10:23:00Z",
  "to": "2025-12-03T10:23:05Z"
}
```

## 🔧 技術スタック

- **言語**: TypeScript
- **ランタイム**: Node.js 20.x
- **IaC**: AWS CDK
- **テスト**: Vitest + fast-check (Property-Based Testing)
- **AWS サービス**:
  - Lambda
  - API Gateway
  - DynamoDB
  - SQS
  - EventBridge
  - CloudWatch Logs

# 📈 パフォーマンス特性

## スループット

- **データ収集頻度**: 5 秒ごと（12 回/分）
- **1 日あたりのデータポイント**: 17,280 件
- **Lambda 実行回数**: 約 34,560 回/日

## レイテンシ

- **Sender Lambda**: < 2 秒
- **Collector Lambda**: < 3 秒
- **API 呼び出し**: < 1 秒
- **DynamoDB 書き込み**: < 100ms

## コスト見積もり（月間）

- **Lambda**: 約 $0.50
- **DynamoDB**: 約 $1.00（オンデマンド）
- **SQS**: 約 $0.10
- **API Gateway**: 約 $0.50
- **合計**: 約 $2.10/月

# 🔐 セキュリティ

## 現在の実装（PoC）

- API 認証: なし
- VPC: なし（パブリックサブネット）
- 暗号化: DynamoDB 暗号化（デフォルト）

## 本番環境推奨

- API Gateway: API Key または カスタムオーソライザー
- Lambda: VPC 内配置
- DynamoDB: カスタマーマネージドキー（CMK）
- CloudWatch Logs: 暗号化

# 📝 運用

## デプロイ

