デベロッパーガイド

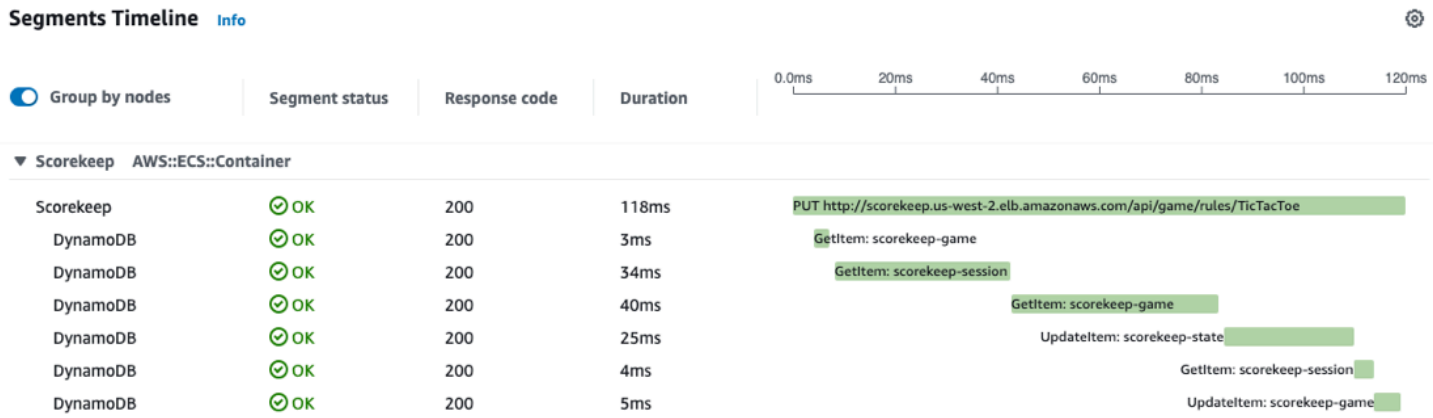
AWS X-Ray

AWS X-Ray: デベロッパーガイド

Table of Contents

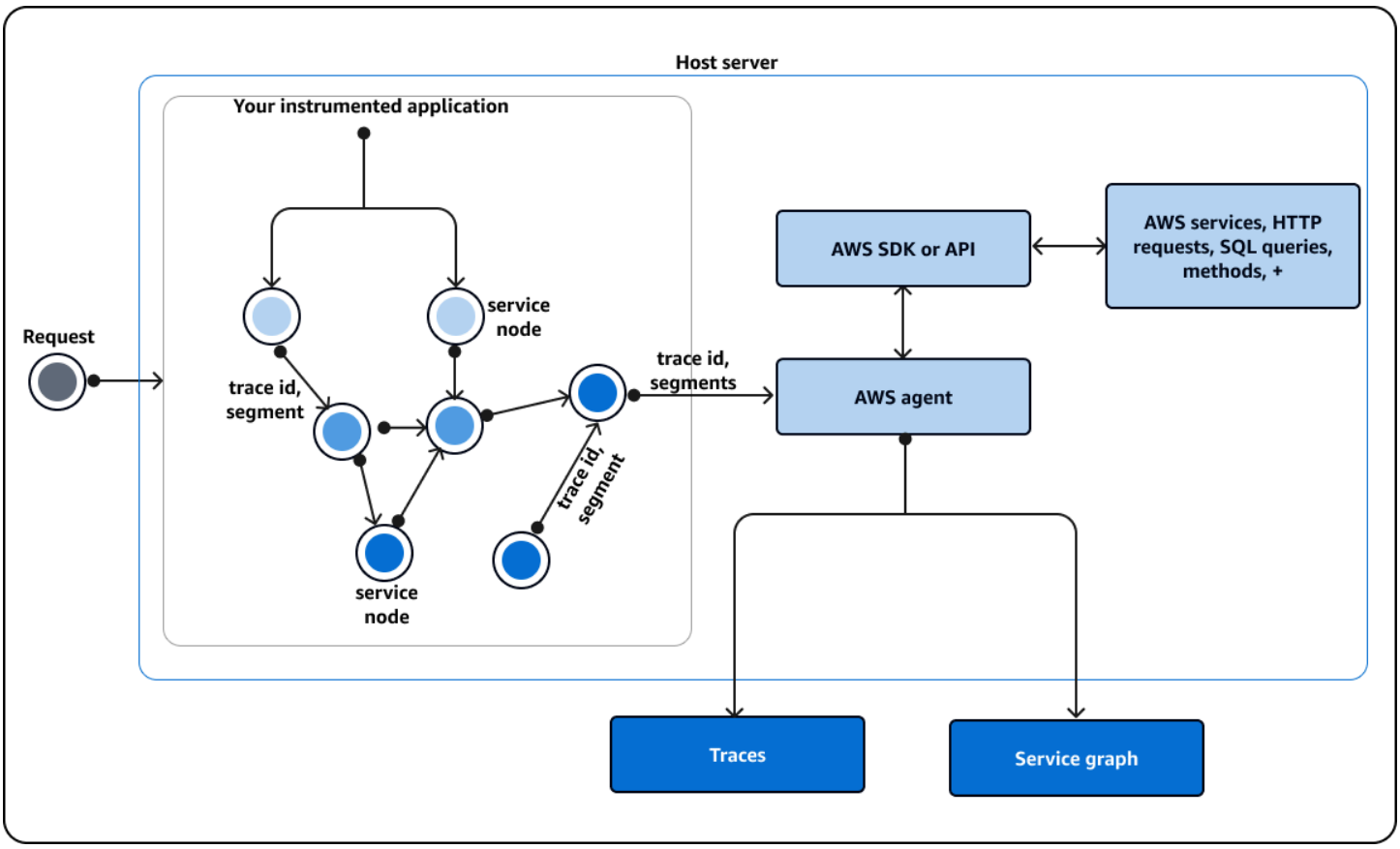
とは AWS X-Ray

AWS X-Ray は、アプリケーションが処理するリクエストに関するデータを収集するサービスであり、そのデータを表示、フィルタリング、インサイトを取得して、最適化の問題と機会を特定するために使用できるツールを提供します。アプリケーションへのトレースされたリクエストについては、リクエストとレスポンスだけでなく、アプリケーションがダウンストリーム AWS リソース、マイクロサービス、データベース、ウェブ APIs に対して行う呼び出しに関する詳細情報も確認できます。



AWS X-Ray は、X-Ray と既に統合されているアプリケーションの使用に加えて、 AWS のサービスアプリケーションからトレースを受け取ります。アプリケーションをインストルメント化するには、アプリケーション内の受信および送信されるリクエストやその他のイベントのトレースデータと、各リクエストに関するメタデータを送信する必要があります。多くの計測シナリオで必要となるのは、設定の変更のみです。たとえば、Java アプリケーション AWS のサービス が行うすべての受信 HTTP リクエストとダウンストリーム呼び出しを計測できます。X-Ray トレースシング用アプリケーションを計測するために使用できる SDK、エージェント、およびツールがいくつかあります。詳細は、アプリケーションの計測 を参照してください。

AWS のサービス X-Ray と統合された は、受信リクエストにトレースヘッダーを追加したり、X-Ray にトレースデータを送信したり、X-Ray デーモンを実行したりできます。たとえば、 AWS Lambda はリクエストに関するトレースデータを Lambda 関数に送信し、ワーカーで X-Ray デーモンを実行して X-Ray SDK の使用を簡素化できます。



各クライアントSDKは、トレースデータを直接X-Rayに送らずに、UDPトラフィックをリッスンしているデーモンプロセスにJSONセグメントドキュメントを送ります。X-Ray デーモン は、セグメントをキューにバッファリングし、バッチでX-Rayにアップロードします。デーモンは Linux、Windows、macOS で使用でき、 および AWS Elastic Beanstalk AWS Lambda プラットフォームに含まれています。

X-Ray は、クラウドアプリケーションを強化する AWS リソースからのトレースデータを使用して、詳細なトレースマップを生成します。トレースマップには、フロントエンドサービスが呼び出してリクエストを処理しデータを維持するクライアント、フロントエンドサービス、バックエンドサービスが表示されます。トレースマップを使用して、ボトルネック、レイテンシーのスパイク、その他の問題を識別して解決し、アプリケーションのパフォーマンスを向上させます。


scorekeep-move
DynamoDB Table
Client
Scorekeep
ECS Container
SNS
SNS
scorekeep-session
DynamoDB Table

X-Ray の使用開始

X-Ray を使用するには、次のステップを実行します。

1. アプリケーションを計測すると、X-Ray はアプリケーションがリクエストを処理する方法を追跡できます。

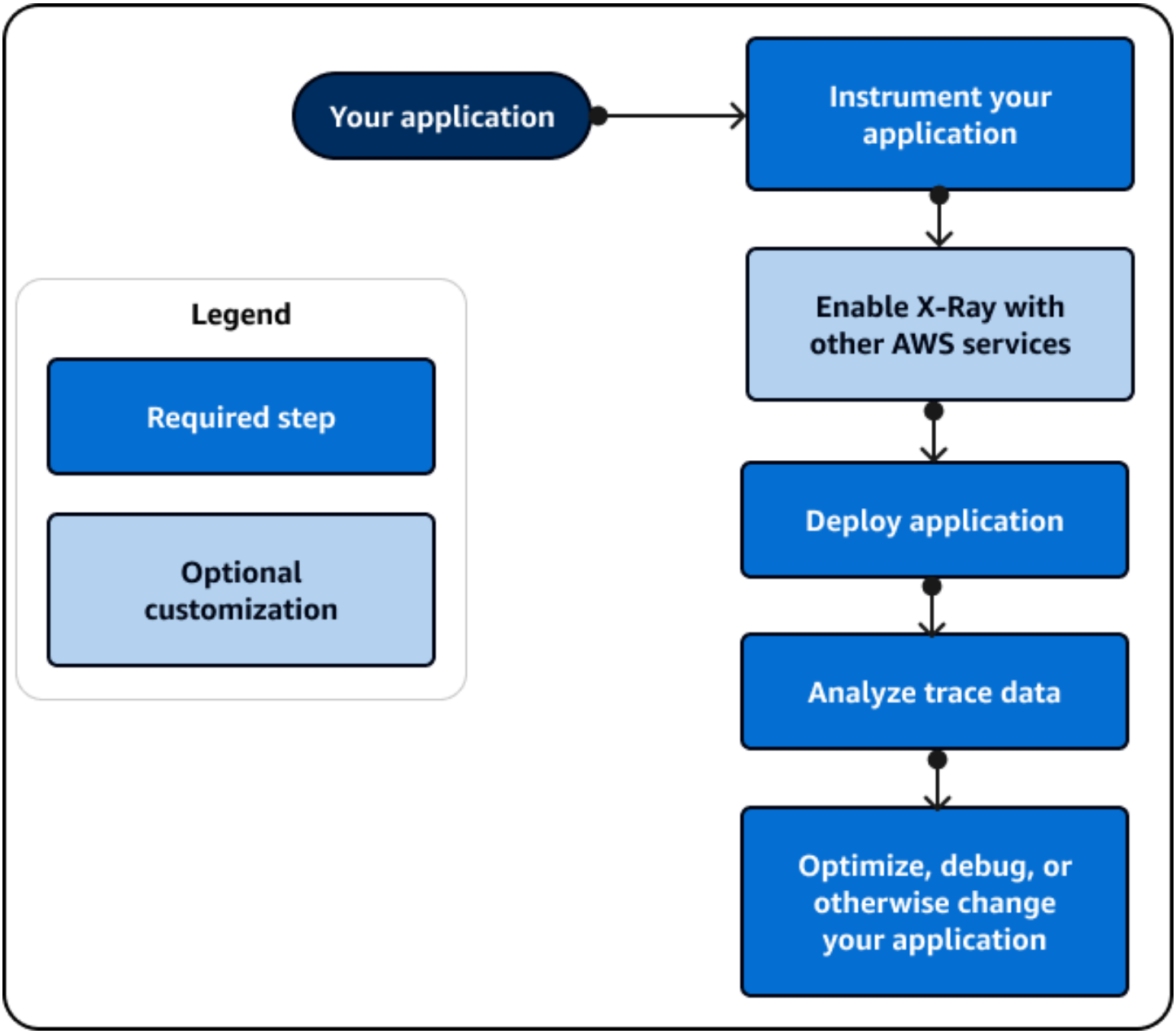
   - X-Ray SDK、X-Ray API、ADOT、または CloudWatch Application Signals を使用して、トレースデータを X-Ray に送信します。使用するインターフェイスの詳細については、「インターフェイスの選択」を参照してください。

   計測の詳細については、「のアプリケーションの計測 AWS X-Ray」を参照してください。

2. （オプション) X-Ray と統合 AWS のサービス する他の と連携するように X-Ray を設定します。トレースをサンプリングして、受信リクエストにヘッダーを追加し、エージェントまたはコレクターを実行し、トレースデータを X-Ray に自動的に送信できます。詳細については、「他の AWS X-Ray との統合 AWS のサービス」を参照してください。
3. 計測したアプリケーションをデプロイします。アプリケーションがリクエストを受信すると、X-Ray SDK はトレース、セグメント、サブセグメントのデータを記録します。このステップでは、IAM ポリシーをセットアップして、エージェントまたはコレクターをデプロイする必要が生じる場合もあります。

   - AWS Distro for OpenTelemetry (ADOT) SDK と CloudWatch エージェントを使用してアプリケーションをさまざまなプラットフォームにデプロイするスクリプトの例については、「Application Signals デモスクリプト」を参照してください。
   - X-Ray SDK と X-Ray デーモンを使用してアプリケーションをデプロイするスクリプトの例については、「AWS X-Ray サンプルアプリケーション」を参照してください。
4. (オプション) コンソールを開いてデータを表示し分析します。トレースマップ、サービスマップなどの GUI 表現を表示してアプリケーションの機能を確認できます。コンソールのグラフィカルな情報を使用して、アプリケーションを最適化、デバッグ、理解します。コンソールの選択の詳細については、「コンソールを使用する」を参照してください。

次の図は、X-Ray の使用を開始する方法を示しています。



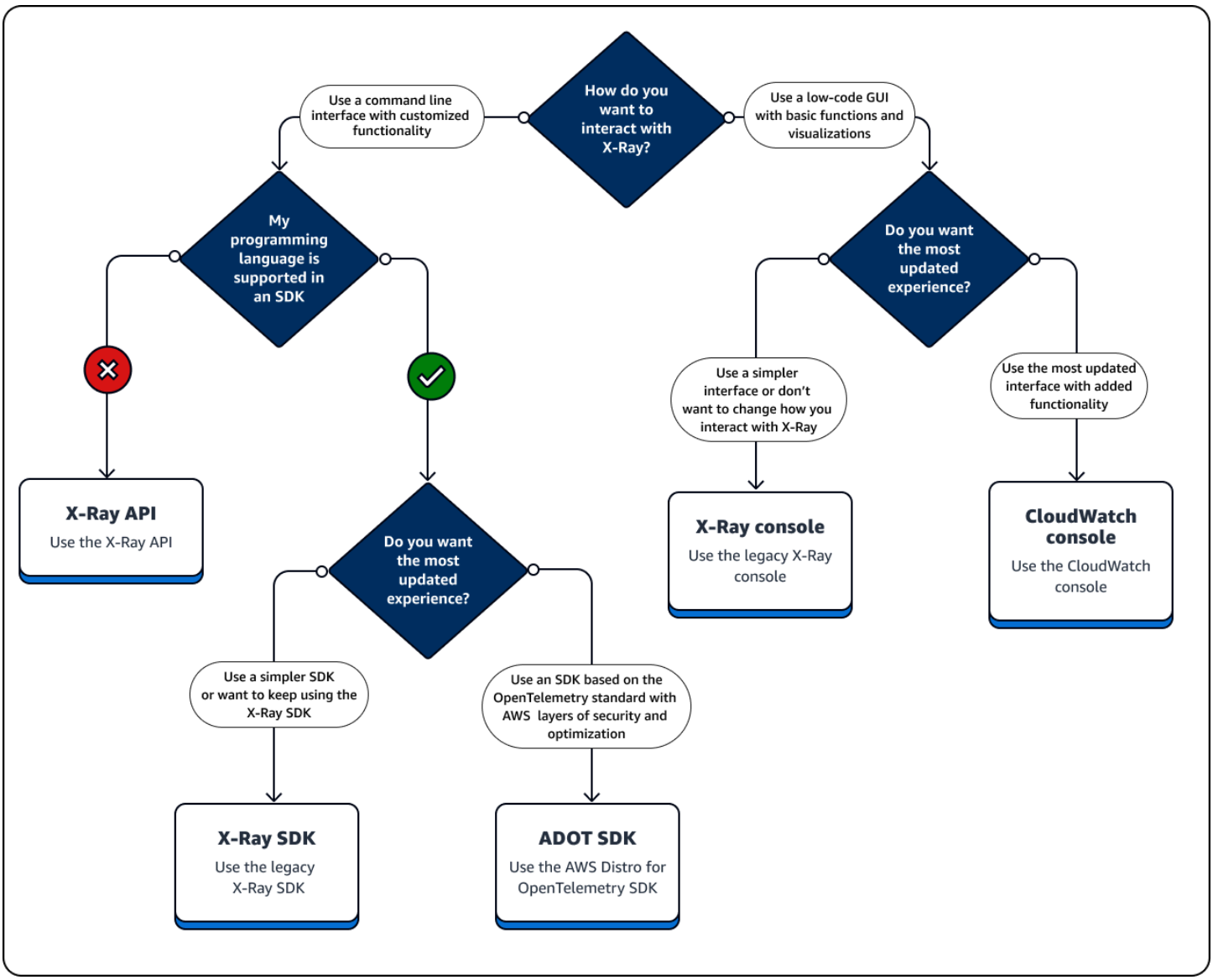
コンソールで使用できるデータとマップの例については、トレースデータを生成するために既に計測済みのサンプルアプリケーションを起動してください。数分で、トラフィックを生成し、セグメントを X-Ray に送信し、トレースとサービスマップを確認できます。

インターフェイスの選択

AWS X-Ray は、アプリケーションの仕組みや、他の のサービスやリソースとやり取りする仕組みに関するインサイトを提供します。アプリケーションの計測または設定を実行すると、X-Ray はアプリケーションがリクエストを処理するときのトレースデータを収集します。このトレースデータを分析すると、パフォーマンスの問題を特定し、エラーのトラブルシューティングを行い、リソースを最適化できます。このガイドでは、以下のガイドラインにより X-Ray を操作する方法について説明します。

- すぐに使用を開始する場合や、構築済みの視覚化を使用して基本的なタスクを実行できる AWS Management Console 場合は、 を使用します。
  - X-Ray コンソールのすべての機能を含む最新のユーザーエクスペリエンスを使用する場合は、Amazon CloudWatch コンソールを選択します。
  - より単純なインターフェイスを使用、または X-Ray の操作方法を変更しない場合は、X-Ray コンソールを使用します。
- が提供するよりも多くのカスタムトレース、モニタリング、またはログ記録機能が必要な場合は、SDK を使用します AWS Management Console 。
  - AWS セキュリティと最適化のレイヤーを追加したオープンソース OpenTelemetry SDK に基づく、ベンダーに依存しない SDK を使用する場合は、ADOT SDK を選択します。
  - より単純な SDK が必要な場合、またはアプリケーションコードを更新しない場合は、X-Ray SDK を選択します。
- SDK がご使用のアプリケーションのプログラミング言語をサポートしていない場合は、X-Ray API オペレーションを使用します。

次の図は、X-Ray の操作方法の選択に役立ちます。



インターフェイスタイプを調べる

- SDK を使用する
- コンソールを使用する
- X-Ray API を使用する

SDK を使用する

コマンドラインインターフェイスを使用する場合、または AWS Management Consoleで利用可能なものよりも多くのカスタムトレース、モニタリング、またはログ機能が必要な場合は、SDK を使

用します。 AWS SDK を使用して、X-Ray APIs を使用するプログラムを開発することもできます。AWS Distro for OpenTelemetry (ADOT) SDK または X-Ray SDK のいずれかを使用できます。

SDK を使用する場合は、アプリケーションを計測するときと、コレクターまたはエージェントを設定するときに、ワークフローにカスタマイズを追加できます。SDK を使用すると、 AWS Management Consoleでは実行できない以下のタスクを実行できます。

- カスタムメトリクスの公開 - 1 秒までの高解像度でメトリクスをサンプリングして、複数のディメンションを使用しメトリクスに関する情報を追加し、データポイントを統計セットに集約します。
- コレクターのカスタマイズ - レシーバー、プロセッサ、エクスポーター、コネクタなど、コレクターの設定の任意の部分をカスタマイズします。
- 計測のカスタマイズ - セグメントとサブセグメントをカスタマイズして、カスタムキーと値のペアを属性として追加し、カスタムメトリクスを作成します。
- プログラムでサンプリングルールを作成および更新します。

AWS セキュリティと最適化のレイヤーを追加した標準化ADOTされた SDK を柔軟に使用する場合は、 OpenTelemetry SDK を使用します。 AWS Distro for OpenTelemetry (ADOT) SDK はベンダーに依存しないパッケージで、コードを再計測することなく、他のベンダーや非AWS サービスのバックエンドと統合できます。

X-Ray SDK を既に使用していて、 AWS バックエンドのみと統合し、X-Ray またはアプリケーションコードの操作方法を変更しない場合は、X-Ray SDK を使用します。

各機能の詳細については、「Distro for OpenTelemetry AWS と X-Ray SDKs の選択」を参照してください。

ADOT SDK を使用する

ADOT SDK は、バックエンドサービスにデータを送信する一連のオープンソース APIs、ライブラリ、エージェントです。 ADOTは AWS、 でサポートされ、複数のバックエンドとエージェントと統合され、OpenTelemetryコミュニティによって管理される多数のオープンソースライブラリを提供します。ADOT SDK を使用してご使用のアプリケーションを計測して、ログ、メタデータ、メトリクス、トレースを収集します。ADOT を使用してサービスをモニタリングし、CloudWatch のメトリクスに基づいてアラームを設定することもできます。

ADOT SDK を使用する場合は、エージェントとの組み合わせで、次のオプションがあります。

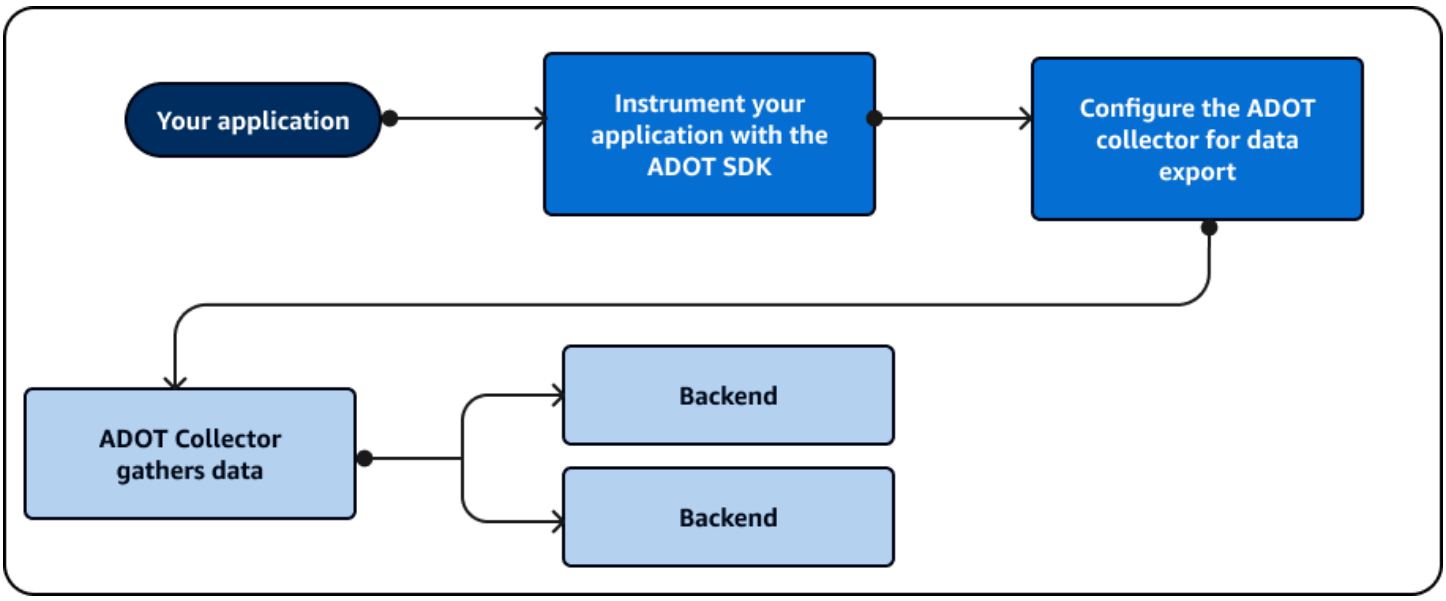
- CloudWatch エージェントで ADOT SDK を使用する - 推奨。

- ADOT コレクターで ADOT SDK を使用する – セキュリティと最適化の AWS レイヤーでベンダーに依存しないソフトウェアを使用する場合は推奨されます。

ADOT SDK を使用する場合は、以下を実行します。

- ADOT SDK を使用してご使用のアプリケーションを計測します。詳細については、「ADOT 技術ドキュメント」の、ご使用のプログラミング言語のドキュメントを参照してください。
- ADOT Collector を設定して、収集データの送信先を指定します。

ADOT コレクターは、データを受信すると、ADOT設定で指定したバックエンドに送信します。 は AWS、次の図に示すように、 以外のベンダーを含む複数のバックエンドにデータを送信ADOTできます。



AWS は定期的に を更新ADOTして機能を追加し、OpenTelemetry フレームワークに合わせます。ADOT の更新および今後の開発計画は一般公開ロードマップの一部です。ADOT は以下を含むいくつかのプログラミング言語をサポートしています。

- Go
- Java
- JavaScript
- Python
- .NET
- Ruby

- PHP

Python を使用する場合、ADOT はアプリケーションを自動的に計測できます。の使用を開始するにはADOT、https://aws-otel.github.io/docs/introductionAWS 「 Distro for OpenTelemetry Collector の概要と使用開始」を参照してください。

X-Ray SDK を使用する

X-Ray SDK は、バックエンドサービスに AWS データを送信する一連の AWS APIsとライブラリです。X-Ray SDK を使用してアプリケーションを計測しトレースデータを収集します。X-Ray SDK を使用したログまたはメトリクスデータの収集はできません。

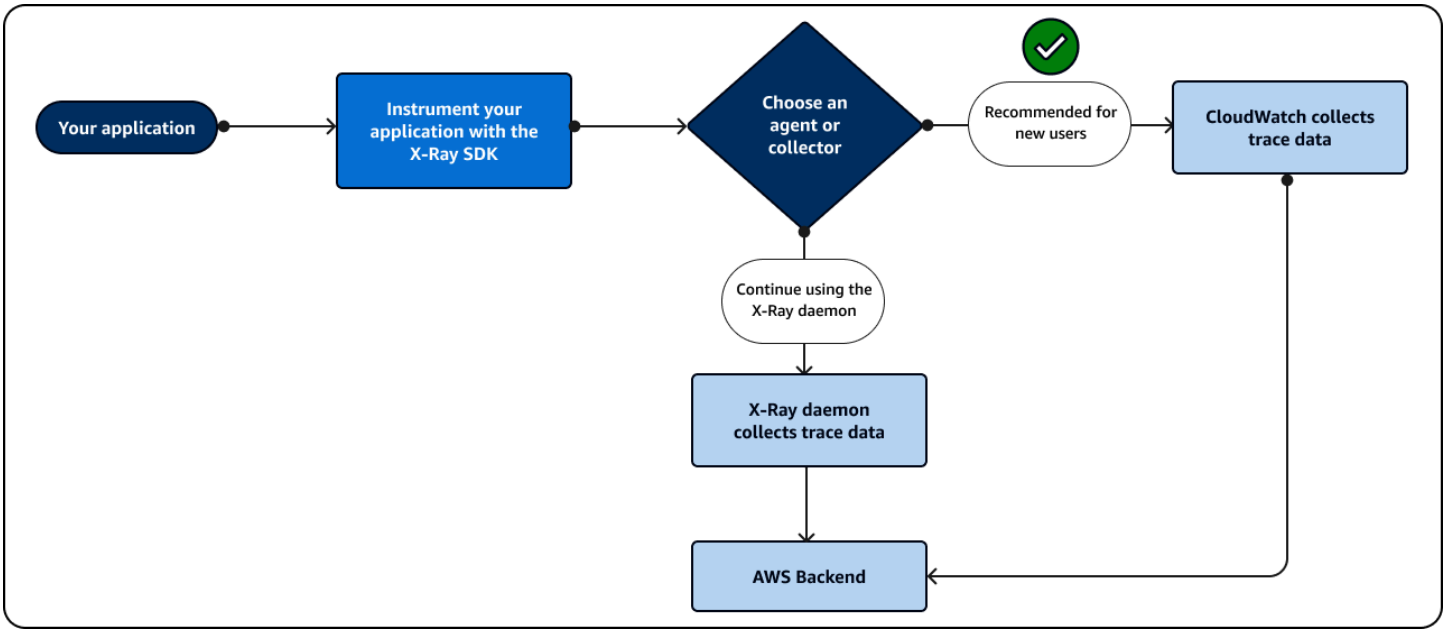
X-Ray SDK を使用する場合は、エージェントとの組み合わせで、次のオプションがあります。

- AWS X-Ray デーモン で X-Ray SDK を使用する - アプリケーションコードを更新しない場合はこれを使用します。
- CloudWatch エージェントで X-Ray SDK を使用する - (推奨) CloudWatch エージェントは X-Ray SDK と互換性があります。

SDK を使用する場合は、以下を実行します。

- X-Ray SDK を使用してご使用のアプリケーションを計測します。
- コレクターを設定して、収集データの送信先を指定します。CloudWatch エージェントまたは X-Ray デーモンを使用すると、トレース情報を収集できます。

コレクターまたはエージェントがデータを受信すると、エージェント設定で指定した AWS バックエンドに送信されます。次の図に示すように、X-Ray SDK は AWS バックエンドにのみデータを送信できます。



Java を使用する場合は、X-Ray SDK を使用してアプリケーションを自動的に計測できます。X-Ray SDK の使用を開始する場合は、次のプログラミング言語に関連付けられているライブラリを参照してください。

- Go
- Java
- Node.js
- Python
- .NET
- Ruby

コンソールを使用する

グラフィカルユーザーインターフェイス (GUI) で必要なコーディングを最小限にする場合は、コンソールを使用します。X-Ray を初めて使用するユーザーは、構築済みの視覚化を使用すると、基本的なタスクをすばやく開始できます。コンソールから以下の内容を直接実行できます。

- X-Ray を有効にします。
- アプリケーションのパフォーマンスについてハイレベルの概要を表示します。
- アプリケーションのヘルスステータスを確認します。
- ハイレベルのエラーを特定します。

- 基本的トレースの概要を表示します。

https://console.aws.amazon.com/cloudwatch/ の Amazon CloudWatch コンソールまたは https://console.aws.amazon.com/xray/home の X-Ray コンソールを使用して、X-Ray を操作できます。

Amazon CloudWatch コンソールを使用する

CloudWatch コンソールには、X-Ray コンソールから再設計された、使いやすい新規 X-Ray 機能が含まれています。CloudWatch コンソールを使用する場合は、X-Ray トレースデータと共に、CloudWatch ログとメトリクスを表示できます。CloudWatch コンソールを使用して以下が含まれるデータを表示および分析します。

- X-Ray トレース - アプリケーションがリクエストを処理する際、アプリケーションに関連付けられたトレースを表示、分析、フィルタリングします。これらのトレースを使用して、高レイテンシーの検出、エラーのデバッグ、アプリケーションワークフローの最適化を実行します。トレースマップとサービスマップを表示して、ご使用のアプリケーションワークフローの視覚的表現を確認します。
- ログ - ご使用のアプリケーションが生成するログを表示、分析、フィルタリングします。ログを使用してエラーをトラブルシューティングし、特定のログ値に基づくモニタリングを設定します。
- メトリクス - ご使用のリソースが出力するメトリクスを使用、または独自のメトリクスを作成して、アプリケーションのパフォーマンスを測定およびモニタリングします。これらのメトリクスをグラフとチャートで表示します。
- ネットワークとインフラストラクチャのモニタリング – コンテナ化されたアプリケーション、その他の AWS サービス、クライアントなど、インフラストラクチャの停止やヘルスとパフォーマンスについて主要なネットワークをモニタリングします。
- 次の X-Ray コンソールを使用するセクションに記載されている X-Ray コンソールのすべての機能。

CloudWatch コンソールの詳細については、「CloudWatch の開始方法」を参照してください。

https://console.aws.amazon.com/cloudwatch/ で Amazon CloudWatch コンソールにログインします。

X-Ray コンソールを使用する

X-Ray コンソールはアプリケーションリクエストの分散トレースを提供しています。よりシンプルなコンソールエクスペリエンスが必要な場合、またはアプリケーションコードを更新しない場合

は、X-Ray コンソールを使用します。 AWS は X-Ray コンソールを開発していません。X-Ray コンソールには計測されたアプリケーションのための以下の機能が含まれています。

- Insights - アプリケーションのパフォーマンスの異常を自動的に検出して根本的な原因を見つけます。Insights は、[Insights] の CloudWatch コンソールに含まれています。詳細については、X-Ray コンソールを使用する の「X-Ray Insights を使用する」を参照してください。
- サービスマップ - アプリケーションのグラフィカル構造、およびクライアント、リソース、サービス、依存関係との接続を表示します。
- トレース - アプリケーションがリクエスト処理の際に生成するトレースの概要を参照します。トレースデータを使用して、HTTP レスポンスや応答時間などの、基本的なメトリクスに対するアプリケーションのパフォーマンスを把握します。
- 分析 - 応答時間の分散のグラフを使用して、トレースデータを解釈、調査、分析します。
- 設定 - カスタマイズしたトレースを作成して次のデフォルト設定を変更します。
  - サンプリング - トレース情報のためのアプリケーションサンプリング頻度を定義するルールを作成します。詳細については、X-Ray コンソールを使用する の「サンプリングルールを設定する」を参照してください。
  - 暗号化 - AWS Key Management Serviceを使用して監査または無効化できるキーで、保管中のデータを暗号化します。
  - グループ - フィルター式を使用して、URL の名前や応答時間などの一般的な特徴量を持つトレースのグループを定義します。詳細については、「グループを設定する」を参照してください。

https://console.aws.amazon.com/xray/home で X-Ray コンソールにログインします。

X-Ray コンソールを使用する

X-Ray コンソールを使用して、アプリケーションが処理するリクエストのサービスと関連するトレースのマップを表示し、トレースが X-Ray に送信される方法に影響するグループとサンプリングルールを設定します。

> **Note**
>
> X-Ray Service マップと CloudWatch ServiceLens マップは、Amazon CloudWatch コンソール内の X-Ray トレースマップに統合済みです。CloudWatch コンソールを開き、左側のナビゲーションペインから [X-Ray トレース] の下の [トレースマップ] を選択します。

CloudWatch には、アプリケーションサービス、クライアント、Synthetics Canary、サービスの依存関係を検出してモニタリングできる Application Signals が含まれるようになりました。Application Signals を使用すると、サービスのリストやビジュアルマップを確認したり、サービスレベル目標 (SLO) に基づくヘルスメトリクスを表示したり、ドリルダウンして相関関係のある X-Ray トレースを確認したりして、より詳細なトラブルシューティングを行うことができます。

X-Ray コンソールの主要ページはトレースマップで、アプリケーションによって生成されたトレースデータから X-Ray が生成した JSON サービスグラフがビジュアルで表現されます。マップは、リクエストを処理するアカウント内の各アプリケーションのサービスノード、リクエストの送信元を示すアップストリームクライアントノード、リクエストの処理中にアプリケーションが使用するウェブサービスとリソースを示すダウンストリームサービスノードで構成されます。他にも、トレースとトレースの詳細を表示したり、グループやサンプリングルールを設定したりするためのページがあります。

X-Ray のコンソールエクスペリエンスを表示し、以下のセクションの CloudWatch コンソールと比較します。

X-Ray コンソールと CloudWatch コンソールの詳細

- X-Ray トレースマップの使用
- トレースとトレースの詳細の表示
- フィルター式の使用
- クロスアカウントトレース
- イベント駆動型アプリケーションのトレース
- レイテンシーヒストグラムの使用
- X-Ray インサイトの使用
- Analytics コンソールとのやり取り
- グループの設定
- サンプリングルールの設定
- コンソールのディープリンク

X-Ray トレースマップの使用

X-Ray トレースマップを表示して、エラーが発生しているサービス、高レイテンシーの接続、失敗したリクエストのトレースを識別します。

> **Note**
>
> CloudWatch には、アプリケーションサービス、クライアント、Synthetics Canary、サービスの依存関係を検出してモニタリングできる Application Signals が含まれるようになりました。Application Signals を使用すると、サービスのリストやビジュアルマップを確認したり、サービスレベル目標 (SLO) に基づくヘルスメトリクスを表示したり、ドリルダウンして相関関係のある X-Ray トレースを確認したりして、より詳細なトラブルシューティングを行うことができます。
> X-Ray サービスマップと CloudWatch ServiceLens マップは、Amazon CloudWatch コンソール内の X-Ray トレースマップに結合されました。CloudWatch コンソールを開き、左側のナビゲーションペインから [X-Ray トレース] の下の [トレースマップ] を選択します。

トレースマップの表示

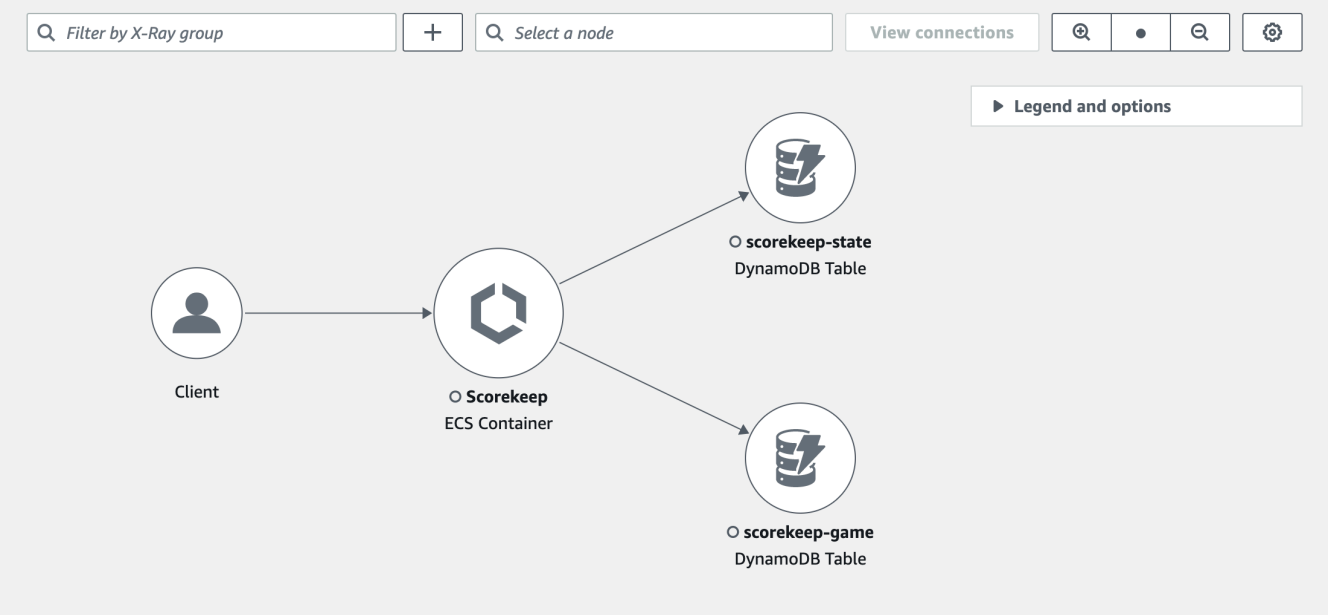
トレースマップは、アプリケーションによって生成されたトレースデータをビジュアルに表現したものです。リクエストを処理するサービスノード、リクエストの送信元を示すアップストリームクライアントノード、リクエストの処理中にアプリケーションが使用するウェブサービスとリソースを示すダウンストリームサービスノードがマップに表示されます。

トレースマップには、Amazon SQS と Lambda を使用するイベント駆動型アプリケーション全体のトレースの接続されたビューが表示されます。詳細については、「イベント駆動型アプリケーションのトレース」を参照してください。トレースマップはクロスアカウントトレーシングもサポートし、複数のアカウントのノードを 1 つのマップに表示します。

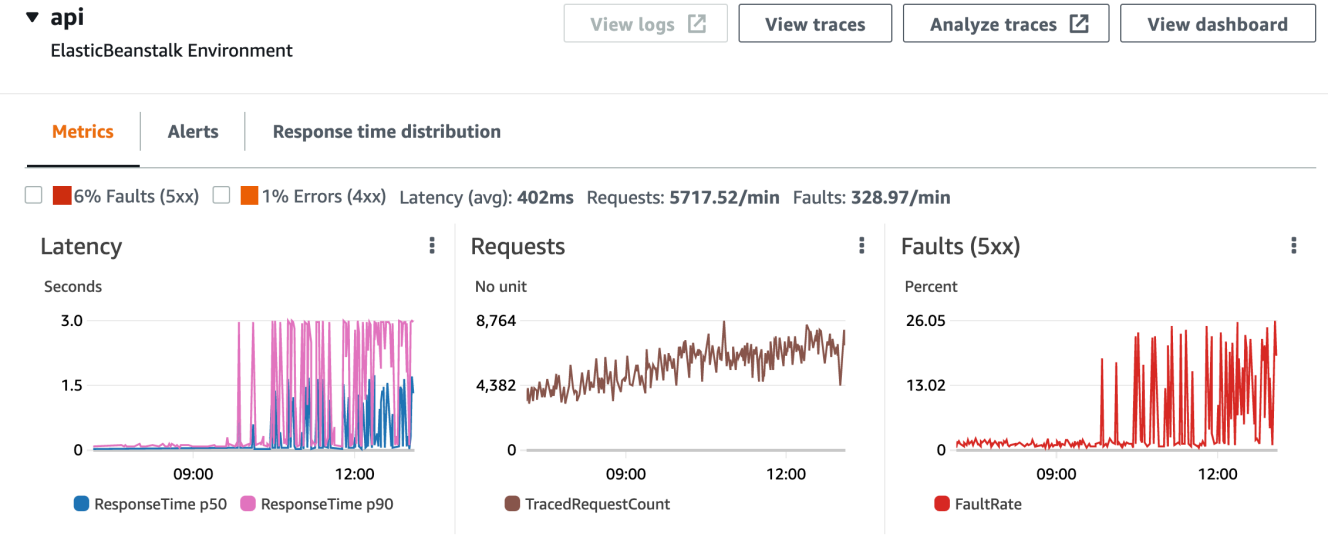
CloudWatch console

CloudWatch コンソールを使用してトレースマップを表示するには

1. CloudWatch コンソールを開きます。左側のナビゲーションペインの [X-Ray トレース] セクションで [トレースマップ] を選択します。



2. そのノードのリクエスト、または 2 つのノード間のエッジを表示するサービスノードを選択して、やり取りされた接続のリクエストを表示します。
3. メトリクス、アラート、応答時間の分布のタブなど、追加情報がトレースマップの下に表示されます。[メトリクス] タブでは、各グラフ内の範囲を選択してドリルダウンして詳細を表示するか、[障害] または [エラー] オプションを選択してトレースをフィルタリングします。[応答時間の分布] タブでは、応答時間でトレースをフィルタリングするグラフ内の範囲を選択します。



4. [トレースを表示] を選択してトレースを表示するか、フィルターが適用されている場合は [フィルタリングされたトレースの表示] を選択します。
5. [ログを表示] を選択すると、選択したノードに関連付けられた CloudWatch ログが表示されます。すべてのトレースマップノードがログの表示をサポートしているわけではありま

せん。詳細については、「CloudWatch ログのトラブルシューティング」を参照してください。

トレースマップでは、各ノード内の問題を色分けして示します。

- 赤は、サーバー障害 (500 系のエラー)
- 黄は、クライアントエラー (400 系のエラー)
- 紫は、スロットリングエラー (429 リクエストが多すぎる)