```
# ビルド
npm run build

# デプロイ
AWS_PROFILE=main AWS_REGION=ap-northeast-1 npx cdk
```

## モニタリング

- CloudWatch Logs: Lambda 実行ログ
- CloudWatch Metrics: Lambda 実行回数、エラー率
- DynamoDB Metrics: 読み取り/書き込みキャパシティ

## トラブルシューティング

1. **データが収集されない**
    - EventBridge Rule の状態確認
    - Sender Lambda のログ確認
    - SQS キューのメッセージ数確認
2. **API 呼び出しエラー**
    - Mock API のログ確認

    - ネットワーク接続確認
    - API Gateway のメトリクス確認
3. **DynamoDB 書き込みエラー**
    - IAM ロールの権限確認
    - スロットリング確認
    - テーブルの状態確認

## 🔗 関連リンク

- 要件定義書
- 設計書
- 実装計画
- AWS CDK スタック

# サポート

質問や問題がある場合は、プロジェクトの
Issue トラッカーまでお問い合わせください。

# 要件定義書

## はじめに

本ドキュメントは、外部 HTTP API を 5 秒間隔でポーリングし、結果を DynamoDB に保存するデータ収集スケジューラの PoC（概念実証）の要件を定義します。EventBridge、SQS、Lambda を使用した完全サーバーレスの AWS アーキテクチャで、5 秒間隔のポーリングを実現します。テスト用のデータソースとして、モック「AI エンプロイー」HTTP API を使用します。

# 用語集

- **Collector Lambda**: AI エンプロイー API からデータを取得し、DynamoDB に保存する AWS Lambda 関数
- **Sender Lambda**: サブミニッツスケジューリングを実現するため、12 個の遅延 SQS メッセージを生成する AWS Lambda 関数
- **AI エンプロイー API**: 指定された時間枠のレスポンス数を返すモック HTTP サーバー
- **スロットタイム**: DynamoDB のソートキーとして使用される 5 秒単位に揃えられたタイムスタンプ（例：2025-12-02T10:23:05Z）
- **タイムウィンドウ**: レスポンス数を照会するための `from` と `to` タイムスタンプで定義される 5 秒間の期間
- **サブミニッツスケジューリング**: EventBridge の 1 分最小間隔より短いポーリング間隔を実現するため、SQS 遅延メッセージを使用するパターン

# 要件

## 要件 1: サブミニッツスケジューリングパターン

**ユーザーストーリー:** システム運用者として、外部 API を 5 秒ごとにポーリングしたい。これにより、きめ細かいメトリクスデータを収集できる。

## 受け入れ基準

1. EventBridge ルールが毎分 0 秒にトリガーされた時、Sender Lambda は正確に 1 回呼び出されるものとする
2. Sender Lambda が実行された時、Sender Lambda は SQS キューに正確に 12 個のメッセージを送信するものとする。各メッセージには対象時間幅（`from` と `to`）を含め、DelaySeconds の値は 0、5、10、15、20、25、30、35、40、45、50、55 とする
3. Sender Lambda が時間幅を生成する時、各メッセージの時間幅は 5 秒間隔で連続するものとする（例：0-5 秒、5-10 秒、...、55-60 秒）

4. SQS メッセージが利用可能になった時、Collector Lambda はそのメッセージを処理するために呼び出されるものとする
5. スケジューリングシステムが 1 分間動作した時、Collector Lambda は約 12 回呼び出されるものとする

## 要件 2: Collector Lambda のデータ取得

**ユーザーストーリー:** データアナリストとして、コレクターが AI エンプロイー API からレスポンス数を取得してほしい。これにより、5 秒ウィンドウごとのメトリクスが記録される。

## 受け入れ基準

1. Collector Lambda が実行された時、Collector Lambda は SQS メッセージから時間幅（`from` と `to`）を取得するものとする
2. Collector Lambda が時間幅を取得した後、Collector Lambda は ISO8601 形式の `from` と `to` クエリパラメータを使用して GET {AI_API_BASE_URL}/response_count を呼び出すものとする
3. AI エンプロイー API がレスポンスを返した時、Collector Lambda

は`from`、`to`、`count`フィールドを含むJSON レスポンスをパースするものとする

## 要件 3: DynamoDB ストレージ

**ユーザーストーリー:** データアナリストとして、メトリクスを 5 秒粒度で DynamoDB に保存してほしい。これにより、後でデータをクエリして可視化できる。

## 受け入れ基準

1. Collector Lambda が有効な API レスポンスを受信した時、Collector Lambda はパーティションキー`metricName`を"ai_response_count"に設定して DynamoDB にレコードを書き込むものとする
2. Collector Lambda がレコードを書き込む時、Collector Lambda はソートキー`slotTime`を最も近い 5 秒境界に切り捨てた ISO8601 タイムスタンプに設定するものとする