トレースマップが大きい場合は、画面のコントロールまたはマウスを使用して、マップを拡大/縮小したり移動したりします。

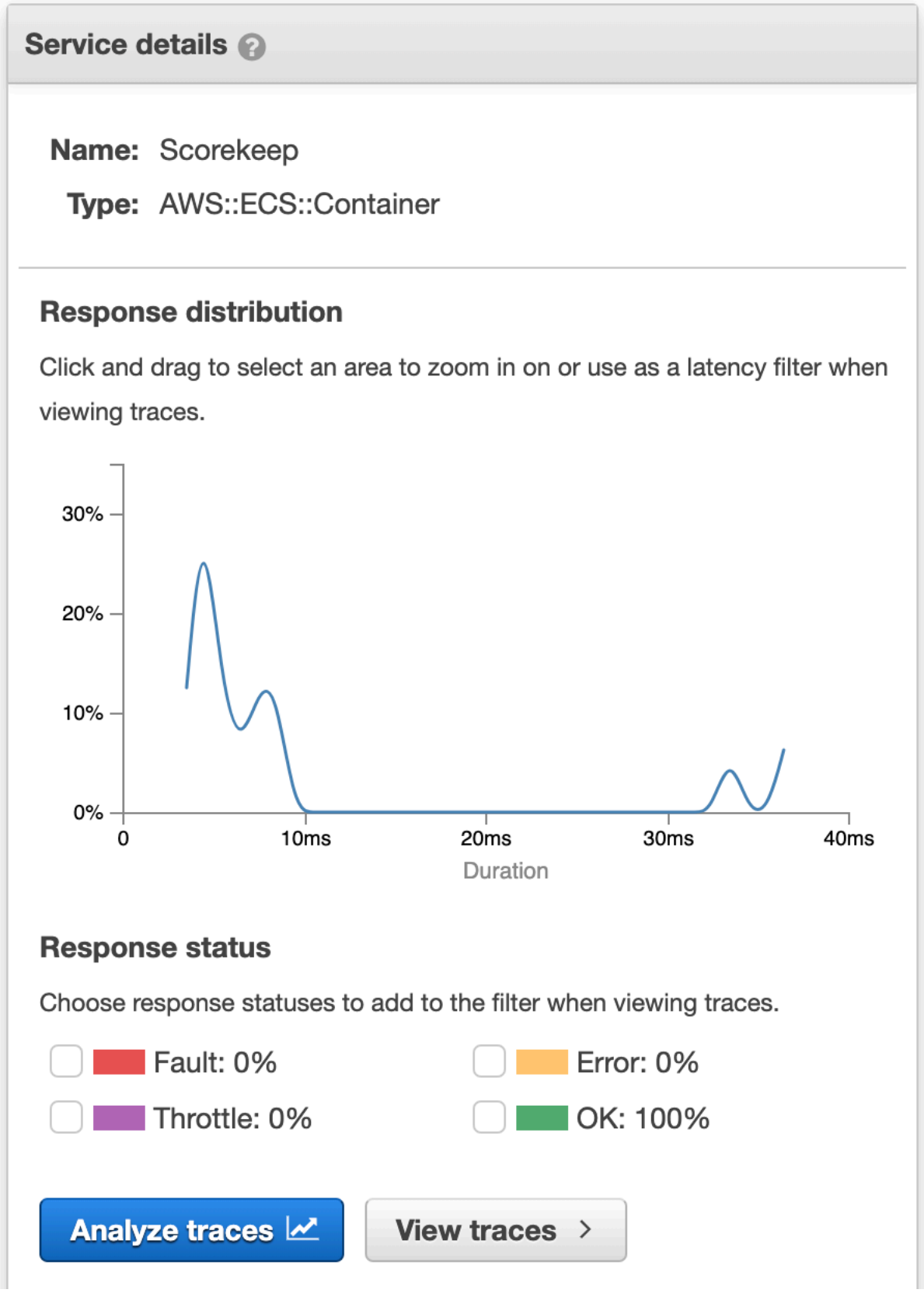
X-Ray console

サービスマップを表示するには

1. [X-Ray console (X-Ray コンソール)] を開きます。デフォルトでは、サービスマップが表示されます。左側のナビゲーションペインで [サービスマップ] を選択することもできます。

2. そのノードのリクエスト、または 2 つのノード間のエッジを表示するサービスノードを選択して、やり取りされた接続のリクエストを表示します。

3. レスポンスディストリビューションヒストグラムを使用して、期間ごとにトレースをフィルタリングし、トレースを表示するステータスコードを選択します。[View traces (トレースの表示)] を選択し、フィルタ式を適用してトレースリストを開きます。


Service details
Name: Scorekeep
Type: AWS::ECS::Container
Response distribution
Click and drag to select an area to zoom in on or use as a latency filter when viewing traces.
30%
20%
10%
0%
0
10ms
20ms
30ms
40ms
Duration
Response status
Choose response statuses to add to the filter when viewing traces.
Fault: 0%
Error: 0%
Throttle: 0%
OK: 100%
Analyze traces
View traces

このサービスマップは、各ノードの状態をエラーと障害に対する正常な呼び出しの比率に基づいて色分けしたものです。

- 緑は、正常な呼び出し
- 赤は、サーバー障害 (500 系のエラー)
- 黄は、クライアントエラー (400 系のエラー)
- 紫は、スロットリングエラー (429 リクエストが多すぎる)

サービスマップが大きい場合は、画面のコントロールまたはマウスを使用して、マップを拡大/縮小したり移動したりします。

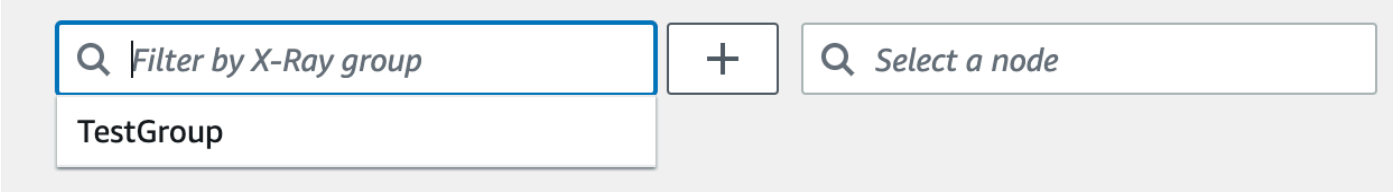
> **Note**
>
> X-Ray トレースマップは、最大 10,000 個のノードを表示できます。まれに、サービスノードの総数がこの制限を超えると、エラーが表示され、コンソールに完全なトレースマップを表示できない場合があります。

グループ別のトレースマップのフィルタリング

フィルター式を使用すると、グループに含めるトレースの基準を定義できます。次のステップに従って、トレースマップにその特定のグループを表示します。

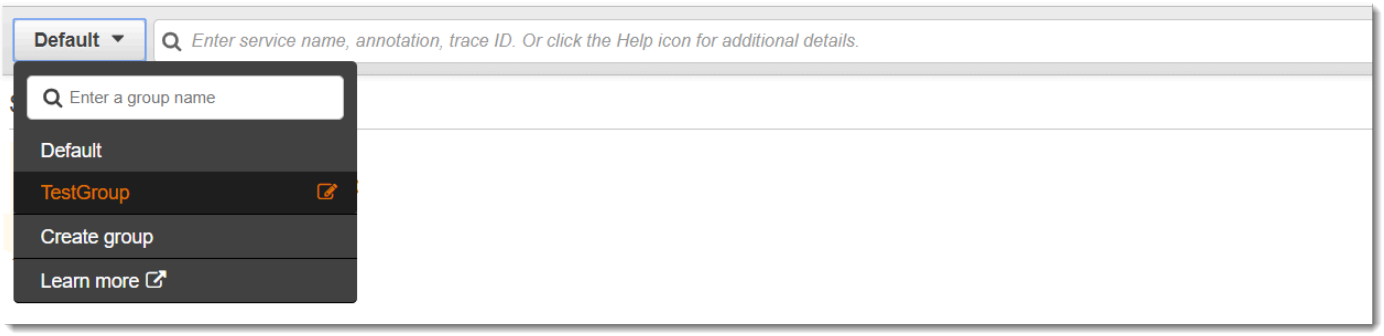
CloudWatch console

トレースマップの左上にあるグループフィルターからグループ名を選択します。



X-Ray console

検索バーの左側にあるドロップダウンメニューからグループ名を選択します。



これで、サービスマップがフィルタリングされ、選択したグループのフィルター式と一致するトレースが表示されます。

トレースマップの凡例とオプション

トレースマップには、マップ表示をカスタマイズするための凡例といくつかのオプションがあります。

CloudWatch console

マップの右上にある [凡例とオプション] ドロップダウンを選択します。以下のノード内に表示する内容を選択します。

- メトリクスには、選択した時間範囲の平均応答時間と 1 分あたりに送信されたトレース数が表示されます。
- ノードには、各ノード内のサービスアイコンが表示されます。

[設定] ペインから追加のマップ設定を選択します。このペインには、マップの右上にある歯車アイコンからアクセスできます。これらの設定には、各ノードのサイズを決定するために使用するメトリクスや、マップに表示する Canary の選択などがあります。

X-Ray console

サービスマップの凡例を表示するには、マップの右上にある [マップの凡例] リンクを選択します。トレースマップの右下で、次のようなサービスマップのオプションを選択できます。

- サービスアイコン 各ノード内の表示内容を切り替えて、サービスアイコンを表示するか、または選択した時間範囲の平均応答時間と 1 分あたりに送信されたトレース数を表示します。
- ノードサイズ : なし すべてのノードを同じサイズに設定します。

- ノードサイズ : ヘルス エラー、障害、スロットリングされたリクエストなど、影響を受けるリクエストの数に応じてノードのサイズを設定します。
- ノードサイズ : トラフィック リクエストの合計数に応じてノードのサイズを設定します。

トレースとトレースの詳細の表示

X-Ray コンソールの [トレース] ページを使用して、URL、レスポンスコード、トレース概要のその他のデータでトレースを検索します。トレースリストからトレースを選択すると、[トレースの詳細] ページには、選択したトレースに関係するサービスノードのマップと、トレースセグメントのタイムラインが表示されます。

トレースの表示

CloudWatch console

CloudWatch コンソールでトレースを表示するには

1. にサインイン AWS Management Console し、https://console.aws.amazon.com/cloudwatch/:://www.com」で CloudWatch コンソールを開きます。
2. 左側のナビゲーションペインで、[X-Ray トレース] を選択してから [トレース] を選択します。グループでフィルタリングすることも、フィルター式を入力することもできます。これにより、ページ下部の [トレース] セクションに表示されるトレースがフィルタリングされます。

   または、サービスマップを使用して特定のサービスノードに移動してから、トレースを表示できます。これにより、クエリを既に適用済みの [トレース] ページが開きます。
3. [クエリリファイナー] セクションでクエリを絞り込みます。共通属性でトレースをフィルタリングするには、[クエリを絞り込む] の横にある下矢印からオプションを選択します。オプションは以下のとおりです。

   - ノード - サービスノードでトレースをフィルタリングします。
   - リソース ARN - トレースに関連付けられたリソースでトレースをフィルタリングします。これらのリソースの例としては、Amazon Elastic Compute Cloud (Amazon EC2) インスタンス、 AWS Lambda 関数、 Amazon DynamoDB テーブルなどがあります。
   - ユーザー - ユーザー ID でトレースをフィルタリングします。
   - エラー根本原因メッセージ - エラー根本原因でトレースをフィルタリングします。
   - URL - アプリケーションで使用される URL パスでトレースをフィルタリングします。

- HTTP ステータスコード - アプリケーションが返した HTTP ステータスコードでトレースをフィルタリングします。カスタムレスポンスコードを指定、または以下から選択できます。
  - `200` - リクエストが成功しました。
  - `401` - リクエストには有効な認証情報がありませんでした。
  - `403` - リクエストには有効なアクセス許可がありませんでした。
  - `404` - サーバーがリクエストされたリソースを見つけることができませんでした。
  - `500` - サーバーにより予期しない状態が検出され、内部エラーを生成しました。

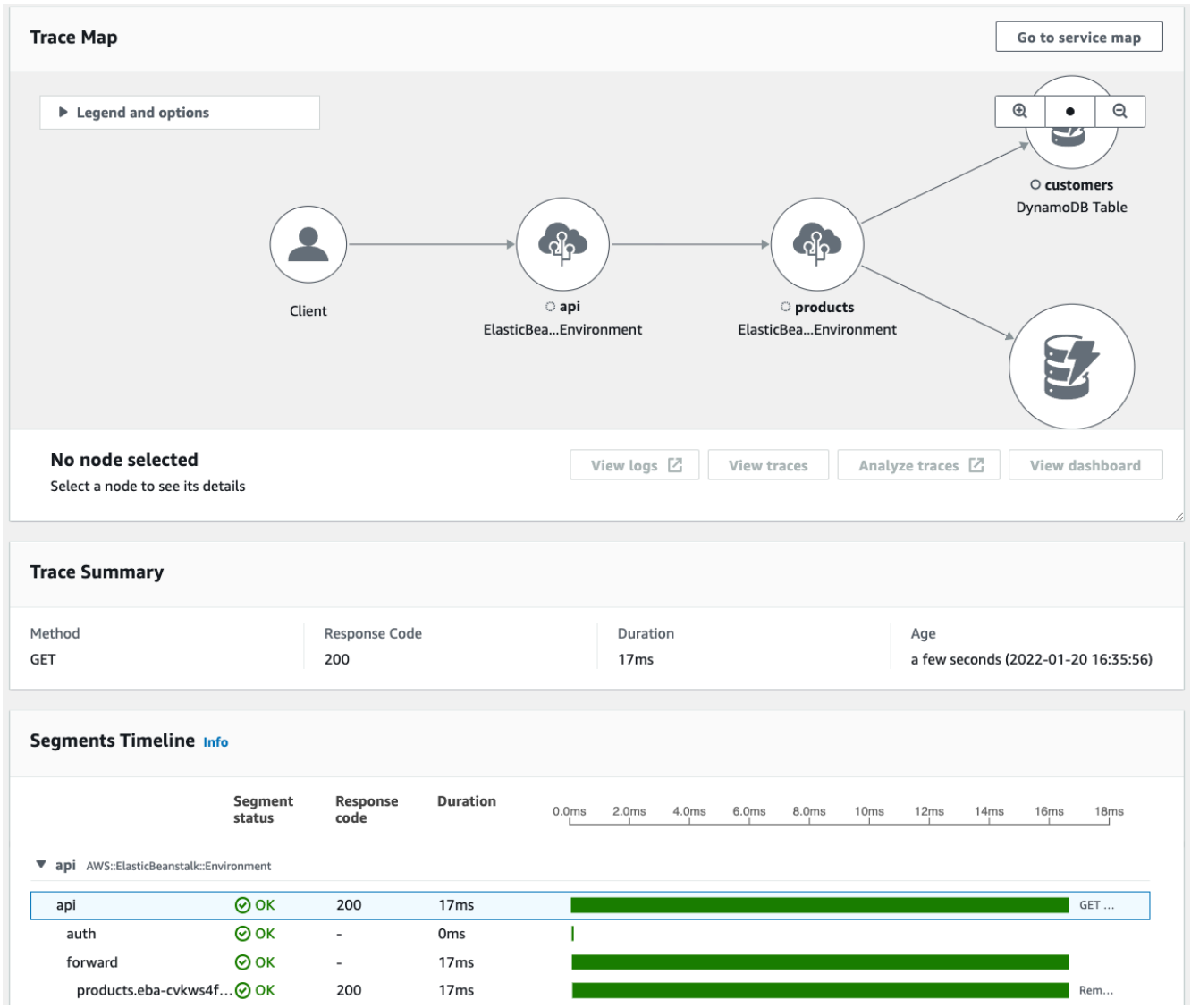
1 つまたは複数のエントリを選択してから [クエリに追加] を選択すると、ページ上部のフィルター式に追加されます。

4. 単一トレースを検索するには、トレース ID をクエリフィールドに直接入力します。X-Ray 形式または World Wide Web Consortium (W3C) 形式を使用できます。例えば、AWS Distro for OpenTelemetry を使用して作成されたトレースは W3C 形式です。

   > **Note**
   >
   > W3C 形式のトレース ID で作成されたトレースをクエリするときは、一致するトレースが X-Ray 形式でコンソールに表示されます。例えば、W3C 形式で `4efaaf4d1e8720b39541901950019ee5` にクエリを実行すると、コンソールには X-Ray に相当する `1-4efaaf4d-1e8720b39541901950019ee5` が表示されます。

5. [クエリを実行] を選択すると、いつでもページ下部の [トレース] セクションに一致するトレースのリストが表示されます。
6. 単一トレースの [トレースの詳細] ページを表示するには、リストからトレース ID を選択します。

   次の図は、トレースに関連付けられたサービスノードと、トレースを構成するセグメントが取得したパスを表すノード間のエッジを含む、[トレースマップ] を示しています。[トレース概要] が [トレースマップ] に続きます。概要には、サンプルの GET オペレーション、[レスポンスコード]、トレースの実行に要した [所要時間]、リクエストの [経過時間] に関する情報が含まれます。[セグメントのタイムライン] がトレースのセグメントとサブセグメントの所要時間を示す [トレース概要] に続きます。



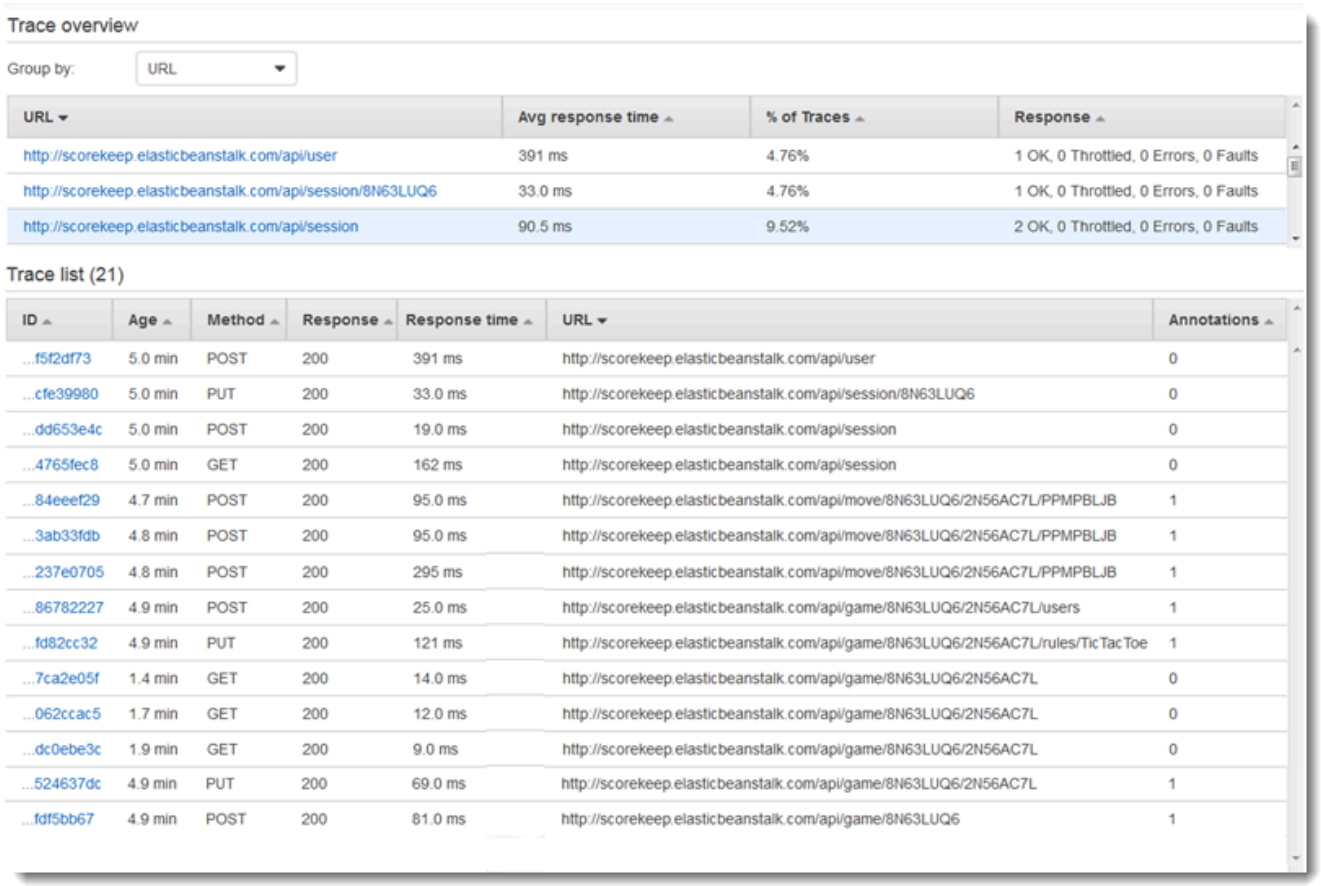
Amazon SQS と Lambda を使用するイベント駆動型アプリケーションがある場合は、[トレースマップ] で各リクエストのトレースが接続された表示を確認できます。マップでは、メッセージプロデューサーからのトレースは AWS Lambda コンシューマーからのトレースにリンクされ、破線のエッジとして表示されます。イベント駆動型アプリケーションの詳細については、「イベント駆動型アプリケーションのトレース」を参照してください。

[トレース] と [トレースの詳細] ページはクロスアカウントトレースにも対応しており、これにより複数のアカウントのトレースをトレースリストと 1 つのトレースマップ内に表示できます。

X-Ray console

X-Ray コンソールでトレースを表示するには

1. X-Ray コンソールのトレースページを開きます。[トレース概要] パネルには、[エラーの根本原因]、[ResourceARN]、[InstanceId] などの、共通機能でグループ化されたトレースのリストが表示されます。
2. トレースのグループ化されたセットを表示する共通機能を選択するには、[グループ別] の横にある下矢印を展開します。次の図は、AWS X-Ray サンプルアプリケーション の URL でグループ化されたトレースのトレース概要と、関連するトレースのリストを示しています。



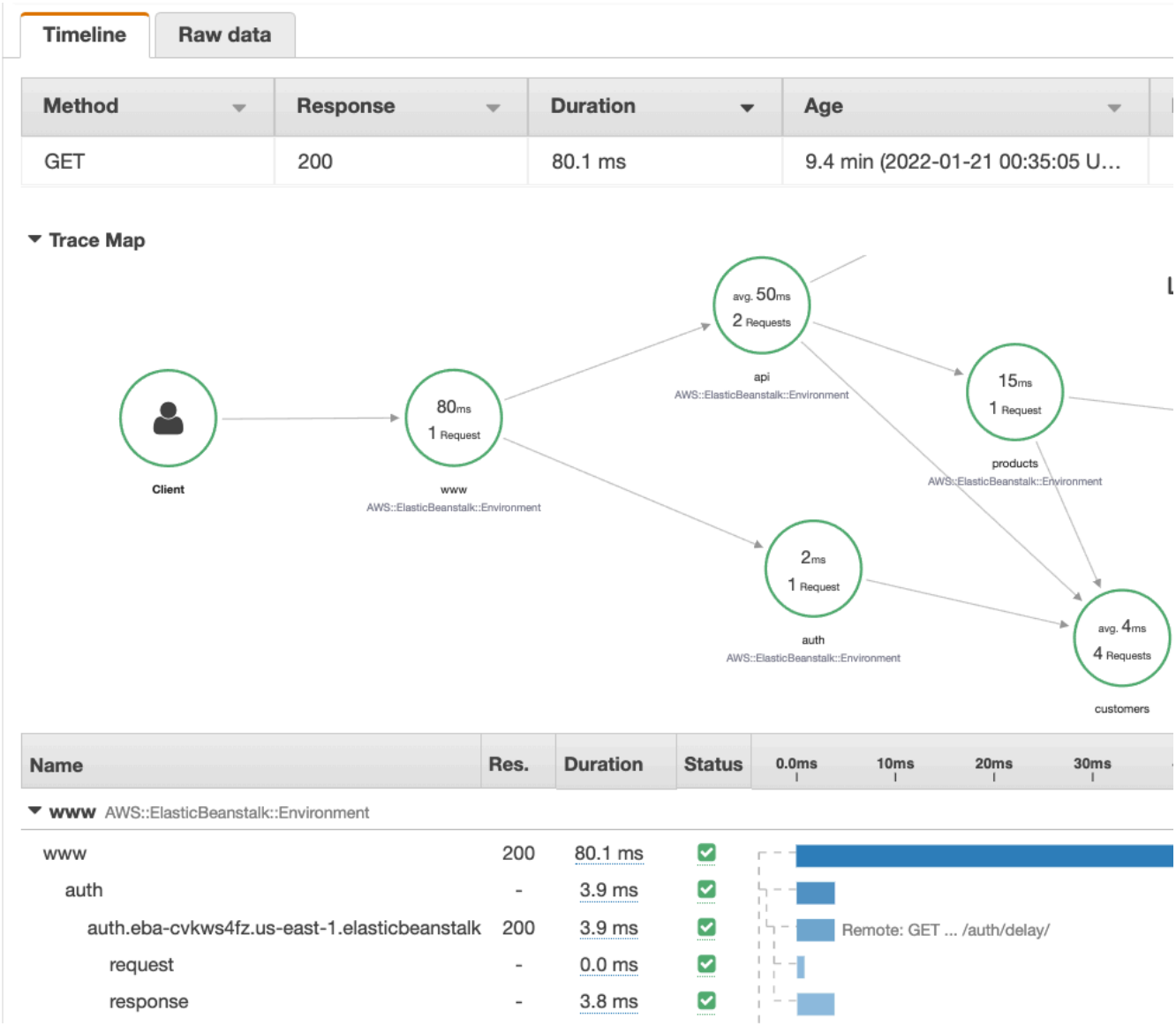
3. トレースの [ID] を選択すると [トレースのリスト] の下に表示されます。ナビゲーションペインで [サービスマップ] を選択して特定のサービスノードのトレースを表示することもできます。そうすると、そのノードに関連付けられているトレースを表示できます。

   [タイムライン] タブにはトレースのリクエストフローが表示されますが、以下のものが含まれます。

   - 各トレース内のセグメントのパスのマップ。

- セグメントがトレースマップのノードに到達するまでにかかった時間。
- トレースマップのノードに対して実行されたリクエスト数。

次の図は、サンプルアプリケーションに対して実行された GET リクエストに関連付けられた、[トレースマップ] の例を示しています。矢印は各セグメントがリクエスト完了のために用いたパスを示しています。サービスノードには GET リクエスト中に実行されたリクエスト数が表示されます。



[タイムライン] タブの詳細については、次の「トレースのタイムラインの探索」セクションを参照してください。

[未加工データ] タブには、トレース、およびトレースを構成するセグメントとサブセグメントに関する情報が JSON 形式で表示されます。この情報には、次の内容が含まれます。

- タイムスタンプ
- 一意の ID
- セグメントまたはサブセグメントに関連付けられたリソース
- セグメントまたはサブセグメントのソース、またはオリジン
- HTTP リクエストのレスポンスなど、アプリケーションへのリクエストに関する追加情報

トレースのタイムラインの探索

[タイムライン] セクションには、タスクの完了にかかった時間に対応する水平バーの横にある、セグメントとサブセグメントの階層が表示されます。リスト内の最初のエントリは、1 回のリクエストに対してサービスによって記録されたすべてのデータを表すセグメントです。サブセグメントはインデントされてセグメントの後に一覧表示されます。列には各セグメントに関する情報が含まれます。

CloudWatch console

CloudWatch コンソールでは、[セグメントのタイムライン] に次の情報が表示されます。

- 最初の列: 選択したトレース内のセグメントとサブセグメントが表示されます。
- [セグメントステータス] 列: 各セグメントとサブセグメントのステータスの結果が表示されます。
- [レスポンスコード] 列: セグメントまたはサブセグメントが実行したブラウザリクエストがある場合、その TTTP レスポンスステータスコードが表示されます。
- [期間] 列: セグメントまたはサブセグメントの実行期間が表示されます。
- [次でホストされています:] 列: 必要に応じて、セグメントまたはサブセグメントが実行される名前空間または環境が表示されます。詳細については、「収集されるディメンションと、ディメンションの組み合わせ」を参照してください。
- 最後の列: タイムライン内の他のセグメントまたはサブセグメントと関連する、セグメントまたはサブセグメントの実行期間に対応する水平バーが表示されます。

サービスノード別にセグメントとサブセグメントのリストをグループ化するには、[ノード別にグループ化] をオンにします。

X-Ray console

トレースの詳細ページで [タイムライン] タブを選択すると、トレースを構成する各セグメントとサブセグメントのタイムラインが表示されます。

X-Ray コンソールの [タイムライン] には次の情報が表示されます。

- [名前] 列: トレース内のセグメントとサブセグメントの名前が表示されます。
- [Res.] 列: セグメントまたはサブセグメントによって実行されたブラウザリクエストがある場合、その HTTP レスポンスステータスコードが表示されます。
- [期間] 列: セグメントまたはサブセグメントの実行期間が表示されます。
- [ステータス] 列: セグメントまたはサブセグメントのステータスの結果が表示されます。
- 最後の列: タイムライン内の他のセグメントまたはサブセグメントと関連する、セグメントまたはサブセグメントの実行期間に対応する水平バーが表示されます。

コンソールがタイムラインの生成のために使用する未加工トレースデータを表示するには、[未加工データ] タブを選択します。未加工データには、トレース、およびトレースを構成するセグメントとサブセグメントに関する情報が JSON 形式で表示されます。この情報には、次の内容が含まれます。

- タイムスタンプ
- 一意の ID
- セグメントまたはサブセグメントに関連付けられたリソース
- セグメントまたはサブセグメントのソース、またはオリジン
- HTTP リクエストのレスポンスなど、アプリケーションへのリクエストに関する追加情報

計測された AWS SDK、HTTP、または SQLクライアントを使用して外部リソースを呼び出すと、X-Ray SDK はサブセグメントを自動的に記録します。また、X-Ray SDK を使用して任意の関数またはコードブロックのカスタムサブセグメントを記録することもできます。カスタムサブセグメントが開いている間に記録された追加サブセグメントは、カスタムサブセグメントの子になります。

セグメントの詳細を表示する

トレースの [タイムライン] から、詳細を表示するセグメントの名前を選択します。

[セグメントの詳細] パネルには、[概要]、[リソース]、[注釈]、[メタデータ]、[例外]、および [SQL] タブが表示されます。以下が適用されます。

- [概要] タブには、リクエストと応答に関する情報が表示されます。情報には、名前、開始時間、終了時間、期間、リクエスト URL、リクエストオペレーション、リクエストレスポンスコード、エラーや障害が含まれます。
- セグメントのリソースタブには、X-Ray SDK からの情報と、アプリケーションを実行している AWS リソースに関する情報が表示されます。X-Ray SDK の Amazon EC2 AWS Elastic Beanstalk、または Amazon ECS プラグインを使用して、サービス固有のリソース情報を記録します。プラグインの詳細については、X-Ray SDK for Java の設定 の「サービスプラグイン」セクションを参照してください。
- その他のタブには、セグメントに記録されている [注釈]、[メタデータ]、[例外] が表示されます。計測されたリクエストから生成されると例外は自動的にキャプチャされます。注釈とメタデータには X-Ray SDK が提供するオペレーションを使用して記録される追加情報が含まれています。セグメントに注釈またはメタデータを追加するには、X-Ray SDK を使用します。詳細については、「」の AWS X-Ray SDKs」に記載されている言語固有のリンクを参照してくださいのアプリケーションの計測 AWS X-Ray。

サブセグメントの詳細を表示する

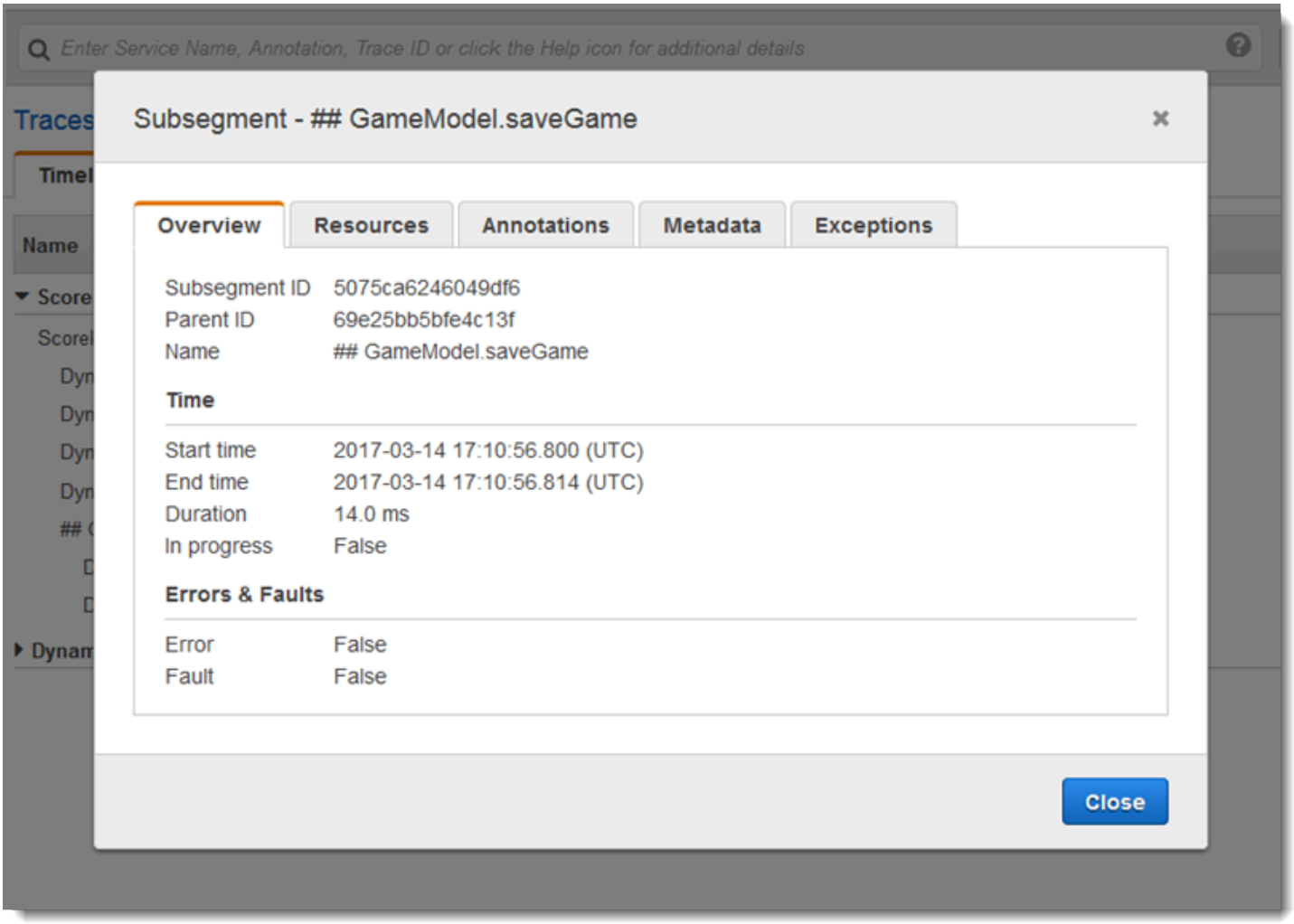
トレースタイムラインから、詳細を表示するサブセグメントの名前を選択します。

- [概要] タブにはリクエストとレスポンスに関する情報が表示されます。これには、名前、開始時間、終了時間、期間、リクエスト URL、リクエストオペレーション、リクエストレスポンスコード、エラーや障害が含まれます。計測されたクライアントを使用して生成されたサブセグメントについては、[概要] タブにアプリケーションの視点からのリクエストと応答に関する情報が含まれています。
- サブセグメントの [リソース] タブには、サブセグメントの実行に使用された AWS リソースの詳細が表示されます。たとえば、リソースタブには、 AWS Lambda 関数 ARN、DynamoDB テーブルに関する情報、呼び出されるオペレーション、およびリクエスト ID を含めることができます。
- その他のタブには、サブセグメントに記録されている [注釈]、[メタデータ]、[例外] が表示されます。計測されたリクエストから生成されると例外は自動的にキャプチャされます。注釈とメタデータには X-Ray SDK が提供するオペレーションを使用して記録される追加情報が含まれています。セグメントに注釈またはメタデータを追加するには、X-Ray SDK を使用します。詳細については、のアプリケーションの計測 AWS X-Ray の「 AWS X-Ray SDK でアプリケーションを計測する」に記載されている、言語固有のリンクを参照してください。

カスタムサブセグメントの場合、[概要] タブには記録するコードまたは関数の領域を指定するために設定できるサブセグメントの名前が表示されます。詳細については、「」の AWS X-Ray SDKs」に

記載されている言語固有のリンクを参照してくださいX-Ray SDK for Java を使用したカスタムサブセグメントの生成。

次の図は、カスタムサブセグメントの [概要] タブを示しています。概要には、サブセグメント ID、親 ID、名前、開始時間と終了時間、期間、ステータス、エラーまたは障害が含まれます。



カスタムサブセグメントの [メタデータ] タブには、そのサブセグメントで使用されるリソースに関する情報が JSON 形式で含まれています。

フィルター式の使用

フィルター式を使用して、特定のリクエスト、サービス、2 つのサービス間の接続 (エッジ)、または条件を満たすリクエストのトレースマップまたはトレースを表示できます。X-Ray にはフィルタ式言語があり、リクエストヘッダー、レスポンスステータス、元セグメントのインデックス付きフィールドのデータに基づいて、リクエスト、サービス、エッジをフィルタリングできます。

X-Ray コンソールで表示するトレースの期間を選択すると、コンソールが表示できる以上の結果が得られることがあります。右上隅には、スキャンしたトレースの数と使用可能なトレースが他にもあるかどうかがコンソールに表示されます。フィルター式を使用して、検索するトレースだけに結果を絞り込むことができます。

トピック

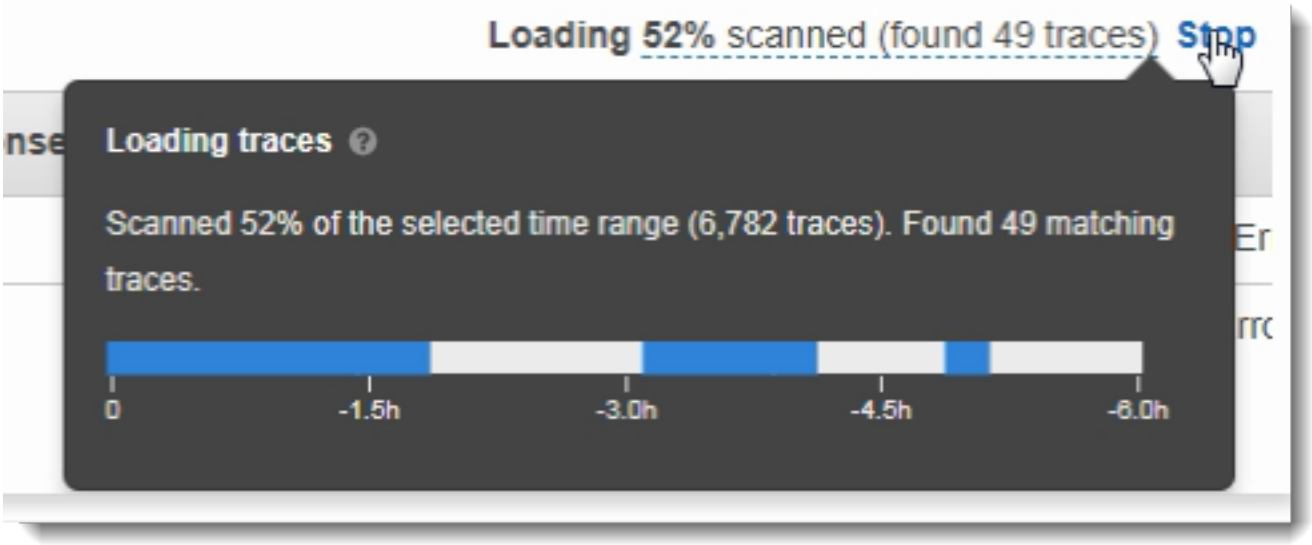
フィルタ式の詳細

トレースマップのノードを選択すると、コンソールはノードのサービス名と選択に基づいて、存在するエラーのタイプに基づいてフィルター式を構成します。パフォーマンスの問題を示すトレースや特定のリクエストに関連するトレースを見つけるには、コンソールが提供する式を調整するか、独自の式を作成します。X-Ray SDK で注釈を追加する場合は、注釈キーまたはキーの値に基づいてフィルターを適用することもできます。

> **Note**
>
> トレースマップで相対的な時間範囲を選択し、ノードを選択した場合、コンソールによって時間範囲が絶対開始時刻と終了時刻に変換されます。ノードのトレースが検索結果に確実に表示され、ノードがアクティブでないときのスキャン時間を避けるために、時間範囲にはノードがトレースを送信した時間だけが含まれます。現在の時刻を基準にして検索するには、トレースページで相対的な時間範囲に戻って再度スキャンすることができます。

コンソールが表示できるものより多くの結果がまだある場合、コンソールには一致したトレースの数とスキャンされたトレースの数が表示されます。表示される割合は、スキャンされた選択済みの時間枠の割合です。結果に表示されているすべての一致するトレースを確認するには、フィルタ式をさらに絞り込むか、より短い時間枠を選択します。

一番新しい結果を得るために、コンソールは時間範囲の終わりにスキャンを開始し、逆方向に動作します。多数のトレースがあるが結果が少ない場合、コンソールは時間範囲をチャンクに分割し、それらを並行してスキャンします。進行状況バーには、スキャンされた時間範囲の一部が表示されます。



グループでフィルタ式を使用する

グループは、フィルタ式で定義されるトレースのコレクションです。グループを使用して、追加のサービスグラフを生成し、Amazon CloudWatch メトリクスを指定できます。

グループは名前または Amazon リソースネーム (ARN) で識別され、フィルタ式を含みます。サービスは着信トレースを式と比較し、それに応じてそれらを保管します。

フィルタ式検索バーの左側にあるドロップダウンメニューを使用して、グループを作成および変更できます。

> **Note**
>
> グループの認定中に、サービスでエラーが検出された場合、そのグループは着信トレースの処理に含まれなくなり、エラーメトリクスが記録されます。

グループの詳細については、グループの設定を参照してください。

フィルタ式の構文

フィルタ式にキーワード、単項またはバイナリの演算子、値を追加して比較することができます。

```
keyword operator value
```

演算子が異なる場合は、異なるタイプのキーワードを使用できます。たとえば、responsetime は、数値型キーワードを指し、数値に関する演算子と比較することができます。

Example - 応答時間が 5 秒を超えたリクエスト

```
responsetime > 5
```

AND 演算子および OR 演算子を使用して、複合式内で複数の式を結合できます。

Example - 総所要時間が 5〜8 秒のリクエスト

```
duration >= 5 AND duration <= 8
```

キーワードおよび演算子をシンプルにすると、トレースレベルでのみ問題を見つけることができます。エラーによってダウンロードが発生したが、アプリケーションによって処理され、ユーザーに返らない場合、error の検索では見つけることができません。

ダウンストリームの問題があるトレースを見つけるには、「複合型キーワード」、service()、および edge() を使用できます。これらのキーワードを使用して、すべてのダウンストリームノード、単一のダウンストリームノード、または 2 つのノード間のエッジにフィルタ式を適用することができます。さらに詳細にする場合は、「id() 関数」を使用して、タイプごとにサービスおよびエッジをフィルタリングすることができます。

ブール型キーワード

ブール値のキーワード値は true または false です。これらのキーワードを使用して、エラーの原因となったトレースを見つけます。

ブール型キーワード

- ok - レスポンスステータスコードは 2XX Success でした。
- error - レスポンスステータスコードは 4XX Client Error でした。
- throttle - レスポンスステータスコードは 429 Too Many Requests でした。
- fault - レスポンスステータスコードは 5XX Server Error でした。
- partial - リクエスト内に不完全なセグメントがあります。

- `inferred` - リクエスト内に推定セグメントがあります。
- `first` - 要素は列挙リストの最初です。
- `last` - 要素は列挙リストの最後です。
- `remote` - 根本原因のエンティティがリモートです。
- `root` - サービスはエントリポイント、またはトレースのルートセグメントです。

ブール演算子は、指定されたキーが `true` または `false` のセグメントを見つけます。

ブール演算子

- none - キーワードが true の場合、式は true と評価されます。
- ! - キーワードが false の場合、式は true と評価されます。
- =,!= - キーワードの値を文字列 `true`または`false` と比較します。これらの演算子は、他の演算子と同じように動作しますが、より明示的です。

Example - レスポンスステータスが 2XX OK である

```
ok
```

Example - レスポンスステータスが 2XX OK ではない

```
!ok
```

Example - レスポンスステータスが 2XX OK ではない

```
ok = false
```

Example - 最後の列挙障害トレースにエラー名「逆シリアル化」がある

```
rootcause.fault.entity { last and name = "deserialize" }
```

Example - カバレッジが 0.7 より大きく、サービス名が「トレース」であるリモートセグメントを持つリクエスト

```
rootcause.responsetime.entity { remote and coverage > 0.7 and name = "traces" }
```

Example - サービスタイプが「AWS:DynamoDB」の推定セグメントがあるリクエスト

```
rootcause.fault.service { inferred and name = traces and type = "AWS::DynamoDB" }
```

Example - ルートとして「data-plane」という名前のセグメントのあるリクエスト

```
service("data-plane") {root = true and fault = true}
```

数値型キーワード

数値型キーワードを使用して、特定の応答時間、期間、応答ステータスを含むリクエストを検索します。

数値型キーワード

- `responsetime` - サーバーでレスポンスの送信に要した時間。
- `duration` - すべてのダウンストリーム呼び出しを含むリクエスト総所要時間。
- `http.status` - レスポンスステータスコード。
- `index` - 列挙リスト内の要素の位置。
- `coverage` - ルートセグメントの応答時間に対するエンティティの応答時間の 10 進数の割合。応答時間の根本原因のエンティティにのみ適用されます。

数値型演算子

数値型キーワードでは、標準の品質と比較演算子を使用しています。

- =,!= - キーワードが数値と同等か、等しくない。
- <,<=, >,>= - キーワードが数値より小さい、または大きい。

Example - レスポンスステータスが 200 OK ではない

```
http.status != 200
```

Example - 総所要時間が 5〜8 秒のリクエスト

```
duration >= 5 AND duration <= 8
```

Example - すべてのダウンストリーム呼び出しを含めて 3 秒未満で正常に完了したリクエスト

```
ok !partial duration <3
```

Example - 5 より大きいインデックスを持つ列挙型リストエンティティ

```
rootcause.fault.service { index > 5 }
```

Example - 最後のエンティティが 0.8 より大きいカバレッジを持つリクエスト

```
rootcause.responsetime.entity { last and coverage > 0.8 }
```

文字列型キーワード

文字列型キーワードを使用すると、リクエストヘッダーに特定のテキストを含むトレースや特定のユーザー ID のトレースを見つけることができます。

文字列型キーワード

- `http.url` - リクエストの URL。
- `http.method` - リクエストメソッド。
- `http.useragent` - リクエストのユーザーエージェント文字列。
- `http.clientip` - リクエスタの IP アドレス。
- `user` - Trace の任意の セグメント におけるユーザーフィールドの値。
- `name` - サービスの名前、または例外。
- `type` - サービスタイプ。
- `message` - 例外メッセージ。
- `availabilityzone` - トレース内の任意のセグメントのアベイラビリティーゾーンフィールドの値。
- `instance.id` - トレース内の任意のセグメントのインスタンス ID フィールドの値。
- `resource.arn` - トレース内の任意のセグメントのリソース ARN フィールドの値。

文字列演算子は、特定のテキストと一致する値、または特定のテキストを含む値を見つけます。値は、必ず引用符で囲います。

文字列演算子

- =,!= - キーワードが数値と同等か、等しくない。
- CONTAINS - キーワードに特定の文字列が含まれている。
- BEGINSWITH , ENDSWITH - キーワードが特定の文字列で始まる、または終わる。

Example - Http.url フィルター

```
http.url CONTAINS "/api/game/"
```

フィールドがトレースに存在するかどうかをテストするには、その値に関係なく、空の文字列が含まれているかどうかを確認します。

Example - ユーザー フィルター

ユーザー ID を持つすべてのトレースを見つけます。

```
user CONTAINS ""
```

Example - 「Auth」という名前のサービスを含む、障害の根本原因を含むトレースの選択

```
rootcause.fault.service { name = "Auth" }
```

Example - 最後のサービスに DynamoDB タイプがある応答時間の根本原因を含むトレースの選択

```
rootcause.responsetime.service { last and type = "AWS::DynamoDB" }
```

Example - 最後の例外に「account_id:1234567890 のアクセスが拒否されました」というメッセージのある障害の根本原因を含むトレースの選択

```
rootcause.fault.exception { last and message = "Access Denied for account_id:
 1234567890"
```

複合型キーワード

複雑なキーワードを使用し、サービス名、エッジ名、または注釈値に基づいてリクエストを見つけます。サービスとエッジについては、サービスまたはエッジに適用される追加のフィルタ式を指定できます。注釈では、ブール値、数値、または文字列演算子を使用して、特定のキーで注釈の値をフィルタリングできます。

複合型キーワード

- annotation[*key*] - フィールド *[key]* (キー)の注釈の値。注釈の値は、ブール値、数値、文字列のいずれかであるため、このタイプの比較演算子のいずれも使用することができます。このキーワードは service または edge キーワードと組み合わせて使用することができます。ドット (ピリオド) を含む注釈キーは角かっこで囲む必要があります ([ ])。
- edge(*source*, *destination*) {*filter*} - *[source]* (ソース)サービスと *[destination]* (宛先) サービス間の接続。オプションの中括弧には、この接続のセグメントに適用されるフィルタ式を含めることができます。
- group.*name* / group.*arn* — グループ名またはグループ ARN で参照されるグループのフィルター式の値。
- json - JSON 根本原因オブジェクト。JSON エンティティをプログラムで作成する手順については、AWS 「X-Ray からデータを取得する」を参照してください。
- service(*name*) {*filter*} - 名前が *[name]* のサービス。オプションの中括弧には、サービスで作成されたセグメントに適用されるフィルタ式を含めることができます。

サービスのキーワードを使用して、トレースマップの特定のノードにヒットするリクエストのトレースを見つけます。

複合型キーワード演算子は、指定されたキーが設定されている、または設定されていないセグメントを見つけます。

複合型キーワード演算子

- none - キーワードが 設定されている場合、式は true と評価されます。キーワードがブール型の場合は、ブール値として評価されます。
- ! - キーワードが setでない場合、式は true と評価されます。キーワードがブール型の場合は、ブール値として評価されます。
- =,!= — キーワードの値を比較します。
- edge(*source*, *destination*) {*filter*} - *[source]* (ソース) サービスと *[destination]* (宛先) サービス間の接続。オプションの中括弧には、この接続のセグメントに適用されるフィルタ式を含めることができます。
- annotation[*key*] - フィールド *[key]* (キー)の注釈の値。注釈の値は、ブール値、数値、文字列のいずれかであるため、このタイプの比較演算子のいずれも使用することができます。このキーワードは service または edge キーワードと組み合わせて使用することができます。

- `json` - JSON 根本原因オブジェクト。JSON エンティティをプログラムで作成する手順については、AWS 「X-Ray からデータを取得する」を参照してください。

サービスのキーワードを使用して、トレースマップの特定のノードにヒットするリクエストのトレースを見つけます。

Example - サービスフィルター

障害 (500 シリーズのエラー) が発生した `api.example.com` の呼び出しを含んだリクエスト。

```
service("api.example.com") { fault }
```

サービス名を除外して、フィルタ式をサービスマップのすべてのノードに適用することができます。

Example - サービスフィルター

トレースマップの任意の場所で障害が発生したリクエスト。

```
service() { fault }
```

エッジキーワードは、フィルタ式を 2 つのノード間の接続に適用します。

Example - エッジ フィルター

サービス `api.example.com` から `backend.example.com` への呼び出しが失敗してエラーが発生したリクエスト。

```
edge("api.example.com", "backend.example.com") { error }
```

また、サービスまたはエッジのキーワードを含む ! 演算子を使用して、サービスまたはエッジを他のフィルタ式の結果から除外することもできます。

Example - サービスおよびリクエスト フィルター

URL が `http://api.example.com/` で始まって `/v2/` を含むが、`api.example.com` という名前のサービスに達しないリクエスト。

```
http.url BEGINSWITH "http://api.example.com/" AND http.url CONTAINS "/v2/" AND !
service("api.example.com")
```

Example — サービスと応答時間のフィルター

http url が設定され、応答時間が 2 秒を超えているトレースを見つけてください。

```
http.url AND responseTime > 2
```

注釈について、annotation[*key*] が設定されているか、値のタイプに対応する比較演算子を使用しているかすべてのトレースを呼び出すことができます。

Example - 文字列値を含む注釈

文字列値 gameid の "817DL6VO" という名前のリクエスト。

```
annotation[gameid] = "817DL6VO"
```

Example — 注釈が設定されている

age 設定という名前の注釈を持つリクエスト。

```
annotation[age]
```

Example — 注釈が設定されていません

age 設定という名前の注釈のないリクエスト。

```
!annotation[age]
```

Example - 数値を含む注釈

アノテーション期間が数値 29 より大きいリクエスト。

```
annotation[age] > 29
```

Example – サービスまたはエッジと組み合わせた注釈

```
service { annotation[request.id] = "917DL6VO" }
```

```
edge { source.annotation[request.id] = "916DL6VO" }
```

```
edge { destination.annotation[request.id] = "918DL6VO" }
```

Example — ユーザーとグループ化

トレースがhigh_response_time グループフィルター (例:responseTime > 3) を満たし、ユーザーの名前が Alice のリクエスト。

```
group.name = "high_response_time" AND user = "alice"
```

Example - 根本原因のエンティティを含む JSON

根本原因エンティティが一致するリクエスト

```
rootcause.json = #[{ "Services": [ { "Name": "GetWeatherData", "EntityPath": [{ "Name":
 "GetWeatherData" }, { "Name": "get_temperature" } ] }, { "Name": "GetTemperature",
 "EntityPath": [ { "Name": "GetTemperature" } ] } ] }]
```

id 関数

service または edge のキーワードにサービス名を入力すると、そのサービス名を含むすべてのノードの結果が得られます。詳細にフィルタリングする場合は、id 関数を使用してサービスのタイプと名前を指定し、同一名を持つノードを区別することができます。

モニタリングアカウント内の複数のアカウントのトレースを表示する場合、account.id 関数を使用してサービスの特定のアカウントを指定します。

```
id(name: "service-name", type:"service::type", account.id:"account-ID")
```

サービスフィルタおよびエッジフィルタで、サービス名ではなく、id 関数を使用することもできます。

```
service(id(name: "service-name", type:"service::type")) { filter }
```

```
edge(id(name: "service-one", type:"service::type"), id(name: "service-two",
 type:"service::type")) { filter }
```

たとえば、 AWS Lambda 関数はトレースマップに 2 つのノードを作成します。1 つは関数呼び出し用、もう 1 つは Lambda サービス用です。この 2 つのノードは同じ名前ですが、タイプが異なります。標準サービスフィルタは、両ノードのトレースを見つけます。

Example - サービスフィルター

random-name という名前を持つすべてのサービスにエラーを含むリクエスト。

```
service("random-name") { error }
```

id 関数を使用して、関数そのもののエラーを絞り込み、サービスからエラーを除外します。

Example - id 関数を使用したサービスフィルター

サービスタイプが random-name の AWS::Lambda::Function という名前のサービスにエラーを含むリクエスト。

```
service(id(name: "random-name", type: "AWS::Lambda::Function")) { error }
```

タイプ別にノードを検索するには、名前を完全に除外します。

Example – id 関数とサービスタイプを使用したサービスフィルター

サービスタイプが AWS::Lambda::Function のサービスにエラーを含むリクエスト。

```
service(id(type: "AWS::Lambda::Function")) { error }
```

特定のノードを検索するには AWS アカウント、アカウント ID を指定します。

Example – id 関数とアカウント ID を使用したサービスフィルター

特定のアカウント ID AWS::Lambda::Function 内のサービスを含むリクエスト。

```
service(id(account.id: "account-id"))
```

クロスアカウントトレース

AWS X-Ray はクロスアカウントオブザーバビリティをサポートしているため、 内の複数のアカウントにまたがるアプリケーションをモニタリングおよびトラブルシューティングできます AWS リージョン。リンクされたアカウントのメトリクス、ログ、トレースをまとめてシームレスに検索、可視化、分析できます。これにより、複数のアカウントにまたがるリクエストの全体像を把握できます。クロスアカウントトレースは X-Ray のトレースマップと、CloudWatch コンソール内のトレースページで表示できます。

共有されるオブザーバビリティデータには、次のいずれかのタイプのテレメトリが含まれます。

- Amazon CloudWatch のメトリクス
- Amazon CloudWatch Logs のロググループ
- のトレース AWS X-Ray
- Amazon CloudWatch Application Insights のアプリケーション

クロスアカウントオブザーバビリティの設定

クロスアカウントオブザーバビリティを有効にするには、1 つ以上の AWS モニタリングアカウントを設定し、複数のソースアカウントにリンクします。モニタリングアカウントは、ソースアカウントから生成されたオブザーバビリティデータを表示して操作 AWS アカウント できる中央アカウントです。ソースアカウントは、含まれ AWS アカウント るリソースのオブザーバビリティデータを生成する個人です。

ソースアカウントは、オブザーバビリティデータをモニタリングアカウントと共有します。トレースは各ソースアカウントから最大 5 つのモニタリングアカウントにコピーされます。ソースアカウントから最初のモニタリングアカウントへのトレースのコピーは無料です。追加のモニタリングアカウントに送信されたトレースのコピーは、標準料金に基づいて各ソースアカウントに請求されます。詳細については、AWS X-Ray 料金表および Amazon CloudWatch 料金表を参照してください。

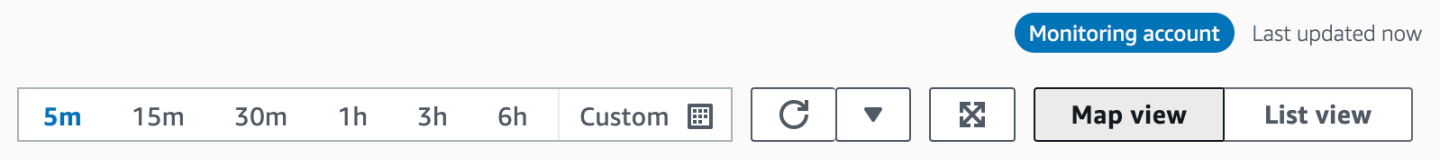
モニタリングアカウントとソースアカウント間のリンクを作成するには、CloudWatch コンソールまたは AWS CLI および API の新しい Observability Access Manager コマンドを使用します。詳細については、「CloudWatch のクロスアカウントオブザーバビリティ」を参照してください。

> **Note**
>
> X-Ray トレースは、受信した AWS アカウント に請求されます。サンプルリクエストが複数のサービスにまたがる場合 AWS アカウント、各アカウントは個別のトレースを記録し、すべてのトレースは同じトレース ID を共有します。クロスアカウントオブザーバビリティ料金の詳細については、AWS X-Ray 料金と Amazon CloudWatch 料金をご覧ください。

クロスアカウントトレースの表示

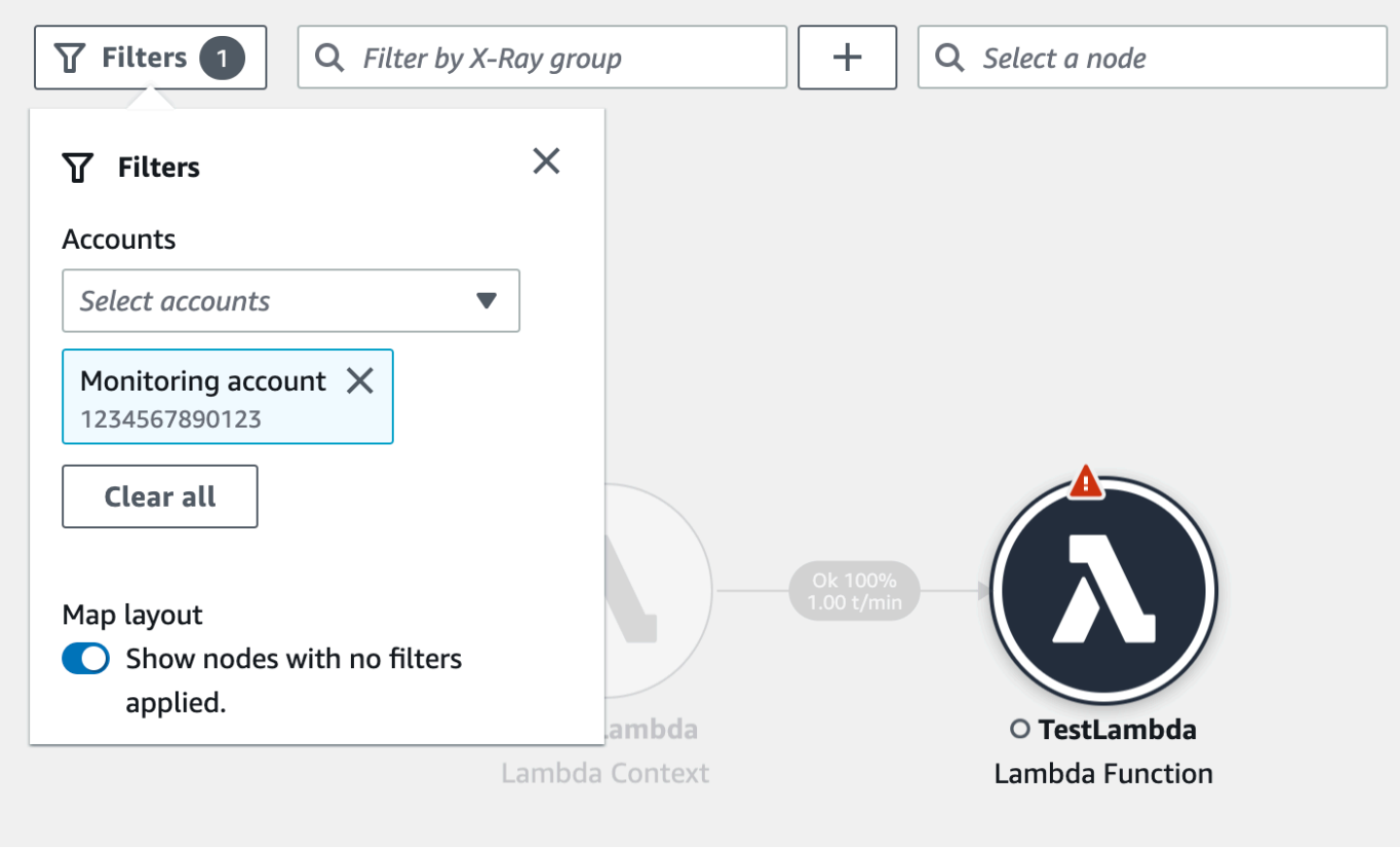
クロスアカウントトレースはモニタリングアカウントに表示されます。各ソースアカウントには、その特定のアカウントのローカルトレースのみが表示されます。以下のセクションでは、モニタリング

アカウントにサインインしていて、Amazon CloudWatch コンソールを開いていることを前提としています。トレースマップページとトレースページの両方で、右上隅にモニタリングアカウントのバッジが表示されます。

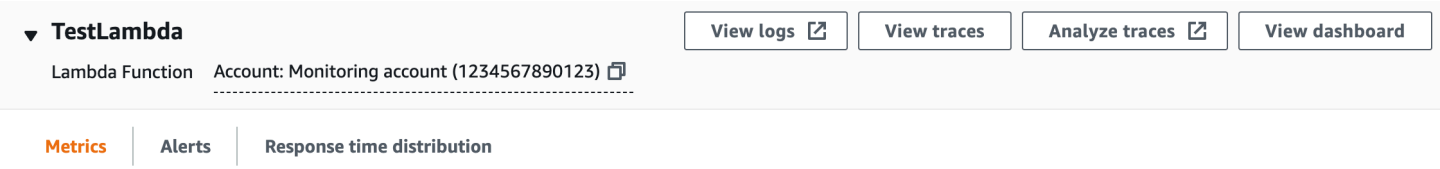


トレースマップ

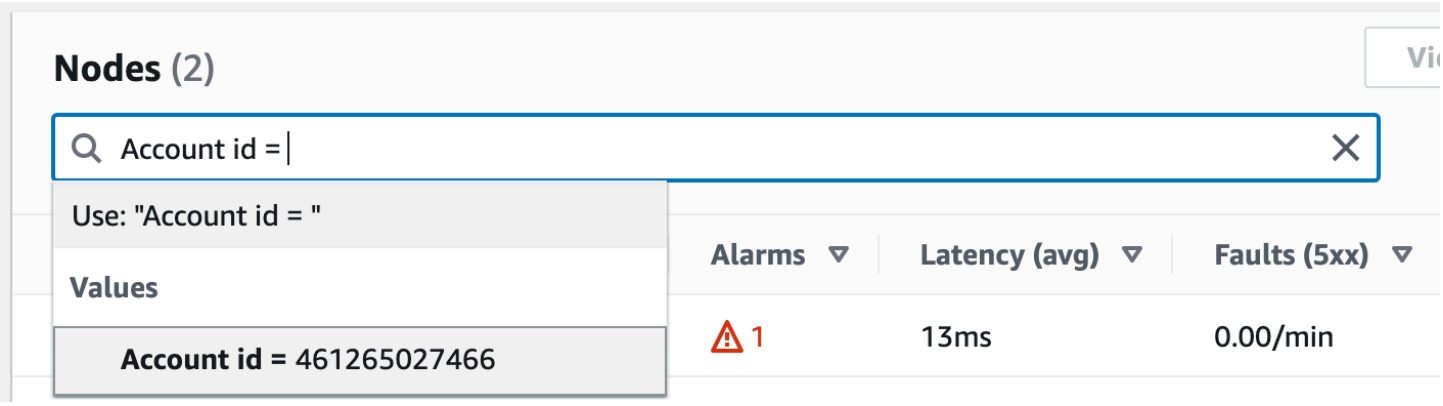
CloudWatch コンソールで、左側のナビゲーションペインから [X-Ray トレース] の下の [トレースマップ] を選択します。デフォルトでは、トレースマップには、モニタリングアカウントにトレースを送信するすべてのソースアカウントのノードと、モニタリングアカウント自体のノードが表示されます。トレースマップの左上から [フィルター] を選択し、[アカウント] ドロップダウンを使用してトレースマップをフィルターします。アカウントフィルターが適用されると、現在のフィルターと一致しないアカウントのサービスノードはグレー表示されます。



サービスノードを選択すると、[ノードの詳細] ペインにサービスのアカウント ID とラベルが表示されます。



トレースマップの右上で、[リストビュー] を選択して、サービスノードのリストを表示します。サービスノードのリストには、モニタリングアカウントのサービスと、ソースアカウントに設定されているすべてのアカウントのサービスが含まれます。[ノード] フィルターで [アカウントラベル] または [アカウント ID] を選択して、ノードのリストをフィルタリングします。

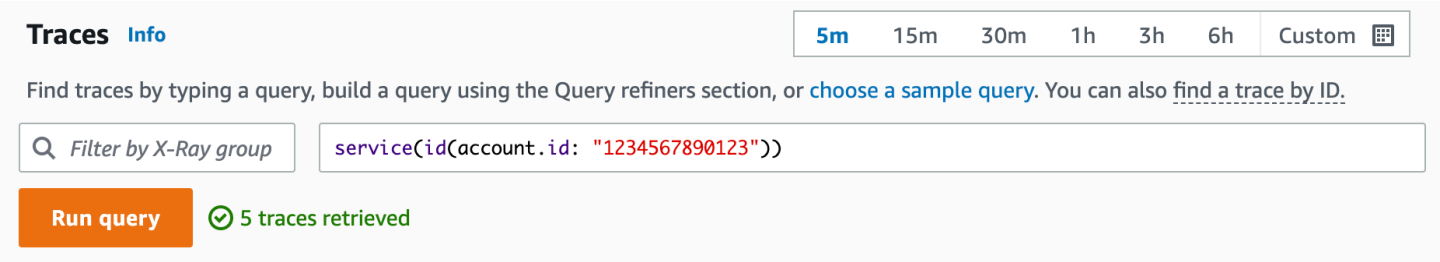


トレース

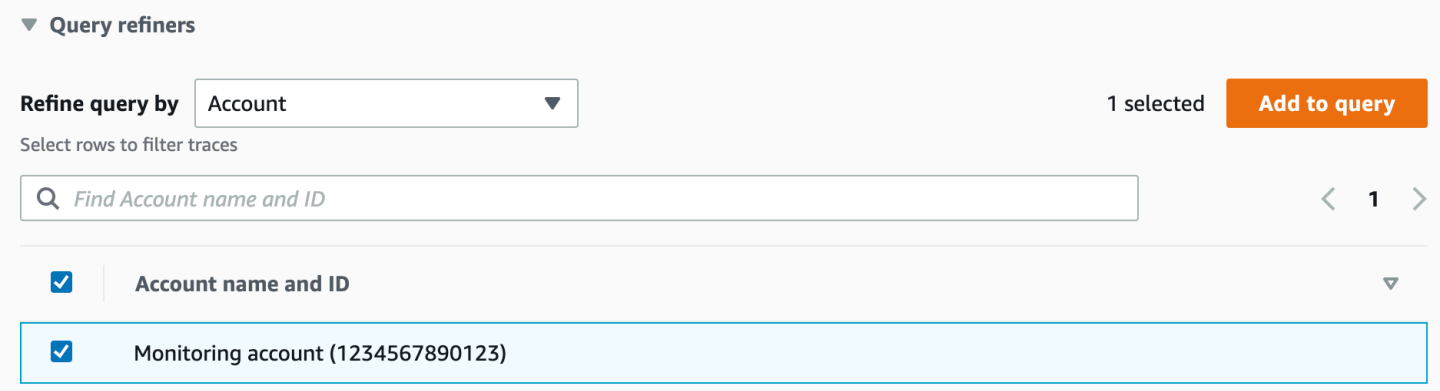
モニタリングアカウントから CloudWatch コンソールを開き、左側のナビゲーションペインの [X-Ray トレース] で [トレース] を選択すると、複数のアカウントにまたがるトレースについてのトレースの詳細が表示されます。このページは、X-Ray [トレースマップ] でノードを選択し、ノードの詳細ペインから [トレースを表示] を選択して開くこともできます。

[トレース] ページでは、アカウント ID によるクエリがサポートされています。はじめに、1 つまたは複数のアカウント ID を含むクエリを入力します。次の例では、アカウント ID Xまたは Y を通過したトレースをクエリします。

```
service(id(account.id:"X")) OR service(id(account.id:"Y"))
```



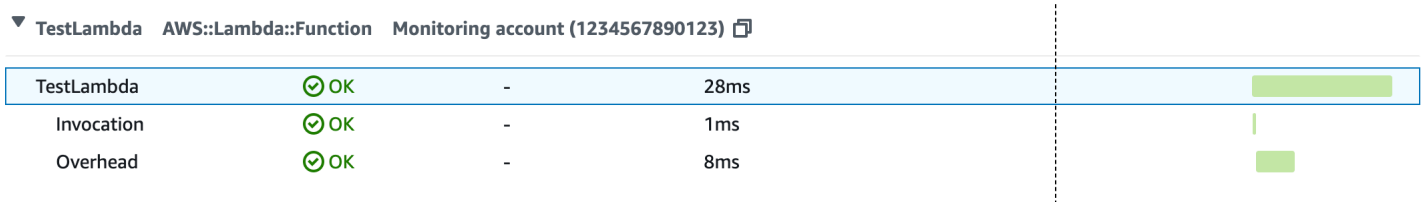
[アカウント] ごとにクエリを絞り込みます。リストから 1 つ以上のアカウントを選択し、[クエリに追加] を選択します。



トレースの詳細

[トレース] ページの下部にある [トレース] リストからトレースを選択すると、トレースの詳細が表示されます。トレースが通過したすべてのアカウントのサービスノードを含むトレース詳細マップなど、[トレースの詳細] が表示されます。特定のサービスノードを選択すると、対応するアカウントが表示されます。

[セグメントのタイムライン] セクションには、タイムラインの各セグメントのアカウント詳細が表示されます。



イベント駆動型アプリケーションのトレース

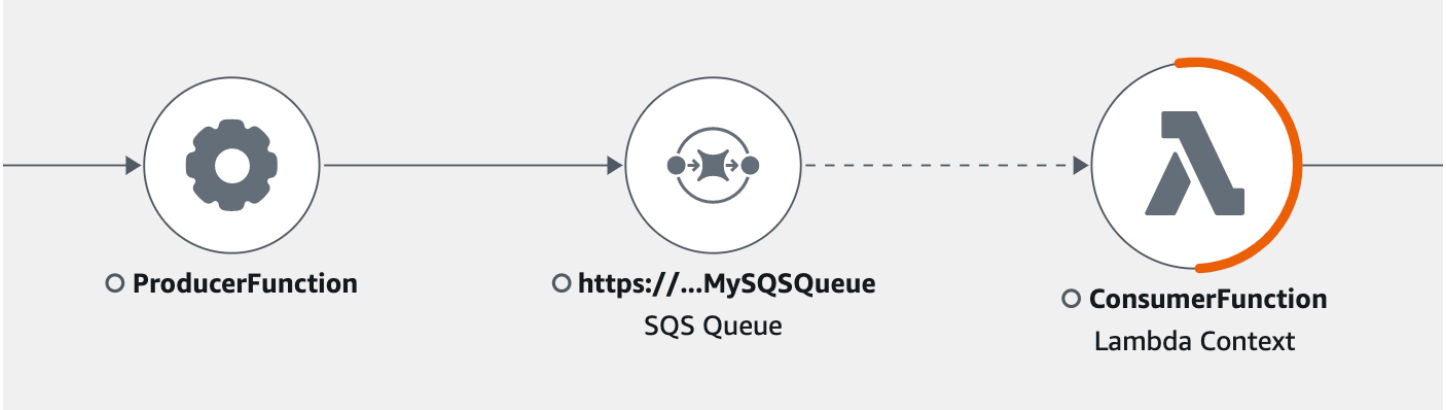
AWS X-Ray は、Amazon SQS と を使用したイベント駆動型アプリケーションのトレースをサポートします AWS Lambda。CloudWatch コンソールを使用すると、各リクエストが Amazon SQS のキューに入れられ、1 つ以上の Lambda 関数によって処理される過程を各リクエストの接続されたビューで確認できます。アップストリームメッセージプロデューサーからのトレースは、ダウンストリーム Lambda コンシューマーノードからのトレースに自動的にリンクされるため、アプリケーションのエンドツーエンドのビューが作成されます。

> **Note**
>
> 各トレースセグメントは最大 20 のトレースにリンクできます。また、1 つのトレースには最大 100 のリンクを含めることができます。シナリオによっては、追加のトレース

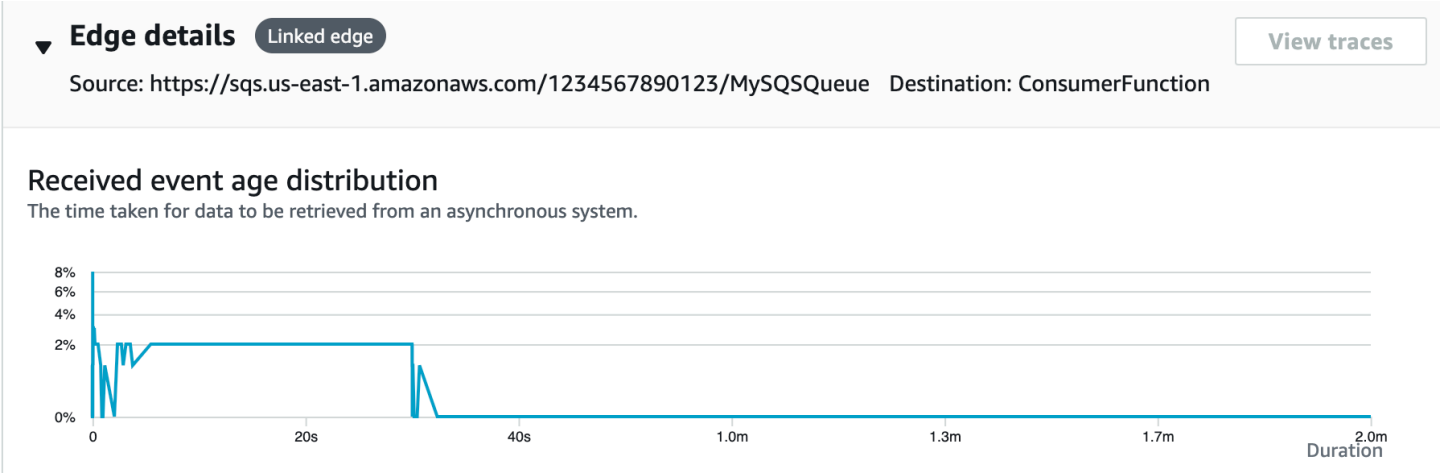
をリンクするとトレースドキュメントの最大サイズを超え、トレースが不完全になる可能性があります。例えば、トレーシングが有効になっている Lambda 関数が 1 回の呼び出しで多数の SQS メッセージをキューに送信した場合に発生することがあります。この問題が発生した場合は、X-Ray SDK を使用する緩和策があります。詳細については、Java、Node.js、Python、Go、または .NET 用の X-Ray SDK を参照してください。

リンクされたトレースをトレースマップに表示

CloudWatch コンソールの [トレースマップ] ページを使用して、Lambda コンシューマーからのトレースにリンクされたメッセージプロデューサーからのトレースを含むサービスマップを表示します。これらのリンクは、Amazon SQS ノードとダウンストリーム Lambda コンシューマーノードを接続する破線のエッジで表示されます。



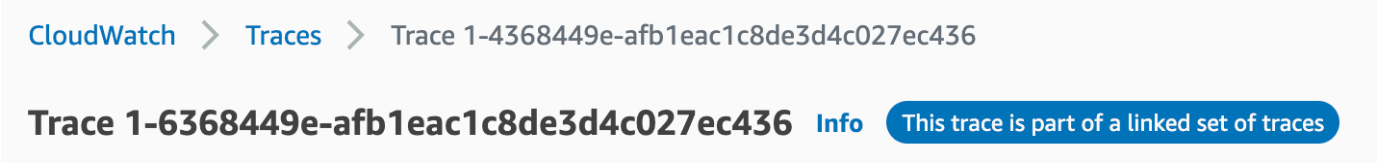
破線のエッジを選択すると、[受け取ったイベントが発生してから経過した時間]ヒストグラムが表示されます。これは、コンシューマーが受け取ったイベントの経過時間の分布をマッピングしたものです。経過時間は、イベントを受信するたびに計算されます。



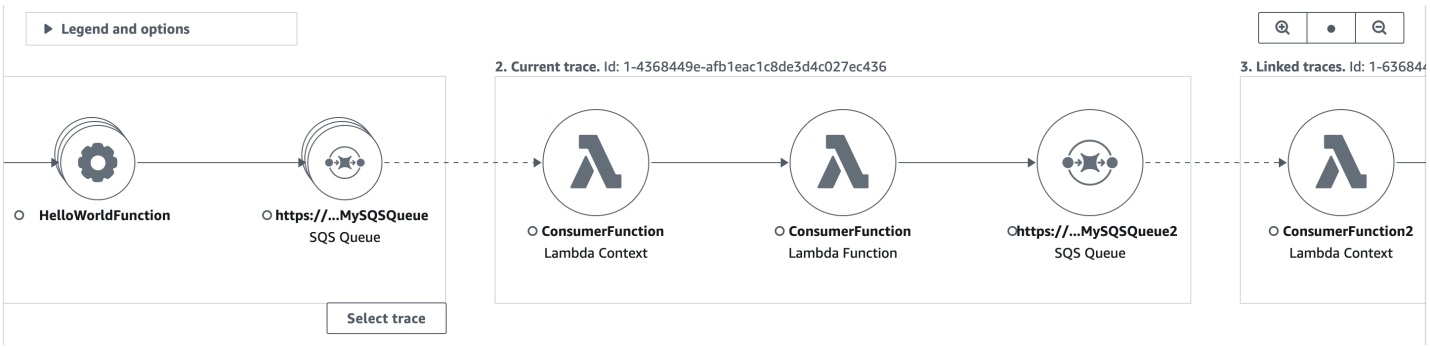
リンクされたトレースの詳細の表示

メッセージプロデューサー、Amazon SQS キュー、または Lambda コンシューマーから送信されたトレースの詳細を表示します。

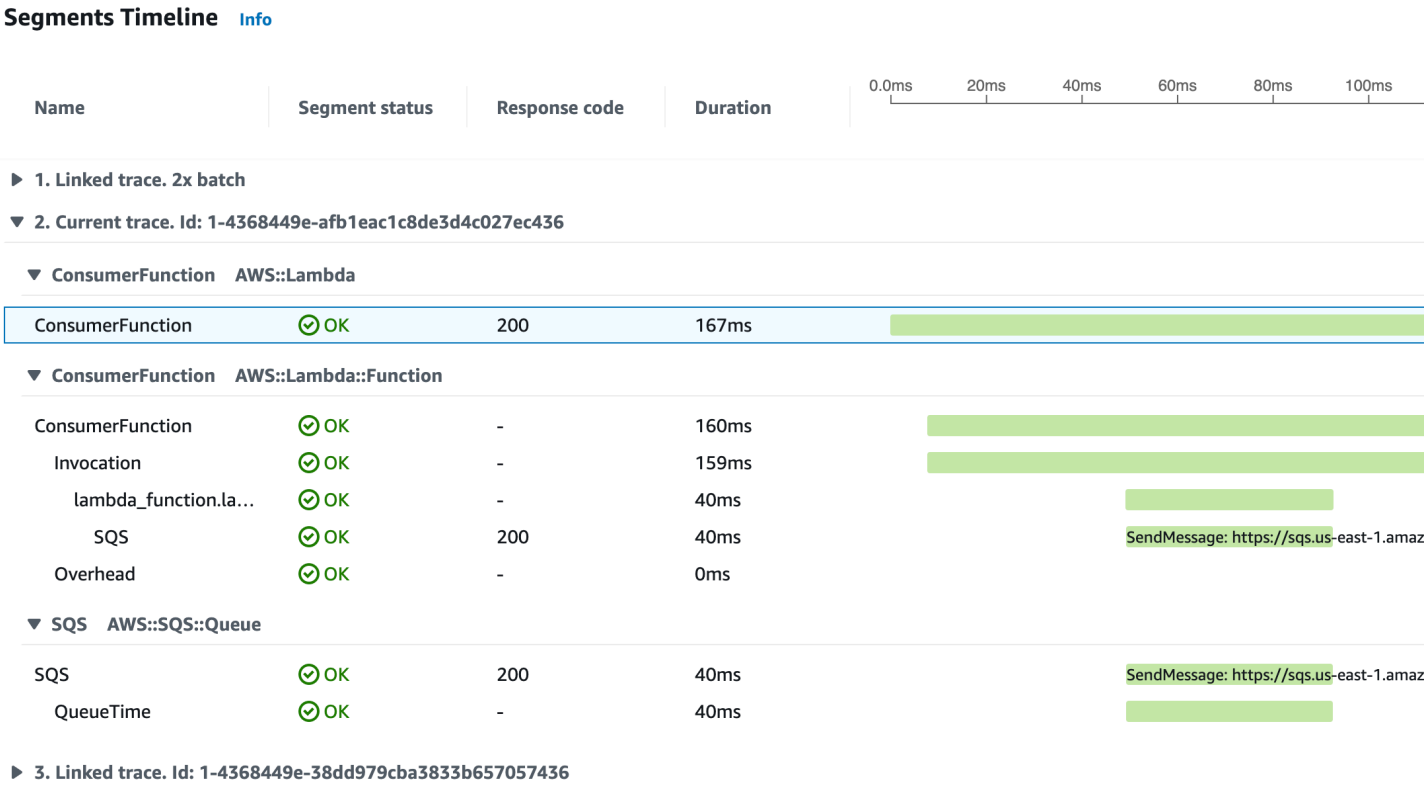
1. [トレースマップ] を使用して、メッセージプロデューサー、Amazon SQS、または Lambda コンシューマーノードを選択します。
2. ノードの詳細ペインから [トレースを表示] を選択すると、トレースのリストが表示されます。CloudWatch コンソール内の [トレース] ページに直接移動することもできます。
3. リストから特定のトレースを選択し、トレースの詳細ページを開きます。選択したトレースが、リンクされたトレースセットの一部である場合、トレースの詳細ページにメッセージが表示されます。



トレース詳細マップには、現在のトレースと、アップストリームとダウンストリームにリンクされたトレースが表示されます。各トレースは、トレースごとの境界を示すボックスに含まれています。現在選択されているトレースがアップストリームまたはダウンストリームの複数のトレースにリンクされている場合、アップストリームまたはダウンストリームにリンクされているトレース内のノードは積み重ねられ、[トレースを選択] ボタンが表示されます。



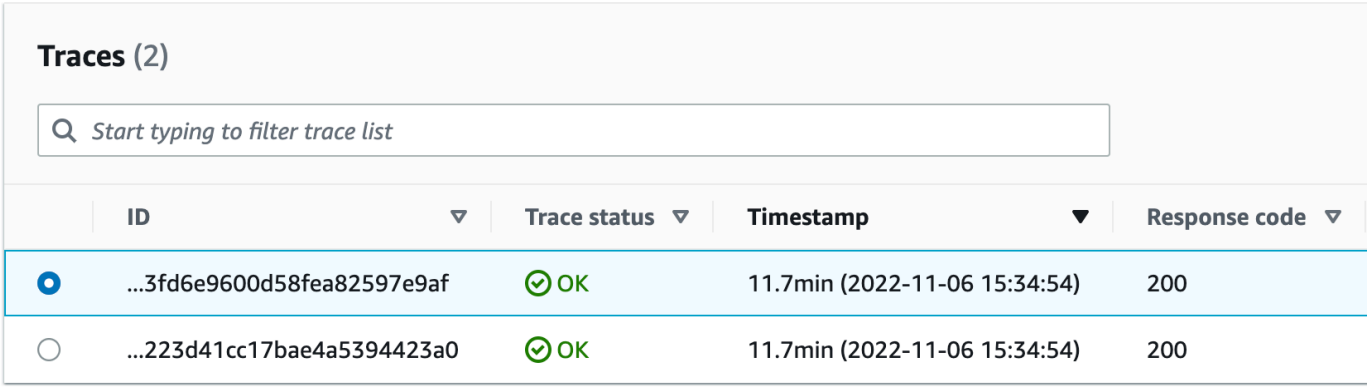
トレース詳細マップの下には、アップストリームとダウンストリームのリンクされたトレースを含むトレースセグメントのタイムラインが表示されます。アップストリームまたはダウンストリームにリンクされたトレースが複数ある場合、それらのセグメントの詳細は表示できません。リンクされたトレースセット内の 1 つのトレースのセグメント詳細を表示するには、以下の説明に従って 1 つのトレースを選択します。



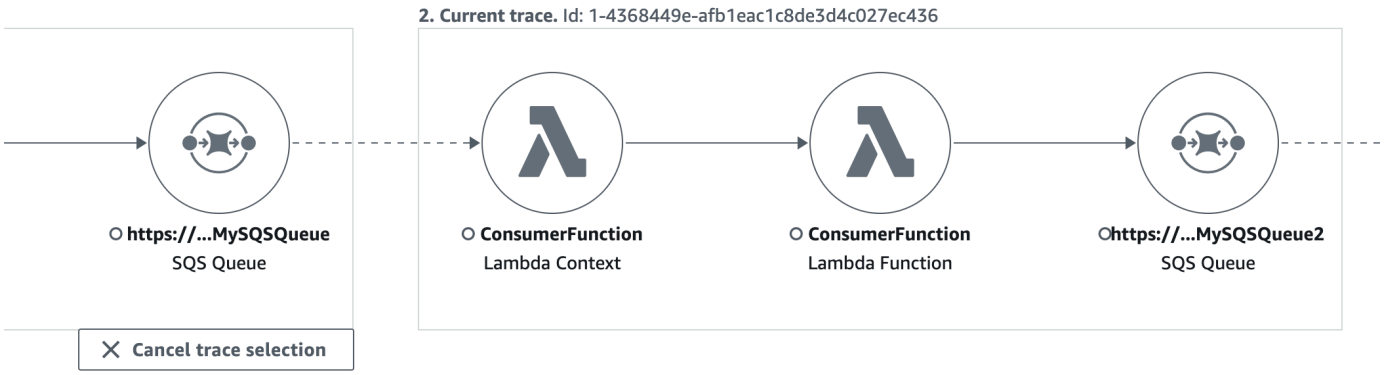
リンクされたトレースのセットから 1 つのトレースを選択

リンクされたトレースセットを 1 つのトレースにフィルタリングすると、タイムラインにセグメントの詳細が表示されます。

1. トレース詳細マップ上のリンクされたトレースの下にある [トレースを選択] を選択します。トレースのリストが表示されます。



2. トレース詳細マップ内に表示するには、トレースの横のラジオボタンを選択します。
3. [トレースの選択をキャンセル] を選択すると、リンクされたトレースのセット全体が表示されます。



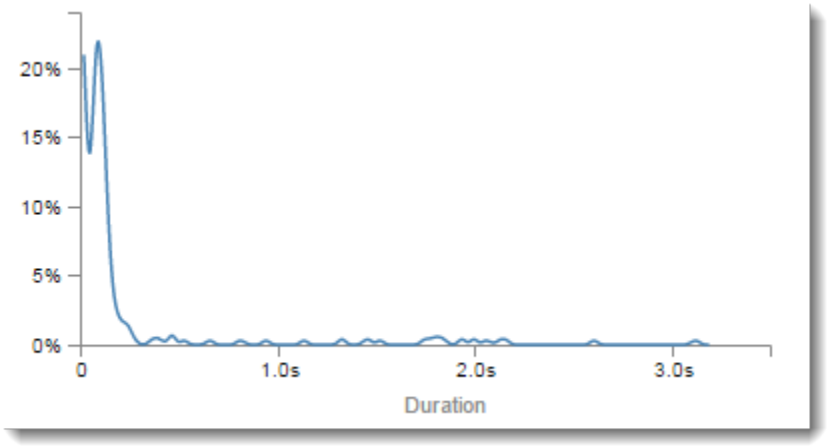
レイテンシーヒストグラムの使用

AWS X-Ray トレースマップでノードまたはエッジを選択すると、X-Ray コンソールにレイテンシー分散ヒストグラムが表示されます。

レイテンシー

レイテンシーは、リクエストが開始してから完了するまでの時間です。ヒストグラムは、レイテンシーの分散を示します。また、期間を x 軸、各期間に一致するリクエストの割合を y 軸に示しています。

このヒストグラムでは、ほとんどのリクエストを 300 ミリ秒 (ms) 以下で完了するサービスを示します。割合が低いリクエストには最大 2 秒、異常値の場合はそれ以上かかります。