3. Collector Lambda がレコードを書き込む時、Collector Lambda は API レスポンスからの数値として`count`属性を保存するものとする

4. Collector Lambda が既存のスロットにレコードを書き込もうとした時、Collector Lambda は重複や二重カウントを防ぐ冪等な書き込み操作を使用するものとする

## 要件 4: スロットタイム計算

**ユーザーストーリー:** 開発者として、スロットタイムを 5 秒境界に揃えたい。これにより、データ集計が一貫して予測可能になる。

## 受け入れ基準

1. タイムスタンプからスロットタイムを計算する時、システムは秒を最も近い 5 の倍数に切り捨てるものとする（例：07 は 05 に、13 は 10 になる）
2. タイムスタンプの秒が正確に 5 秒境界上にある時、システムはその境界値を変更せずに保持するものとする
3. スロットタイムをフォーマットする時、システムは秒精度の ISO8601 形式で出力するものとする（例：2025-12-02T10:23:05Z）

## 要件 5: モック AI エンプロイー API

**ユーザーストーリー:** 開発者として、決定論的なレスポンス数を返すモック API が欲しい。これにより、データ収集パイプラインをエンドツーエンドでテストできる。

## 受け入れ基準

1. 有効な`from`と`to`クエリパラメータを持つ GET リクエストが/response_count に送信された時、モック API は`from`、`to`、`count`フィールドを含む JSON レスポンスを返すものとする
2. モック API がカウントを計算する時、モック API はタイムウィンドウに基づく決定論的な式を使用して 0 から 10 の間の値を生成するものとする
3. モック API がリクエストを受信した時、モック API は`from`と`to`パラメータを ISO8601 日時文字列としてパースするものとする
4. モック API が起動した時、モック API はポート 3000 でリッスンするものとする

## 要件 6: Infrastructure as Code

**ユーザーストーリー:** DevOps エンジニアとして、すべての AWS リソースを CDK TypeScript

で定義したい。これにより、インフラストラクチャがバージョン管理され、再現可能になる。

## 受け入れ基準

1. CDK スタックが合成された時、スタックはパーティションキー `metricName`（文字列）とソートキー `slotTime`（文字列）を持つ DynamoDB テーブルを定義するものとする
2. CDK スタックが合成された時、スタックはスケジューリングメッセージ用の SQS キューを定義するものとする
3. CDK スタックが合成された時、スタックは SQS キューへのメッセージ送信権限を持つ Sender Lambda を定義するものとする
4. CDK スタックが合成された時、スタックは DynamoDB への書き込み権限を持つ Collector Lambda を定義するものとする
5. CDK スタックが合成された時、スタックは 1 分ごとに Sender Lambda をトリガーする EventBridge ルールを定義するものとする
6. CDK スタックが合成された時、スタックは Lambda 関数を Node.js 20.x ランタイムで設定するものとする

## 要件 7: エラーハンドリングとロギング

**ユーザーストーリー:** 開発者として、Lambda 関数に基本的なエラーハンドリングとロギングが欲しい。これにより、テスト中の問題をデバッグできる。

## 受け入れ基準

1. Collector Lambda が AI エンプロイー API の呼び出しに失敗した時、Collector Lambda はエラー詳細をログに記録し、例外を再スローするものとする
2. Collector Lambda が DynamoDB への書き込みに失敗した時、Collector Lambda はエラー詳細をログに記録し、例外を再スローするものとする
3. Sender Lambda が SQS メッセージの送信に失敗した時、Sender Lambda はエラー詳細をログに記録し、残りのメッセージの処理を続行するものとする
4. Lambda 関数が実行された時、Lambda 関数はタイムスタンプと関連パラメータを含む主要な操作詳細をログに記録するものとする

# 設計書: データ収集スケジューラ

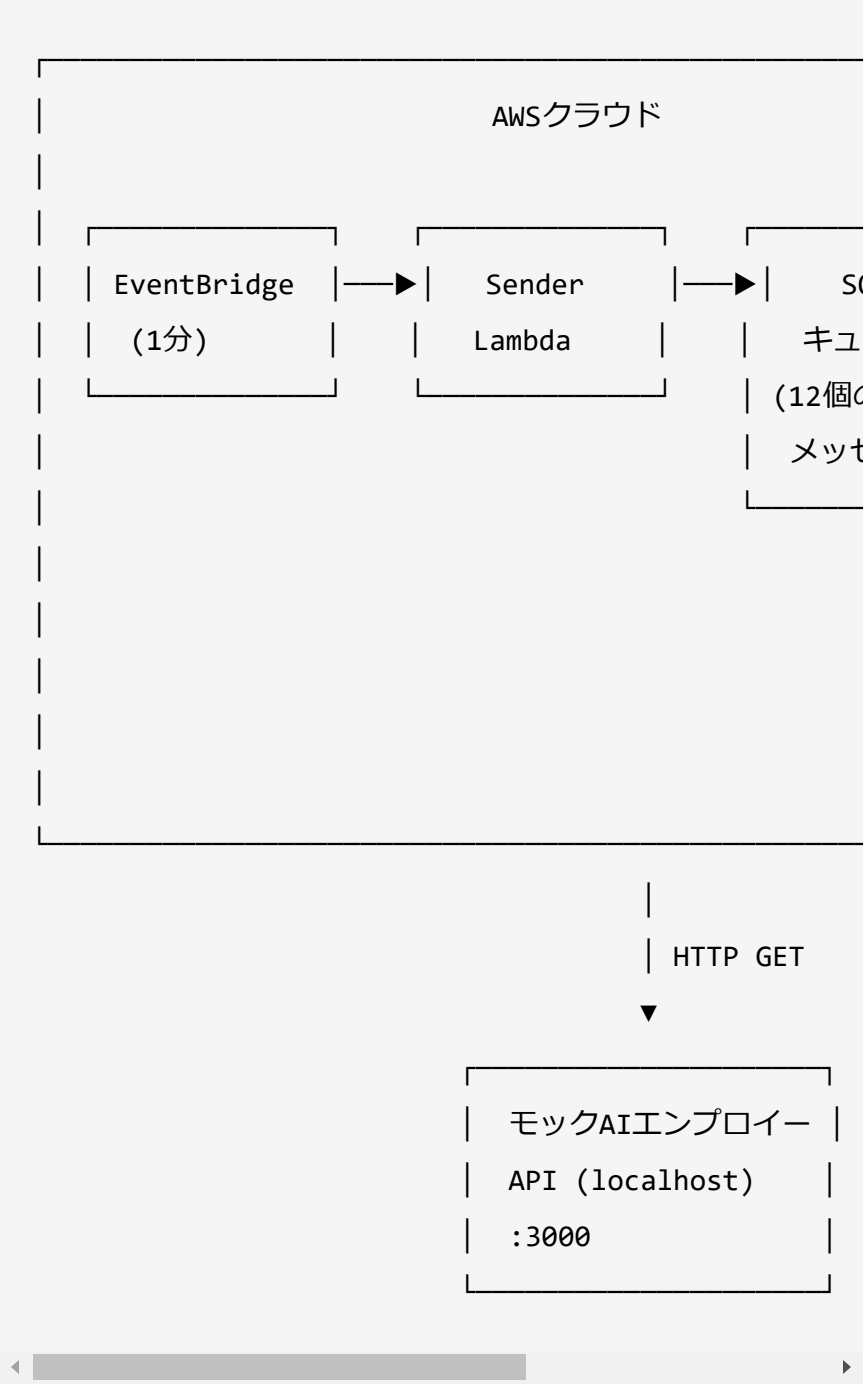
## 概要

本ドキュメントは、完全サーバーレスの AWS アーキテクチャを使用して、外部 HTTP API のサブミニッツ（5 秒）ポーリングを実現する PoC データ収集スケジューラの設計を記述します。システムはメトリクスを DynamoDB に保存し、後で可視化できるようにします。

## 主要な設計決定

1. **時刻形式**: すべてのタイムスタンプに ISO8601 秒精度形式を使用（例：`2025-12-02T10:23:05Z`）
2. **AWS リージョン**: CDK コンテキスト/環境で設定可能（デフォルト：us-east-1）
3. **冪等性戦略**: 重複エントリを防ぐため、`attribute_not_exists`を使用した DynamoDB 条件付き書き込み
4. **モック API カウント式**: 可視パターンを作成するため、分の値に基づく決定論的計算

# アーキテクチャ


AWSクラウド
EventBridge
(1分)
Sender
Lambda
キュ
(12個
メッセ
HTTP GET
モックAIエンプロイー
API (localhost)
:3000

# サブミニッツスケジューリングフロー

```
N分目:
├── EventBridgeがSender Lambdaをトリガー
├── Sender Lambdaが12個のSQSメッセージを送信:
│   ├── メッセージ1: DelaySeconds=0  → CollectorがN+
│   ├── メッセージ2: DelaySeconds=5  → CollectorがN+
│   ├── メッセージ3: DelaySeconds=10 → CollectorがN+
│   ├── ...
│   └── メッセージ12: DelaySeconds=55 → CollectorがN
└── 各メッセージは遅延が終了するとCollector Lambdaをト
```

# コンポーネントとインターフェース

## 1. Sender Lambda

**目的**: 1 分間に 5 秒間隔のポーリングを作成するため、12 個の遅延 SQS メッセージを生成する。各メッセージには対象時間幅（FROM, TO）を含める。

**インターフェース**:

```
// 入力: EventBridgeスケジュールイベント（毎分0秒にト
// 出力: void（副作用：12個のSQSメッセージ送信）

interface SenderConfig {
  queueUrl: string; // 環境変数からのSQSキューURL
  intervalSeconds: number; // 5秒
  messagesPerMinute: number; // 12メッセージ
}

interface ScheduleMessage {
  from: string; // ISO8601タイムスタンプ（時間幅の開
  to: string; // ISO8601タイムスタンプ（時間幅の終了）
}
```

**動作**:

1. EventBridge トリガー時刻（毎分 0 秒）を基準時刻として取得
2. 12 個のスロットを計算:
    - スロット 0: from=基準時刻+0 秒, to=基準時刻+5 秒, delay=0
    - スロット 1: from=基準時刻+5 秒, to=基準時刻+10 秒, delay=5
    - ...
    - スロット 11: from=基準時刻+55 秒, to=基準時刻+60 秒, delay=55
3. 対応する`DelaySeconds`で各メッセージを SQS に送信
4. デバッグ用に送信した各メッセージをログ記録
5. 1 つが失敗しても残りのメッセージの処理を続行

## 2. Collector Lambda

**目的**: SQS メッセージから時間幅を受け取り、AI エンプロイー API からメトリクスを取得し、DynamoDB に保存する。

**インターフェース**:

```
// 入力: 時間幅（from, to）を含むSQSイベント
// 出力: void（副作用：DynamoDBレコード書き込み）
```

```
interface CollectorConfig {
  apiBaseUrl: string; // AI_API_BASE_URL環境変数
  tableName: string; // AI_METRICS_TABLE_NAME環境変
}

interface ScheduleMessage {
  from: string; // ISO8601タイムスタンプ（時間幅の開始
  to: string; // ISO8601タイムスタンプ（時間幅の終了）
}

interface ApiResponse {
  from: string;
  to: string;
  count: number;
}

interface MetricRecord {
  metricName: string; // パーティションキー: "ai_res
  slotTime: string; // ソートキー: fromのISO8601（5秒
  count: number;
}
```

**動作**:

1. SQS メッセージから時間幅（from, to）を取得
2. `from` をスロットタイムとして使用（既に 5 秒境界に揃っている）

3. API を呼び出し: `GET` `{baseUrl}/response_count?from={from}&to={to}`
4. JSON レスポンスをパース
5. 冪等な条件式で DynamoDB に書き込み
6. デバッグ用にすべての操作をログ記録

## 3. モック AI エンプロイー API

**目的**: 収集パイプライン用の決定論的テストデータを提供する。

**インターフェース**:

```
// エンドポイント: GET /response_count
// クエリパラメータ: from (ISO8601), to (ISO8601)
// レスポンス: { from: string, to: string, count: nu

interface ResponseCountQuery {
  from: string; // ISO8601日時
  to: string; // ISO8601日時
}

interface ResponseCountResult {
  from: string;
  to: string;
  count: number; // 0-10、時刻に基づく決定論的値
}
```

**カウント式**（決定論的）:

```
// 可視パターンを作成するため分の値を使用
const minute = new Date(to).getMinutes();
const count = minute % 11; // 11分ごとに0-10の値が循
```

# 4. スロットタイム計算機

**目的**: タイムスタンプを 5 秒境界に切り捨てる純粋関数。

**インターフェース**:

```
function calculateSlotTime(timestamp: Date): string
// 入力: 任意のDate
// 出力: 秒を5の倍数に切り捨てたISO8601文字列
// 例: 2025-12-02T10:23:07Z → "2025-12-02T10:23:05Z
```

**アルゴリズム**:

```
const seconds = timestamp.getUTCSeconds();
const truncatedSeconds = Math.floor(seconds / 5) *
// 秒を切り捨て値に、ミリ秒を0に設定
// ISO8601文字列を返す
```

# データモデル

## DynamoDB テーブル: AiResponseMetrics

| 属性 | 型 | キータイプ |
| --- | --- | --- |
| metricName | String | パーティションキー |
| slotTime | String | ソートキー |
| count | Number | - |

**レコード例**:

```
{
  "metricName": "ai_response_count",
  "slotTime": "2025-12-02T10:23:05Z",
  "count": 7
}
```

## SQS メッセージスキーマ