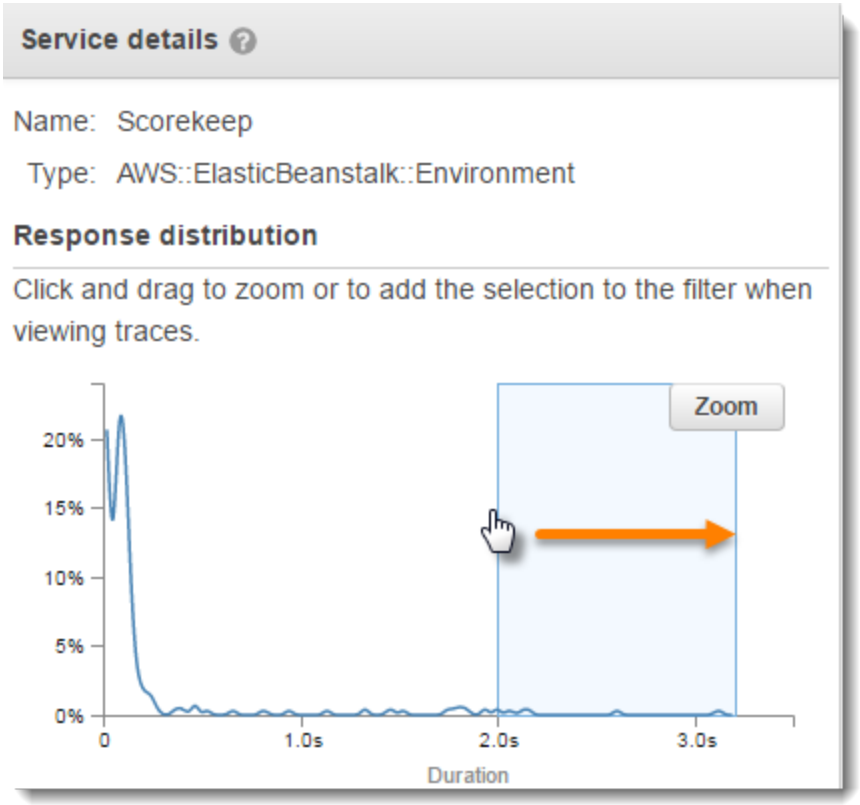
サービスの詳細の解釈

サービス ヒストグラムおよびエッジヒストグラムは、レイテンシーをサービスまたはリクエスト元の視点からビジュアルに表現したものです。

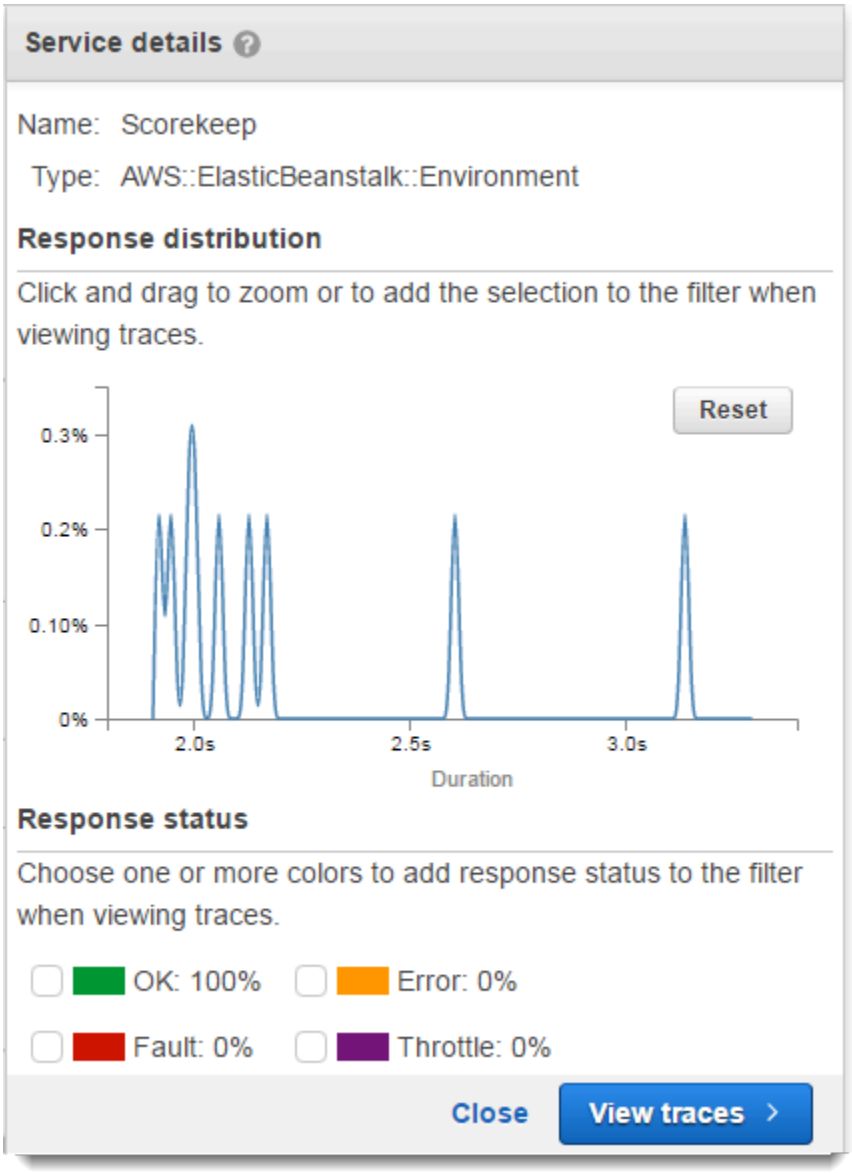
- 円をクリックして、[service node] (サービス ノード)を選択します。X-Ray は、サービスによる依頼者のヒストグラムを示します。このレイテンシーは、サービスによって記録されたものであり、サービスと依頼者の間のネットワークレイテンシーは含まれません。
- 2 つのサービスの間のエッジの線、または矢の先をクリックして、[edge] (エッジ) を選択します。X-Rayは、ダウンストリームサービスによって行われる依頼者のリクエストのヒストグラムを示します。このレイテンシーは、依頼者によって記録されたものであり、2 つのサービスの間のネットワーク接続のレイテンシーは含まれません。

[サービスの詳細] パネルヒストグラムを解釈するには、ヒストグラムの値の大部分と異なる値を探します。このような 異常値 は、ヒストグラムのピークまたは急増としてみなすことができるため、特定のエリアのトレースを表示して、現在の状況を調査することができます。

レイテンシーでフィルタリングされたトレースを表示するには、ヒストグラムの範囲を選択します。選択の開始位置をクリックし、左から右にドラッグして、トレースフィルタに含むレイテンシーの範囲をハイライト表示します。



範囲を選択したら、[Zoom] を選択してヒストグラムの部分のみを表示し、選択範囲を絞り込みます。



表示するエリアを絞って設定したら、[View traces] を選択します。

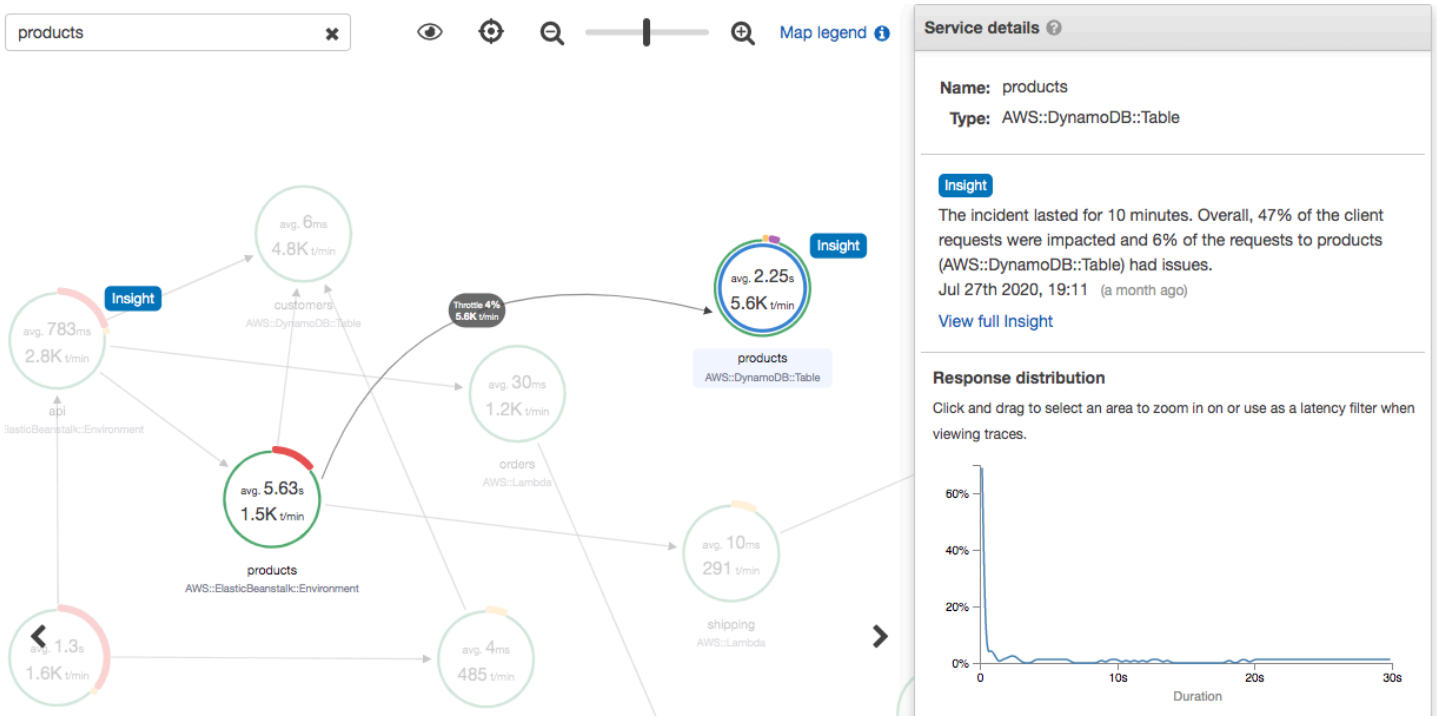
X-Ray インサイトの使用

AWS X-Ray は、アカウント内のトレースデータを継続的に分析して、アプリケーションの緊急の問題を特定します。故障率が想定範囲を超えた場合、その問題を記録し、解決されるまで影響を追跡するインサイトを作成します。インサイトを使用すると、次のことが可能になります。

- アプリケーションで問題が発生している場所、問題の根本原因、および関連する影響を特定します。　インサイトによって提供される影響分析により、問題の重要度と優先度を導き出すことができます。

- 時間の経過とともに課題が変化すると、通知を受信します。 Amazon EventBridge を使用して、インサイト通知をモニタリングおよびアラートソリューションと統合することができます。 この統合により、問題の重要度に基づいて自動メールまたはアラートを送信できます。

X-Ray コンソールは、トレースマップで進行中のインシデントがあるノードを特定します。インサイトの概要を確認するには、影響を受けるノードを選択します。左側のナビゲーションペインから[Insights] (インサイト)を選択すると、インサイトを確認およびフィルタリングすることもできます。



X-Ray は、サービスマップの 1 つ以上のノードで[anomaly] (異常)を検出すると、インサイトを作成します。このサービスは、統計モデリングを使用して、アプリケーション内のサービスの予想故障率を予測します。前の例では、異常は からの障害の増加です AWS Elastic Beanstalk。Elastic Beanstalk サーバーで複数の API コール タイムアウトが発生し、ダウンストリームノードで異常が発生しました。

X-Ray コンソールでインサイトの有効にします

インサイト機能を使用する場合は、グループごとに、インサイトを有効にする必要があります。インサイトを有効にするには、[Groups] (グループ)ページから行います。

1. [X-Ray console] (X-Ray コンソール)を開きます。

2. 既存のグループを選択するか、[Create group] (グループの作成)を選択して新しいグループを作成し、[Enable Insights] (インサイトを有効にする)を選択します。X-Ray コンソールでのグループの構成の詳細については、グループの設定 を参照してください。
3. 左側のナビゲーションペインで、[Insights] (インサイト)を選択し、表示するインサイトを選択します。

> **Note**
>
> X-rayはGetInsightSummaries、GetInsight、GetInsightEvents、GetInsightImpactGraph API オペレーションを使用して、インサイトからデータを取得できます。
> 詳細については、「AWS X-Ray が IAM と連携する方法」を参照してください。

インサイトの通知を有効にします。

インサイト通知を使用すると、インサイトが作成されたとき、大幅な変更、クローズされたときなど、インサイトイベントごとに通知が作成されます。　お客様は Amazon EventBridge イベントを通じてこれらの通知を受け取り、条件付きルールを使用して SNS 通知、Lambda 呼び出し、SQS キューへのメッセージの投稿、または EventBridge がサポートする任意のターゲットなどのアクションを実行できます。　インサイト 通知がベストで出され -ベスト工フォートですが、保証されていません。ターゲットの詳細については、Amazon EventBridge Targets を参照してください。

インサイトが有効なグループに対してのインサイト通知は、[Groups] (グループ)ページから有効にすることができます。

X-Ray グループの通知を有効にするには

1. [X-Ray console].(X-Ray コンソール)を開きます。
2. 既存のグループを選択するか、[Create group] グループの作成)を選択して新しいグループを作成し、[Enable Insights] (インサイトを有効にする)が選択されていることを確認し、[Enable Notifications] (通知を有効にする)を選択します。X-Ray コンソールでのグループの設定の詳細については、グループの設定 を参照してください。

Amazon EventBridge 条件付きルールを設定するには

1. [Amazon EventBridge console] (Amazon EventBridge コンソール) を開きます。
2. 左側のナビゲーションバーで、[Rules] (ルール) に移動し[Create rule] (ルールの作成) を選択します。
3. ルールの名前と説明を入力します。
4. [Event pattern] (イベントパターン)を選択してから、[Custom pattern] (カスタムパターン)を選択します。`"source": [ "aws.xray" ]` とならびに `"detail-type": [ "AWS X-Ray Insight Update" ]` を含むパターンを指定します。考えられるパターンとしては、以下のようなものがあります。
   - X-Ray インサイト からのすべての受信イベントを照合するイベントパターン:

     ```
     {
     "source": [ "aws.xray" ],
     "detail-type": [ "AWS X-Ray Insight Update" ]
     }
     ```

   - 指定した **`state`** ならびに **`category`** に一致するイベントパターン:

     ```
     {
     "source": [ "aws.xray" ],
     "detail-type": [ "AWS X-Ray Insight Update" ],
     "detail": {
             "State": [ "ACTIVE" ],
             "Category": [ "FAULT" ]
       }
     }
     ```

5. イベントがこのルールに一致したときに呼び出すターゲットを選択して設定します。
6. （オプション）このルールをより簡単に識別して選択するためのタグを指定します。
7. [Create] を選択します。

> **Note**
>
> X-Ray インサイト通知は Amazon EventBridge にイベントを送信しますが、Amazon EventBridge は現在カスタマー管理キーをサポートしていません。詳細については、「での データ保護 AWS X-Ray」を参照してください。

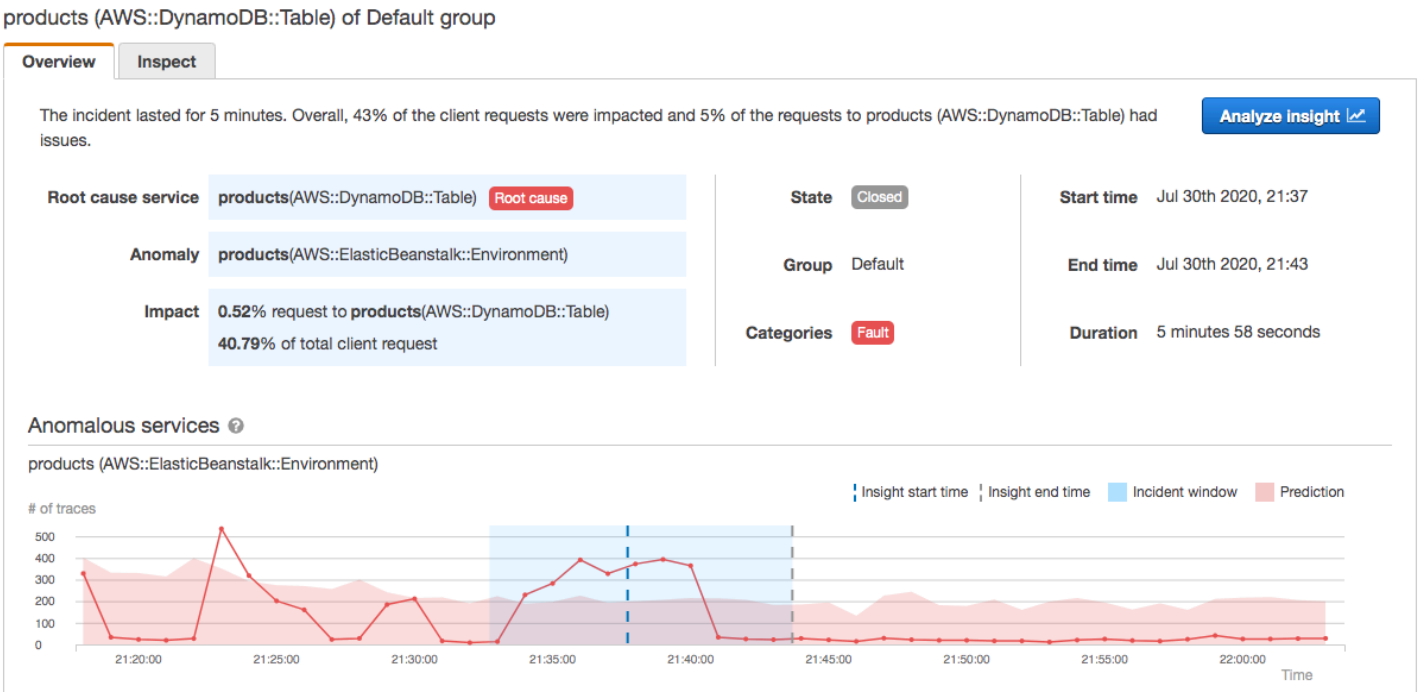
インサイト 概要

インサイト 試行tの概要ページでは、三つの重要な質問に答えます。

- 根本的な問題は何ですか？
- 根本原因は何ですか？
- その影響は何ですか？

[Anomalous services] (異常サービス)セクションには、インシデント中の故障率の変化を示す各サービスのタイムラインが表示されます。タイムラインには、記録されたトラフィック量に基づいて予想される障害数を示すソリッドバンドに障害が発生したトレース数を重ねて表示されます。インサイトの期間は、[Incident window] (インシデント ウインドウ)で表示されます。インシデントウィンドウは、X-Ray がメトリックの異常を観測したときに始まり、インサイトがアクティブである間、継続します。

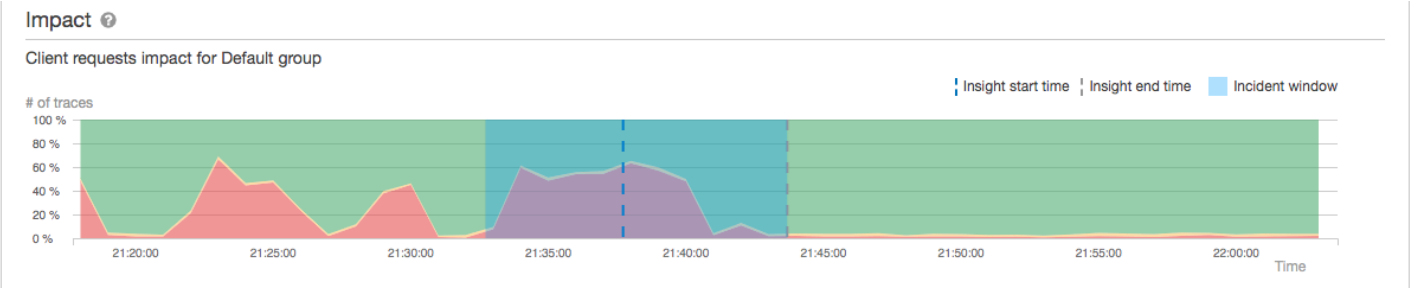
次の例は、インシデントの原因となった障害の増加を示しています。



[根本原因] セクションは、根本原因のサービスと影響を受けるパスに焦点を当てたトレースマップを示しています。根本原因マップの右上にある目のアイコンを選択すると、影響を受けないノードを非表示にすることができます。　根本原因サービスは、X-Ray が異常を特定した最も遠いダウンストリームノードです。　これは、インストルメントしたサービス、またはサービスがインストルメントクライアントで呼び出した外部サービスを表すことができます。　例えば、計測された AWS SDK クライアントを使用して Amazon DynamoDB を呼び出すと、DynamoDB からの障害が増加すると、DynamoDB を根本原因とするインサイトが得られます。

根本原因をさらに調査するには、根本原因グラフの [View root cause details] (根本原因の詳細を表示)を選択します。[Analytics] (分析)ページを使用して、根本原因と関連メッセージを調査することができます。詳細については、「Analytics コンソールとのやり取り」を参照してください。

マップ内で続くアップストリームの障害は、複数のノードに影響し、複数の異常を引き起こす可能性があります。リクエストを行ったユーザーに障害を引き渡された場合、結果は[client fault] (クライアント障害)となります。これは、トレースマップのルートノードに発生した障害です。[Impact] (影響)グラフは、グループ全体のクライアント体験のタイムラインを表したものです。この経験は、次の状態のパーセンテージに基づいて計算されます。Fault, Error, Throttle, および Okay。



この例では、インシデントの発生時にルートノードで障害が発生したトレースの増加を示します。ダウンストリームサービスのインシデントは、クライアントエラーの増加に必ずしも対応しているとは限りません。

[Analyze insight] (インサイトを分析)を選択すると X-Ray Analytics コンソールがウィンドウで開き、インサイトの原因となる一連のトレースを深く掘り下げることができます。詳細については、「Analytics コンソールとのやり取り」を参照してください。

[Understanding impact] (影響の把握)

AWS X-Ray は、インサイトと通知の生成の一環として、進行中の問題によって引き起こされる影響を測定します。影響は、次の 2 つの方法で測定されます。

- X-Ray [group] (グループ)への影響
- 根本原因サービスへの影響

この影響は、一定の期間内に失敗またはエラーを引き起こしているリクエストの割合によって決まります。この影響分析では、特定のシナリオに基づいて、問題の重要度と優先度を導き出すことができます。この影響は、インサイト通知に加えて、コンソール体験の一部として利用できます。

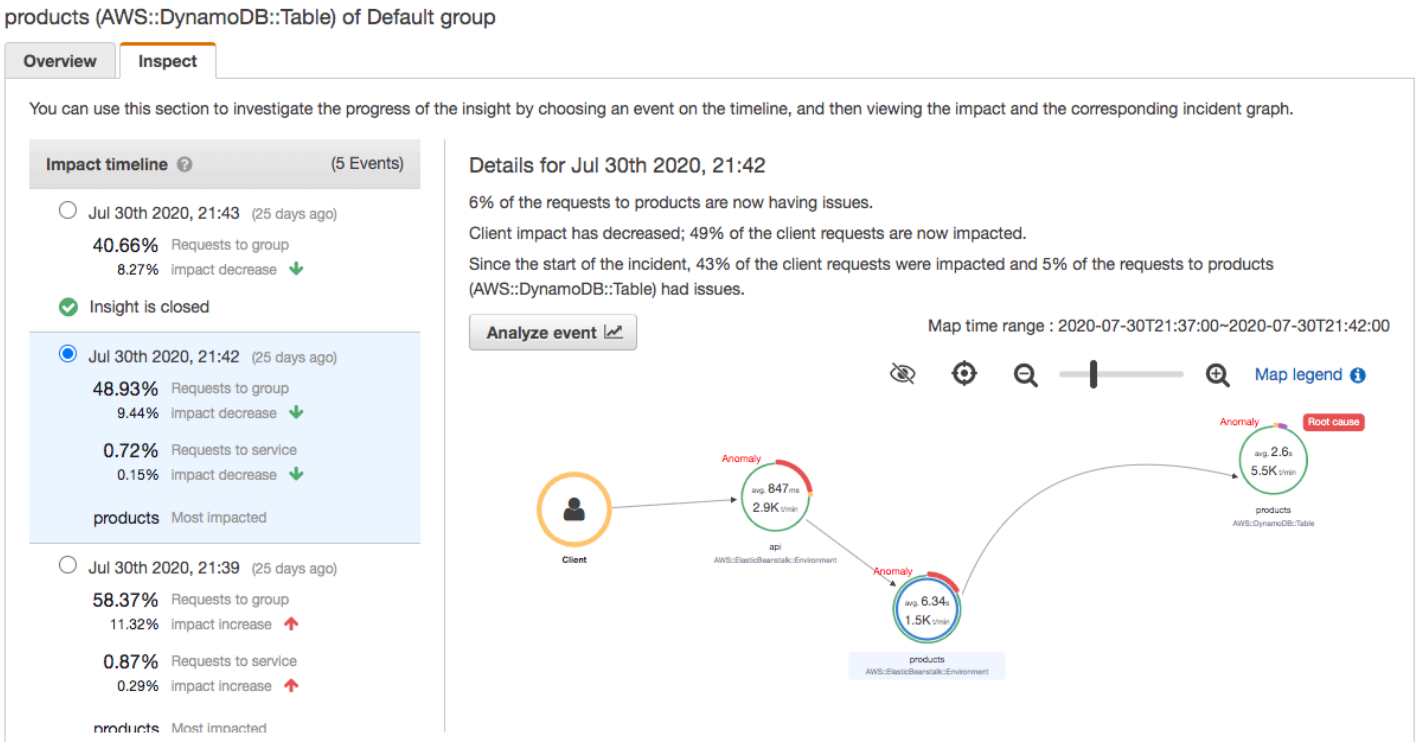
[Deduplication] (重複排除)

AWS X-Ray Insights は、複数のマイクロサービス間で問題を重複排除します。異常検出により、問題の根本原因であるサービスを特定し、他の関連サービスが同じ根本原因で異常な動作を示しているかどうかを判断し、結果を単一のインサイトとして記録します。

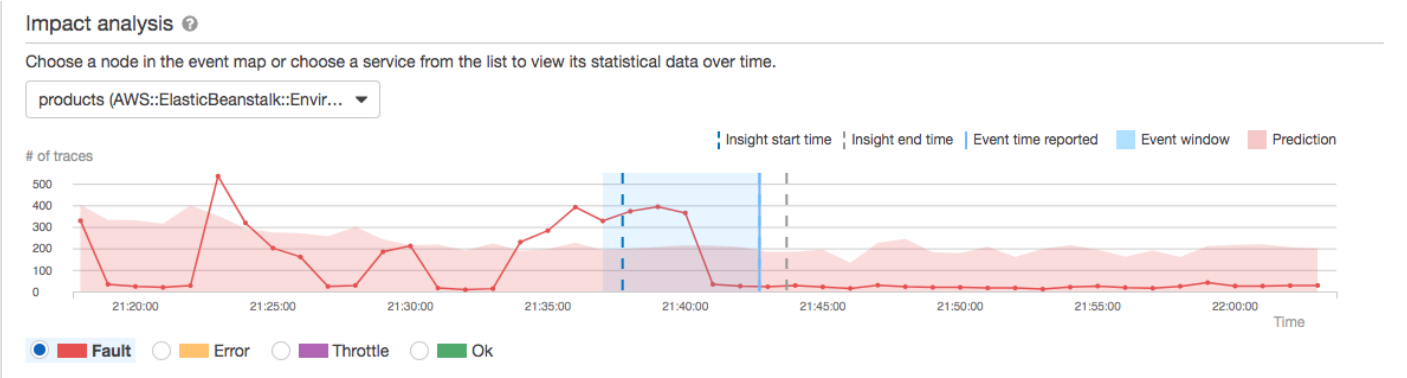
インサイトの進捗状況を確認します

X-Ray は、インサイトが解決されるまで定期的に再評価し、注目すべき中間変化をそれぞれ Amazon EventBridge イベントとして送信できる 通知 として記録します。これにより、プロセスとワークフローを構築して、問題が時間の経過とともにどのように変化したかを判断し、EventBridge を使用して電子メールの送信やアラートシステムとの統合などの適切なアクションを実行できます。

インシデントイベントは、[Inspect] (検査) ページの [Impact Timeline] (影響タイムライン) で確認することができます。デフォルトでは、別のサービスを選択するまで、タイムラインには最も影響のあるサービスが表示されます。



イベントのトレースマップとグラフを表示するには、影響タイムラインから選択します。　トレースマップには、インシデントの影響を受けるアプリケーションのサービスが表示されます。[Impact analysis] (影響分析)では、選択したノードおよびグループ内のクライアントの障害タイムラインがグラフで表示されます。



インシデントに関連するトレースを詳しく調べるには、[Inspect] (検査) ページで [Analyze event] (イベントの分析) を選択します。[Analytics] (分析) ページで、トレースのリストを絞り込み、影響を受けるユーザーを特定することができます。詳細については、「Analytics コンソールとのやり取り」を参照してください。

Analytics コンソールとのやり取り

AWS X-Ray Analytics コンソールは、トレースデータを解釈して、アプリケーションとその基盤となるサービスのパフォーマンスをすばやく理解するためのインタラクティブなツールです。このコンソールを使用すると、インタラクティブな応答時間グラフと時系列グラフを使用して、トレースを調査、分析、および視覚化できます。

Analytics コンソールで選択すると、コンソールは選択したすべてのトレースのサブセットを反映するようにフィルタを作成します。現在のトレースセットに関連付けられているグラフとメトリクスおよびフィールドのパネルをクリックして、アクティブなデータセットをきめ細かく絞り込むことができます。

トピック

- コンソールの機能
- 応答時間ディストリビューション
- 時系列アクティビティ
- ワークフローの例
- サービスグラフの障害を確認する
- ピーク応答時間を特定する
- ステータスコードでマークされているすべてのトレースを表示する
- サブグループ内のすべての項目と、ユーザーに関連付けられているすべての項目を表示する
- 異なる条件のあるトレースのセット 2 つを比較する

- 目的のトレースを特定して、詳細を表示する

コンソールの機能

X-Ray Analytics コンソールでは、以下の主要な機能を使用して、トレースデータをグループ化、フィルタリング、比較、定量化します。

機能

機能	説明
グループ	最初に選択されるグループは、`Default` です。取得されたグループを変更するには、メインフィルタ式検索バーの右側にあるメニューから別のグループを選択します。グループの詳細については、「グループでフィルタ式を使用する」を参照してください。
Retrieved traces (取得されたトレース)	デフォルトでは、Analytics コンソールは選択したグループのすべてのトレースに基づいてグラフを生成します。取得されたトレースは、作業セットのすべてのトレースを表します。このタイルにトレースカウントがあります。メイン検索バーに適用したフィルタ式は、取得されたトレースを絞り込んで更新します。
Show in charts/Hide from charts (グラフに表示/グラフに非表示)	アクティブなグループを切り替え、取得されたトレースと比較します。グループに関連するデータをアクティブなフィルタと比較するには、[Show in charts (グラフに表示)]を選択します。グラフからこのビューを削除するには、[Hide from charts (グラフに非表示)] を選択します。
Filtered trace set A (フィルタリングされたトレースセット A)	グラフやテーブルとの対話を通じて、フィルターを適用して[Filtered trace set A] (フィルタリングされたトレースセット A)の基準を作成します。フィルターが適用されると、適用可

機能	説明
	能なトレースの数と取得された合計に対するトレースの割合がこのタイル内で計算されます。フィルタは [Filtered trace set A] タイル内のタグとして入力され、タイルから削除することもできます。
Refine (絞り込み)	この関数は、トレースセット A に適用されたフィルタに基づいて、取得したトレースのセットを更新します。取得されたトレースセットを絞り込むと、トレースセット A のフィルタに基づいて取得したすべてのトレースの処理中のセットが更新されます。取得されたトレースのワーキングセットは、グループ内のすべてのトレースをサンプリングしたサブセットです。
Filtered trace set B (フィルタリングされたトレースセット B)	作成した場合、[フィルタリングされたトレースセット B] は、[フィルタリングされたトレースセット A] の写しです。2 つのトレースセットを比較するには、トレースセット A を固定したままトレースセット B に適用する新しいフィルターを選択します。フィルタが適用されると、適用可能なトレースの数と取得された合計からのトレースの割合がこのタイル内で計算されます。フィルタは、[Filtered trace set B] タイル内にタグとして入力され、タイルから削除することもできます。

機能	説明
Response time root cause entity paths (応答時間根本原因エンティティパス)	記録されたエンティティパスのテーブル。X-Rayは、トレース内のどのパスが応答時間の最大の原因であるかを判別します。この形式は、検出されたエンティティの階層を示し、最後に応答時間の根本原因を示します。これらの行を使用して、定期的な応答時間障害をフィルタリングします。根本原因フィルタのカスタマイズと API 経由のデータ取得に関する詳細については、「根本原因分析の取得と絞り込み」を参照してください。
Delta (�) (デルタ)	トレースセット A とトレースセット B の両方がアクティブなときにメトリクステーブルに追加される列。Deltaデルタ列は、トレースセット A とトレースセット B の間のトレースの割合の差を計算します。

応答時間ディストリビューション

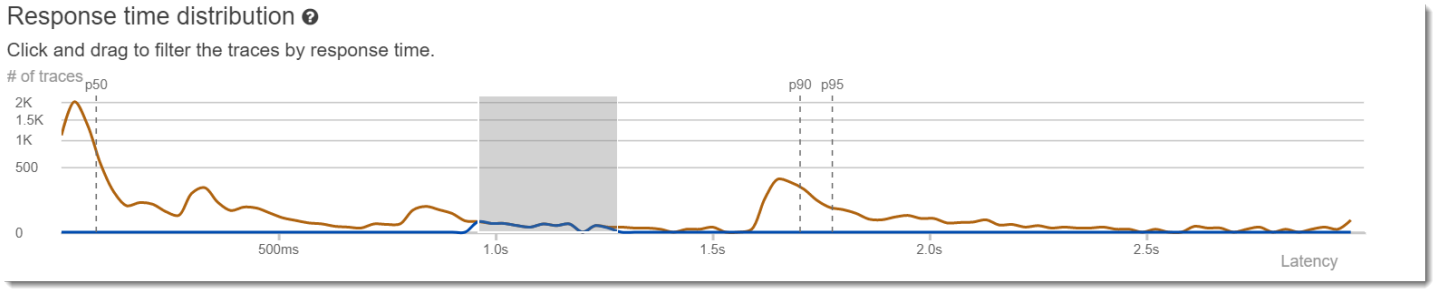
X-Ray Analytics コンソールは、トレースの視覚化に役立つ、[Response Time Distribution] (応答時間ディストリビューション) および [Time Series Activity] (時系列アクティビティ) の 2 つの主要なグラフが生成されます。このセクションと続く部分ではそれぞれの例をあげて、グラフを読み取る方法の基本について説明します。

応答時系列グラフに関連する色は次のとおりです (時系列グラフは同じカラースキームを使用します)。

- [All traces in the group] (グループのすべてのトレース) - グレー
- [Retrieved traces] (取得されたトレース) - オレンジ
- [Filtered trace set A] (フィルタリングされたトレースセット A) - 緑
- [Filtered trace set B] (フィルタリングされたトレースセット B) - 青

Example - 応答時間ディストリビューション

応答時間ディストリビューションは、指定された応答時間でのトレース数を示すグラフです。クリックしてドラッグし、応答時間ディストリビューションで選択を行います。これは、特定の応答時間内のすべてのトレースに対して、`responseTime`という名前の作業トレースセットにフィルターを選択して作成します。

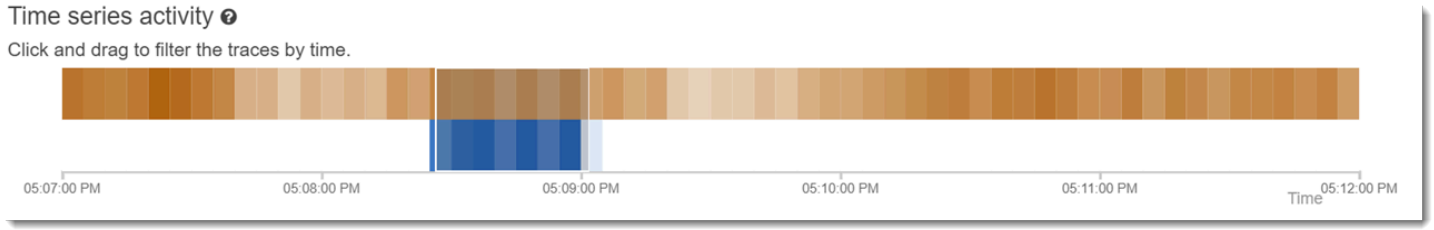


時系列アクティビティ

時系列アクティビティグラフには、指定された期間のトレースの数が表示されます。カラーインジケータは、応答時間ディストリビューションの折れ線グラフの色を反映しています。アクティビティシリーズのカラーブロックが濃く、密になるほど、より多くのトレースが指定された期間に表示されます。

Example - 時系列アクティビティ

クリックしてドラッグし、時系列アクティビティグラフ内で選択を行います。これは、特定の時間範囲内のすべてのトレースに対して、作業トレースセットに指定されている `timerange` という名前のフィルタを選択して作成します。



ワークフローの例

以下の例では、X-Ray Analytics コンソールの一般的なユースケースを示します。各例では、コンソールエクスペリエンスの主要な機能を示します。グループとして、例では基本的なトラブルシューティングワークフローに従います。このステップでは、最初に異常なノードを特定し、次に Analytics コンソールを操作して比較クエリを自動的に生成する方法を示します。クエリによってス

コープを絞り込んだら、最終的に関心のあるトレースの詳細を調べて、サービスの健全性を損なう原因を特定します。

サービスグラフの障害を確認する

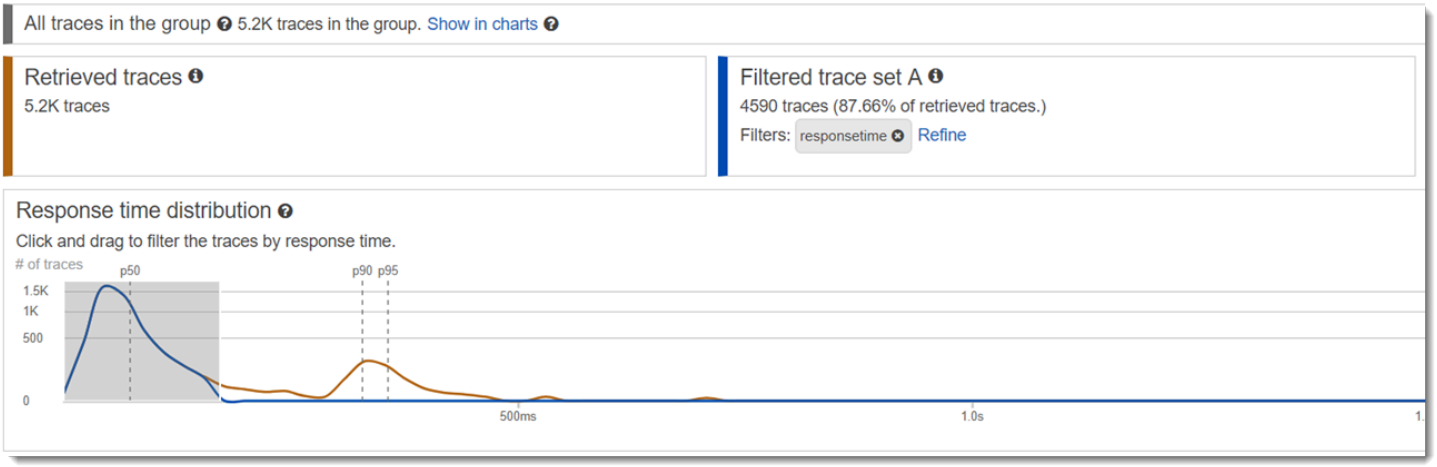
このトレースマップは、各ノードの状態をエラーと障害に対する正常な呼び出しの比率に基づいて色分けしたものです。赤色の割合が表示されたら、そのシグナルノードに障害が発生しています。X-Ray Analytics コンソールを使用して、調査を実施します。

トレースマップを読み取る方法の詳細については、「トレースマップの表示」を参照してください。

ピーク応答時間を特定する

応答時間ディストリビューションを使用して、応答時間のピークを確認できます。応答時間のthe ピークを選択すると、グラフの下のテーブルが更新され、ステータスコードなど、関連付けられているすべてのメトリクスが表示されます。

クリックしてドラッグすると、X-Rayによってフィルターが選択され、作成されます。これは、グラフ化された線の上にグレーの影で表示されます。影をディストリビューションに沿って左右にドラッグして選択内容とフィルタを更新できるようになりました。



ステータスコードでマークされているすべてのトレースを表示する

グラフの下にあるメトリクステーブルを使用して、選択したピーク内のトレースを詳しく調べることができます。[HTTP STATUS CODE] テーブルの行をクリックすると、作業データセットにフィルタが自動的に作成されます。たとえば、ステータスコード 500 のトレースをすべて表示することができます。これにより `http.status` という名前のトレースセットタイルにフィルタタグが作成されます。

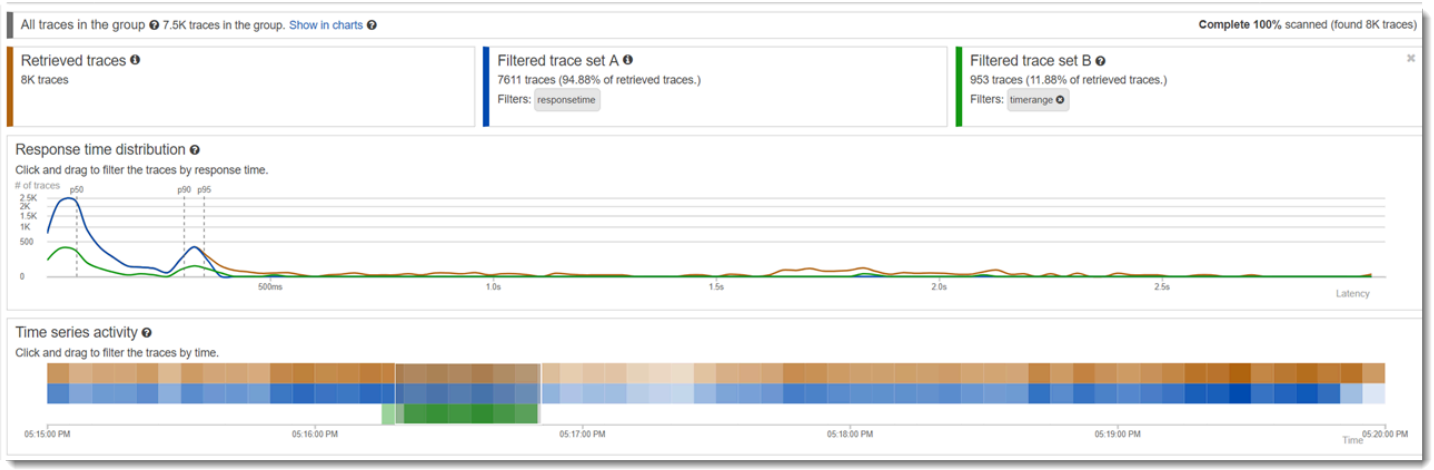
サブグループ内のすべての項目と、ユーザーに関連付けられているすべての項目を表示する

ユーザー、URL、応答時間の根本原因、またはその他の事前定義された属性に基づいてエラーセットを詳しく調べます。たとえば、500 ステータスコードでトレースのセットをさらにフィルタリングするには、[USERS] テーブルから行を選択します。これにより、トレースセットタイルに以前に指定された `http.status`、および `user` の 2 つのフィルタタグが作成されます。

異なる条件のあるトレースのセット 2 つを比較する

さまざまなユーザーとその POST リクエストを比較して、他の不一致や相関関係を見つけます。最初のフィルタのセットを適用します。これらは応答時間ディストリビューションの青い線で定義されます。次に、[Compare (比較)] を選択します。最初に、これによりトレースセット A にフィルタのコピーが作成されます。

続行するには、トレースセット B に適用する新しいフィルタセットを定義します。この 2 番目のセットは緑色の線で表されます。次の例は、青と緑のカラースキームに従って異なる線を示しています。



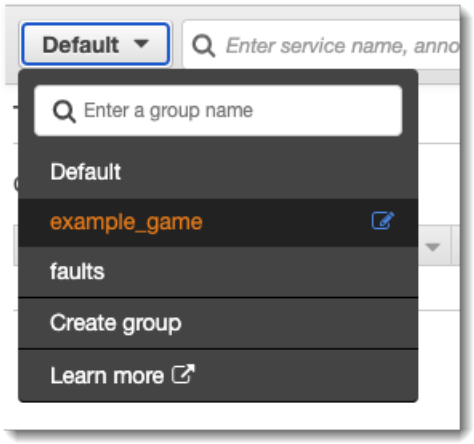
目的のトレースを特定して、詳細を表示する

コンソールフィルタを使用してスコープを絞り込むと、メトリクステーブルの下のトレースリストがより有益になります。トレースリストテーブルは、[URL]、[USER]、および [STATUS CODE] に関する情報を 1 つのビューにまとめたものです。詳細については、このテーブルから行を選択して、トレースの詳細ページを開き、タイムライン、raw データを表示します。

グループの設定

グループは、フィルタ式で定義されるトレースのコレクションです。グループを使用して、追加のサービスグラフを生成し、Amazon CloudWatch メトリクスを指定できます。 AWS X-Ray コンソールまたは X-Ray APIを使って、サービスのグループを作成および管理することができます。このトピックでは、X-Ray コンソールを使用してグループを作成および管理する方法について説明します。X-Ray API を使用してグループを管理する方法については、グループ を参照してください。

トレースマップ、トレース、または分析のトレースのグループを作成できます。グループを作成すると、グループが 3 つのページ [トレースマップ]、[トレース]、[分析] すべてのグループドロップダウンメニューでフィルターとして使用できるようになります。



グループは名前または Amazon リソースネーム (ARN) で識別され、フィルタ式を含みます。サービスは着信トレースを式と比較し、それに応じてそれらを保管します。フィルター式の作成方法の詳細については、フィルター式の使用 を参照してください。

グループのフィルタ式を更新しても、すでに記録されているデータは変わりません。更新は後続のトレースにのみ適用されます。これにより、新しい式と古い式がマージされたグラフが表示される場合があります。これを回避するには、現在のグループを削除し、新しいグループを作成します。

> **Note**
>
> グループは、フィルタ式と一致する取得済みのトレースの数で請求されます。詳細については、AWS X-Ray の料金を参照してください。

トピック

- グループを作成する
- グループを適用します
- グループを編集します
- グループのクローンを作成します
- グループの削除
- Amazon CloudWatchでグループメトリクスを表示します

グループを作成する

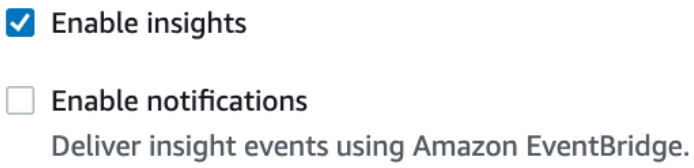
> **Note**
>
> Amazon CloudWatch コンソール内から X-Ray グループを設定できるようになりました。X-Ray コンソールを引き続き使用することもできます。

CloudWatch console

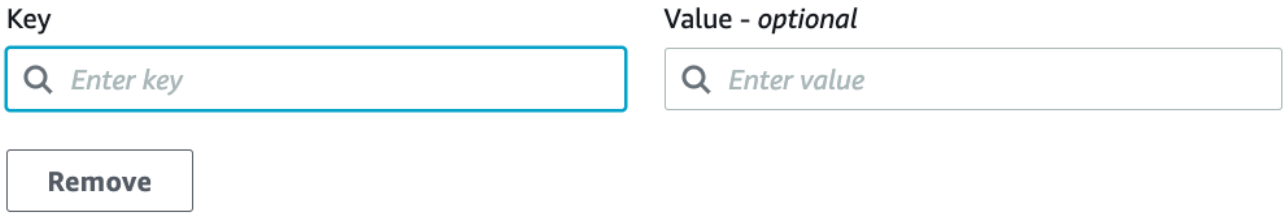
1. にサインイン AWS Management Console し、「https://https://console.aws.amazon.com/cloudwatch/.com」で CloudWatch コンソールを開きます。
2. 左側のナビゲーションペインの [設定] を選択します。
3. [X-Ray トレース] セクションの [グループ] の下にある [設定の表示] を選択します。
4. グループのリストの上にある [グループを作成] を選択します。
5. [Create group] (グループの作成) ページで、グループの名前を入力します。グループ名は最大 32 文字で、英数字とダッシュを使用できます。グループ名では大文字と小文字が区別されます。
6. フィルター式を入力します。フィルター式の作成方法の詳細については、フィルター式の使用 を参照してください。次の例では、グループによってサービスapi.example.com からの障害トレースと応答時間が 5 秒以上であったサービスへのリクエストがフィルタリングされます。

   ```
   fault = true AND http.url CONTAINS "example/game" AND responsetime >= 5
   ```

7. [Insights] (インサイト)で、グループのインサイトアクセスを有効または無効にします。インサイトの詳細については、「X-Ray インサイトの使用」を参照してください。

8. [タグ] で [新しいタグを追加] を選択してタグキーを入力し、オプションでタグ値を入力します。必要に応じて、続けてタグを追加します。タグ キーは一意である必要があります。タグを削除するには、タグの下にある [削除] を選択します。タグの詳細については、X-Ray のサンプリングルールとグループのタグ付けを参照してください。



9. [グループの作成] を選択してください。

X-Ray console

1. にサインイン AWS Management Console し、https://console.aws.amazon.com/xray/home:// www.com」で X-Ray コンソールを開きます。
2. 左側のナビゲーションペインにある [グループ] ページ、または次のいずれかのページ、[トレースマップ]、[トレース]、[分析] いずれかのページのグループメニューから[グループの作成] ページを開きます。
3. [Create group] (グループの作成) ページで、グループの名前を入力します。グループ名は最大 32 文字で、英数字とダッシュを使用できます。グループ名では大文字と小文字が区別されます。
4. フィルター式を入力します。フィルター式の作成方法の詳細については、フィルター式の使用 を参照してください。次の例では、グループによってサービスapi.example.com からの障害トレースと応答時間が 5 秒以上であったサービスへのリクエストがフィルタリングされます。

```
fault = true AND http.url CONTAINS "example/game" AND responsetime >= 5
```

5. [Insights] (インサイト)で、グループのインサイトアクセスを有効または無効にします。インサイトの詳細については、「X-Ray インサイトの使用」を参照してください。

6. [Tags] (タグ) で、タグ キー、および必要に応じてタグ値を入力します。タグを追加すると、別のタグを入力するための新しい行が表示されます。タグ キーは一意である必要があります。タグを削除するには、タグの行の最後にあるX を選択します。タグの詳細については、X-Ray のサンプリングルールとグループのタグ付けを参照してください。

7. [グループの作成] を選択してください。

グループを適用します

CloudWatch console

1. にサインイン AWS Management Console し、「https://https://console.aws.amazon.com/cloudwatch/.com」で CloudWatch コンソールを開きます。
2. X-Ray トレースの下のナビゲーションペインで、次のいずれかのページを開きます:
   - トレースマップ
   - トレース
3. [X-Ray グループでフィルタリング] フィルターにグループ名を入力します。ページに表示されるデータは、グループに設定されているフィルター式と一致するように変更されます。

X-Ray console

1. にサインイン AWS Management Console し、https://console.aws.amazon.com/xray/home:// www.com」で X-Ray コンソールを開きます。
2. ナビゲーションペイン から次のいずれかのページを開きます:
   - トレースマップ
   - トレース
   - 分析
3. グループ メニューで、the section called “グループを作成する”で作成したグループを選択します。ページに表示されるデータは、グループに設定されているフィルター式と一致するように変更されます。

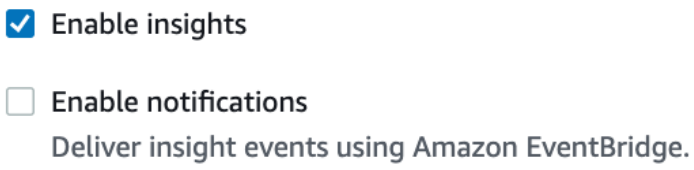
グループを編集します

CloudWatch console

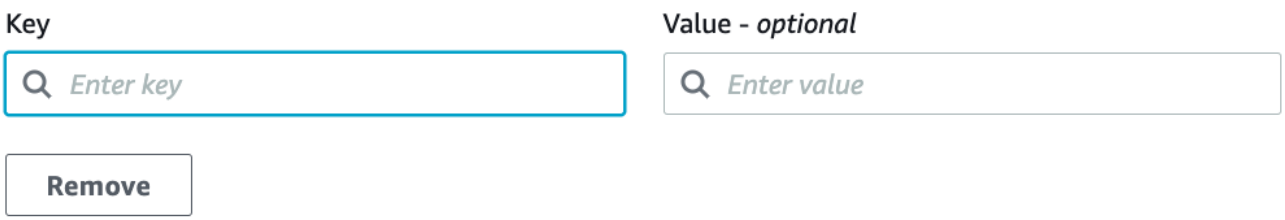
1. にサインイン AWS Management Console し、「https://https://console.aws.amazon.com/cloudwatch/.com」で CloudWatch コンソールを開きます。
2. 左側のナビゲーションペインの [設定] を選択します。
3. [X-Ray トレース] セクションの [グループ] の下にある [設定を表示] を選択します。
4. [グループ] セクションからグループを選択し、[編集] を選択します。
5. グループの名前を変更することはできませんが、フィルター式を更新することはできます。フィルター式の作成方法の詳細については、フィルター式の使用 を参照してください。次の例では、リクエストのURLアドレスに`example/game` が含まれ、リクエストの応答時間が5秒以上であるサービス `api.example.com` からの障害トレース を、グループでフィルタリングしています。

```
fault = true AND http.url CONTAINS "example/game" AND responsetime >= 5
```

6. [Insights] (インサイト)で、グループのインサイトアクセスを有効または無効にします。インサイトの詳細については、「X-Ray インサイトの使用」を参照してください。



7. [タグ] で [新しいタグを追加] を選択してタグキーを入力し、オプションでタグ値を入力します。必要に応じて、続けてタグを追加します。タグ キーは一意である必要があります。タグを削除するには、タグの下にある [削除] を選択します。タグの詳細については、X-Ray のサンプリングルールとグループのタグ付けを参照してください。



8. グループの更新が完了したら、[グループの更新] を選択します。

X-Ray console

1. にサインイン AWS Management Console し、https://console.aws.amazon.com/xray/home:// www.com」で X-Ray コンソールを開きます。
2. 次のいずれかを実行して [Edit group] (グループの編集ページ) を開きます。

   a. [Groups] (グループ) ページで、グループ名を選択して編集します。

   b. 次のいずれかのページの グループ メニューで、グループを挙げて、[Edit] (編集) を選びます。

   - トレースマップ
   - トレース
   - 分析

3. グループの名前を変更することはできませんが、フィルター式を更新することはできます。フィルター式の作成方法の詳細については、フィルター式の使用 を参照してください。次の例では、リクエストのURLアドレスに`example/game` が含まれ、リクエストの応答時間が5秒以上であるサービス `api.example.com` からの障害トレース を、グループでフィルタリングしています。

```
fault = true AND http.url CONTAINS "example/game" AND responsetime >= 5
```

4. Insights (インサイト)では、グループのインサイトおよびインサイト通知を有効または無効にします。インサイトの詳細については、「X-Ray インサイトの使用」を参照してください。

5. [Tags] (タグ) で、タグキーと値を編集します。タグキーは一意である必要があります。タグ値は任意であり、 必要であれば値 を削除できます。タグを削除するには、タグの行の最後にあるXを選択します。タグの詳細については、X-Ray のサンプリングルールとグループのタグ付けを参照してください。

6. グループの更新が完了したら、[グループの更新] を選択します。

グループのクローンを作成します

グループを複製すると、既存のグループのフィルター式とタグを持つ新しいグループが作成されます。グループのクローンを作成すると、新しいグループの名前はクローンの元のグループの名前に-clone 付加された名前になります。

CloudWatch console

1. にサインイン AWS Management Console し、「https://https://console.aws.amazon.com/cloudwatch/.com」で CloudWatch コンソールを開きます。
2. 左側のナビゲーションペインの [設定] を選択します。
3. [X-Ray トレース] セクションの [グループ] の下にある [設定を表示] を選択します。
4. [グループ] セクションからグループを選択し、[クローン] を選択します。
5. [Create group] (グループを作成する)ページで、グループの名前は*[group-name]* (グループ名) -clone。必要に応じて、グループの新しい名前を入力します。グループ名は最大 32 文字で、英数字とダッシュを使用できます。グループ名では大文字と小文字が区別されます。
6. フィルター式を既存のグループから保持するか、オプションで新しいフィルター式を入力できます。フィルター式の作成方法の詳細については、フィルター式の使用 を参照してください。次の例では、グループによってサービス api.example.com からの障害トレースと応答時間が 5 秒以上であったサービスへのリクエストがフィルタリングされます。

   ```
   service("api.example.com") { fault = true OR responsetime >= 5 }
   ```

7. [Tags] (タグ) で、必要に応じてタグのキーと値を編集します。タグキーは一意である必要があります。タグ値は任意であり、必要に応じて値を削除できます。タグを削除するには、タグの行の最後にある[X] を選択します。タグの詳細については、X-Ray のサンプリングルールとグループのタグ付けを参照してください。
8. [グループの作成] を選択してください。

X-Ray console

1. にサインイン AWS Management Console し、https://console.aws.amazon.com/xray/home://www.com」で X-Ray コンソールを開きます。

2. 左側のナビゲーションペインから [グループ] ページを開き、クローンを作成するグループの名前を選択します。
3. [アクション] メニューから、[グループのクローン作成] を選択します。
4. [Create group] (グループを作成する)ページで、グループの名前は*[group-name]* (グループ名) `-clone`。必要に応じて、グループの新しい名前を入力します。グループ名は最大 32 文字で、英数字とダッシュを使用できます。グループ名では大文字と小文字が区別されます。
5. フィルター式を既存のグループから保持するか、オプションで新しいフィルター式を入力できます。フィルター式の作成方法の詳細については、フィルター式の使用 を参照してください。次の例では、グループによってサービス `api.example.com` からの障害トレースと応答時間が 5 秒以上であったサービスへのリクエストがフィルタリングされます。

```
service("api.example.com") { fault = true OR responsetime >= 5 }
```

6. [Tags] (タグ) で、必要に応じてタグのキーと値を編集します。タグキーは一意である必要があります。タグ値は任意であり、必要に応じて値を削除できます。タグを削除するには、タグの行の最後にある[X] を選択します。タグの詳細については、X-Ray のサンプリングルールとグループのタグ付けを参照してください。
7. [グループの作成] を選択してください。

グループの削除

グループを削除するには、このセクションの手順に従います。[Default] (デフォルト) グループを削除することはできません。

CloudWatch console

1. にサインイン AWS Management Console し、「https://https://console.aws.amazon.com/cloudwatch/.com」で CloudWatch コンソールを開きます。
2. 左側のナビゲーションペインの [設定] を選択します。
3. [X-Ray トレース] セクションの [グループ] の下にある [設定を表示] を選択します。
4. [グループ] セクションからグループを選択し、[削除] を選択します。
5. 確認を求められたら [Delete] を選択します。

X-Ray console

1. にサインイン AWS Management Console し、https://console.aws.amazon.com/xray/home://www.com」で X-Ray コンソールを開きます。
2. 左側のナビゲーションペインから [グループ] ページを開き、削除するグループの名前を選択します。
3. [Actions] メニューで、[Delete] を選択します。
4. 確認を求められたら [Delete] を選択します。

Amazon CloudWatchでグループメトリクスを表示します

グループの作成後、着信トレースは、X-Ray サービスに格納されるときにグループのフィルター式と照合されます。各基準に一致するトレースの数を確認するメトリクスがAmazon CloudWatchに断続的に公開されます。[Edit group] (グループの編集) ページで [View metric] (メトリクスを表示)を選択すると、CloudWatch コンソールに [Metric] (メトリクス) ページが表示されます。CloudWatch メトリクスを使用する方法の詳細については、[Amazon CloudWatch User Guide] (Amazon CloudWatch ユーザーガイド) の[Using Amazon CloudWatch Metrics] (Amazon CloudWatch メトリクスの使用) を参照してください。

CloudWatch console

1. にサインイン AWS Management Console し、https://console.aws.amazon.com/cloudwatch/://www.com」で CloudWatch コンソールを開きます。
2. 左側のナビゲーションペインの [設定] を選択します。
3. [X-Ray トレース] セクションの [グループ] の下にある [設定を表示] を選択します。
4. [グループ] セクションからグループを選択し、[編集] を選択します。
5. [Edit group] (グループ編集) ページで、[View metric] (メトリック表示) を選択します。

   CloudWatch コンソール [Metrics] (メトリクス) ページを新しいタブで開きます。

X-Ray console

1. にサインイン AWS Management Console し、https://console.aws.amazon.com/xray/home://www.com」で X-Ray コンソールを開きます。
2. 左側のナビゲーションペインから [グループ] ページを開き、メトリクスを表示するグループの名前を選択します。

3. [Edit group] (グループ編集) ページで、[View metric] (メトリック表示) を選択します。

   CloudWatch コンソール [Metrics] (メトリクス) ページを新しいタブで開きます。

サンプリングルールの設定

AWS X-Ray コンソールを使用して、 サービスのサンプリングルールを設定できます。サンプリング設定でアクティブなトレースをサポートする X-Ray SDK と AWS のサービス は、サンプリングルールを使用して、記録するリクエストを決定します。

トピック

サンプリングルールの設定

以下のユースケースのサンプリングを設定できます。

- API Gateway Entrypoint — API Gateway は、サンプリングとアクティブトレースをサポートします。API ステージでアクティブトレースを有効にする方法については、「の Amazon API Gateway アクティブトレースのサポート AWS X-Ray」を参照してください。
- AWS AppSync – サンプリングとアクティブトレース AWS AppSync をサポートします。リクエストで AWS AppSync アクティブなトレースを有効にするには、AWS 「X-Ray を使用したトレース」を参照してください。
- コンピューティングプラットフォームでの X-Ray SDK の計測 – Amazon EC2、Amazon ECS、などのコンピューティングプラットフォームを使用する場合 AWS Elastic Beanstalk、アプリケーションが最新の X-Ray SDK で計測されているとサンプリングがサポートされます。

サンプリングルールのカスタマイズ

サンプリングルールをカスタマイズすることで、記録するデータの量を制御できます。また、コードを変更したり再デプロイしたりすることなく、サンプリング動作を変更することもできます。サンプリングルールにより、X-Ray SDK に一連の基準に対して記録するリクエスト数を指示します。X-Ray SDK はデフォルトで、1 秒ごとに最初のリクエストを記録し、それ以降のリクエストは 5% ずつ記録します。1 秒あたり 1 つのリクエストがリザーバです。これにより、サービスがリクエストを処理している限り、毎秒少なくとも 1 つのトレースが記録されます。5% は、リザーバサイズを超えて追加リクエストがサンプリングされるレートです。

X-Ray SDK を設定して、使用するコードに含める JSON ドキュメントからサンプリングルールを読み取ることができます。ただし、サービスで複数のインスタンスを実行するときに、各インスタンスは個別にサンプリングを実行します。これによりサンプリングされたリクエストの全体的な割合 (パーセント) が増加します。すべてのインスタンスのリザーバが実質的に結合されるからです。さらに、ローカルサンプリングルールを更新するために、コードを再デプロイする必要があります。

X-Ray コンソールでサンプリングルールを定義し、[configuring the SDK] (SDK を設定)して、X-Ray サービスからルールを読み取ることで、これら両方の問題を回避できます。このサービスでは、各ルールのリザーバを管理し、実行されているインスタンスの数に基づいて、リザーバを均等に分散させるため、使用するサービスの各インスタンスにクォータを割り当てます。リザーバの制限は、設定したルールに従って計算されます。サービスでルールが設定されているので、追加のデプロイを行わずにルールを管理できます。

> **Note**
>
> X-Ray では、ベストエフォート型の方法でサンプリングルールを適用しているため、場合によっては、有効なサンプリングレートが、設定されたサンプリングルールと完全に一致しないことがあります。しかし、時間の経過とともに、サンプリングされるリクエスト数が設定したパーセンテージに近づくはずです。

Amazon CloudWatch コンソール内から X-Ray サンプリングルールを設定できるようになりました。X-Ray コンソールを引き続き使用することもできます。

CloudWatch console

CloudWatch コンソールでサンプリングルールを設定するには

1. にサインイン AWS Management Console し、https://console.aws.amazon.com/cloudwatch/://www.com」で CloudWatch コンソールを開きます。
2. 左側のナビゲーションペインの [設定] を選択します。
3. [X-Ray トレース] セクションの [サンプリングルール] の下にある [設定を表示] を選択します。
4. ルールを作成するには、[サンプリングルールの作成] を選択します。

   ルールを編集するには、ルールを選択してから [編集] を選択します。

   ルールを削除するには、ルールを選択してから [削除] を選択します。

X-Ray console

X-Ray コンソールでのサンプリングルールを設定するには

1. [X-Ray console] (X-Ray コンソール)を開きます。
2. 左側のナビゲーションペインで [サンプリング] を選択します。
3. ルールを作成するには、[Create sampling rule] (サンプリングルールの作成) を選択します。

   ルールを編集するには、ルールの名前を選択します。

   ルールを削除するには、ルールを選択し、[Actions] (アクション) メニューを使用して削除します。

サンプリングルールオプション

次のオプションが各ルールで利用できます。文字列値では、ワイルドカードを使用して、1 つの文字列 (?) またはゼロ以上の文字 (*) に一致させることができます。

サンプリングルールオプション

- [Rule name] (ルール名) (文字列) - ルールの一意の名前。
- [Priority] (優先度) (1 ~ 9999 の整数) - サンプリングルールの優先度。サービスでは、優先度の昇順でルールが評価され、一致する最初のルールを使用してサンプリングの決定が行われます。

- [Reservoir] (リザーバ) (負ではない整数) - 固定レートを適用する前に、1 秒あたりに計測する、一致するリクエストの固定数。リザーバはサービスで直接使用されませんが、ルールを一括して使用するすべてのサービスに適用されます。
- [レート] (0 ~ 100) - リザーバの上限に達した後に機器と一致するリクエストのパーセンテージ。コンソールでサンプリングルールを設定するときは、0 から 100 までのパーセンテージを選択します。JSON ドキュメントを使用してクライアント SDK でサンプリングルールを設定する場合は、0 から 1 までのパーセンテージ値を指定します。
- [サービス名] (文字列) - トレースマップに表示される、計測されたサービスの名前です。
  - X-Ray SDK - レコーダーで設定したサービス名。
  - Amazon API Gateway - *api-name*/*stage*。
- [サービスタイプ] (文字列) - トレースマップに表示される、サービスタイプ。X-Ray SDK の場合、適切なプラグインを適用することで、サービスタイプを設定します。
  - `AWS::ElasticBeanstalk::Environment` – AWS Elastic Beanstalk 環境 (プラグイン) 。
  - `AWS::EC2::Instance` — Amazon EC2 インスタンス (プラグイン)。
  - `AWS::ECS::Container` — Amazon ECS コンテナ (プラグイン)。
  - `AWS::APIGateway::Stage` - Amazon API Gateway ステージ。
  - `AWS::AppSync::GraphQLAPI` – AWS AppSync API リクエスト。
- [Host] (ホスト) (文字列) - HTTP ホスト ヘッダーからの ホスト名。
- [HTTP method] (HTTP メソッド) (文字列) - HTTP リクエストのメソッド。
- [URL path] (URL パス) (文字列) - リクエストの URL パス。
  - X-Ray SDK - HTTP リクエスト URL のパス部分。
- リソース ARN (文字列) — サービスを実行している AWS リソースの ARN。
  - X-Ray SDK — サポートされていません。SDK では [リソース ARN] に * が設定されているルールのみを使用できます。
  - Amazon API Gateway — ステージ ARN。
- (オプション) [Attributes] (属性) (キーと値) - サンプリングデシジョンが行われたときに認識されるセグメント属性。
  - X-Ray SDK — サポートされていません。SDK は属性を指定するルールを無視します。
  - Amazon API Gateway — 元の HTTP リクエストからのヘッダー。

サンプリングルールの例

Example - リザーバなし、低レートのデフォルトルール

デフォルトルールのリザーバとレートを変更することができます。他のルールに一致しないリクエストにデフォルトのルールが適用されます。

- [Reservoir] (リザーバ): **0**
- レート: **5** (**0.05** JSON ドキュメントを使用して設定した場合)

Example - 問題のあるルートのすべてのリクエストをトレースするデバッグルール

デバッグ用に一時的に適用される、高優先度のルールです。

- [Rule name] (ルール名): **DEBUG - history updates**
- [Priority] (優先度): **1**
- [Reservoir] (リザーバ): **1**
- レート: **100** (**1** JSON ドキュメントを使用して設定した場合)
- [Service name] (サービス名): **Scorekeep**
- [Service type] (サービスタイプ): *****
- [Host] (ホスト): *****
- [HTTP method] (HTTP メソッド): **PUT**
- [URL path] (URL パス): **/history/***
- [Resource ARN] (リソース ARN): *****

Example - POST 用の最小レートが高い

- [Rule name] (ルール名): **POST minimum**
- [Priority] (優先度): **100**
- [Reservoir] (リザーバ): **10**
- レート: **10** (**.1** JSON ドキュメントを使用して設定した場合)
- [Service name] (サービス名): *****
- [Service type] (サービスタイプ): *****
- [Host] (ホスト): *****

- [HTTP method] (HTTP メソッド): **POST**
- [URL path] (URL パス): *****
- [Resource ARN] (リソース ARN): *****

サンプリングルールを使用するようにサービスを設定する

X-Ray SDK ではコンソールで設定したサンプリングルールを使用するため追加の設定が必要です。サンプリング戦略を設定する方法の詳細については、使用言語の「設定」トピックを参照してください。

- Java: サンプリングルール
- Go: サンプリングルール
- Node.js: サンプリングルール
- Python: サンプリングルール
- Ruby: サンプリングルール
- .NET: サンプリングルール

API Gatewayについては、の Amazon API Gateway アクティブトレースのサポート AWS X-Ray を参照してください。

サンプリング結果の表示

X-Ray コンソールの [Sampling] (サンプリング) ページには、サービスでの各サンプリングルールの使用方法に関する詳細情報が表示されます。

[Trend (トレンド)] 列には、直近の数分でルールがどのように使用されたのかを表示します。各列に、10 秒ウィンドウの統計が表示されます。

サンプリング統計

- [Total matched rule] (ルールに一致した総数): 対象ルールに一致するリクエストの数。この数には、対象ルールに一致する可能性があるすべてのリクエストが含まれるわけではありません。優先度の高いルールに最初に一致したリクエストのみが含まれます。
- [Total sampled] (サンプリングされた総数): 記録されたリクエスト数。
- [Sampled with fixed rate] (サンプリングの固定レート): ルールの固定レートを適用してサンプリングされたリクエスト数。

- [Sampled with reservoir limit] (サンプリングのリザーバ制限):X-Rayによって割り当てられたクォータを使用してサンプリングされたリクエスト数。
- [Borrowed from reservoir] (リザーバから借用): リザーバからの借用によって、サンプリングされたリクエスト数。サービスで初めてリクエストがルールに一致するとき、X-Rayによりクォータがまだ割り当てられていません。ただし、リザーバが少なくとも 1 である場合、X-Rayがクォータを割り当てるまで、サービスは 1 秒あたり 1 つのトレースを借用します。

サンプリングの統計およびサービスがサンプリングルールを使用する方法の詳細については、「X-Ray API でのサンプリングルールの使用」を参照してください。

次のステップ

X-Ray API を使用して、サンプリングルールを管理できます。API では、スケジュールに従って、またはアラームや通知に応答して、プログラムでルールを作成および更新することができます。手順と追加のルールの例については、AWS X-Ray API を使用したサンプリング、グループ、暗号化設定の構成 を参照してください。

X-Ray SDK および は、X-Ray API を使用してサンプリングルールの読み取り、サンプリング結果のレポート、サンプリングターゲットの取得 AWS のサービス も行います。X-Ray がサービスにクォータを割り当てていないルールにリクエストが一致するとき、サービスは、各ルールの適用、優先度に基づくルールの評価、およびリザーバからの借用に関する頻度を追跡する必要があります。サービスがサンプリングに API を使用する方法の詳細については、「X-Ray API でのサンプリングルールの使用」を参照してください。

X-Ray SDK がサンプリング API を呼び出すと、プロキシとして X-Ray デーモンを使用します。TCP ポート 2000 をすでに使用している場合、別のポートでプロキシを実行するようにデーモンを設定できます。詳細については、「AWS X-Ray デーモンの設定」を参照してください。

コンソールのディープリンク

ルートとクエリを使用して、特定のトレース、またはトレースのフィルタリングされたビュー、およびトレースマップにディープリンクできます。

コンソールページ

- ようこそページ - [xray/home#/welcom] (xray/ホーム#/ようこそ)
- はじめに - [xray/home#/getting-starte] (xray/ホーム#/はじめに)
- トレースマップ - [xray/home#/service-ma] (xray/ホーム#/サービスマップ)

- トレース - [xray/home#/trace] (xray/ホーム#/トレース)

トレース

個別のトレースのタイムライン、raw、マップビューのリンクを生成できます。

[Trace timeline] (トレースのタイムライン) - `xray/home#/traces/`*`trace-id`*

[Raw trace data] (未加工のトレースデータ) - `xray/home#/traces/`*`trace-id`*`/raw`

Example - 未加工のトレースデータ

```
https://console.aws.amazon.com/xray/home#/traces/1-57f5498f-d91047849216d0f2ea3b6442/
raw
```

フィルタ式

フィルタリングされたトレースリストへのリンク

[Filtered traces view] (フィルタリングされたトレースビュー) - `xray/home#/traces?filter=`*`filter-expression`*

Example - フィルター表現

```
https://console.aws.amazon.com/xray/home#/traces?filter=service("api.amazon.com")
 { fault = true OR responsetime > 2.5 } AND annotation.foo = "bar"
```

Example - フィルター表現 (URL エンコード)

```
https://console.aws.amazon.com/xray/home#/traces?filter=service(%22api.amazon.com
%22)%20%7B%20fault%20%3D%20true%20OR%20responsetime%20%3E%202.5%20%7D%20AND
%20annotation.foo%20%3D%20%22bar%22
```

フィルタ式の詳細については、「フィルター式の使用」を参照してください。

[時間範囲]

期間または開始時刻と終了時刻を ISO8601 形式で指定します。時間範囲は UTC で、最大で6時間にすることができます。

[Length of time] (期間) - `xray/home#/`*`page`*`?timeRange=`*`range-in-minutes`*

Example - 過去 1 時間のトレースマップ

```
https://console.aws.amazon.com/xray/home#/service-map?timeRange=PT1H
```

[Start and end time] (開始と終了の時刻) - `xray/home#/`*`page`*`?timeRange=`*`start`*`~`*`end`*

Example - 秒単位の正確な時間範囲

```
https://console.aws.amazon.com/xray/home#/traces?
timeRange=2023-7-01T16:00:00~2023-7-01T22:00:00
```

Example - 分単位の正確な時間範囲

```
https://console.aws.amazon.com/xray/home#/traces?
timeRange=2023-7-01T16:00~2023-7-01T22:00
```

リージョン

を指定 AWS リージョン して、そのリージョンのページにリンクします。リージョンを指定しない場合、コンソールは最後に利用したリージョンにリダイレクトされます。

[Region] (リージョン) - `xray/home`**`?region=`*`region`***`#/`*`page`*

Example - 米国西部 (オレゴン) (us-west-2) のトレースマップ

```
https://console.aws.amazon.com/xray/home?region=us-west-2#/service-map
```

その他のクエリパラメーターを使用してリージョンを含めると、リージョンクエリは、ハッシュ前、X-Ray 固有のクエリはページ名の後に指定されます。

Example - 米国西部 (オレゴン) (us-west-2)の直近 1 時間のトレースマップ

```
https://console.aws.amazon.com/xray/home?region=us-west-2#/service-map?timeRange=PT1H
```

結合

Example - 期間フィルターを含む最近のトレース

```
https://console.aws.amazon.com/xray/home#/traces?timeRange=PT15M&filter=duration%20%3E
%3D%205%20AND%20duration%20%3C%3D%208
```

Output

- ページ - トレース
- 時間範囲 - 最後の 15 分
- フィルター - 期間 >= 5 および 期間 <= 8

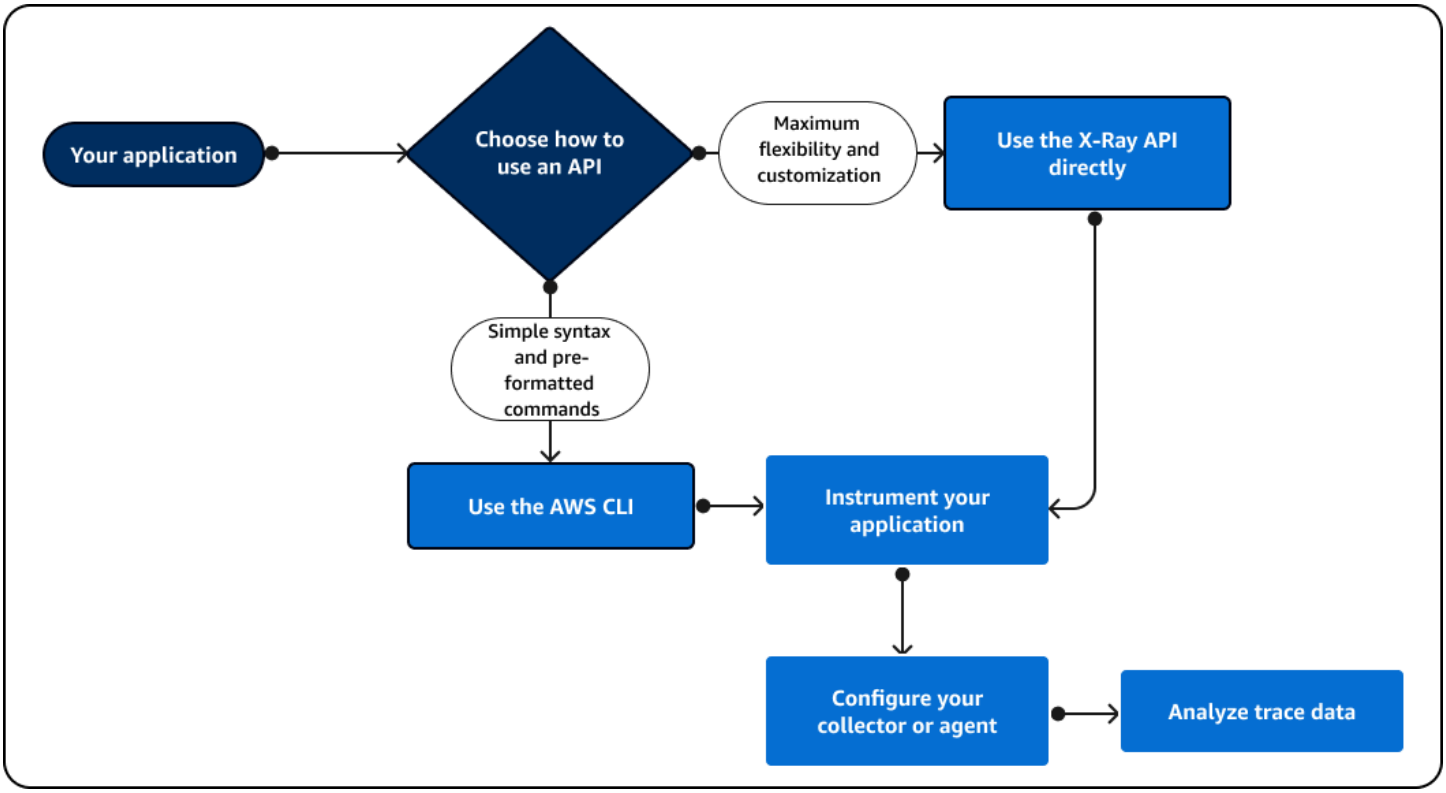
X-Ray API を使用する

X-Ray SDK がプログラミング言語をサポートしていない場合は、X-Ray APIsを直接使用するか、AWS Command Line Interface（ AWS CLI) を使用して X-Ray API コマンドを呼び出すことができます。次のガイダンスを使用して、API の操作方法を選択します。

- 事前フォーマットされたコマンドを使用するか、リクエスト内のオプションを使用して、よりシンプルな構文 AWS CLI に を使用します。
- X-Ray API を直接使用して、X-Ray に対するリクエストの柔軟性とカスタマイズを最大限活用します。

の代わりに X-Ray API を直接使用する場合は AWS CLI、リクエストを正しいデータ形式でパラメータ化する必要があり、認証と エラー処理を設定する必要もあります。

次の図は、X-Ray API の操作方法の選択のためのガイダンスを示しています。



X-Ray API を使用してトレースデータを X-Ray に直接送信します。X-Ray API は、以下の一般的なアクションを含む、X-Ray SDK で使用可能なすべての関数をサポートしています。

- PutTraceSegments - セグメントのドキュメントを X-Ray にアップロードします。
- BatchGetTraces - トレース ID のリスト内のトレースのリストを取得します。取得した各トレースは、単一のリクエストからのセグメントドキュメントのコレクションです。
- GetTraceSummaries - トレースの ID と注釈を取得します。`FilterExpression` を指定するとトレース概要のサブセットを取得できます。
- GetTraceGraph - 特定のトレース ID のサービスグラフを取得します。
- GetServiceGraph - 受信リクエストを処理してダウンストリームリクエストを呼び出すサービスを説明する、JSON 形式のドキュメントを取得します。

アプリケーションコード内の AWS Command Line Interface （ AWS CLI) を使用して、プログラムで X-Ray とやり取りすることもできます。は、他の の関数を含め、X-Ray SDK で使用できるすべての関数 AWS CLI をサポートします AWS のサービス。次の関数は以前にリストされた API オペレーションのバージョンで、より単純な形式になっています。

- put-trace-segments - セグメントのドキュメントを X-Ray にアップロードします。

- batch-get-traces - トレース ID のリスト内のトレースのリストを取得します。取得した各トレースは、単一のリクエストからのセグメントドキュメントのコレクションです。
- get-trace-summaries - トレースの ID と注釈を取得します。`FilterExpression` を指定するとトレース概要のサブセットを取得できます。
- get-trace-graph - 特定のトレース ID のサービスグラフを取得します。
- get-service-graph - 受信リクエストを処理してダウンストリームリクエストを呼び出すサービスを説明する、`JSON` 形式のドキュメントを取得します。

開始するには、オペレーティングシステムAWS CLIの をインストールする必要があります。 は Linux、、macOSおよび Windows オペレーティングシステム AWS をサポートしています。X-Ray コマンドのリストの詳細については、「X-Ray のAWS CLI コマンドリファレンスガイド」を参照してください。

トピック

- CLI での AWS X-Ray API AWS の使用
- トレースデータを AWS X-Rayに送信する
- からのデータの取得 AWS X-Ray
- AWS X-Ray API を使用したサンプリング、グループ、暗号化設定の構成
- X-Ray API でのサンプリングルールの使用
- AWS X-Ray セグメントドキュメント

CLI での AWS X-Ray API AWS の使用

AWS CLI を使用すると、X-Ray サービスに直接アクセスし、X-Ray コンソールがサービスグラフと raw トレースデータを取得するために使用するのと同じ APIs を使用できます。サンプルアプリケーションには、 CLI でこれらの APIs AWS を使用する方法を示すスクリプトが含まれています。

前提条件

このチュートリアルでは、Scorekeep サンプルアプリケーションと、それに含まれるトレーシングデータとサービスマップを生成するスクリプトを使用します。入門チュートリアルの指示に従って、アプリケーションを起動します。

このチュートリアルでは AWS CLI 、 を使用して X-Ray API の基本的な使用方法を示します。Windows、Linux、および OS-X で使用できる AWS CLI は、すべての APIs へのコマンドラインアクセスを提供します AWS のサービス。

> **Note**
>
> AWS CLI が Scorekeep サンプルアプリケーションが作成されたリージョンと同じリージョンに設定されていることを確認する必要があります。

サンプルアプリケーションに含まれるテスト用のスクリプトは、cURL を使用してトラフィックを API および jq に送信し、出力を解析します。jq 実行可能ファイルは stedolan.github.io から、curl 実行可能ファイルは https://curl.haxx.se/download.html からダウンロードできます。ほとんどの Linux および OS X インストールには cURL が含まれています。

トレースデータの生成

ゲームの進行中、ウェブアプリケーションは数秒ごとに API にトラフィックを生成し続けますが、生成されるリクエストのタイプは 1 つだけです。test-api.sh スクリプトを使用して、エンドツーエンドのシナリオを実行し、API のテスト中により多様なトレースデータを生成します。

test-api.sh スクリプトを使用するには

1. Elastic Beanstalk コンソールを開きます。
2. 環境に対応するマネジメントコンソールに移動します。
3. 環境 [URL] をページヘッダーからコピーします。
4. bin/test-api.sh を開いて API の値を環境の URL に置き換えます。

```
#!/bin/bash
API=scorekeep.9hbtbm23t2.us-west-2.elasticbeanstalk.com/api
```

5. スクリプトを実行して API へのトラフィックを生成します。

```
~/debugger-tutorial$ ./bin/test-api.sh
Creating users,
session,
game,
configuring game,
playing game,
ending game,
game complete.
{"id":"MTBP8BAS","session":"HUF6IT64","name":"tic-tac-toe-test","users":
["QFF3HBGM","KL6JR98D"],"rules":"102","startTime":1476314241,"endTime":1476314245,"states":
```

```
["JQVLEOM2","D67QLPIC","VF9BM9NC","OEAA6GK9","2A705O73","1U2LFTLJ","HUKIDD70","BAN1C8FI","G
["BS8F8LQ","4MTTSPKP","463OETES","SVEBCL3N","N7CQ1GHP","O84ONEPD","EG4BPROQ","V4BLIDJ3","9R
```

X-Ray API を使用する

AWS CLI は、 GetServiceGraphや など、X-Ray が提供するすべての API アクションのコマンドを提供しますGetTraceSummaries。サポートされているすべてのアクションおよびそこで使用するデータ型の詳細については、「AWS X-Ray API リファレンス」を参照してください。

Example bin/service-graph.sh

```
EPOCH=$(date +%s)
aws xray get-service-graph --start-time $(($EPOCH-600)) --end-time $EPOCH
```

このスクリプトは直近 10 分のサービスグラフを取得します。

```
~/eb-java-scorekeep$ ./bin/service-graph.sh | less
{
    "StartTime": 1479068648.0,
    "Services": [
        {
            "StartTime": 1479068648.0,
            "ReferenceId": 0,
            "State": "unknown",
            "EndTime": 1479068651.0,
            "Type": "client",
            "Edges": [
                {
                    "StartTime": 1479068648.0,
                    "ReferenceId": 1,
                    "SummaryStatistics": {
                        "ErrorStatistics": {
                            "ThrottleCount": 0,
                            "TotalCount": 0,
                            "OtherCount": 0
                        },
                        "FaultStatistics": {
                            "TotalCount": 0,
                            "OtherCount": 0
                        },
                        "TotalCount": 2,
                        "OkCount": 2,
```

```
                        "TotalResponseTime": 0.054000139236450195
                    },
                    "EndTime": 1479068651.0,
                    "Aliases": []
                }
            ]
        },
        {
            "StartTime": 1479068648.0,
            "Names": [
                "scorekeep.elasticbeanstalk.com"
            ],
            "ReferenceId": 1,
            "State": "active",
            "EndTime": 1479068651.0,
            "Root": true,
            "Name": "scorekeep.elasticbeanstalk.com",
...
```

Example bin/trace-urls.sh

```
EPOCH=$(date +%s)
aws xray get-trace-summaries --start-time $(($EPOCH-120)) --end-time $(($EPOCH-60)) --
query 'TraceSummaries[*].Http.HttpURL'
```

このスクリプトは 1 分前から 2 分前の間に生成されたトレースの URL を取得します。

```
~/eb-java-scorekeep$ ./bin/trace-urls.sh
[
    "http://scorekeep.elasticbeanstalk.com/api/game/6Q0UE1DG/5FGLM9U3/
endtime/1479069438",
    "http://scorekeep.elasticbeanstalk.com/api/session/KH4341QH",
    "http://scorekeep.elasticbeanstalk.com/api/game/GLQBJ3K5/153AHDIA",
    "http://scorekeep.elasticbeanstalk.com/api/game/VPDL672J/G2V41HM6/
endtime/1479069466"
]
```

Example bin/full-traces.sh

```
EPOCH=$(date +%s)
TRACEIDS=$(aws xray get-trace-summaries --start-time $(($EPOCH-120)) --end-time
 $(($EPOCH-60)) --query 'TraceSummaries[*].Id' --output text)
```

```
aws xray batch-get-traces --trace-ids $TRACEIDS --query 'Traces[*]'
```

このスクリプトは 1 分前から 2 分前の間に生成されたトレース全体を取得します。

```
~/eb-java-scorekeep$ ./bin/full-traces.sh | less
[
    {
        "Segments": [
            {
                "Id": "3f212bc237bafd5d",
                "Document": "{\"id\":\"3f212bc237bafd5d\",\"name\":\"DynamoDB\",
\"trace_id\":\"1-5828d9f2-a90669393f4343211bc1cf75\",\"start_time\":1.479072242459E9,
\"end_time\":1.479072242477E9,\"parent_id\":\"72a08dcf87991ca9\",\"http\":
{\"response\":{\"content_length\":60,\"status\":200}},\"inferred\":true,\"aws\":
{\"consistent_read\":false,\"table_name\":\"scorekeep-session-xray\",\"operation\":
\"GetItem\",\"request_id\":\"QAKE0S8DD0LJM245KAOPMA746BVV4KQNSO5AEMVJF66Q9ASUAAJG\",
\"resource_names\":[\"scorekeep-session-xray\"]},\"origin\":\"AWS::DynamoDB::Table\"}"
            },
            {
                "Id": "309e355f1148347f",
                "Document": "{\"id\":\"309e355f1148347f\",\"name\":\"DynamoDB\",
\"trace_id\":\"1-5828d9f2-a90669393f4343211bc1cf75\",\"start_time\":1.479072242477E9,
\"end_time\":1.479072242494E9,\"parent_id\":\"37f14ef837f00022\",\"http\":
{\"response\":{\"content_length\":606,\"status\":200}},\"inferred\":true,\"aws\":
{\"table_name\":\"scorekeep-game-xray\",\"operation\":\"UpdateItem\",\"request_id
\":\"388GEROC4PCA6D59ED3CTI5EEJVV4KQNSO5AEMVJF66Q9ASUAAJG\",\"resource_names\":
[\"scorekeep-game-xray\"]},\"origin\":\"AWS::DynamoDB::Table\"}"
            }
        ],
        "Id": "1-5828d9f2-a90669393f4343211bc1cf75",
        "Duration": 0.05099987983703613
    }
...
```

クリーンアップ

Elastic Beanstalk 環境を終了し、Amazon EC2 インスタンス、DynamoDB テーブル、およびその他のリソースをシャットダウンします。

Elastic Beanstalk 環境を終了するには

1. Elastic Beanstalk コンソールを開きます。

2. 環境に対応するマネジメントコンソールに移動します。
3. [アクション] を選択します。
4. [Terminate Environment] を選択します。
5. [Terminate] (終了) を選択します。

30 日後、トレースデータは自動的に X-Ray から削除されます。

トレースデータを AWS X-Rayに送信する

トレースデータは、セグメントドキュメントの形式で X-Ray に送信できます。セグメントドキュメントは、アプリケーションがリクエストのサービスで行う作業に関する情報を含む JSON 形式の文字列です。セグメント内で行われる作業、またはサブセグメントのダウンストリームサービスおよびリソースを使用する作業に関するデータはアプリケーションに記録できます。

セグメントは、アプリケーションで行われる作業に関する情報を記録します。セグメントには、少なくとも、タスク、名前、2 つの ID で使用される時間が記録されます。トレース ID は、サービス間でやり取りされるリクエストを追跡します。セグメント ID は、単一のサービスのリクエストで行われる作業を追跡します。

Example 最小完了セグメント

```
{
  "name" : "Scorekeep",
  "id" : "70de5b6f19ff9a0a",
  "start_time" : 1.478293361271E9,
  "trace_id" : "1-581cf771-a006649127e371903a2de979",
  "end_time" : 1.478293361449E9
}
```

リクエストを受信したら完了するまで、プレースホルダーとして進行中のセグメントを送信できます。

Example 進行中セグメント

```
{
  "name" : "Scorekeep",
  "id" : "70de5b6f19ff9a0b",
  "start_time" : 1.478293361271E9,
```

```
  "trace_id" : "1-581cf771-a006649127e371903a2de979",
  “in_progress”: true
}
```

セグメントは、「PutTraceSegments」、または「X-Ray デーモンを通じて」直接 X-Ray に送信できます。

ほとんどのアプリケーションは、 AWS SDK を使用して他の サービスを呼び出したり、リソースにアクセスしたりします。サブセグメントのダウンストリーム呼び出しに関する情報を記録します。X-Ray はサブセグメントを使用して、セグメントを送信しないダウンストリームサービスを識別し、そのエントリをサービスグラフに作成します。

サブセグメントはフルセグメントドキュメントに埋め込むことも、個別に送信することもできます。サブセグメントを個別に送信して、長期実行されているリクエストのダウンストリーム呼び出しを非同期でトレースしたり、セグメントドキュメントの最大サイズ (64 kB) を超えないようにしたりできます。