```
{
  "from": "2025-12-02T10:23:00Z",
```

```
  "to": "2025-12-02T10:23:05Z"
}
```

# 正確性プロパティ

プロパティとは、システムのすべての有効な実行において真であるべき特性または動作です。本質的に、システムが何をすべきかについての形式的な記述です。プロパティは、人間が読める仕様と機械で検証可能な正確性保証の橋渡しをします。

## プロパティ 1: Sender が正しい遅延シーケンスを生成

任意のSender Lambda 実行において、関数は正確に 12 個のメッセージを生成し、DelaySeconds 値は[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55]のシーケンスを形成するものとする。

**検証対象: 要件 1.2**

## プロパティ 2: Sender のタイムウィンドウ生成の正確性

任意の基準時刻（毎分 0 秒）において、Sender Lambda が生成する 12 個のメッセージは、それぞれ正確に 5 秒間隔の連続した時間幅（from,

to）を持ち、`to - from = 5秒`かつ`from < to`であるものとする。

**検証対象: 要件 1.2, 2.1**

## プロパティ 3: ISO8601 形式での API URL 構築

任意の有効な from/to 日付を持つタイムウィンドウにおいて、構築された URL は適切にフォーマットされた ISO8601 クエリパラメータを含み、同等のタイムスタンプにパースバック可能であるものとする。

**検証対象: 要件 2.3**

## プロパティ 4: API レスポンスパースのラウンドトリップ

任意の`from`、`to`、`count`フィールドを含む有効な JSON レスポンスにおいて、パースは同等の値を持つ構造化オブジェクトを生成し、同等の JSON にシリアライズバック可能であるものとする。

**検証対象: 要件 2.4**

## プロパティ 5: スロットタイムの 5 秒境界への切り捨て

任意のタイムスタンプにおいて、計算されたスロットタイムは秒が 5 の倍数であり、元のタイムスタンプ以下であり、元のタイムスタンプとの差が 5 秒未満であるものとする。

**検証対象: 要件 4.1, 3.2**

## プロパティ 6: スロットタイム ISO8601 形式

任意の計算されたスロットタイムにおいて、出力文字列は ISO8601 パターン `YYYY-MM-DDTHH:mm:ssZ` に一致し、同等の Date にパースバック可能であるものとする。

**検証対象: 要件 4.3**

## プロパティ 7: モック API レスポンス構造と境界

任意の有効なタイムウィンドウクエリにおいて、レスポンスは `from`、`to`、`count` フィールドを含み、count は 0 から 10 の整数であり、同じ入力は常に同じ count を生成する（決定論性）ものとする。

**検証対象: 要件 5.1, 5.2**

## プロパティ 8: モック API の ISO8601 パース

任意の有効な ISO8601 日時文字列において、モック API はそれを Date オブジェクトにパースし、ISO8601 にフォーマットバックすると同等のタイムスタンプを生成するものとする。

**検証対象: 要件 5.3**

## プロパティ 9: 冪等な DynamoDB 書き込み

任意のメトリクスレコードにおいて、同じレコード（同じ metricName と slotTime）を複数回書き込むと、テーブルには一貫したデータを持つ正確に 1 つのレコードが存在するものとする。

**検証対象: 要件 3.4**

# エラーハンドリング

## Sender Lambda エラー

| エラーシナリオ | ハンドリング戦略 |
| --- | --- |
| SQS SendMessage 失敗 | エラーをログ記録、残りのメッセージを続行 |
| 無効なキュー URL | エラーをログ記録、例外をスロー（フェイルファスト） |

## Collector Lambda エラー

| エラーシナリオ | ハンドリング戦 |
| --- | --- |
| API 呼び出し失敗（ネットワーク/タイムアウト） | エラー詳細をログ記録、S リトライをトリガーするた |
| API が非 200 ステータスを返す | エラー詳細をログ記録、S リトライをトリガーするた |
| API が無効な JSON を返す | エラー詳細をログ記録、S リトライをトリガーするた |

| エラーシナリオ | ハンドリング戦 |
| --- | --- |
| DynamoDB<br>書き込み失敗 | エラー詳細をログ記録、S<br>リトライをトリガーするた |
| 条件チェック失敗<br>（重複） | 情報としてログ記録<br>（冪等性のため想定内）、 |

## モック API エラー

| エラーシナリオ | ハンドリング戦略 |
| --- | --- |
| クエリパラメータ欠落 | エラーメッセージ付き<br>400 を返す |
| 無効な ISO8601 形式 | エラーメッセージ付き<br>400 を返す |

# テスト戦略

## プロパティベーステストライブラリ

**ライブラリ**: TypeScript 用fast-check

**設定**: プロパティテストごとに最低 100 回の反復

## プロパティベーステスト

各正確性プロパティは fast-check を使用したプロパティベーステストとして実装されます：

1. **Sender 遅延シーケンス** - 任意の実行コンテキストを生成、正しい遅延を持つ 12 メッセージを検証
2. **タイムウィンドウ計算** - 任意のタイムスタンプとウィンドウサイズを生成、from/to 関係を検証
3. **URL 構築** - 任意のタイムウィンドウを生成、ISO8601 フォーマットとパース可能性を検証
4. **レスポンスパース** - 任意の有効な JSON レスポンスを生成、ラウンドトリップ一貫性を検証

5. **スロットタイム切り捨て** - 任意のタイムスタンプを生成、5 秒境界揃えを検証
6. **スロットタイムフォーマット** - 任意のスロットタイムを生成、ISO8601 形式とパース可能性を検証
7. **モック API レスポンス** - 任意のタイムウィンドウを生成、レスポンス構造と決定論性を検証
8. **ISO8601 パース** - 任意の有効な ISO8601 文字列を生成、パース/フォーマットラウンドトリップを検証
9. **冪等書き込み** - 任意のレコードを生成、複数書き込み後の単一レコードを検証

# ユニットテスト

ユニットテストは以下をカバーします：

1. **スロットタイム計算のエッジケース**:
    - 正確に 5 秒境界上のタイムスタンプ
    - 分/時境界のタイムスタンプ
    - ミリ秒を持つタイムスタンプ
2. **エラーハンドリングパス**:
    - API タイムアウトシミュレーション
    - 無効な JSON レスポンス
    - DynamoDB 条件チェック失敗
3. **設定読み込み**:
    - 環境変数のデフォルト値

- 必須設定の欠落

## 統合テスト（手動）

PoC では、統合テストは手動で行います：

1. CDK スタックをデプロイ
2. モック API をローカルで起動
3. CloudWatch ログで Lambda 実行を観察
4. DynamoDB をクエリしてレコードを検証

## テストアノテーション形式

すべてのプロパティベーステストには以下のコメント形式を含める必要があります：