Example サブセグメント

サブセグメントには subsegment の type および親セグメントを識別する parent_id があります。

```
{
  "name" : "www2.example.com",
  "id" : "70de5b6f19ff9a0c",
  "start_time" : 1.478293361271E9,
  "trace_id" : "1-581cf771-a006649127e371903a2de979"
  “end_time” : 1.478293361449E9,
  “type” : “subsegment”,
  “parent_id” : “70de5b6f19ff9a0b”
}
```

セグメントとサブセグメントに含めることができるフィールドと値の詳細については、「AWS X-Ray セグメントドキュメント」を参照してください。

セクション

- トレース ID を生成する
- PutTraceSegments を使用する
- セグメントドキュメントを X-Ray デーモンに送信する

トレース ID を生成する

X-Ray にデータを送信するには、リクエストごとに一意のトレース ID を生成する必要があります。

X-Ray のトレース ID 形式

X-Ray `trace_id` は、ハイフンで区切られた 3 つの数字で構成されています。例えば、`1-58406520-a006649127e371903a2de979` と指定します。これには、以下のものが含まれます：

- バージョン番号、すなわち、1。
- 元のリクエストの時刻。ユニックスエポックタイムで、16 進数 8 桁で表示されます。

  例えば、エポックタイムで 2016 年 12 月 1 日 10:00AM PST (太平洋標準時刻) は 1480615200 秒、または 16 進数で `58406520` と表示されます。
- グローバルに一意なトレースの 96 ビットの識別子で、24 桁の 16 進数で表示されます。

> **Note**
>
> X-Ray は、OpenTelemetry、および W3C トレースコンテキスト仕様に準拠するその他のフレームワークを使用して作成されたトレース ID をサポートするようになりました。W3C トレース ID は X-Ray に送信するとき、X-Ray トレース ID 形式でフォーマットする必要があります。例えば、W3C トレース ID `4efaaf4d1e8720b39541901950019ee5` は X-Ray に送信するとき、`1-4efaaf4d-1e8720b39541901950019ee5` の形式にする必要があります。X-Ray トレース ID には元のリクエストの Unix エポックタイムのタイムスタンプが含まれますが、これは W3C トレース ID を X-Ray 形式で送信する場合に必要ありません。

テスト用の X-Ray トレース ID を生成するためのスクリプトを記述することができます。これらはその 2 つの例です。

Python

```
import time
import os
import binascii

START_TIME = time.time()
```

```
HEX=hex(int(START_TIME))[2:]
TRACE_ID="1-{}-{}".format(HEX, binascii.hexlify(os.urandom(12)).decode('utf-8'))
```

Bash

```
START_TIME=$(date +%s)
HEX_TIME=$(printf '%x\n' $START_TIME)
GUID=$(dd if=/dev/random bs=12 count=1 2>/dev/null | od -An -tx1 | tr -d ' \t\n')
TRACE_ID="1-$HEX_TIME-$GUID"
```

トレース ID を作成し、X-Ray デーモンにセグメントを送信するスクリプトについては、Scorekeep サンプルアプリケーションを参照してください。

- Python – xray_start.py
- Bash – xray_start.sh

PutTraceSegments を使用する

セグメントドキュメントは、PutTraceSegments API を使用してアップロードできます。API には、単一のパラメーター (TraceSegmentDocuments) があり、これを実行すると、JSON セグメントドキュメントのリストが取得されます。

AWS CLI では、aws xray put-trace-segments コマンドを使用してセグメントドキュメントを直接 X-Ray に送信します。

```
$ DOC='{"trace_id": "1-5960082b-ab52431b496add878434aa25", "id": "6226467e3f845502",
 "start_time": 1498082657.37518, "end_time": 1498082695.4042, "name":
 "test.elasticbeanstalk.com"}'
$ aws xray put-trace-segments --trace-segment-documents "$DOC"
{
    "UnprocessedTraceSegments": []
}
```

> **Note**
>
> Windows コマンドプロセッサおよび Windows PowerShell では、JSON 文字列に引用符の使用やエスケープに関する要件が異なります。詳細については、ユーザーガイドの「文字列の引用 AWS CLI 」を参照してください。

この出力には、処理に失敗したセグメントが表示されます。たとえば、トレース ID の日付が遠い過去の場合は、次のようなエラーが表示されます。

```
{
    "UnprocessedTraceSegments": [
        {
            "ErrorCode": "InvalidTraceId",
            "Message": "Invalid segment. ErrorCode: InvalidTraceId",
            "Id": "6226467e3f845502"
        }
    ]
}
```

複数のセグメントドキュメントは、スペースで区切って同時に渡すことができます。

```
$ aws xray put-trace-segments --trace-segment-documents "$DOC1" "$DOC2"
```

セグメントドキュメントを X-Ray デーモンに送信する

セグメントドキュメントを X-Ray API に送信する代わりに、セグメントおよびサブセグメントを X-Ray デーモンに送信できます。これにより、バッファされた後、バッチで X-Ray API にアップロードされます。X-Ray SDK は、セグメントドキュメントをデーモンに送信して、 AWS が直接呼び出されないようにします。

> **Note**
>
> デーモンを実行する方法については、「ローカルで X-Ray; デーモンを実行する」を参照してください。

UDP ポート 2000 経由で JSON でセグメントを送信します。先頭にはデーモンのヘッダー `{"format": "json", "version": 1}\n` を追加します。

```
{"format": "json", "version": 1}\n{"trace_id": "1-5759e988-bd862e3fe1be46a994272793",
 "id": "defdfd9912dc5a56", "start_time": 1461096053.37518, "end_time": 1461096053.4042,
 "name": "test.elasticbeanstalk.com"}
```

Linux では、セグメントドキュメントを Bash ターミナルからデーモンに送信できます。ヘッダーおよびセグメントドキュメントをテキストファイルに保存し、`cat` を使用して `/dev/udp` にパイプします。

```
$ cat segment.txt > /dev/udp/127.0.0.1/2000
```

Example segment.txt

```
{"format": "json", "version": 1}
{"trace_id": "1-594aed87-ad72e26896b3f9d3a27054bb", "id": "6226467e3f845502",
 "start_time": 1498082657.37518, "end_time": 1498082695.4042, "name":
 "test.elasticbeanstalk.com"}
```

デーモンのログを確認し、セグメントが X-Ray に送信されていることを確認します。

```
2017-07-07T01:57:24Z [Debug] processor: sending partial batch
2017-07-07T01:57:24Z [Debug] processor: segment batch size: 1. capacity: 50
2017-07-07T01:57:24Z [Info] Successfully sent batch of 1 segments (0.020 seconds)
```

からのデータの取得 AWS X-Ray

AWS X-Ray は、送信するトレースデータを処理して、JSON で完全なトレース、トレース概要、サービスグラフを生成します。生成されたデータは、 CLI を使用して API AWS から直接取得できます。

セクション

- サービスグラフの取得
- グループ別サービスグラフの取得
- トレースの取得
- 根本原因分析の取得と絞り込み

サービスグラフの取得

JSON サービスグラフを取得するには、`GetServiceGraph` API を使用することができます。この API には、開始時間と終了時間を設定する必要があります。これらは `date` コマンドを使用して Linux 端末から計算することができます。

```
$ date +%s
1499394617
```

`date +%s` は、日付を秒単位で出力します。この数字を終了時間として使用し、日付から差し引いて、開始時間を取得します。

Example 最後の 10 分間のサービスグラフを取得するスクリプト。

```
EPOCH=$(date +%s)
aws xray get-service-graph --start-time $(($EPOCH-600)) --end-time $EPOCH
```

次の例では、4 つのノードを持つサービスグラフを示します。これには、クライアントノード、EC2 インスタンス、DynamoDB テーブル、および Amazon SNS トピックが含まれます。

Example GetServiceGraph 出力

```
{
    "Services": [
        {
            "ReferenceId": 0,
            "Name": "xray-sample.elasticbeanstalk.com",
            "Names": [
                "xray-sample.elasticbeanstalk.com"
            ],
            "Type": "client",
            "State": "unknown",
            "StartTime": 1528317567.0,
            "EndTime": 1528317589.0,
            "Edges": [
                {
                    "ReferenceId": 2,
                    "StartTime": 1528317567.0,
                    "EndTime": 1528317589.0,
                    "SummaryStatistics": {
                        "OkCount": 3,
                        "ErrorStatistics": {
                            "ThrottleCount": 0,
                            "OtherCount": 1,
                            "TotalCount": 1
                        },
                        "FaultStatistics": {
                            "OtherCount": 0,
                            "TotalCount": 0
                        },
                        "TotalCount": 4,
                        "TotalResponseTime": 0.273
                    },
                    "ResponseTimeHistogram": [
                        {
```

```
                        "Value": 0.005,
                        "Count": 1
                    },
                    {
                        "Value": 0.015,
                        "Count": 1
                    },
                    {
                        "Value": 0.157,
                        "Count": 1
                    },
                    {
                        "Value": 0.096,
                        "Count": 1
                    }
                ],
                "Aliases": []
            }
        ]
    },
    {
        "ReferenceId": 1,
        "Name": "awseb-e-dixzws4s9p-stack-StartupSignupsTable-4IMSMHAYX2BA",
        "Names": [
            "awseb-e-dixzws4s9p-stack-StartupSignupsTable-4IMSMHAYX2BA"
        ],
        "Type": "AWS::DynamoDB::Table",
        "State": "unknown",
        "StartTime": 1528317583.0,
        "EndTime": 1528317589.0,
        "Edges": [],
        "SummaryStatistics": {
            "OkCount": 2,
            "ErrorStatistics": {
                "ThrottleCount": 0,
                "OtherCount": 0,
                "TotalCount": 0
            },
            "FaultStatistics": {
                "OtherCount": 0,
                "TotalCount": 0
            },
            "TotalCount": 2,
            "TotalResponseTime": 0.12
```

```
            },
            "DurationHistogram": [
                {
                    "Value": 0.076,
                    "Count": 1
                },
                {
                    "Value": 0.044,
                    "Count": 1
                }
            ],
            "ResponseTimeHistogram": [
                {
                    "Value": 0.076,
                    "Count": 1
                },
                {
                    "Value": 0.044,
                    "Count": 1
                }
            ]
        },
        {
            "ReferenceId": 2,
            "Name": "xray-sample.elasticbeanstalk.com",
            "Names": [
                "xray-sample.elasticbeanstalk.com"
            ],
            "Root": true,
            "Type": "AWS::EC2::Instance",
            "State": "active",
            "StartTime": 1528317567.0,
            "EndTime": 1528317589.0,
            "Edges": [
                {
                    "ReferenceId": 1,
                    "StartTime": 1528317567.0,
                    "EndTime": 1528317589.0,
                    "SummaryStatistics": {
                        "OkCount": 2,
                        "ErrorStatistics": {
                            "ThrottleCount": 0,
                            "OtherCount": 0,
                            "TotalCount": 0
```

```
                    },
                    "FaultStatistics": {
                        "OtherCount": 0,
                        "TotalCount": 0
                    },
                    "TotalCount": 2,
                    "TotalResponseTime": 0.12
                },
                "ResponseTimeHistogram": [
                    {
                        "Value": 0.076,
                        "Count": 1
                    },
                    {
                        "Value": 0.044,
                        "Count": 1
                    }
                ],
                "Aliases": []
            },
            {
                "ReferenceId": 3,
                "StartTime": 1528317567.0,
                "EndTime": 1528317589.0,
                "SummaryStatistics": {
                    "OkCount": 2,
                    "ErrorStatistics": {
                        "ThrottleCount": 0,
                        "OtherCount": 0,
                        "TotalCount": 0
                    },
                    "FaultStatistics": {
                        "OtherCount": 0,
                        "TotalCount": 0
                    },
                    "TotalCount": 2,
                    "TotalResponseTime": 0.125
                },
                "ResponseTimeHistogram": [
                    {
                        "Value": 0.049,
                        "Count": 1
                    },
                    {
```

```
                    "Value": 0.076,
                    "Count": 1
                }
            ],
            "Aliases": []
        }
    ],
    "SummaryStatistics": {
        "OkCount": 3,
        "ErrorStatistics": {
            "ThrottleCount": 0,
            "OtherCount": 1,
            "TotalCount": 1
        },
        "FaultStatistics": {
            "OtherCount": 0,
            "TotalCount": 0
        },
        "TotalCount": 4,
        "TotalResponseTime": 0.273
    },
    "DurationHistogram": [
        {
            "Value": 0.005,
            "Count": 1
        },
        {
            "Value": 0.015,
            "Count": 1
        },
        {
            "Value": 0.157,
            "Count": 1
        },
        {
            "Value": 0.096,
            "Count": 1
        }
    ],
    "ResponseTimeHistogram": [
        {
            "Value": 0.005,
            "Count": 1
        },
```

```
                {
                    "Value": 0.015,
                    "Count": 1
                },
                {
                    "Value": 0.157,
                    "Count": 1
                },
                {
                    "Value": 0.096,
                    "Count": 1
                }
            ]
        },
        {
            "ReferenceId": 3,
            "Name": "SNS",
            "Names": [
                "SNS"
            ],
            "Type": "AWS::SNS",
            "State": "unknown",
            "StartTime": 1528317583.0,
            "EndTime": 1528317589.0,
            "Edges": [],
            "SummaryStatistics": {
                "OkCount": 2,
                "ErrorStatistics": {
                    "ThrottleCount": 0,
                    "OtherCount": 0,
                    "TotalCount": 0
                },
                "FaultStatistics": {
                    "OtherCount": 0,
                    "TotalCount": 0
                },
                "TotalCount": 2,
                "TotalResponseTime": 0.125
            },
            "DurationHistogram": [
                {
                    "Value": 0.049,
                    "Count": 1
                },
```

```
                {
                    "Value": 0.076,
                    "Count": 1
                }
            ],
            "ResponseTimeHistogram": [
                {
                    "Value": 0.049,
                    "Count": 1
                },
                {
                    "Value": 0.076,
                    "Count": 1
                }
            ]
        }
    ]
}
```

グループ別サービスグラフの取得

グループの内容に基づきサービスグラフを呼び出すには、groupName または groupARN を含めます。以下の例では、Example1 という名前のグループへのサービスグラフの呼び出しを示します。

Example グループ Example1 の名前別サービスグラフを取得するスクリプト

```
aws xray get-service-graph --group-name "Example1"
```

トレースの取得

GetTraceSummaries API を使用して、トレースサマリのリストを取得します。トレースサマリには、ダウンロードするトレース全体 (注釈、リクエストと応答に関する情報、ID) などを識別するのに使用できる情報が含まれます。

`aws xray get-trace-summaries` を呼び出すときに、2 つの TimeRangeType フラグを使用できます。

- TraceId – デフォルト `GetTraceSummaries` の検索は TraceID 時間を使用し、計算された `[start_time, end_time)` の範囲内で開始されたトレースを返します。このタイムスタンプの範囲は、TraceId 内のタイムスタンプのエンコードに基づいて計算されるか、手動で定義できます。

- イベント時間 – AWS X-Ray では、経時的に発生するイベントを検索するために、イベントのタイムスタンプを使用してトレースを検索できます。イベント時間では、トレースの開始時間に関係なく、[start_time, end_time) の範囲内でアクティブなトレースが返されます。

aws xray get-trace-summaries コマンドを使用して、トレースサマリのリストを取得します。以下のコマンドは、デフォルトの TraceId 時間を使用して、過去 1 ～ 2 分間のトレースサマリーのリストを取得します。

Example トレースサマリを取得するスクリプト

```
EPOCH=$(date +%s)
aws xray get-trace-summaries --start-time $(($EPOCH-120)) --end-time $(($EPOCH-60))
```

Example GetTraceSummaries 出力

```
{
    "TraceSummaries": [
        {
            "HasError": false,
            "Http": {
                "HttpStatus": 200,
                "ClientIp": "205.255.255.183",
                "HttpURL": "http://scorekeep.elasticbeanstalk.com/api/session",
                "UserAgent": "Mozilla/5.0 (Windows NT 6.1; Win64; x64)
 AppleWebKit/537.36 (KHTML, like Gecko) Chrome/59.0.3071.115 Safari/537.36",
                "HttpMethod": "POST"
            },
            "Users": [],
            "HasFault": false,
            "Annotations": {},
            "ResponseTime": 0.084,
            "Duration": 0.084,
            "Id": "1-59602606-a43a1ac52fc7ee0eea12a82c",
            "HasThrottle": false
        },
        {
            "HasError": false,
            "Http": {
                "HttpStatus": 200,
                "ClientIp": "205.255.255.183",
                "HttpURL": "http://scorekeep.elasticbeanstalk.com/api/user",
```

```
                "UserAgent": "Mozilla/5.0 (Windows NT 6.1; Win64; x64)
 AppleWebKit/537.36 (KHTML, like Gecko) Chrome/59.0.3071.115 Safari/537.36",
                "HttpMethod": "POST"
            },
            "Users": [
                {
                    "UserName": "5M388M1E"
                }
            ],
            "HasFault": false,
            "Annotations": {
                "UserID": [
                    {
                        "AnnotationValue": {
                            "StringValue": "5M388M1E"
                        }
                    }
                ],
                "Name": [
                    {
                        "AnnotationValue": {
                            "StringValue": "Ola"
                        }
                    }
                ]
            },
            "ResponseTime": 3.232,
            "Duration": 3.232,
            "Id": "1-59602603-23fc5b688855d396af79b496",
            "HasThrottle": false
        }
    ],
    "ApproximateTime": 1499473304.0,
    "TracesProcessedCount": 2
}
```

出力のトレース ID を使用して、BatchGetTraces API でトレース全体を取得します。

Example BatchGetTraces コマンド

```
$ aws xray batch-get-traces --trace-ids 1-596025b4-7170afe49f7aa708b1dd4a6b
```

Example BatchGetTraces 出力


```
{
    "Traces": [
        {
            "Duration": 3.232,
            "Segments": [
                {
                    "Document": "{\"id\":\"1fb07842d944e714\",\"name\":
\"random-name\",\"start_time\":1.499473411677E9,\"end_time\":1.499473414572E9,
\"parent_id\":\"0c544c1b1bbff948\",\"http\":{\"response\":{\"status\":200}},
\"aws\":{\"request_id\":\"ac086670-6373-11e7-a174-f31b3397f190\"},\"trace_id\":
\"1-59602603-23fc5b688855d396af79b496\",\"origin\":\"AWS::Lambda\",\"resource_arn\":
\"arn:aws:lambda:us-west-2:123456789012:function:random-name\"}",
                    "Id": "1fb07842d944e714"
                },
                {
                    "Document": "{\"id\":\"194fcc8747581230\",\"name\":\"Scorekeep
\",\"start_time\":1.499473411562E9,\"end_time\":1.499473414794E9,\"http\":{\"request
\":{\"url\":\"http://scorekeep.elasticbeanstalk.com/api/user\",\"method\":\"POST\",
\"user_agent\":\"Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML,
 like Gecko) Chrome/59.0.3071.115 Safari/537.36\",\"client_ip\":\"205.251.233.183\"},
\"response\":{\"status\":200}},\"aws\":{\"elastic_beanstalk\":{\"version_label\":\"app-
abb9-170708_002045\",\"deployment_id\":406,\"environment_name\":\"scorekeep-dev\"},
\"ec2\":{\"availability_zone\":\"us-west-2c\",\"instance_id\":\"i-0cd9e448944061b4a
\"},\"xray\":{\"sdk_version\":\"1.1.2\",\"sdk\":\"X-Ray for Java\"}},\"service
\":{},\"trace_id\":\"1-59602603-23fc5b688855d396af79b496\",\"user\":\"5M388M1E
\",\"origin\":\"AWS::ElasticBeanstalk::Environment\",\"subsegments\":[{\"id\":
\"0c544c1b1bbff948\",\"name\":\"Lambda\",\"start_time\":1.499473411629E9,\"end_time
\":1.499473414572E9,\"http\":{\"response\":{\"status\":200,\"content_length\":14}},
\"aws\":{\"log_type\":\"None\",\"status_code\":200,\"function_name\":\"random-name
\",\"invocation_type\":\"RequestResponse\",\"operation\":\"Invoke\",\"request_id
\":\"ac086670-6373-11e7-a174-f31b3397f190\",\"resource_names\":[\"random-name\"]},
\"namespace\":\"aws\"},{\"id\":\"071684f2e555e571\",\"name\":\"## UserModel.saveUser
\",\"start_time\":1.499473414581E9,\"end_time\":1.499473414769E9,\"metadata\":{\"debug
\":{\"test\":\"Metadata string from UserModel.saveUser\"}},\"subsegments\":[{\"id\":
\"4cd3f10b76c624b4\",\"name\":\"DynamoDB\",\"start_time\":1.49947341469E9,\"end_time
\":1.499473414769E9,\"http\":{\"response\":{\"status\":200,\"content_length\":57}},
\"aws\":{\"table_name\":\"scorekeep-user\",\"operation\":\"UpdateItem\",\"request_id
\":\"MFQ8CGJ3JTDDVVVASUAAJGQ6NJ82F738BOB4KQNSO5AEMVJF66Q9\",\"resource_names\":
[\"scorekeep-user\"]},\"namespace\":\"aws\"}]}]}",
                    "Id": "194fcc8747581230"
                },
                {
```

```
                    "Document": "{\"id\":\"00f91aa01f4984fd\",\"name\":
\"random-name\",\"start_time\":1.49947341283E9,\"end_time\":1.49947341457E9,
\"parent_id\":\"1fb07842d944e714\",\"aws\":{\"function_arn\":\"arn:aws:lambda:us-
west-2:123456789012:function:random-name\",\"resource_names\":[\"random-name\"],
\"account_id\":\"123456789012\"},\"trace_id\":\"1-59602603-23fc5b688855d396af79b496\",
\"origin\":\"AWS::Lambda::Function\",\"subsegments\":[{\"id\":\"e6d2fe619f827804\",
\"name\":\"annotations\",\"start_time\":1.499473413012E9,\"end_time\":1.499473413069E9,
\"annotations\":{\"UserID\":\"5M388M1E\",\"Name\":\"Ola\"}},{\"id\":\"b29b548af4d54a0f
\",\"name\":\"SNS\",\"start_time\":1.499473413112E9,\"end_time\":1.499473414071E9,
\"http\":{\"response\":{\"status\":200}},\"aws\":{\"operation\":\"Publish\",
\"region\":\"us-west-2\",\"request_id\":\"a2137970-f6fc-5029-83e8-28aadeb99198\",
\"retries\":0,\"topic_arn\":\"arn:aws:sns:us-west-2:123456789012:awseb-e-
ruag3jyweb-stack-NotificationTopic-6B829NT9V5O9\"},\"namespace\":\"aws\"},{\"id\":
\"2279c0030c955e52\",\"name\":\"Initialization\",\"start_time\":1.499473412064E9,
\"end_time\":1.499473412819E9,\"aws\":{\"function_arn\":\"arn:aws:lambda:us-
west-2:123456789012:function:random-name\"}}]}",
                    "Id": "00f91aa01f4984fd"
                },
                {
                    "Document": "{\"id\":\"17ba309b32c7fbaf\",\"name\":
\"DynamoDB\",\"start_time\":1.49947341469E9,\"end_time\":1.499473414769E9,
\"parent_id\":\"4cd3f10b76c624b4\",\"inferred\":true,\"http\":{\"response
\":{\"status\":200,\"content_length\":57}},\"aws\":{\"table_name
\":\"scorekeep-user\",\"operation\":\"UpdateItem\",\"request_id\":
\"MFQ8CGJ3JTDDVVVASUAAJGQ6NJ82F738BOB4KQNSO5AEMVJF66Q9\",\"resource_names\":
[\"scorekeep-user\"]},\"trace_id\":\"1-59602603-23fc5b688855d396af79b496\",\"origin\":
\"AWS::DynamoDB::Table\"}",
                    "Id": "17ba309b32c7fbaf"
                },
                {
                    "Document": "{\"id\":\"1ee3c4a523f89ca5\",\"name\":\"SNS
\",\"start_time\":1.499473413112E9,\"end_time\":1.499473414071E9,\"parent_id\":
\"b29b548af4d54a0f\",\"inferred\":true,\"http\":{\"response\":{\"status\":200}},\"aws
\":{\"operation\":\"Publish\",\"region\":\"us-west-2\",\"request_id\":\"a2137970-
f6fc-5029-83e8-28aadeb99198\",\"retries\":0,\"topic_arn\":\"arn:aws:sns:us-
west-2:123456789012:awseb-e-ruag3jyweb-stack-NotificationTopic-6B829NT9V5O9\"},
\"trace_id\":\"1-59602603-23fc5b688855d396af79b496\",\"origin\":\"AWS::SNS\"}",
                    "Id": "1ee3c4a523f89ca5"
                }
            ],
            "Id": "1-59602603-23fc5b688855d396af79b496"
        }
    ],
    "UnprocessedTraceIds": []
```

```
}
```

トレース全体には、同一のトレース ID を使用して取得されるすべてのセグメントドキュメントからコンパイルされた、各セグメントのドキュメントが含まれます。これらのドキュメントは、アプリケーションによって X-Ray に送信されたデータを表していません。その代わりに、X-Ray サービスによって生成された処理済みドキュメントを表します。X-Ray はアプリケーションによって送信されたセグメントドキュメントをコンパイルして、完全なトレースドキュメントを作成し、セグメントドキュメントスキーマに準拠しないデータを削除します。

X-Ray は、セグメント自体を送信しないサービスへのダウンストリーム呼び出しの推測セグメントを作成します。たとえば、計測されたクライアントを使用して DynamoDB を呼び出したときに、X-Ray SDK の視点からの呼び出しに関する詳細をサブセグメントに記録します。ただし、DynamoDB は対応するセグメントを送信しません。X-Ray は、サブセグメントにある情報を使用して、トレースマップで DynamoDB リソースを表す推測セグメントを作成し、トレースドキュメントに追加します。

API から複数のトレースを取得するには、`get-trace-summaries` の出力から AWS CLI クエリを使用して抽出できるトレース ID のリストが必要です。リストから `batch-get-traces` の入力にリダイレクトし、特定の時間のトレース全体を取得します。

Example 1 分間のトレース全体を取得するスクリプト。

```
EPOCH=$(date +%s)
TRACEIDS=$(aws xray get-trace-summaries --start-time $(($EPOCH-120)) --end-time
 $(($EPOCH-60)) --query 'TraceSummaries[*].Id' --output text)
aws xray batch-get-traces --trace-ids $TRACEIDS --query 'Traces[*]'
```

根本原因分析の取得と絞り込み

GetTraceSummaries API を使用して、トレースサマリを生成すると、根本原因に基づいて絞り込まれたフィルタ式を作成するために、部分的なトレースサマリを JSON 形式で再利用できます。絞り込みのステップのウォークスルーについては、以下の例を参照してください。

Example 例 GetTraceSummaries 出力 - 応答時間の根本原因セクション

```
{
  "Services": [
    {
      "Name": "GetWeatherData",
```

```
      "Names": ["GetWeatherData"],
      "AccountId": 123456789012,
      "Type": null,
      "Inferred": false,
      "EntityPath": [
        {
          "Name": "GetWeatherData",
          "Coverage": 1.0,
          'Remote": false
        },
        {
          "Name": "get_temperature",
          "Coverage": 0.8,
          "Remote": false
        }
      ]
    },
    {
      "Name": "GetTemperature",
      "Names": ["GetTemperature"],
      "AccountId": 123456789012,
      "Type": null,
      "Inferred": false,
      "EntityPath": [
        {
          "Name": "GetTemperature",
          "Coverage": 0.7,
          "Remote": false
        }
      ]
    }
  ]
}
```

上記の出力を編集して省略することで、この JSON は一致した根本原因のエンティティのフィルタになる可能性があります。JSON に存在するすべてのフィールドについて、候補は完全に一致する必要があります。そうしないと、トレースが返されません。削除されたフィールドはワイルドカード値になります。これは、フィルタ式クエリ構造と互換性のある形式です。

Example 再フォーマットされた応答時間の根本原因

```
{
  "Services": [
```

```
    {
      "Name": "GetWeatherData",
      "EntityPath": [
        {
          "Name": "GetWeatherData"
        },
        {
          "Name": "get_temperature"
        }
      ]
    },
    {
      "Name": "GetTemperature",
      "EntityPath": [
        {
          "Name": "GetTemperature"
        }
      ]
    }
  ]
}
```

この JSON は、`rootcause.json = #[{}]` への呼び出しを通じてフィルタ式の一部として使用されます。フィルタ式を使用するクエリの詳細については、「フィルタ式」の章を参照してください。

Example JSON フィルタの例

```
rootcause.json = #[{ "Services": [ { "Name": "GetWeatherData", "EntityPath": [{ "Name":
 "GetWeatherData" }, { "Name": "get_temperature" } ] }, { "Name": "GetTemperature",
 "EntityPath": [ { "Name": "GetTemperature" } ] } ] }]
```

AWS X-Ray API を使用したサンプリング、グループ、暗号化設定の構成

AWS X-Ray にはAPIs が用意されています。 ??? ???

セクション

- 暗号化設定
- サンプリングルール
- グループ

暗号化設定

PutEncryptionConfig を使用して、暗号化に使用する AWS Key Management Service（ AWS KMS) キーを指定します。

> **Note**
>
> X-Ray は非対称 KMS キーをサポートしていません。

```
$ aws xray put-encryption-config --type KMS --key-id alias/aws/xray
{
    "EncryptionConfig": {
        "KeyId": "arn:aws:kms:us-east-2:123456789012:key/c234g4e8-39e9-4gb0-84e2-
b0ea215cbba5",
        "Status": "UPDATING",
        "Type": "KMS"
    }
}
```

キー ID には、(例に示すような) エイリアス、キー ID、または Amazon リソースネーム (ARN) を使用できます。

GetEncryptionConfigを使用して現在の設定を取得します。X-Ray が設定の適用を終了すると、ステータスが [UPDATING] から [ACTIVE] に変わります。

```
$ aws xray get-encryption-config
{
    "EncryptionConfig": {
        "KeyId": "arn:aws:kms:us-east-2:123456789012:key/c234g4e8-39e9-4gb0-84e2-
b0ea215cbba5",
        "Status": "ACTIVE",
        "Type": "KMS"
    }
}
```

KMS の使用を停止し、デフォルトの暗号化を使用するには、暗号化タイプを NONE に設定します。

```
$ aws xray put-encryption-config --type NONE
{
```

```
    "EncryptionConfig": {
        "Status": "UPDATING",
        "Type": "NONE"
    }
}
```

サンプリングルール

X-Ray API を使用して、アカウントのサンプリングルールを管理できます。タグの追加と管理の詳細については、「X-Ray のサンプリングルールとグループのタグ付け」を参照してください。

GetSamplingRulesですべてのサンプリングルールを取得します。

```
$ aws xray get-sampling-rules
{
    "SamplingRuleRecords": [
        {
            "SamplingRule": {
                "RuleName": "Default",
                "RuleARN": "arn:aws:xray:us-east-2:123456789012:sampling-rule/Default",
                "ResourceARN": "*",
                "Priority": 10000,
                "FixedRate": 0.05,
                "ReservoirSize": 1,
                "ServiceName": "*",
                "ServiceType": "*",
                "Host": "*",
                "HTTPMethod": "*",
                "URLPath": "*",
                "Version": 1,
                "Attributes": {}
            },
            "CreatedAt": 0.0,
            "ModifiedAt": 1529959993.0
        }
    ]
}
```

別のルールに一致しないすべてのリクエストにデフォルトのルールが適用されます。このルールは最も優先度が低く、削除することはできません。ただし、UpdateSamplingRuleを使用してレートとリザーバのサイズを変更できます。

Example **UpdateSamplingRule**の API 入力 – 10000-default.json

```
{
    "SamplingRuleUpdate": {
        "RuleName": "Default",
        "FixedRate": 0.01,
        "ReservoirSize": 0
    }
}
```

次の例では、以前のファイルを入力として使用し、デフォルトのルールをリザーバなしの 1% に変更します。タグはオプションです。タグを追加する場合は、タグキーが必要で、タグ値はオプションです。サンプリングルールから既存のタグを削除するための、UntagResourceの使用

```
$ aws xray update-sampling-rule --cli-input-json file://1000-default.json --tags
 [{"Key": "key_name","Value": "value"},{"Key": "key_name","Value": "value"}]
{
    "SamplingRuleRecords": [
        {
            "SamplingRule": {
                "RuleName": "Default",
                "RuleARN": "arn:aws:xray:us-east-2:123456789012:sampling-rule/Default",
                "ResourceARN": "*",
                "Priority": 10000,
                "FixedRate": 0.01,
                "ReservoirSize": 0,
                "ServiceName": "*",
                "ServiceType": "*",
                "Host": "*",
                "HTTPMethod": "*",
                "URLPath": "*",
                "Version": 1,
                "Attributes": {}
            },
            "CreatedAt": 0.0,
            "ModifiedAt": 1529959993.0
        },
```

CreateSamplingRuleを使用して追加のサンプリングルールを作成します。ルールを作成するときは、ルールフィールドの大部分を指定する必要があります。次の例では 2 つのルールを作成します。この最初のルールでは、Scorekeep サンプルアプリケーションの基本レートを設定します。これは、より優先度の高いルールに一致しない API からのすべてのリクエストに一致します。

Example **UpdateSamplingRule**の API 入力 – 9000-base-scorekeep.json

```
{
    "SamplingRule": {
        "RuleName": "base-scorekeep",
        "ResourceARN": "*",
        "Priority": 9000,
        "FixedRate": 0.1,
        "ReservoirSize": 5,
        "ServiceName": "Scorekeep",
        "ServiceType": "*",
        "Host": "*",
        "HTTPMethod": "*",
        "URLPath": "*",
        "Version": 1
    }
}
```

2 つ目のルールも Scorekeep に適用されますが、このルールはより優先度が高く具体的です。このルールは、ポーリングリクエストに関して非常に低いサンプリングレートを設定します。これらは、ゲームの状態の変更を確認するためにクライアントによって数秒ごとに行われる GET リクエストです。

Example **UpdateSamplingRule**の API 入力 – 5000-polling-scorekeep.json

```
{
    "SamplingRule": {
        "RuleName": "polling-scorekeep",
        "ResourceARN": "*",
        "Priority": 5000,
        "FixedRate": 0.003,
        "ReservoirSize": 0,
        "ServiceName": "Scorekeep",
        "ServiceType": "*",
        "Host": "*",
        "HTTPMethod": "GET",
        "URLPath": "/api/state/*",
        "Version": 1
    }
}
```

タグはオプションです。タグを追加する場合は、タグキーが必要で、タグ値はオプションです。

```
$ aws xray create-sampling-rule --cli-input-json file://5000-polling-scorekeep.json --
tags [{"Key": "key_name","Value": "value"},{"Key": "key_name","Value": "value"}]
{
    "SamplingRuleRecord": {
        "SamplingRule": {
            "RuleName": "polling-scorekeep",
            "RuleARN": "arn:aws:xray:us-east-1:123456789012:sampling-rule/polling-
scorekeep",
            "ResourceARN": "*",
            "Priority": 5000,
            "FixedRate": 0.003,
            "ReservoirSize": 0,
            "ServiceName": "Scorekeep",
            "ServiceType": "*",
            "Host": "*",
            "HTTPMethod": "GET",
            "URLPath": "/api/state/*",
            "Version": 1,
            "Attributes": {}
        },
        "CreatedAt": 1530574399.0,
        "ModifiedAt": 1530574399.0
    }
}
$ aws xray create-sampling-rule --cli-input-json file://9000-base-scorekeep.json
{
    "SamplingRuleRecord": {
        "SamplingRule": {
            "RuleName": "base-scorekeep",
            "RuleARN": "arn:aws:xray:us-east-1:123456789012:sampling-rule/base-
scorekeep",
            "ResourceARN": "*",
            "Priority": 9000,
            "FixedRate": 0.1,
            "ReservoirSize": 5,
            "ServiceName": "Scorekeep",
            "ServiceType": "*",
            "Host": "*",
            "HTTPMethod": "*",
            "URLPath": "*",
            "Version": 1,
            "Attributes": {}
        },
```

```
        "CreatedAt": 1530574410.0,
        "ModifiedAt": 1530574410.0
    }
}
```

サンプリングルールを削除するには、DeleteSamplingRuleを使用します。

```
$ aws xray delete-sampling-rule --rule-name polling-scorekeep
{
    "SamplingRuleRecord": {
        "SamplingRule": {
            "RuleName": "polling-scorekeep",
            "RuleARN": "arn:aws:xray:us-east-1:123456789012:sampling-rule/polling-
scorekeep",
            "ResourceARN": "*",
            "Priority": 5000,
            "FixedRate": 0.003,
            "ReservoirSize": 0,
            "ServiceName": "Scorekeep",
            "ServiceType": "*",
            "Host": "*",
            "HTTPMethod": "GET",
            "URLPath": "/api/state/*",
            "Version": 1,
            "Attributes": {}
        },
        "CreatedAt": 1530574399.0,
        "ModifiedAt": 1530574399.0
    }
}
```

グループ

X-Ray API を使用して、アカウントのグループを管理することができます。グループは、フィルタ式で定義されるトレースのコレクションです。グループを使用して、追加のサービスグラフを生成し、Amazon CloudWatch メトリクスを指定できます。X-Ray API を使用したサービスグラフとメトリクスの操作の詳細については、「からのデータの取得 AWS X-Ray」を参照してください。グループの詳細については、「グループの設定」を参照してください。タグの追加と管理の詳細については、「X-Ray のサンプリングルールとグループのタグ付け」を参照してください。

CreateGroup を使用してグループを作成します。タグはオプションです。タグを追加する場合は、タグキーが必要で、タグ値はオプションです。

```
$ aws xray create-group --group-name "TestGroup" --filter-expression
 "service(\"example.com\") {fault}" --tags [{"Key": "key_name","Value": "value"},
{"Key": "key_name","Value": "value"}]
{
    "GroupName": "TestGroup",
    "GroupARN": "arn:aws:xray:us-east-2:123456789012:group/TestGroup/UniqueID",
    "FilterExpression": "service(\"example.com\") {fault OR error}"
}
```

GetGroups を使用して既存のグループをすべて取得します。

```
$ aws xray get-groups
{
    "Groups": [
        {
            "GroupName": "TestGroup",
            "GroupARN": "arn:aws:xray:us-east-2:123456789012:group/TestGroup/UniqueID",
            "FilterExpression": "service(\"example.com\") {fault OR error}"
        },
  {
            "GroupName": "TestGroup2",
            "GroupARN": "arn:aws:xray:us-east-2:123456789012:group/TestGroup2/
UniqueID",
            "FilterExpression": "responsetime > 2"
        }
    ],
 "NextToken": "tokenstring"
}
```

UpdateGroup を使用してグループを更新します。タグはオプションです。タグを追加する場合は、タグキーが必要で、タグ値はオプションです。グループから既存のタグを削除するには、UntagResourceを使用します。

```
$ aws xray update-group --group-name "TestGroup" --group-arn "arn:aws:xray:us-
east-2:123456789012:group/TestGroup/UniqueID" --filter-expression
 "service(\"example.com\") {fault OR error}" --tags [{"Key": "Stage","Value": "Prod"},
{"Key": "Department","Value": "QA"}]
{
    "GroupName": "TestGroup",
    "GroupARN": "arn:aws:xray:us-east-2:123456789012:group/TestGroup/UniqueID",
    "FilterExpression": "service(\"example.com\") {fault OR error}"
```

```
}
```

DeleteGroup を使用してグループを削除します。

```
$ aws xray delete-group --group-name "TestGroup" --group-arn "arn:aws:xray:us-
east-2:123456789012:group/TestGroup/UniqueID"
    {
    }
```

X-Ray API でのサンプリングルールの使用

AWS X-Ray SDK は X-Ray API を使用して、サンプリングルールの取得、サンプリング結果のレポート、クォータの取得を行います。これらの API を使用すれば、サンプリングルールの仕組みを理解したり、X-Ray SDK でサポートされていない言語でサンプリングを実行したりできます。

まず、GetSamplingRulesを使用してすべてのサンプリングルールを取得します。

```
$ aws xray get-sampling-rules
{
    "SamplingRuleRecords": [
        {
            "SamplingRule": {
                "RuleName": "Default",
                "RuleARN": "arn:aws:xray:us-east-1::sampling-rule/Default",
                "ResourceARN": "*",
                "Priority": 10000,
                "FixedRate": 0.01,
                "ReservoirSize": 0,
                "ServiceName": "*",
                "ServiceType": "*",
                "Host": "*",
                "HTTPMethod": "*",
                "URLPath": "*",
                "Version": 1,
                "Attributes": {}
            },
            "CreatedAt": 0.0,
            "ModifiedAt": 1530558121.0
        },
        {
            "SamplingRule": {
```

```
                "RuleName": "base-scorekeep",
                "RuleARN": "arn:aws:xray:us-east-1::sampling-rule/base-scorekeep",
                "ResourceARN": "*",
                "Priority": 9000,
                "FixedRate": 0.1,
                "ReservoirSize": 2,
                "ServiceName": "Scorekeep",
                "ServiceType": "*",
                "Host": "*",
                "HTTPMethod": "*",
                "URLPath": "*",
                "Version": 1,
                "Attributes": {}
            },
            "CreatedAt": 1530573954.0,
            "ModifiedAt": 1530920505.0
        },
        {
            "SamplingRule": {
                "RuleName": "polling-scorekeep",
                "RuleARN": "arn:aws:xray:us-east-1::sampling-rule/polling-scorekeep",
                "ResourceARN": "*",
                "Priority": 5000,
                "FixedRate": 0.003,
                "ReservoirSize": 0,
                "ServiceName": "Scorekeep",
                "ServiceType": "*",
                "Host": "*",
                "HTTPMethod": "GET",
                "URLPath": "/api/state/*",
                "Version": 1,
                "Attributes": {}
            },
            "CreatedAt": 1530918163.0,
            "ModifiedAt": 1530918163.0
        }
    ]
}
```

出力には、デフォルトルールとカスタムルールが含まれています。まだサンプリングルールを作成していない場合は、「サンプリングルール」を参照してください。

優先度の昇順で受信リクエストのルールを評価します。ルールが一致したら、固定レートとリザーバのサイズを使用してサンプリングデシジョンを作成します。サンプリングされたリクエストを記録し、(トレースを目的とする) サンプリングされていないリクエストは無視します。サンプリングデシジョンが作成されたら、ルールの評価を停止します。

ルールのリザーバのサイズは、固定レートを適用する前に記録する 1 秒あたりのトレースの目標数です。リザーバはすべてのサービスに累積的に適用されるため、直接使用することはできません。ただし、0 以外の場合は、X-Ray がクォータを割り当てるまでリザーバから 1 秒に 1 個トレースを借りることが可能です。クォータを受信する前に、1 秒ごとに最初のリクエストを記録し、追加のリクエストに固定レートを適用します。固定レートは、0 ~ 1.00 (100%) の 10 進数です。

次の例は、過去 10 秒間に作成されたサンプリングデシジョンの詳細を含むGetSamplingTargetsの呼び出しを示したものです。

```
$ aws xray get-sampling-targets --sampling-statistics-documents '[
    {
        "RuleName": "base-scorekeep",
        "ClientID": "ABCDEF1234567890ABCDEF10",
        "Timestamp": "2018-07-07T00:20:06",
        "RequestCount": 110,
        "SampledCount": 20,
        "BorrowCount": 10
    },
    {
        "RuleName": "polling-scorekeep",
        "ClientID": "ABCDEF1234567890ABCDEF10",
        "Timestamp": "2018-07-07T00:20:06",
        "RequestCount": 10500,
        "SampledCount": 31,
        "BorrowCount": 0
    }
]'
{
    "SamplingTargetDocuments": [
        {
            "RuleName": "base-scorekeep",
            "FixedRate": 0.1,
            "ReservoirQuota": 2,
            "ReservoirQuotaTTL": 1530923107.0,
            "Interval": 10
        },
        {
```

```
            "RuleName": "polling-scorekeep",
            "FixedRate": 0.003,
            "ReservoirQuota": 0,
            "ReservoirQuotaTTL": 1530923107.0,
            "Interval": 10
        }
    ],
    "LastRuleModification": 1530920505.0,
    "UnprocessedStatistics": []
}
```

X-Ray からのレスポンスには、リザーバから借りる代わりに使用するクォータが含まれています。この例では、サービスがリザーバから 10 秒間に 10 個のトレースを借り、他の 100 個のリクエストに 10% の固定レートを適用した結果、サンプリングされたリクエストの合計数が 20 個になりました。クォータは (有効期限で示される) 5 分間、または新しいクォータが割り当てられるまで有効です。X-Ray では、ここで割り当てられなかったとしても、デフォルトより長いレポート間隔を割り当てる場合があります。

> **Note**
>
> 最初の呼び出しのときには、X-Ray からのレスポンスにクォータが含まれていない可能性があります。その場合は、クォータが割り当てられるまで、リザーバからクォータを借り続けてください。

レスポンスの他の 2 つのフィールドは、入力の問題を示している可能性があるため、前回呼び出したGetSamplingRulesの LastRuleModification を確認します。より新しい場合は、そのルールの新しいコピーを取得します。UnprocessedStatistics には、ルールが削除されたこと、入力の統計ドキュメントが古すぎること、またはアクセス許可のエラーが発生していることを示すエラーが含まれている可能性があります。

AWS X-Ray セグメントドキュメント

トレースセグメントは、アプリケーションが対応するリクエストの JSON 表現です。トレースセグメントは、元のリクエストに関する情報、アプリケーションがローカルで行う作業に関する情報、およびアプリケーションが AWS リソース、HTTP APIs、SQL データベースに対して行うダウンストリーム呼び出しに関する情報を含むサブセグメントを記録します。

セグメントドキュメントは、セグメントに関する情報を X-Ray に伝えます。セグメントドキュメントは最大で 64 kB とし、サブセグメントを含むセグメント全体、リクエストが進行中であることを示すセグメントのフラグメント、または別個に送信される単一のサブセグメントを含むことができます。セグメントドキュメントは、`PutTraceSegments` API を使用して直接 X-Ray に送信できます。