```
// **Feature: data-collection-scheduler, Property {
```

# 実装計画

- [x] 1. プロジェクト構造と CDK セットアップ
  - [x] 1.1 CDK プロジェクトの初期化とディレクトリ構造の作成
    - `bin/app.ts`、`lib/stack.ts`、`lambda/` ディレクトリを作成
    - package.json に CDK 依存関係を追加
    - tsconfig.json を設定
    - 要件*: 6.1, 6.2, 6.3, 6.4, 6.5, 6.6*
  - [x] 1.2 共通ユーティリティとインターフェースの作成
    - `lambda/shared/types.ts` に共通型定義を作成（ScheduleMessage, ApiResponse, MetricRecord）
    - `lambda/shared/slot-time.ts` にスロットタイム計算関数を作成
    - 要件*: 4.1, 4.2, 4.3*
  - [x] 1.3 スロットタイム計算のプロパティテストを作成
    - **プロパティ 5: スロットタイムの 5 秒境界への切り捨て**
    - **プロパティ 6: スロットタイム ISO8601 形式**

        - **検証対象: 要件 4.1, 4.3**
- [x] 2. モック AI エンプロイー API の実装
    - [x] 2.1 Express サーバーのセットアップ
        - `mock-api/` ディレクトリを作成
        - package.json と tsconfig.json を設定
        - Express アプリケーションの基本構造を作成
        - 要件*: 5.4*
    - [x] 2.2 /response_count エンドポイントの実装
        - ISO8601 パラメータのパースを実装
        - 決定論的カウント計算（minute % 11）を実装
        - JSON レスポンス構造を実装
        - エラーハンドリング（400 レスポンス）を実装
        - 要件*: 5.1, 5.2, 5.3*
    - [x] 2.3 モック API のプロパティテストを作成
        - **プロパティ 7: モック API レスポンス構造と境界**
        - **プロパティ 8: モック API の ISO8601 パース**
        - **検証対象: 要件 5.1, 5.2, 5.3**
- [x] 3. Sender Lambda の実装

- [x] 3.1 Sender Lambda 関数の作成
  - `lambda/sender/index.ts` を作成
  - 基準時刻から 12 個の時間幅（from, to）を生成するロジックを実装
  - 各メッセージに対応する DelaySeconds（0, 5, 10, ... 55）を設定
  - SQS メッセージ送信ロジックを実装
  - エラーハンドリングとロギングを実装
  - 要件*: 1.2, 1.3, 7.3, 7.4*
- [x] 3.2 Sender Lambda のプロパティテストを作成
  - **プロパティ 1: Sender が正しい遅延シーケンスを生成**
  - **プロパティ 2: Sender のタイムウィンドウ生成の正確性**
  - **検証対象: 要件 1.2, 1.3**
- [x] 4. Collector Lambda の実装
  - [x] 4.1 API 呼び出しロジックの実装
    - `lambda/collector/api-client.ts` を作成
    - SQS メッセージから from/to を取得

    - URL 構築と ISO8601 フォーマットを実装
    - HTTP リクエストとレスポンスパースを実装
    - 要件*: 2.1, 2.2, 2.3*
- [x] 4.2 API 呼び出しのプロパティテストを作成
    - **プロパティ 3: ISO8601 形式での API URL 構築**
    - **プロパティ 4: API レスポンスパースのラウンドトリップ**
    - **検証対象: 要件 2.2, 2.3**
- [x] 4.3 DynamoDB 書き込みロジックの実装
    - `lambda/collector/dynamodb-writer.ts` を作成
    - SQS メッセージの from をスロットタイムとして使用
    - 冪等な条件付き書き込みを実装（attribute_not_exists）
    - 要件*: 3.1, 3.2, 3.3, 3.4*
- [x] 4.4 DynamoDB 書き込みのプロパティテストを作成
    - **プロパティ 9: 冪等な DynamoDB 書き込み**
    - **検証対象: 要件 3.4**
- [x] 4.5 Collector Lambda ハンドラーの統合
    - `lambda/collector/index.ts` を作成

        - 環境変数読み込みを実装（AI_API_BASE_URL, AI_METRICS_TABLE_NAME）
        - SQS イベントからメッセージを取得
        - 各コンポーネントを統合
        - エラーハンドリングとロギングを実装
        - 要件: *7.1, 7.2, 7.4*
- [x] 5. チェックポイント - すべてのテストが通ることを確認
    - すべてのテストが通ることを確認し、質問があればユーザーに確認する
- [x] 6. CDK スタックの実装
    - [x] 6.1 DynamoDB テーブルの定義
        - AiResponseMetrics テーブルを定義
        - パーティションキー（metricName）とソートキー（slotTime）を設定
        - 要件: *6.1*
    - [x] 6.2 SQS キューの定義
        - スケジューリングメッセージ用キューを定義
        - 要件: *6.2*
    - [x] 6.3 Sender Lambda の定義

    - Lambda 関数を定義（Node.js 20.x）
    - SQS キューへの送信権限を付与
    - 環境変数を設定（QUEUE_URL）
    - 要件*: 6.3, 6.6*
- [x] 6.4 Collector Lambda の定義
    - Lambda 関数を定義（Node.js 20.x）
    - DynamoDB への書き込み権限を付与
    - SQS トリガーを設定
    - 環境変数を設定（AI_API_BASE_URL, AI_METRICS_TABLE_NAME）
    - 要件*: 6.4, 6.6*
- [x] 6.5 EventBridge ルールの定義
    - 1 分間隔のスケジュールルールを定義
    - Sender Lambda をターゲットに設定
    - 要件*: 6.5*
- [x] 6.6 CDK アプリケーションエントリポイントの作成
    - `bin/app.ts` を作成
    - スタックをインスタンス化
    - 要件*: 6.1*

- [x] 7. 最終チェックポイント - すべてのテストが通ることを確認
    - すべてのテストが通ることを確認し、質問があればユーザーに確認する