X-Ray はセグメントドキュメントをコンパイルおよび処理し、それぞれ`GetTraceSummaries`および`BatchGetTraces` API を使用してアクセスできる、クエリ可能なトレースサマリおよびトレース全体を生成します。このサービスは、X-Ray に送信するセグメントとサブセグメントに加えて、サブセグメントの情報を使用して推測セグメントを生成し、トレース全体に追加します。推定セグメントは、トレースマップのダウンストリームサービスとリソースを表します。

X-Ray は、セグメントドキュメントの JSON スキーマを提供します。スキーマは、「xray-segmentdocument-schema-v1.0.0」からダウンロードできます。スキーマに示されたフィールドとオブジェクトについては、以下のセクションで詳しく説明します。

セグメントフィールドのサブセットは、フィルタ式で使用するために X-Ray によってインデックスが作成されます。たとえば、セグメントの `user` フィールドを一意の ID に設定した場合、X-Ray コンソールで、または `GetTraceSummaries` API を使用して、特定のユーザーに関連付けられたセグメントを検索できます。詳細については、「フィルター式の使用」を参照してください。

X-Ray SDK でアプリケーションを計測すると、SDK によりセグメントドキュメントが生成されます。セグメントドキュメントを直接 X-Ray に送信する代わりに、SDK がそれらのドキュメントをローカル UDP ポート経由で X-Ray デーモンに送信します。詳細については、「セグメントドキュメントを X-Ray デーモンに送信する」を参照してください。

セクション

- セグメントフィールド
- サブセグメント
- HTTP リクエストデータ
- 注釈
- メタデータ
- AWS リソースデータ
- エラーと例外
- SQL クエリ

セグメントフィールド

セグメントは、アプリケーションが対応するリクエストに関する追跡情報を記録します。セグメントは、少なくともリクエストの名前、ID、開始時間、トレース ID、および終了時間を記録します。

Example 最小完了セグメント

```
{
  "name" : "example.com",
  "id" : "70de5b6f19ff9a0a",
  "start_time" : 1.478293361271E9,
  "trace_id" : "1-581cf771-a006649127e371903a2de979",
  "end_time" : 1.478293361449E9
}
```

次のフィールドは、セグメントで必須、または条件付きで必須です。

> **Note**
>
> 特に明記されていない限り、値は文字列である必要があります (最大 250 文字)。

必須のセグメントフィールド

- `name` – リクエストを処理したサービスの論理名 (最大 200 文字)。たとえば、アプリケーション名やドメイン名です。名前には、Unicode 文字、数字、空白、および次の記号を含めることができます: _、.、:、/、%、&、#、=、+、\、-、@
- `id` – セグメントの 64 ビット識別子。16 進数の数字であり、同じトレース内のセグメント間で一意です。
- `trace_id` – 1 つのクライアントリクエストから送信されるすべてのセグメントとサブセグメントに接続する一意の識別子です。

  X-Ray のトレース ID 形式

  X-Ray `trace_id` は、ハイフンで区切られた 3 つの数字で構成されています。例えば、`1-58406520-a006649127e371903a2de979` と指定します。これには、以下のものが含まれます :

  - バージョン番号、すなわち、1。
  - 元のリクエストの時刻。ユニックスエポックタイムで、16 進数 8 桁で表示されます。

例えば、エポックタイムで 2016 年 12 月 1 日 10:00AM PST (太平洋標準時刻) は 1480615200 秒、または 16 進数で 58406520 と表示されます。

- グローバルに一意なトレースの 96 ビットの識別子で、24 桁の 16 進数で表示されます。

> **Note**
>
> X-Ray は、OpenTelemetry、および W3C トレースコンテキスト仕様に準拠するその他のフレームワークを使用して作成されたトレース ID をサポートするようになりました。W3C トレース ID は X-Ray に送信するとき、X-Ray トレース ID 形式でフォーマットする必要があります。例えば、W3C トレース ID 4efaaf4d1e8720b39541901950019ee5 は X-Ray に送信するとき、1-4efaaf4d-1e8720b39541901950019ee5 の形式にする必要があります。X-Ray トレース ID には元のリクエストの Unix エポックタイムのタイムスタンプが含まれますが、これは W3C トレース ID を X-Ray 形式で送信する場合に必要ありません。

> **トレース ID セキュリティ**
>
> トレース ID はレスポンスヘッダーに表示されます。攻撃者が将来のトレース ID を計算できないように安全なランダムのアルゴリズムを使用してトレース ID を生成し、その ID を使用してアプリケーションにリクエストを送信します。

- `start_time` – セグメントが作成された時間の数値 (エポック時間の浮動小数点で表した秒数)。例えば、1480615200.010、1.480615200010E9 です。必要なだけ桁数を使用します。利用できる場合は、マイクロ秒の精度をお勧めします。
- `end_time` – セグメントが切断された時間を表す数値。例えば、1480615200.090、1.480615200090E9 です。`end_time` または `in_progress` のいずれかを指定します。
- `in_progress` – ではなく を設定して、開始されたが完了していないセグメントを記録するtrueブール値end_time。アプリケーションが処理に時間がかかるリクエストを受信したときに、進行中のセグメントを送信して、リクエストの受信を追跡します。レスポンスが送信されると、完了したセグメントが送信され進行中のセグメントを上書きします。リクエストごとに、1 つの完全なセグメントと、1 つまたは 0 個の進行中のセグメントのみを送信します。

> **サービス名**
>
> セグメントの name は、セグメントを生成するサービスのドメイン名または論理名と一致する必要があります。ただし、これは強制ではありません。権限を持つアプリケーションPutTraceSegmentsは、任意の名前でセグメントを送信できます。

次のフィールドは、セグメントではオプションです。

オプションのセグメントフィールド

- service – アプリケーションに関する情報を含むオブジェクト。
  - version – リクエストに対応したアプリケーションのバージョンを識別する文字列。
- user – リクエストを送信したユーザーを識別する文字列。
- origin – アプリケーションを実行している AWS リソースのタイプ。

  サポートされる値

  - AWS::EC2::Instance – Amazon EC2 インスタンス。
  - AWS::ECS::Container – Amazon ECS コンテナ。
  - AWS::ElasticBeanstalk::Environment – Elastic Beanstalk 環境

  複数の値をアプリケーションに適用する場合は、最も具体的な値を使用します。たとえば、複数コンテナの Docker Elastic Beanstalk の環境では、Amazon ECS コンテナでアプリケーションが実行され、そのコンテナは Amazon EC2 インスタンスで実行されます。この場合、環境は他の 2 つのリソースの親として、オリジンを AWS::ElasticBeanstalk::Environment に設定します。
- parent_id – 計測したアプリケーションからリクエストが発信された場合に指定するサブセグメント ID。X-Ray SDK は親サブセグメント ID をダウンストリーム HTTP 呼び出しのトレースヘッダーに追加します。ネストされたサブセグメントの場合、サブセグメントは親としてセグメントまたはサブセグメントを持つことができます。
- http – 元の HTTP リクエストに関する情報を含むhttpオブジェクト。
- aws – アプリケーションがリクエストを処理した AWS リソースに関する情報を含む aws オブジェクト。
- error、throttle、fault、cause – エラーが発生したことを示し、エラーの原因となった例外に関する情報を含む error フィールド。
- annotations – X-Ray で検索用にインデックスを作成するキーと値のペアを含むannotationsオブジェクト。

- metadata – セグメントに保存する追加のデータを含むmetadataオブジェクト。
- subsegments – オブジェクトの配列subsegment。

サブセグメント

サブセグメントを作成して、 AWS SDK で行った AWS のサービス およびリソースへの呼び出し、内部または外部の HTTP ウェブ APIs への呼び出し、または SQL データベースクエリを記録できます。また、サブセグメントを作成してアプリケーションでコードブロックをデバッグしたり、注釈を付けたりできます。サブセグメントには他のサブセグメントを含めることができるため、内部関数呼び出しに関するメタデータを記録するカスタムサブセグメントには、他のカスタムサブセグメントおよびダウンストリーム呼び出し用のサブセグメントを含めることができます。

サブセグメントは、ダウンストリーム呼び出しを、それを呼び出したサービスの視点から記録します。X-Ray はサブセグメントを使用して、セグメントを送信しないダウンストリームサービスを識別し、そのエントリをサービスグラフに作成します。

サブセグメントはフルセグメントドキュメントに埋め込むことも、個別に送信することもできます。サブセグメントを個別に送信して、長期実行されているリクエストのダウンストリーム呼び出しを非同期でトレースしたり、セグメントドキュメントの最大サイズを超えないようにしたりできます。

Example 埋め込みサブセグメントを含むセグメント

独立したサブセグメントには、親セグメントを識別する type の subsegment、および parent_id があります。

```
{
  "trace_id"   : "1-5759e988-bd862e3fe1be46a994272793",
  "id"         : "defdfd9912dc5a56",
  "start_time" : 1461096053.37518,
  "end_time"   : 1461096053.4042,
  "name"       : "www.example.com",
  "http"       : {
    "request"  : {
      "url"        : "https://www.example.com/health",
      "method"     : "GET",
      "user_agent" : "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6)
 AppleWebKit/601.7.7",
      "client_ip"  : "11.0.3.111"
    },
    "response" : {
      "status"         : 200,
```

```
      "content_length" : 86
    }
  },
  "subsegments" : [
    {
      "id"         : "53995c3f42cd8ad8",
      "name"       : "api.example.com",
      "start_time" : 1461096053.37769,
      "end_time"   : 1461096053.40379,
      "namespace"  : "remote",
      "http"       : {
        "request"  : {
          "url"    : "https://api.example.com/health",
          "method" : "POST",
          "traced" : true
        },
        "response" : {
          "status"         : 200,
          "content_length" : 861
        }
      }
    }
  ]
}
```

長期間実行されるリクエストについては、進行中のセグメントを送信してリクエストが受信されたことを X-Ray に通知し、セグメントを個別に送信して追跡してから、元のリクエストを完了することができます。

Example 進行中セグメント

```
{
  "name" : "example.com",
  "id" : "70de5b6f19ff9a0b",
  "start_time" : 1.478293361271E9,
  "trace_id" : "1-581cf771-a006649127e371903a2de979",
  "in_progress": true
}
```

Example 独立したサブセグメント

独立したサブセグメントには、親セグメントを識別する `type` の `subsegment`、`trace_id`、および `parent_id` があります。

```
{
  "name" : "api.example.com",
  "id" : "53995c3f42cd8ad8",
  "start_time" : 1.478293361271E9,
  "end_time" : 1.478293361449E9,
  "type" : "subsegment",
  "trace_id" : "1-581cf771-a006649127e371903a2de979"
  "parent_id" : "defdfd9912dc5a56",
  "namespace"  : "remote",
  "http"       : {
      "request"  : {
          "url"    : "https://api.example.com/health",
          "method" : "POST",
          "traced" : true
      },
      "response" : {
          "status"         : 200,
          "content_length" : 861
      }
  }
}
```

リクエストが完了したら、end_time とともに再送信してセグメントを閉じます。完了セグメントは進行中のセグメントを上書きします。

非同期ワークフローをトリガーした、完了したリクエストに対してサブセグメントを個別に送信することもできます。たとえば、ウェブ API は、ユーザーがリクエストした作業を開始する直前に OK 200 応答を返す場合があります。応答が送信されたらすぐに、完全なセグメントを X-Ray に送信し、それに続いて後で完了する作業のサブセグメントを送信できます。セグメントと同様に、サブセグメントフラグメントを送信して、サブセグメントが開始されたことを記録した後で、ダウンストリーム呼び出しが完了したら完全なサブセグメントでそれを上書きできます。

次のフィールドは、サブセグメントで必須、または条件付きで必須です。

> **Note**
>
> 特に明記されていない限り、値は文字列です (最大 250 文字)。

必須のサブセグメントフィールド

- id – サブセグメントの 64 ビット識別子。16 進数の数字であり、同じトレース内のセグメント間で一意です。
- name – サブセグメントの論理名。ダウンストリーム呼び出しの場合は、リソースまたはサービスを呼び出した後のサブセグメントの名前。カスタムサブセグメントの場合は、計測するコードの後にサブセグメントの名前を付けます (関数名など)。
- start_time – サブセグメントが作成された時間を表す数値で、エポック時間を浮動小数点で表した秒 (ミリ秒)。例えば、1480615200.010、1.480615200010E9 です。
- end_time – サブセグメントが切断された時間を表す数値。例えば、1480615200.090、1.480615200090E9 です。end_time または in_progress を指定します。
- in_progress – ではなく を設定して、開始されたが完了していないサブセグメントを記録するtrueブール値end_time。ダウンストリームリクエストごとに、1 つの完全なサブセグメントと、1 つまたは 0 個の進行中のサブセグメントのみを送信します。
- trace_id – サブセグメントの親セグメントのトレース ID。サブセグメントを個別に送信する場合にのみ必要です。

  X-Ray のトレース ID 形式

  X-Ray trace_id は、ハイフンで区切られた 3 つの数字で構成されています。例えば、1-58406520-a006649127e371903a2de979 と指定します。これには、以下のものが含まれます：

  - バージョン番号、すなわち、1。
  - 元のリクエストの時刻。ユニックスエポックタイムで、16 進数 8 桁で表示されます。

    例えば、エポックタイムで 2016 年 12 月 1 日 10:00AM PST (太平洋標準時刻) は 1480615200 秒、または 16 進数で 58406520 と表示されます。
  - グローバルに一意なトレースの 96 ビットの識別子で、24 桁の 16 進数で表示されます。

  > **Note**
  >
  > X-Ray は、OpenTelemetry、および W3C トレースコンテキスト仕様に準拠するその他のフレームワークを使用して作成されたトレース ID をサポートするようになりました。W3C トレース ID は X-Ray に送信するとき、X-Ray トレース ID 形式でフォーマットする必要があります。例えば、W3C ト

レース ID 4efaaf4d1e8720b39541901950019ee5 は X-Ray に送信するとき、1-4efaaf4d-1e8720b39541901950019ee5 の形式にする必要があります。X-Ray トレース ID には元のリクエストの Unix エポックタイムのタイムスタンプが含まれますが、これは W3C トレース ID を X-Ray 形式で送信する場合に必要ありません。

- parent_id – サブセグメントの親セグメントのセグメント ID。サブセグメントを個別に送信する場合にのみ必要です。ネストされたサブセグメントの場合、サブセグメントは親としてセグメントまたはサブセグメントを持つことができます。
- type – subsegment。サブセグメントを個別に送信する場合にのみ必要です。

次のフィールドは、サブセグメントではオプションです。

オプションのサブセグメントフィールド

- namespace – AWS SDK 呼び出しの場合は aws、他のダウンストリーム呼び出しの場合は remote。
- http – 送信 HTTP 呼び出しに関する情報を含むhttpオブジェクト。
- aws – アプリケーションが呼び出したダウンストリーム AWS リソースに関する情報を含む aws オブジェクト。
- error、throttle、fault、cause – エラーが発生したことを示し、エラーの原因となった例外に関する情報を含む error フィールド。
- annotations – X-Ray で検索用にインデックスを作成するキーと値のペアを含むannotationsオブジェクト。
- metadata – セグメントに保存する追加のデータを含むmetadataオブジェクト。
- subsegments – subsegmentオブジェクトの配列。
- precursor_ids – このサブセグメントの前に完了した同じ親を持つサブセグメントを識別するサブセグメント ID の配列。

HTTP リクエストデータ

HTTP ブロックを使用して、(セグメントで) アプリケーションが対応した HTTP リクエスト、または (サブセグメントで) アプリケーションがダウンストリーム HTTP API に対して行ったリクエストの詳細を記録します。このオブジェクトのほとんどのフィールドは、HTTP リクエストと応答で見つかった情報にマッピングされます。

http

すべてのフィールドはオプションです。

- request – リクエストに関する情報。
  - method – リクエストメソッド。例えば、GET と指定します。
  - url – リクエストのプロトコル、ホスト名、およびパスからコンパイルされた、リクエストの完全な URL。
  - user_agent – リクエスタのクライアントからのユーザーエージェント文字列。
  - client_ip – リクエスタの IP アドレス。IP パケットの Source Address から、または転送リクエストの場合は X-Forwarded-For ヘッダーから取得できます。
  - x_forwarded_for – (セグメントのみ) が ヘッダーから読み取られ、偽造されている可能性があるため信頼できないことを示すclient_ipブール値X-Forwarded-For。
  - traced – (サブセグメントのみ) ダウンストリーム呼び出しが別の追跡されたサービスであることを示すブール値。このフィールドが true に設定されている場合、このブロックを含むサブセグメントの parent_id に一致する id を含むセグメントをダウンストリームサービスがアップロードするまで、X-Ray はトレースが壊れていると見なします。
- response – レスポンスに関する情報。
  - status – レスポンスの HTTP ステータスを示す整数。
  - content_length – レスポンス本文の長さをバイト単位で示す整数。

ダウンストリームウェブ API に対する呼び出しを計測するときは、HTTP リクエストおよびレスポンスに関する情報を含むセグメントを記録します。X-Ray はサブセグメントを使用してリモート API の推測セグメントを生成します。

Example Amazon EC2 で実行しているアプリケーションにより提供される HTTP 呼び出し用のセグメント

```
{
  "id": "6b55dcc497934f1a",
  "start_time": 1484789387.126,
  "end_time": 1484789387.535,
  "trace_id": "1-5880168b-fd5158284b67678a3bb5a78c",
  "name": "www.example.com",
  "origin": "AWS::EC2::Instance",
  "aws": {
    "ec2": {
```

```
      "availability_zone": "us-west-2c",
      "instance_id": "i-0b5a4678fc325bg98"
    },
    "xray": {
        "sdk_version": "2.11.0 for Java"
    },
  },
  "http": {
    "request": {
      "method": "POST",
      "client_ip": "78.255.233.48",
      "url": "http://www.example.com/api/user",
      "user_agent": "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:45.0) Gecko/20100101
 Firefox/45.0",
      "x_forwarded_for": true
    },
    "response": {
      "status": 200
    }
  }
```

Example ダウンストリーム HTTP 呼び出しのサブセグメント

```
{
  "id": "004f72be19cddc2a",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "name": "names.example.com",
  "namespace": "remote",
  "http": {
    "request": {
      "method": "GET",
      "url": "https://names.example.com/"
    },
    "response": {
      "content_length": -1,
      "status": 200
    }
  }
}
```

Example ダウンストリーム HTTP 呼び出しの推定セグメント

```
{
  "id": "168416dc2ea97781",
  "name": "names.example.com",
  "trace_id": "1-62be1272-1b71c4274f39f122afa64eab",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "parent_id": "004f72be19cddc2a",
  "http": {
    "request": {
      "method": "GET",
      "url": "https://names.example.com/"
    },
    "response": {
      "content_length": -1,
      "status": 200
    }
  },
  "inferred": true
}
```

注釈

セグメントとサブセグメントは、X-Ray がフィルタ式で使用するためにインデックスを作成する 1 つ以上のフィールドが含まれた annotations オブジェクトを含むことができます。フィールドは、文字列、数値、またはブール値を持つことができます (オブジェクトや配列を含むことはできません)。X-Ray は、トレースごとに 50 の注釈までインデックスを付けます。

Example 注釈を使用した HTTP 呼び出しのセグメント

```
{
  "id": "6b55dcc497932f1a",
  "start_time": 1484789187.126,
  "end_time": 1484789187.535,
  "trace_id": "1-5880168b-fd515828bs07678a3bb5a78c",
  "name": "www.example.com",
  "origin": "AWS::EC2::Instance",
  "aws": {
    "ec2": {
      "availability_zone": "us-west-2c",
      "instance_id": "i-0b5a4678fc325bg98"
```

```
    },
    "xray": {
        "sdk_version": "2.11.0 for Java"
    },
  },
  "annotations": {
    "customer_category" : 124,
    "zip_code" : 98101,
    "country" : "United States",
    "internal" : false
  },
  "http": {
    "request": {
      "method": "POST",
      "client_ip": "78.255.233.48",
      "url": "http://www.example.com/api/user",
      "user_agent": "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:45.0) Gecko/20100101
Firefox/45.0",
      "x_forwarded_for": true
    },
    "response": {
      "status": 200
    }
  }
```

キーはフィルタで動作するために英数字である必要があります。アンダースコアは使用できます。その他の記号や空白は使用できません。

メタデータ

セグメントとサブセグメントは、オブジェクトと配列を含めて、任意の型の値を持つ 1 つ以上のフィールドが含まれた metadata オブジェクトを含むことができます。X-Ray はメタデータのインデックスを作成せず、セグメントドキュメントが最大サイズ (64 kB) を超えない限り、任意のサイズにすることができます。BatchGetTraces API によって返された完全なセグメントドキュメントで、メタデータを表示できます。で始まるフィールドキー (debug次の例では) AWS.は、 が提供する SDKs AWSとクライアント用に予約されています。

Example カスタムサブセグメントとメタデータ

```
{
  "id": "0e58d2918e9038e8",
  "start_time": 1484789387.502,
```

```
  "end_time": 1484789387.534,
  "name": "## UserModel.saveUser",
  "metadata": {
    "debug": {
      "test": "Metadata string from UserModel.saveUser"
    }
  },
  "subsegments": [
    {
      "id": "0f910026178b71eb",
      "start_time": 1484789387.502,
      "end_time": 1484789387.534,
      "name": "DynamoDB",
      "namespace": "aws",
      "http": {
        "response": {
          "content_length": 58,
          "status": 200
        }
      },
      "aws": {
        "table_name": "scorekeep-user",
        "operation": "UpdateItem",
        "request_id": "3AIENM5J4ELQ3SPODHKBIRVIC3VV4KQNSO5AEMVJF66Q9ASUAAJG",
        "resource_names": [
          "scorekeep-user"
        ]
      }
    }
  ]
}
```

AWS リソースデータ

セグメントの場合、aws オブジェクトはアプリケーションが実行されているリソースに関する情報を含みます。複数のフィールドを単一のリソースに適用できます。たとえば、Elastic Beanstalk の複数コンテナの Docker 環境で実行されているアプリケーションは、Amazon EC2 インスタンス、インスタンス上で実行されている Amazon ECS コンテナ、および Elastic Beanstalk 環境自体に関する情報を持つことができます。

aws (セグメント)

すべてのフィールドはオプションです。

- account_id – アプリケーションがセグメントを別の に送信する場合は AWS アカウント、アプリケーションを実行しているアカウントの ID を記録します。
- cloudwatch_logs – 単一の CloudWatch ロググループを記述するオブジェクトの配列。
  - log_group – CloudWatch のロググループ名。
  - arn – CloudWatch のロググループ ARN。
- ec2 – Amazon EC2 インスタンスに関する情報。
  - instance_id – EC2 インスタンスのインスタンス ID。
  - instance_size – EC2 インスタンスのタイプ。
  - ami_id – Amazon マシンイメージ ID。
  - availability_zone – インスタンスが実行されているアベイラビリティーゾーン。
- ecs – Amazon ECS コンテナに関する詳細。
  - container – コンテナのホスト名。
  - container_id – コンテナの完全なコンテナ ID。
  - container_arn – コンテナインスタンスの ARN。
- eks – Amazon EKS クラスターに関する情報。
  - pod – EKS ポッドのホスト名。
  - cluster_name – EKS クラスター名。
  - container_id – コンテナの完全なコンテナ ID。
- elastic_beanstalk – Elastic Beanstalk 環境に関する情報。この情報は、最新の Elastic Beanstalk プラットフォームの /var/elasticbeanstalk/xray/environment.conf という名前のファイルにあります。
  - environment_name – 環境の名前。
  - version_label – リクエストに対応したインスタンスに現在デプロイされているアプリケーションバージョンの名前。
  - deployment_id – リクエストに対応したインスタンスに対して最後に成功したデプロイの ID を示す数値。
- xray – 使用した計測のタイプとバージョンに関するメタデータ。
  - auto_instrumentation – 自動計測が使用されたかどうかを示すブール値 (例えば、Java Agent)。
  - sdk_version – 使用中の SDK またはエージェントのバージョン。
  - sdk – SDK のタイプ。

Example AWS プラグインを使用した ブロック

```
"aws":{
   "elastic_beanstalk":{
      "version_label":"app-5a56-170119_190650-stage-170119_190650",
      "deployment_id":32,
      "environment_name":"scorekeep"
   },
   "ec2":{
      "availability_zone":"us-west-2c",
      "instance_id":"i-075ad396f12bc325a",
      "ami_id":
   },
   "cloudwatch_logs":[
      {
         "log_group":"my-cw-log-group",
         "arn":"arn:aws:logs:us-west-2:012345678912:log-group:my-cw-log-group"
      }
   ],
   "xray":{
      "auto_instrumentation":false,
      "sdk":"X-Ray for Java",
      "sdk_version":"2.8.0"
   }
}
```

サブセグメントの場合は、アプリケーションがアクセスする AWS のサービス およびリソースに関する情報を記録します。X-Ray はこの情報を使用して、サービスマップのダウンストリームサービスを表す推定セグメントを作成します。

aws (サブセグメント)

すべてのフィールドはオプションです。

- operation – AWS のサービス または リソースに対して呼び出された API アクションの名前。
- account_id – アプリケーションが別のアカウントのリソースにアクセスする場合、またはセグメントを別のアカウントに送信する場合は、アプリケーションがアクセスした AWS リソースを所有するアカウントの ID を記録します。
- region – リソースがアプリケーションとは異なるリージョンにある場合は、そのリージョンを記録します。例えば、us-west-2 と指定します。
- request_id – リクエストの一意の識別子。

- queue_url – Amazon SQS キューのオペレーションの場合は、キューの URL。
- table_name – DynamoDB テーブルのオペレーションの場合、テーブルの名前。

Example 項目を保存するための DynamoDB に対する呼び出しのサブセグメント

```
{
  "id": "24756640c0d0978a",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "DynamoDB",
  "namespace": "aws",
  "http": {
    "response": {
      "content_length": 60,
      "status": 200
    }
  },
  "aws": {
    "table_name": "scorekeep-user",
    "operation": "UpdateItem",
    "request_id": "UBQNSO5AEM8T4FDA4RQDEB94OVTDRVV4K4HIRGVJF66Q9ASUAAJG",
  }
}
```

エラーと例外

エラーが発生した場合は、エラーと生成された例外に関する詳細を記録できます。アプリケーションがユーザーにエラーを返す場合はセグメントにエラーを記録し、ダウンストリーム呼び出しがエラーを返す場合はサブセグメントにエラーを記録します。

エラーのタイプ

次の 1 つ以上のフィールドを true に設定して、エラーが発生したことを示します。複合エラーの場合は、複数のタイプを適用できます。たとえば、ダウンストリーム呼び出しからの 429 Too Many Requests エラーにより、アプリケーションは 500 Internal Server Error を返すことがあり、その場合は 3 つすべてのタイプが適用されます。

- error – クライアントエラーが発生したことを示すブール値 (レスポンスステータスコードは 4XX Client Error でした)。

- `throttle` – リクエストが調整されたことを示すブール値 (レスポンスステータスコードは 429 Too Many Requests でした)。
- `fault` – サーバーエラーが発生したことを示すブール値 (レスポンスステータスコードは 5XX Server Error でした)。

セグメントまたはサブセグメントに cause オブジェクトを含めてエラーの原因を示します。

`cause`

原因は、16 文字の例外 ID、または次のフィールドを含むオブジェクトとすることができます。

- `working_directory` – 例外が発生したときの作業ディレクトリのフルパス。
- `paths` – 例外が発生したときに使用されているライブラリまたはモジュールへのパスの配列。
- `exceptions` – 例外オブジェクトの配列。

1 つまたは複数の例外オブジェクトのエラーに関する詳細情報を含めます。

`exception`

すべてのフィールドはオプションです。

- `id` – 例外の 64 ビット識別子。16 進数の数字であり、同じトレース内のセグメント間で一意です。
- `message` – 例外メッセージ。
- `type` – 例外のタイプ。
- `remote` – ダウンストリームサービスによって返されたエラーが原因で例外が発生したことを示すブール値。
- `truncated` – から省略されたスタックフレームの数を示す整数`stack`。
- `skipped` – この例外とその子の間でスキップされた例外 (発生した例外) の数を示す整数。
- `cause` – 例外の親 (この例外を発生させた例外) の例外 ID。
- `stack` – stackFrame オブジェクトの配列。

使用可能な場合、コールスタックに関する情報を stackFrame オブジェクトに記録します。

`stackFrame`

すべてのフィールドはオプションです。

- path – ファイルの相対パス。
- line – ファイルの行。
- label – 関数またはメソッド名。

SQL クエリ

アプリケーションが SQL データベースに対して実行するクエリのサブセグメントを作成できます。

sql

すべてのフィールドはオプションです。

- connection_string – SQL Server または URL 接続文字列を使用しないその他のデータベース接続の場合は、パスワードを除く接続文字列を記録します。
- url – URL 接続文字列を使用するデータベース接続の場合は、パスワードを除く URL を記録します。
- sanitized_query – データベースクエリと、プレースホルダーによって削除または置換されたユーザー指定の値。
- database_type – データベースエンジンの名前。
- database_version – データベースエンジンのバージョン番号。
- driver_version – アプリケーションが使用するデータベースエンジンドライバーの名前とバージョン番号。
- user – データベースユーザー名。
- preparation – クエリで call を使用した場合は PreparedCall、クエリで statement を使用した場合は PreparedStatement。

Example サブセグメントと SQL クエリ

```
{
  "id": "3fd8634e78ca9560",
  "start_time": 1484872218.696,
  "end_time": 1484872218.697,
  "name": "ebdb@aawijb5u25wdoy.cpamxznpdoq8.us-west-2.rds.amazonaws.com",
  "namespace": "remote",
  "sql" : {
    "url": "jdbc:postgresql://aawijb5u25wdoy.cpamxznpdoq8.us-
west-2.rds.amazonaws.com:5432/ebdb",
```

```
    "preparation": "statement",
    "database_type": "PostgreSQL",
    "database_version": "9.5.4",
    "driver_version": "PostgreSQL 9.4.1211.jre7",
    "user" : "dbuser",
    "sanitized_query" : "SELECT  *  FROM  customers  WHERE  customer_id=?;"
  }
}
```

AWS X-Ray の概念

AWS X-Ray は、 サービスからデータをセグメントとして受け取ります。X-Ray は、トレースへの共通リクエストを含むセグメントをグループ化します。X-Ray は、トレースを処理して、アプリケーションのビジュアル表現を提供するサービスグラフを生成します。

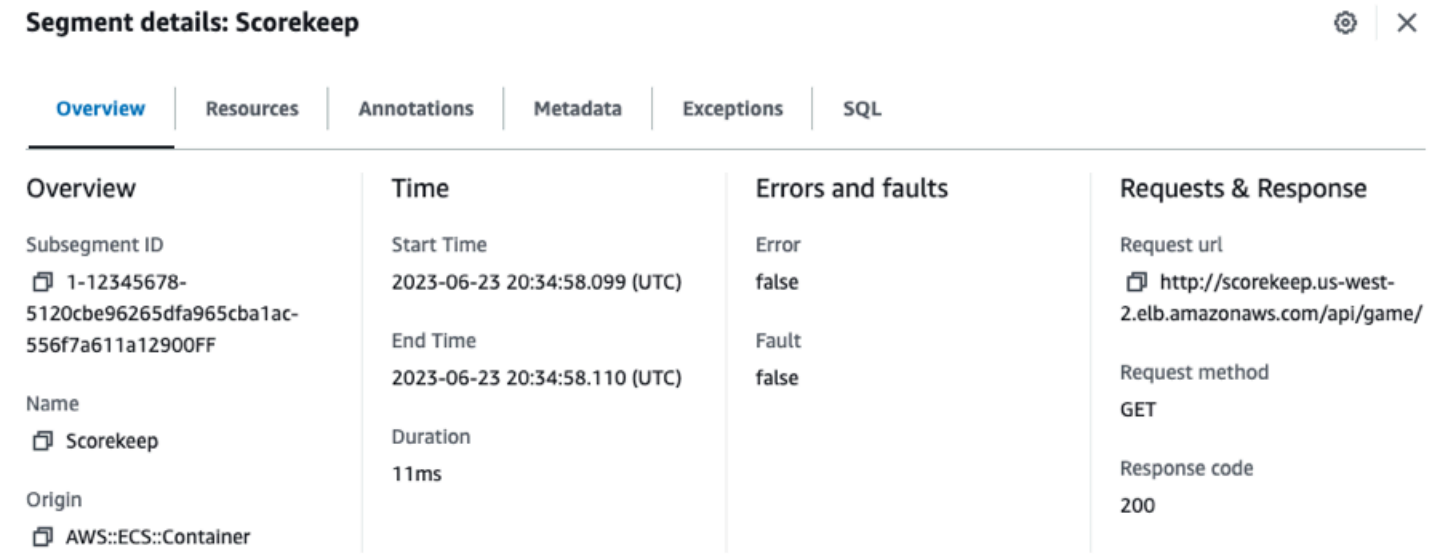
概念

- セグメント
- サブセグメント
- サービスグラフ
- トレース
- サンプリング
- トレースヘッダー
- フィルタ式
- グループ
- 注釈とメタデータ
- エラー、障害、および例外

セグメント

アプリケーションロジックを実行しているコンピューティングリソースは、セグメントとしての動作に関するデータを送信します。セグメントには、リソース名、リクエストの詳細、行った作業の詳細が含まれています。たとえば、HTTP リクエストがアプリケーションに到達すると、次のデータが記録されます。

- ホスト – ホスト名、エイリアス、または IP アドレス
- リクエスト – メソッド、クライアントアドレス、パス、ユーザーエージェント
- レスポンス – ステータス、コンテンツ
- 行った作業 – 開始および終了時刻、サブセグメント
- 発生する問題 – エラー、障害、例外 (例外スタックの自動取得を含む)。



X-Ray SDK は、リクエストヘッダーとレスポンスヘッダー、アプリケーション内のコード、および実行する AWS リソースに関するメタデータから情報を収集します。収集するデータを選択するには、アプリケーション設定またはコードを変更して、受信リクエスト、ダウンストリームリクエスト、および AWS SDK クライアントを計測します。

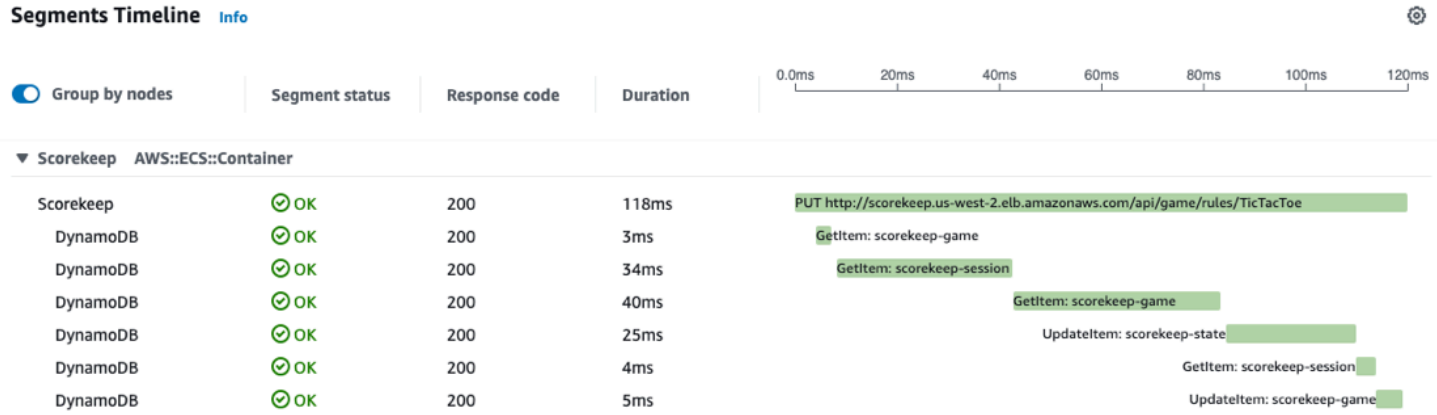
> **転送されたリクエスト**
>
> ロードバランサーまたは他の仲介者がアプリケーションにリクエストを転送する場合、X-Ray は、クライアントの IP をIP パケットの送信元 IP からではなく、リクエストのX-Forwarded-Forヘッダーから取得します。転送されたリクエストに対して、記録されたクライアント IP は偽造される可能性があるので、信頼できるものではありません。

X-Ray SDK を使用して、注釈やメタデータなどの追加情報を記録します。セグメントとサブセグメントで記録される構造と情報の詳細については、「AWS X-Ray セグメントドキュメント」を参照してください。セグメントのドキュメントのサイズは最大 64 kB まで可能です。

サブセグメント

セグメントでは、完了した作業に関するデータをサブセグメントに分けることができます。サブセグメントには、詳細なタイミング情報や、元のリクエストを満たすためにアプリケーションが実行したダウンストリーム呼び出しに関する詳細を含みます。サブセグメントには、、外部 HTTP API AWS のサービス、または SQL データベースへの呼び出しに関する追加の詳細を含めることができます。

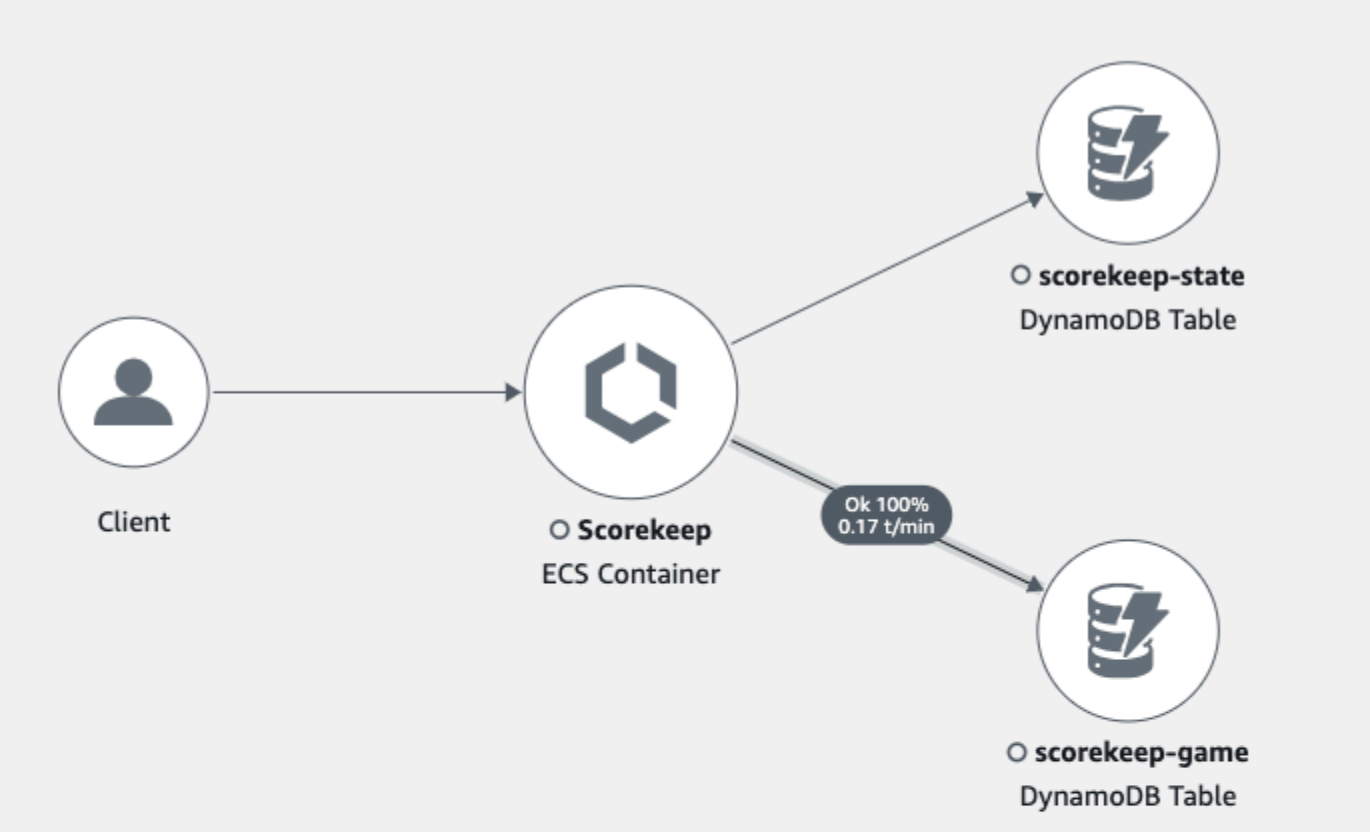
アプリケーションのコードの特定の関数または行を計測する任意のサブセグメントを定義することもできます。

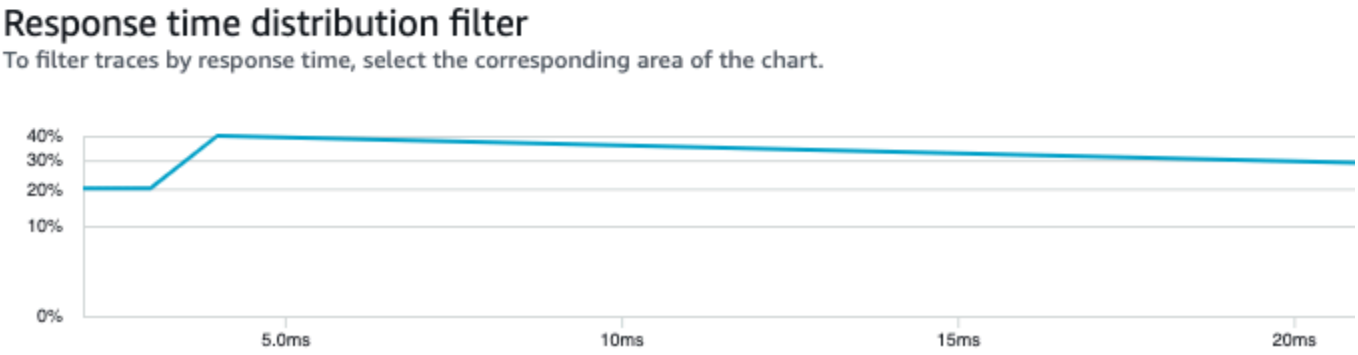


Amazon DynamoDB のような独自のセグメントを送信しないサービスでは、X-Ray はサブセグメントを使用してトレースマップ上に推測セグメントとダウンストリームノードを生成します。これにより、トレースをサポートしていない場合でも、外部の場合でも、すべてのダウンストリーム依存関係を表示することができます。

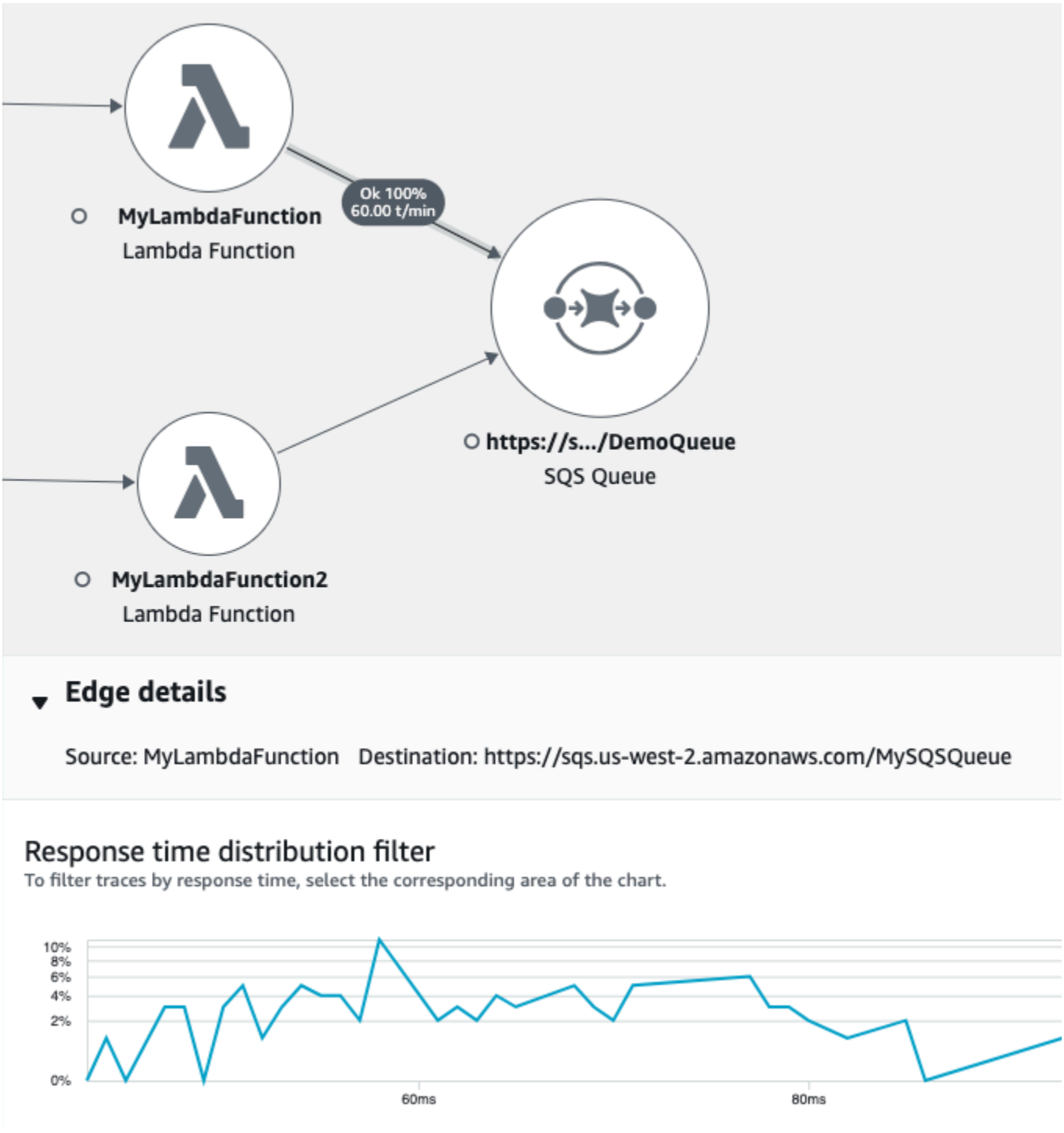
サブセグメントは、クライアントとしてのダウンストリーム呼び出しのアプリケーションビューを表します。ダウンストリームサービスも計測されている場合、送信するセグメントは、アップストリームクライアントのサブセグメントから生成された推測セグメントを置き換えます。利用可能であれば、サービスグラフのノードはサービスのセグメントからの情報を常に使用しますが、2 つのノード間のエッジはアップストリームサービスのサブセグメントを使用します。