```
import * as cdk from 'aws-cdk-lib';
import * as dynamodb from 'aws-cdk-lib/aws-dynamodb';
import * as sqs from 'aws-cdk-lib/aws-sqs';
import * as lambda from 'aws-cdk-lib/aws-lambda';
import * as lambdaEventSources from 'aws-cdk-lib/aws-lambda-event-sources';
import * as events from 'aws-cdk-lib/aws-events';
import * as targets from 'aws-cdk-lib/aws-events-targets';
import * as apigateway from 'aws-cdk-lib/aws-apigateway';
import { Construct } from 'constructs';
import * as path from 'path';

export class DataCollectionSchedulerStack extends cdk.Stack {
constructor(scope: Construct, id: string, props?: cdk.StackProps) {
super(scope, id, props);
```

```
    // Mock API Lambda - simulates AI
    const mockApiLambda = new lambda.F
        runtime: lambda.Runtime.NODEJS
        handler: 'index.handler',
        code: lambda.Code.fromAsset(pa
        timeout: cdk.Duration.seconds(
    });

    // API Gateway for Mock API
    const mockApi = new apigateway.Res
        restApiName: 'AI Employee Mock
        description: 'Mock API for tes
        deployOptions: {
            stageName: 'prod',
        },
    });

    // Add /response_count endpoint
    const responseCountResource = mock
    responseCountResource.addMethod('G

    // Add /health endpoint
    const healthResource = mockApi.roo
    healthResource.addMethod('GET', ne

    // DynamoDB Table for AI Response
    const metricsTable = new dynamodb.
        partitionKey: {
            name: 'metricName',
            type: dynamodb.AttributeTy
```

```
        },
        sortKey: {
            name: 'slotTime',
            type: dynamodb.AttributeTy
        },
        billingMode: dynamodb.BillingM
        removalPolicy: cdk.RemovalPoli
    });

    // SQS Queue for scheduling messag
    const schedulingQueue = new sqs.Qu
        visibilityTimeout: cdk.Duratio
        retentionPeriod: cdk.Duration.
    });

    // Sender Lambda - generates 12 de
    const senderLambda = new lambda.Fu
        runtime: lambda.Runtime.NODEJS
        handler: 'index.handler',
        code: lambda.Code.fromAsset(pa
        environment: {
            QUEUE_URL: schedulingQueue
        },
        timeout: cdk.Duration.seconds(
    });

    // Grant Sender Lambda permission
    schedulingQueue.grantSendMessages(

    // Collector Lambda - fetches data
```

```
    const collectorLambda = new lambda
        runtime: lambda.Runtime.NODEJS
        handler: 'index.handler',
        code: lambda.Code.fromAsset(pa
        environment: {
            AI_API_BASE_URL: mockApi.u
            AI_METRICS_TABLE_NAME: met
        },
        timeout: cdk.Duration.seconds(
    });

    // Grant Collector Lambda permissi
    metricsTable.grantWriteData(collec

    // Add SQS trigger to Collector La
    collectorLambda.addEventSource(
        new lambdaEventSources.SqsEven
            batchSize: 1,
        })
    );

    // EventBridge rule to trigger Sen
    const scheduleRule = new events.Ru
        schedule: events.Schedule.rate
        description: 'Triggers Sender
    });

    // Add Sender Lambda as target for
    scheduleRule.addTarget(new targets
```

```
        // Output the Mock API URL
        new cdk.CfnOutput(this, 'MockApiUr
            value: mockApi.url,
            description: 'Mock AI Employee
        });

        // Output the DynamoDB Table Name
        new cdk.CfnOutput(this, 'MetricsTa
            value: metricsTable.tableName,
            description: 'DynamoDB table n
        });
    }

}
```