たとえば、計測された AWS SDK クライアントを使用して DynamoDB を呼び出すと、X-Ray SDK はその呼び出しのサブセグメントを記録します。DynamoDB はセグメントを送信しないので、トレースの推測セグメント、サービスグラフの DynamoDB ノード、サービスと DynamoDB の間のエッジにはサブセグメントの情報が含まれます。





実装されたアプリケーションを使用して実装された別のサービスを呼び出すと、ダウンストリームサービスは、サブセグメントに記録されたアップストリームサービスと同じ呼び出しのビューを記録するために、独自のセグメントを送信します。サービスグラフでは、両方のサービスのノードに、それらのサービスのセグメントからのタイミング情報とエラー情報が含まれています。その間には、アップストリームサービスのサブセグメントの情報が含まれています。



ダウンストリームサービスは要求の処理を開始および終了した時点を正確に記録し、アップストリームサービスは要求が 2 つのサービス間を移動した時間を含めて往復遅延を記録するため、両方の視点が役立ちます。

サービスグラフ

X-Ray はアプリケーションが送信したデータを使用してサービスグラフを生成します。X-Ray にデータを送信する各 AWS リソースは、グラフにサービスとして表示されます。エッジは、リクエスト処理に一緒に使用するサービスに接続されています。エッジは、クライアントをアプリケーションに接続し、アプリケーションを、使用しているダウンストリームサービスおよびリソースに接続します。

> **サービス名**
>
> セグメントの `name` は、セグメントを生成するサービスのドメイン名または論理名と一致する必要があります。ただし、これは強制ではありません。`PutTraceSegments`にアクセス許可を持つアプリケーションは、任意の名前でセグメントを送信できます。

サービスグラフは、アプリケーションを構成するサービスおよびリソースに関する情報を含む JSON ドキュメントです。X-Ray コンソールでは、サービスグラフを使用して、視覚化またはサービスマップを生成します。

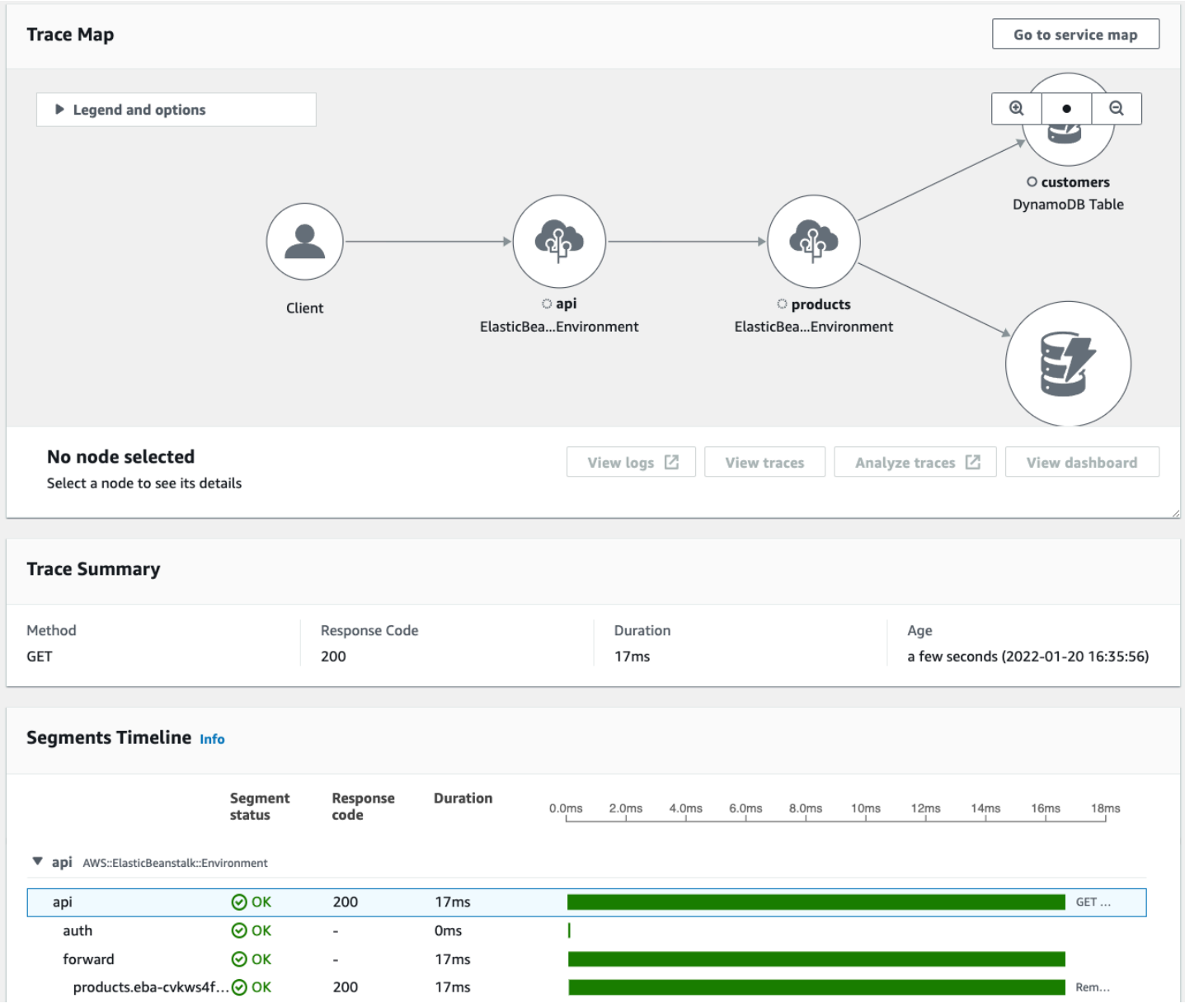
分散アプリケーションの場合、X-Ray は、同一のトレース ID を含むリクエストを処理するすべてのサービスのノードを、単一のサービスグラフに組み合わせます。リクエストがヒットする最初のサービスによって、フロントエンドと呼び出すサービスの間で伝達される「トレースヘッダー」が追加されます。

たとえば、Scorekeep は、マイクロサービス (AWS Lambda 関数) を呼び出し、Node.js ライブラリを使用してランダム名を生成するウェブ API を実行します。X-Ray SDK for Java はトレース ID を生成し、Lambda への呼び出しにそれを含めます。Lambda はトレースデータを送信し、トレース ID を関数に渡します。また、X-Ray SDK for Node.js は、トレース ID を使用してデータを送信します。その結果、API、Lambda サービス、Lambda 関数のノードはすべて別々ですが、接続されたノードとしてトレースマップに表示されます。

サービスグラフのデータは 30 日間保持されます。

トレース

トレース ID はアプリケーションを経由するリクエストのパスを追跡します。トレースでは、1 つのリクエストで生成されたセグメントをすべて収集します。このリクエストは、通常ロードバランサーを経由してやり取りされる HTTP GET または POST リクエストですが、アプリケーションコードにヒットし、他の AWS サービスまたは外部の ウェブ API のダウンストリーム呼び出しを生成します。HTTP リクエストでやり取りされる最初にサポートされたサービスでは、トレース ID のヘッダーをリクエストに追加し、ダウンストリームに伝達して、レイテンシー、処理、その他のリクエストデータを追跡します。



X-Ray トレースの請求方法については、「AWS X-Ray の料金」をご覧ください。トレースデータは 30 日間保持されます。

サンプリング

効率的にトレースを行ってアプリケーションが処理するリクエストの代表的なサンプルを提供するため、X-Ray SDK によってサンプリングアルゴリズムが適用され、トレースするリクエストが決定されます。デフォルトでは、X-Ray SDK は 1 秒ごとに最初のリクエストを記録し、追加のリクエストの 5% を記録します。

開始時にサービス料がかからないように、デフォルトのサンプリングレートは控えめになっています。デフォルトのサンプリングルールを変更し、サービスまたはリクエストのプロパティに基づいてサンプリングを適用する追加のルールを設定するように X-Ray を設定できます。

例えば、サンプリングを無効にして、状態を変更したり、ユーザーやトランザクションを処理したりする呼び出しのすべての要求をトレースすることができます。バックグラウンドポーリング、ヘルスチェック、接続保守などの大量の読み取り専用呼び出しでは、低いレートでサンプリングを行っても、問題が発生した場合にも十分なデータを得ることができます。

詳細については、「サンプリングルールの設定」を参照してください。

トレースヘッダー

構成可能な最小値まではすべてのリクエストがトレースされます。その後はリクエストの一定の割合がトレースされ、不要なコストを避けます。サンプリングデシジョンおよびトレース ID は、HTTP リクエストの という名前のトレースヘッダーX-Amzn-Trace-Idに追加されます。トレースヘッダーは、リクエストがヒットした最初の X-Ray 組み込みサービスによって追加されます。また、X-Ray SDK によって読み取られ、レスポンスに含められます。

Example ルートトレース ID 突きのトレースヘッダーおよびサンプリングデシジョン

```
X-Amzn-Trace-Id: Root=1-5759e988-
bd862e3fe1be46a994272793;Parent=53995c3f42cd8ad8;Sampled=1
```

> **トレースヘッダーのセキュリティ**
>
> トレースヘッダーは、X-Ray SDK、 AWS のサービス、またはクライアントリクエストから作成できます。アプリケーションで受信リクエストから X-Amzn-Trace-Id を削除し、

ユーザーが自分のリクエストにトレース ID またはサンプリングデシジョンを追加することで発生する問題を回避できます。

計測対象アプリケーションからのリクエストの場合は、親セグメント ID をトレースヘッダーに含めることもできます。たとえば、アプリケーションで計測対象 HTTP クライアントを使用してダウンストリーム HTTP ウェブ API を呼び出す場合、X-Ray SDK は元のリクエストのセグメントの ID をダウンストリームリクエストのトレースヘッダーに追加します。ダウンストリームリクエストを処理する計測対象アプリケーションで、親セグメント ID を記録して 2 つのリクエストを接続できます。

Example ルートトレース ID、親セグメント ID およびサンプリングデシジョンを含むトレースヘッダー

```
X-Amzn-Trace-Id: Root=1-5759e988-
bd862e3fe1be46a994272793;Parent=53995c3f42cd8ad8;Sampled=1
```

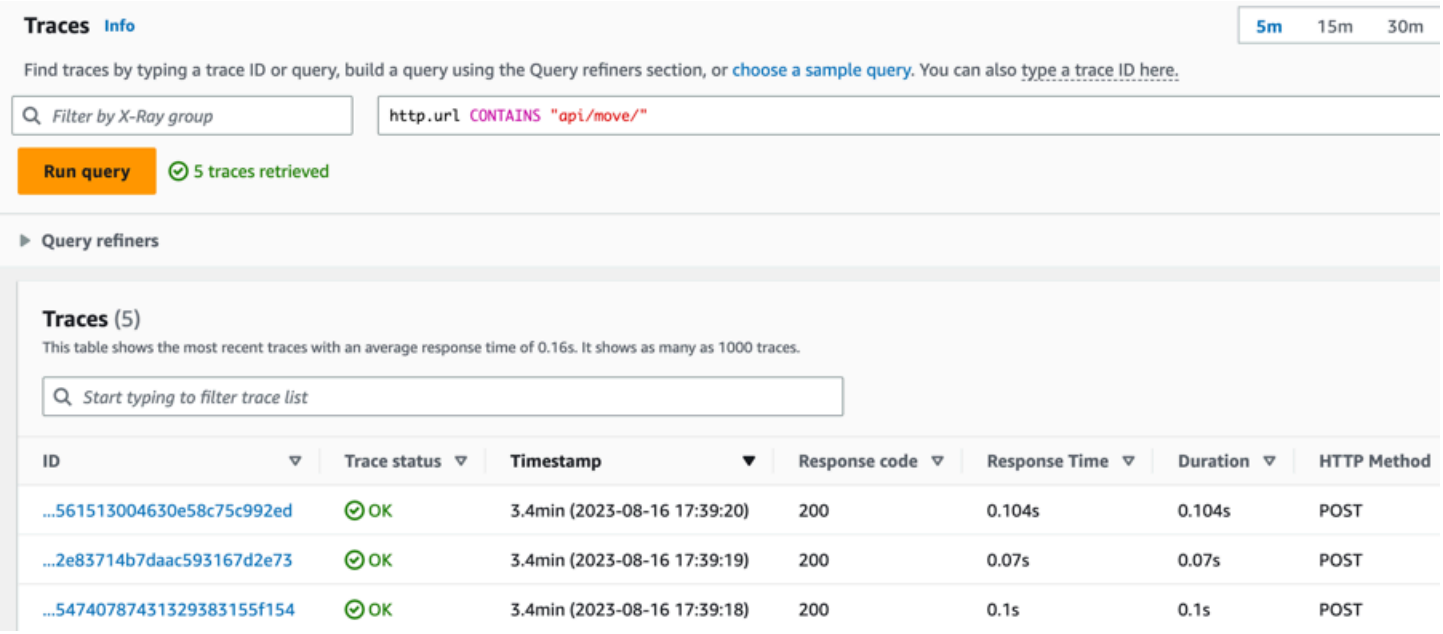
Lineage は、処理メカニズム AWS のサービス の一部として Lambda およびその他の によってトレースヘッダーに追加される可能性があるため、直接使用しないでください。

Example Lineage を含むトレースヘッダー

```
X-Amzn-Trace-Id: Root=1-5759e988-
bd862e3fe1be46a994272793;Parent=53995c3f42cd8ad8;Sampled=1;Lineage=a87bd80c:1|
68fd508a:5|c512fbe3:2
```

フィルタ式

サンプリングされている場合でも、複雑なアプリケーションでは大量のデータが生成されます。AWS X-Ray コンソールには、サービスグラフがeasy-to-navigateビューが表示されます。問題を識別しアプリケーションを最適化するために役立つ健全性とパフォーマンス情報が表示されます。高度なトレースでは、個別のリクエストのトレースを掘り下げたり、フィルタ式を使用して特定のパスまたはユーザーに関連するトレースを検索したりできます。



グループ

フィルタ式が拡張され、X-Ray でもグループ機能がサポートされます。フィルタ式を使用すると、グループへのトレースを受け入れる基準を定義できます。

グループを名前別または Amazon リソースネーム (ARN) 別に呼び出すことで、独自のサービスグラフ、トレースサマリ、および Amazon CloudWatch メトリクスを生成することができます。グループの作成後、着信トレースは、X-Ray サービスに格納されるときにグループのフィルタ式と照合されます。各基準に一致するトレース数のメトリクスは、1 分ごとに CloudWatch に公開されます。

グループのフィルタ式を更新しても、すでに記録されているデータは変わりません。更新は後続のトレースにのみ適用されます。これにより、新しい式と古い式がマージされたグラフが表示される場合があります。これを回避するには、現在のグループを削除し、新しいグループを作成します。

> **Note**
>
> グループは、フィルタ式と一致する取得済みのトレースの数で請求されます。詳細については、AWS X-Ray の料金を参照してください。

グループの詳細については、「グループの設定」を参照してください。

注釈とメタデータ

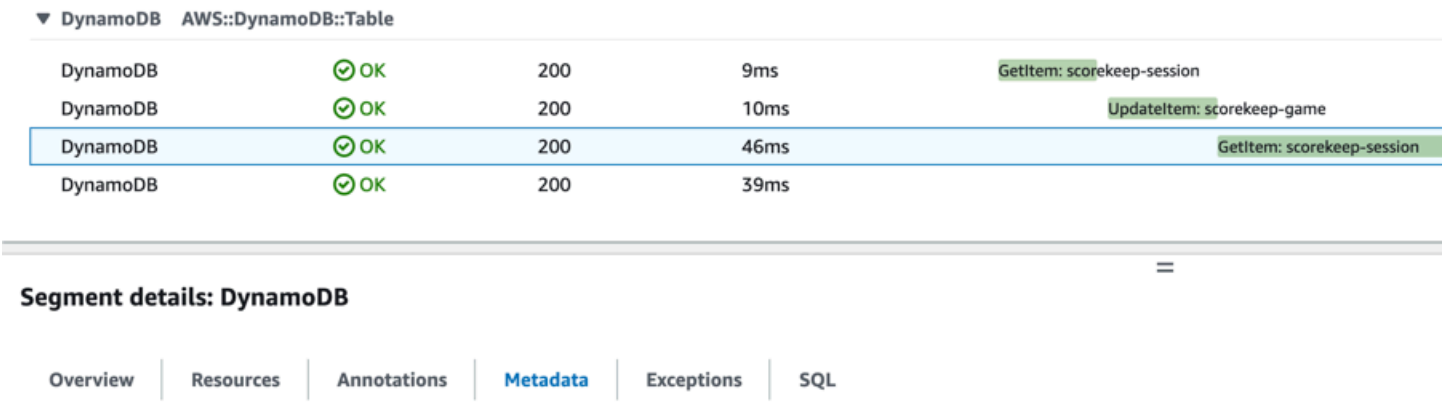
アプリケーションを計測すると、X-Ray SDK は受信リクエストと送信リクエスト、使用される AWS リソース、およびアプリケーション自体に関する情報を記録します。その他の情報を注釈およびメタデータとして、セグメントドキュメントに追加することもできます。注釈とメタデータはトレースレベルで集約され、任意のセグメントまたはサブセグメントに追加できます。

注釈は、「フィルタ式」で使用するためにインデックス化されたシンプルなキーと値のペアです。注釈を使用して、コンソールでトレースをグループ化するため、または`GetTraceSummaries` API を呼び出すときに使用するデータを記録します。

X-Ray は、トレースごとに 50 の注釈までインデックスを付けます。

メタデータは、オブジェクトとリストを含む、任意のタイプの値を持つことができるキーと値のペアですが、フィルタ式に使用するためにインデックスは作成されません。メタデータを使用してトレースに保存するデータを記録しますが、トレースの検索用に使用する必要はありません。

セグメントまたはサブセグメントの詳細ウィンドウ内の注釈およびメタデータは、CloudWatch コンソールでトレースの詳細ページに表示できます。



エラー、障害、および例外

X-Ray は、アプリケーションコードで発生するエラーと、ダウンストリームサービスから返されるエラーをトレースします。エラーは次のように分類されます。

- **`Error`** – クライアントエラー (400 系のエラー)
- **`Fault`** – サーバー障害 (500 系のエラー)
- **`Throttle`** – スロットリングエラー (429 Too Many Requests)

アプリケーションが実装されたリクエストを処理しているときに例外が発生すると、X-Ray SDK はスタックトレースを含む例外に関する詳細を記録します (利用可能な場合)。X-Ray コンソールでセグメント詳細の例外を表示できます。

のセキュリティ AWS X-Ray

でのクラウドセキュリティが最優先事項 AWS です。 AWS カスタマーは、最もセキュリティの影響を受けやすい組織の要件を満たすように構築されたデータセンターとネットワークアーキテクチャを活用できます。

セキュリティは、 AWS とユーザーの間で共有される責任です。責任共有モデルでは、この責任がクラウドのセキュリティおよびクラウド内のセキュリティとして説明されています。

- クラウドのセキュリティ – AWS は、 AWS のサービス で実行されるインフラストラクチャを保護する責任を担います AWS クラウド。 は、安全に使用できるサービス AWS も提供します。セキュリティの有効性は、AWS コンプライアンスプログラムの一環として、サードパーティーの審査機関によって定期的にテストおよび検証されています。X-Ray に適用されるコンプライアンスプログラムの詳細については、「コンプライアンスプログラムの対象範囲となるAWS のサービス」を参照してください。
- クラウド内のセキュリティ – お客様の責任は、使用する によって決まり AWS のサービス ます。また、お客様は、お客様のデータの機密性、組織の要件、および適用可能な法律および規制などの他の要因についても責任を担います。

このドキュメントは、X-Ray を使用するときに、共有責任モデルを適用する方法を理解するのに役立ちます。以下のトピックでは、セキュリティおよびコンプライアンスの目的を達成するために X-Ray を設定する方法を示します。また、X-Ray リソースのモニタリングと保護 AWS のサービス に役立つ他の の使用方法についても説明します。

トピック

- でのデータ保護 AWS X-Ray
- の ID とアクセスの管理 AWS X-Ray
- のコンプライアンス検証 AWS X-Ray
- の耐障害性 AWS X-Ray
- のインフラストラクチャセキュリティ AWS X-Ray

でのデータ保護 AWS X-Ray

AWS X-Ray は、保管中のトレースおよび関連データを常に暗号化します。コンプライアンスまたは内部要件について暗号化キーを監査および無効化する必要がある場合は、 AWS Key Management Service （ AWS KMS) キーを使用してデータを暗号化するように X-Ray を設定できます。

X-Ray は AWS マネージドキー という名前の を提供しますaws/xray。AWS CloudTrailでキーの使用状況を監査するだけで、キー自体を管理する必要がない場合は、このキーを使用します。キーへのアクセスを管理したり、キーの更新を設定したりする必要がある場合は、カスタマー管理のキーを作成できます。

暗号化設定を変更すると、X-Ray でのデータキーの生成および伝達に少し時間がかかります。新しいキーの処理中に、X-Ray は新しい設定と古い設定を組み合わせてデータを暗号化することがあります。暗号化設定を変更するときに、既存のデータは再暗号化されません。

> **Note**
>
> AWS KMS は、X-Ray が KMS キーを使用してトレースデータを暗号化または復号するときに課金されます。
>
> - デフォルトの暗号化 – 無料。
> - AWS マネージドキー – キーの使用について料金が発生します。
> - カスタマー管理キー – キーの保存と使用について料金が発生します。
>
> 詳細については、「AWS Key Management Service の料金」を参照してください。

> **Note**
>
> X-Ray インサイト通知は現在カスタマー管理キーをサポートしていない Amazon EventBridge にイベントを送信します。詳細については、「Amazon EventBridge におけるデータ保護』を参照してください。

カスタマー管理キーを使用して暗号化されたトレースを表示するように X-Ray を設定するには、カスタマー管理キーに対するユーザーレベルのアクセス権が必要です。詳細については「暗号化のユーザーアクセス許可」を参照してください。

CloudWatch console

CloudWatch コンソールを使用して暗号化に KMS キーを使用するように X-Ray を設定するには

1. にサインイン AWS Management Console し、https://console.aws.amazon.com/cloudwatch/://www.com」で CloudWatch コンソールを開きます。
2. 左側のナビゲーションペインの [設定] を選択します。
3. X-Ray トレースセクションの [暗号化] の下にある [設定を表示] を選択します。
4. [暗号化の設定] セクションで、[編集] を選択します。
5. [KMS キーを使用] を選択します。
6. ドロップダウンメニューからキーを選択します。
   - aws/xray – AWS マネージドキーを使用します。
   - キーエイリアス – アカウントでカスタマー管理キーを使用します。
   - キー ARN を手動で入力 – 別のアカウントのカスタマー管理キーを使用します。表示されるフィールドに、キーの完全な Amazon リソースネーム (ARN) を入力します。
7. [暗号化の更新] を選択します。

X-Ray console

X-Ray コンソールを使用して暗号化に KMS キーを使用するように X-Ray を設定するには

1. [X-Ray console (X-Ray コンソール)] を開きます。
2. [暗号化] を選択します。
3. [Use a KMS key (KMS キーを使用する)] を選択します。
4. ドロップダウンメニューからキーを選択します。
   - aws/xray – AWS マネージドキーを使用します。
   - キーエイリアス – アカウントでカスタマー管理キーを使用します。
   - キー ARN を手動で入力 – 別のアカウントのカスタマー管理キーを使用します。表示されるフィールドに、キーの完全な Amazon リソースネーム (ARN) を入力します。
5. [Apply] を選択します。

> **Note**
>
> X-Ray は非対称 KMS キー をサポートしていません。

X-Ray が暗号化キーにアクセスできない場合、データの保存を停止します。これは、ユーザーが KMS キーにアクセスできなくなった場合や、現在使用されているキーを無効にした場合に発生する可能性があります。この場合、X-Ray はナビゲーションバーに通知を表示します。

X-Ray API を使用して暗号化設定を指定する方法については、「AWS X-Ray API を使用したサンプリング、グループ、暗号化設定の構成」を参照してください。

の ID とアクセスの管理 AWS X-Ray

AWS Identity and Access Management (IAM) は、管理者が AWS リソースへのアクセスを安全に制御 AWS のサービス するのに役立つ です。IAM 管理者は、誰を認証 (サインイン) し、誰に X-Ray リソースの使用を許可する (アクセス許可を持たせる) かを制御します。IAM は、追加料金なしで使用できる AWS のサービス です。

トピック

- 対象者
- アイデンティティを使用した認証
- ポリシーを使用したアクセスの管理
- AWS X-Ray が IAM と連携する方法
- AWS X-Ray アイデンティティベースのポリシーの例
- AWS X-Ray ID とアクセスのトラブルシューティング

対象者

AWS Identity and Access Management (IAM) の使用方法は、X-Ray で行う作業によって異なります。

サービスユーザー - X-Ray サービスを使用してジョブを実行する場合は、必要なアクセス許可と認証情報を管理者が用意します。作業を実行するためにさらに多くの X-Ray の特徴を使用するとき、追加の許可が必要になる場合があります。アクセスの管理方法を理解すると、管理者に適切なアクセス

許可をリクエストするのに役に立ちます。X-Ray の特徴にアクセスできない場合は、「AWS X-Ray ID とアクセスのトラブルシューティング」を参照してください。

サービス管理者 - 社内の X-Ray リソースを担当している場合は、通常、X-Ray へのフルアクセスがあります。サービスのユーザーがどの X-Ray 機能やリソースにアクセスするかを決めるのは管理者の仕事です。その後、IAM 管理者にリクエストを送信して、サービスユーザーの権限を変更する必要があります。このページの情報を点検して、IAM の基本概念を理解してください。貴社が X-Ray で IAM を利用する方法の詳細については、「AWS X-Ray が IAM と連携する方法」を参照してください。

IAM 管理者 - IAM 管理者は、X-Ray へのアクセスを管理するポリシーの作成方法の詳細について確認する場合があります。IAM で使用できる X-Ray アイデンティティベースのポリシーの例を表示するには、「AWS X-Ray アイデンティティベースのポリシーの例」を参照してください。

アイデンティティを使用した認証

認証とは、ID 認証情報 AWS を使用して にサインインする方法です。として、IAM ユーザーとして AWS アカウントのルートユーザー、または IAM ロールを引き受けることによって、認証 (にサインイン AWS) される必要があります。

ID ソースを介して提供された認証情報を使用して、フェデレーティッド ID AWS として にサインインできます。 AWS IAM Identity Center （ IAM Identity Center) ユーザー、会社のシングルサインオン認証、Google または Facebook 認証情報は、フェデレーション ID の例です。フェデレーティッド ID としてサインインする場合、IAM ロールを使用して、前もって管理者により ID フェデレーションが設定されています。フェデレーション AWS を使用して にアクセスすると、間接的にロールを引き受けることになります。

ユーザーの種類に応じて、 AWS Management Console または AWS アクセスポータルにサインインできます。へのサインインの詳細については AWS、「 AWS サインイン ユーザーガイド」の「 へのサインイン AWS アカウント方法」を参照してください。

AWS プログラムで にアクセスする場合、 はソフトウェア開発キット (SDK) とコマンドラインインターフェイス (CLI) AWS を提供し、認証情報を使用してリクエストを暗号化して署名します。AWS ツールを使用しない場合は、自分でリクエストに署名する必要があります。リクエストに自分で署名する推奨方法の使用については、「IAM ユーザーガイド」の「API リクエストに対するAWS Signature Version 4」を参照してください。

使用する認証方法を問わず、追加セキュリティ情報の提供をリクエストされる場合もあります。たとえば、 では、アカウントのセキュリティを高めるために多要素認証 (MFA) を使用する AWS ことを

お勧めします。詳細については、「AWS IAM Identity Center ユーザーガイド」の「多要素認証」および「IAM ユーザーガイド」の「IAM のAWS 多要素認証」を参照してください。

AWS アカウント ルートユーザー

を作成するときは AWS アカウント、アカウント内のすべての およびリソースへの AWS のサービス完全なアクセス権を持つ 1 つのサインインアイデンティティから始めます。この ID は AWS アカウント ルートユーザーと呼ばれ、アカウントの作成に使用した E メールアドレスとパスワードでサインインすることでアクセスできます。日常的なタスクには、ルートユーザーを使用しないことを強くお勧めします。ルートユーザーの認証情報は保護し、ルートユーザーでしか実行できないタスクを実行するときに使用します。ルートユーザーとしてサインインする必要があるタスクの完全なリストについては、「IAM ユーザーガイド」の「ルートユーザー認証情報が必要なタスク」を参照してください。

IAM ユーザーとグループ

IAM ユーザーは、単一のユーザーまたはアプリケーションに対して特定のアクセス許可 AWS アカウント を持つ 内の ID です。可能であれば、パスワードやアクセスキーなどの長期的な認証情報を保有する IAM ユーザーを作成する代わりに、一時的な認証情報を使用することをお勧めします。ただし、IAM ユーザーでの長期的な認証情報が必要な特定のユースケースがある場合は、アクセスキーをローテーションすることをお勧めします。詳細については、「IAM ユーザーガイド」の「長期的な認証情報を必要とするユースケースのためにアクセスキーを定期的にローテーションする」を参照してください。

IAM グループは、IAM ユーザーの集団を指定するアイデンティティです。グループとしてサインインすることはできません。グループを使用して、複数のユーザーに対して一度に権限を指定できます。多数のユーザーグループがある場合、グループを使用することで権限の管理が容易になります。例えば、IAMAdmins という名前のグループを設定して、そのグループに IAM リソースを管理する許可を与えることができます。

ユーザーは、ロールとは異なります。ユーザーは 1 人の人または 1 つのアプリケーションに一意に関連付けられますが、ロールはそれを必要とする任意の人が引き受けるようになっています。ユーザーには永続的な長期の認証情報がありますが、ロールでは一時認証情報が提供されます。詳細については、「IAM ユーザーガイド」の「IAM ユーザーに関するユースケース」を参照してください。

IAM ロール

IAM ロールは、特定のアクセス許可 AWS アカウント を持つ 内の ID です。これは IAM ユーザーに似ていますが、特定のユーザーには関連付けられていません。で IAM ロールを一時的に引き受ける

には AWS Management Console、ユーザーから IAM ロール (コンソール) に切り替えることができます。ロールを引き受けるには、 または AWS API オペレーションを AWS CLI 呼び出すか、カスタム URL を使用します。ロールを使用する方法の詳細については、「IAM ユーザーガイド」の「ロールを引き受けるための各種方法」を参照してください。

IAM ロールと一時的な認証情報は、次の状況で役立ちます:

- フェデレーションユーザーアクセス – フェデレーティッド ID に許可を割り当てるには、ロールを作成してそのロールの許可を定義します。フェデレーティッド ID が認証されると、その ID はロールに関連付けられ、ロールで定義されている許可が付与されます。フェデレーションのロールについては、「IAM ユーザーガイド」の「サードパーティー ID プロバイダー (フェデレーション) 用のロールを作成する」を参照してください。IAM Identity Center を使用する場合は、許可セットを設定します。アイデンティティが認証後にアクセスできるものを制御するため、IAM Identity Center は、権限セットを IAM のロールに関連付けます。アクセス許可セットの詳細については、「AWS IAM Identity Center User Guide」の「Permission sets」を参照してください。
- 一時的な IAM ユーザー権限 - IAM ユーザーまたはロールは、特定のタスクに対して複数の異なる権限を一時的に IAM ロールで引き受けることができます。
- クロスアカウントアクセス - IAM ロールを使用して、自分のアカウントのリソースにアクセスすることを、別のアカウントの人物 (信頼済みプリンシパル) に許可できます。クロスアカウントアクセス権を付与する主な方法は、ロールを使用することです。ただし、一部の では AWS のサービス、 (プロキシとしてロールを使用する代わりに) リソースに直接ポリシーをアタッチできます。クロスアカウントアクセスにおけるロールとリソースベースのポリシーの違いについては、「IAM ユーザーガイド」の「IAM でのクロスアカウントのリソースへのアクセス」を参照してください。
- クロスサービスアクセス — 一部の は他の の機能 AWS のサービス を使用します AWS のサービス。例えば、あるサービスで呼び出しを行うと、通常そのサービスによって Amazon EC2 でアプリケーションが実行されたり、Amazon S3 にオブジェクトが保存されたりします。サービスでは、呼び出し元プリンシパルの許可、サービスロール、またはサービスリンクロールを使用してこれを行う場合があります。
  - 転送アクセスセッション (FAS) – IAM ユーザーまたはロールを使用してアクションを実行すると AWS、プリンシパルと見なされます。一部のサービスを使用する際に、アクションを実行することで、別のサービスの別のアクションがトリガーされることがあります。FAS は、 を呼び出すプリンシパルのアクセス許可を AWS のサービス、ダウンストリームサービス AWS のサービス へのリクエストをリクエストする と組み合わせて使用します。FAS リクエストは、サービスが他の AWS のサービス またはリソースとのやり取りを完了する必要があるリクエストを受け取った場合にのみ行われます。この場合、両方のアクションを実行するためのアクセス許可が必

要です。FAS リクエストを行う際のポリシーの詳細については、「転送アクセスセッション」を参照してください。

- サービスロール - サービスがユーザーに代わってアクションを実行するために引き受ける IAM ロールです。IAM 管理者は、IAM 内からサービスロールを作成、変更、削除することができます。詳細については、「IAM ユーザーガイド」の「AWS のサービスに許可を委任するロールを作成する」を参照してください。
- サービスにリンクされたロール – サービスにリンクされたロールは、 にリンクされたサービスロールの一種です AWS のサービス。サービスは、ユーザーに代わってアクションを実行するロールを引き受けることができます。サービスにリンクされたロールは に表示され AWS アカウント 、サービスによって所有されます。IAM 管理者は、サービスリンクロールのアクセス許可を表示できますが、編集することはできません。

- Amazon EC2 で実行されているアプリケーション – IAM ロールを使用して、EC2 インスタンスで実行され、 AWS CLI または AWS API リクエストを行うアプリケーションの一時的な認証情報を管理できます。これは、EC2 インスタンス内でのアクセスキーの保存に推奨されます。EC2 インスタンスに AWS ロールを割り当て、そのすべてのアプリケーションで使用できるようにするには、インスタンスにアタッチされたインスタンスプロファイルを作成します。インスタンスプロファイルにはロールが含まれ、EC2 インスタンスで実行されるプログラムは一時的な認証情報を取得できます。詳細については、「IAM ユーザーガイド」の「Amazon EC2 インスタンスで実行されるアプリケーションに IAM ロールを使用して許可を付与する」を参照してください。

ポリシーを使用したアクセスの管理

でアクセスを制御する AWS には、ポリシーを作成し、ID AWS またはリソースにアタッチします。ポリシーは AWS 、アイデンティティまたはリソースに関連付けられているときにアクセス許可を定義する のオブジェクトです。 は、プリンシパル (ユーザー、ルートユーザー、またはロールセッション) がリクエストを行うときに、これらのポリシー AWS を評価します。ポリシーでの権限により、リクエストが許可されるか拒否されるかが決まります。ほとんどのポリシーは JSON ドキュメント AWS として に保存されます。JSON ポリシードキュメントの構造と内容の詳細については、「IAM ユーザーガイド」の「JSON ポリシー概要」を参照してください。

管理者は JSON AWS ポリシーを使用して、誰が何にアクセスできるかを指定できます。つまり、どのプリンシパルがどのリソースに対してどのような条件下でアクションを実行できるかということです。

デフォルトでは、ユーザーやロールに権限はありません。IAM 管理者は、リソースで必要なアクションを実行するための権限をユーザーに付与する IAM ポリシーを作成できます。その後、管理者はロールに IAM ポリシーを追加し、ユーザーはロールを引き受けることができます。

IAM ポリシーは、オペレーションの実行方法を問わず、アクションの許可を定義します。例えば、`iam:GetRole` アクションを許可するポリシーがあるとします。そのポリシーを持つユーザーは、 AWS Management Console、、 AWS CLIまたは AWS API からロール情報を取得できます。

アイデンティティベースのポリシー

アイデンティティベースポリシーは、IAM ユーザーグループ、ユーザーのグループ、ロールなど、アイデンティティにアタッチできる JSON 許可ポリシードキュメントです。これらのポリシーは、ユーザーとロールが実行できるアクション、リソース、および条件をコントロールします。アイデンティティベースポリシーの作成方法については、「IAM ユーザーガイド」の「カスタマー管理ポリシーでカスタム IAM アクセス許可を定義する」を参照してください。

アイデンティティベースのポリシーは、さらにインラインポリシーまたはマネージドポリシーに分類できます。インラインポリシーは、単一のユーザー、グループ、またはロールに直接埋め込まれています。管理ポリシーは、 内の複数のユーザー、グループ、ロールにアタッチできるスタンドアロンポリシーです AWS アカウント。管理ポリシーには、 AWS 管理ポリシーとカスタマー管理ポリシーが含まれます。マネージドポリシーまたはインラインポリシーのいずれかを選択する方法については、「IAM ユーザーガイド」の「管理ポリシーとインラインポリシーのいずれかを選択する」を参照してください。

リソースベースのポリシー

リソースベースのポリシーは、リソースに添付する JSON ポリシードキュメントです。リソースベースのポリシーには例として、IAM ロールの信頼ポリシーや Amazon S3 バケットポリシーがあげられます。リソースベースのポリシーをサポートするサービスでは、サービス管理者はポリシーを使用して特定のリソースへのアクセスを制御できます。ポリシーがアタッチされているリソースの場合、指定されたプリンシパルがそのリソースに対して実行できるアクションと条件は、ポリシーによって定義されます。リソースベースのポリシーでは、プリンシパルを指定する必要があります。プリンシパルには、アカウント、ユーザー、ロール、フェデレーティッドユーザー、または を含めることができます AWS のサービス。

リソースベースのポリシーは、そのサービス内にあるインラインポリシーです。リソースベースのポリシーでは、IAM の AWS マネージドポリシーを使用できません。

アクセスコントロールリスト (ACL)

アクセスコントロールリスト (ACL) は、どのプリンシパル (アカウントメンバー、ユーザー、またはロール) がリソースにアクセスするための許可を持つかを制御します。ACL はリソースベースのポリシーに似ていますが、JSON ポリシードキュメント形式は使用しません。

Amazon S3、および Amazon VPC は AWS WAF、ACLs。ACL の詳細については、「Amazon Simple Storage Service デベロッパーガイド」の「アクセスコントロールリスト (ACL) の概要」を参照してください。

その他のポリシータイプ

AWS は、追加のあまり一般的ではないポリシータイプをサポートしています。これらのポリシータイプでは、より一般的なポリシータイプで付与された最大の権限を設定できます。

- アクセス許可の境界 - アクセス許可の境界は、アイデンティティベースポリシーによって IAM エンティティ (IAM ユーザーまたはロール) に付与できる権限の上限を設定する高度な機能です。エンティティにアクセス許可の境界を設定できます。結果として得られる権限は、エンティティのアイデンティティベースポリシーとそのアクセス許可の境界の共通部分になります。`Principal` フィールドでユーザーまたはロールを指定するリソースベースのポリシーでは、アクセス許可の境界は制限されません。これらのポリシーのいずれかを明示的に拒否した場合、権限は無効になります。アクセス許可の境界の詳細については、「IAM ユーザーガイド」の「IAM エンティティのアクセス許可の境界」を参照してください。
- サービスコントロールポリシー (SCPs) – SCPsは、 の組織または組織単位 (OU) の最大アクセス許可を指定する JSON ポリシーです AWS Organizations。 AWS Organizations は、ビジネスが所有する複数の をグループ化して一元管理するためのサービス AWS アカウント です。組織内のすべての機能を有効にすると、サービスコントロールポリシー (SCP) を一部またはすべてのアカウントに適用できます。SCP は、各 を含むメンバーアカウントのエンティティのアクセス許可を制限します AWS アカウントのルートユーザー。Organizations と SCP の詳細については、「AWS Organizations ユーザーガイド」の「サービスコントロールポリシー (SCP)」を参照してください。
- リソースコントロールポリシー (RCP) – RCP は、所有する各リソースにアタッチされた IAM ポリシーを更新することなく、アカウント内のリソースに利用可能な最大数のアクセス許可を設定するために使用できる JSON ポリシーです。RCP は、メンバーアカウントのリソースのアクセス許可を制限し、組織に属しているかどうかにかかわらず AWS アカウントのルートユーザー、 を含む ID の有効なアクセス許可に影響を与える可能性があります。RCP をサポートする のリストを含む Organizations と RCP の詳細については、AWS Organizations RCPs「リソースコントロールポリシー (RCPs」を参照してください。 AWS のサービス
- セッションポリシー - セッションポリシーは、ロールまたはフェデレーションユーザーの一時的なセッションをプログラムで作成する際にパラメータとして渡す高度なポリシーです。結果としてセッションの権限は、ユーザーまたはロールのアイデンティティベースポリシーとセッションポリシーの共通部分になります。また、リソースベースのポリシーから権限が派生する場合もありま

す。これらのポリシーのいずれかを明示的に拒否した場合、権限は無効になります。詳細については、「IAM ユーザーガイド」の「セッションポリシー」を参照してください。

複数のポリシータイプ

1 つのリクエストに複数のタイプのポリシーが適用されると、結果として作成される権限を理解するのがさらに難しくなります。が複数のポリシータイプが関与する場合にリクエストを許可するかどうか AWS を決定する方法については、「IAM ユーザーガイド」の「ポリシー評価ロジック」を参照してください。

AWS X-Ray が IAM と連携する方法

X-Ray へのアクセスを管理するために IAM を使用する前に、X-Ray でどの IAM 機能が使用できるかを理解しておく必要があります。X-Ray およびその他の が IAM と AWS のサービス 連携する方法の概要を把握するには、IAM ユーザーガイドのAWS のサービス 「IAM と連携する」を参照してください。

AWS Identity and Access Management (IAM) を使用して、アカウントのユーザーとコンピューティングリソースに X-Ray アクセス許可を付与できます。IAM は、ユーザーが採用するクライアント (コンソール、 AWS SDK AWS CLI) に関係なく、アクセス許可を均一に強制するために、API レベルで X-Ray サービスへのアクセスを制御します。

X-Ray コンソールを使用してトレースマップやセグメントを表示する場合に必要なのは、読み取りアクセス許可だけです。コンソールアクセスを有効にするには、`AWSXrayReadOnlyAccess` 管理ポリシーを IAM ユーザーに追加します。

ローカルの開発とテストには、読み書きのアクセス許可を持つ IAM ロールを作成します。ロールを引き受け、そのロールの一時的な認証情報を保存します。これらの認証情報は、X-Ray デーモン、AWS CLI、および AWS SDK で使用できます。詳細については、「AWS CLIでの一時的なセキュリティ認証情報の使用」を参照してください。

計測されたアプリケーションを にデプロイ AWSするには、書き込みアクセス許可を持つ IAM ロールを作成し、アプリケーションを実行しているリソースに割り当てます。`AWSXRayDaemonWriteAccess`には、トレースをアップロードするアクセス許可と、サンプリングルールの使用をサポートするいくつかの読み取りアクセス許可が含まれています。

読み書きポリシーには、暗号化キー設定とサンプリングルールを指定するためのアクセス許可は含まれていません。`AWSXrayFullAccess` を使用して、これらの設定にアクセスするか、カスタムポリ

シーに設定 API を追加します。作成したカスタマー管理キーで暗号化と複合を行うには、キーを使用するためのアクセス許可も必要です。

トピック

- X-Ray アイデンティティベースのポリシー
- X-Ray リソースベースのポリシー
- X-Ray タグに基づいた承認
- アプリケーションをローカルで実行する
- でアプリケーションを実行する AWS
- 暗号化のユーザーアクセス許可

X-Ray アイデンティティベースのポリシー

IAM アイデンティティベースのポリシーでは許可または拒否するアクションとリソース、またアクションを許可または拒否する条件を指定できます。X-Ray は、特定のアクション、リソース、および条件キーをサポートしています。JSON ポリシーで使用するすべての要素については、「IAM ユーザーガイド」の「IAM JSON ポリシー要素のリファレンス」を参照してください。

アクション

管理者は JSON AWS ポリシーを使用して、誰が何にアクセスできるかを指定できます。つまり、どのプリンシパルがどのリソースに対してどのような条件下でアクションを実行できるかということです。

JSON ポリシーの `Action` 要素にはポリシー内のアクセスを許可または拒否するために使用できるアクションが記述されます。ポリシーアクションの名前は通常、関連付けられた AWS API オペレーションと同じです。一致する API オペレーションのない許可のみのアクションなど、いくつかの例外があります。また、ポリシーに複数のアクションが必要なオペレーションもあります。これらの追加アクションは依存アクションと呼ばれます。

このアクションは関連付けられたオペレーションを実行するためのアクセス許可を付与するポリシーで使用されます。

X-Ray のポリシーアクションは、アクションの前に以下のプレフィックス を使用します: `xray:`。たとえば、X-Ray `GetGroup` API オペレーションを使用してグループリソースの詳細を取得するためのアクセス許可をユーザーに付与するには、ポリシーに `xray:GetGroup` アクションを含めます。

ポリシーステートメントにはAction または NotAction 要素を含める必要があります。X-Ray は、このサービスで実行できるタスクを記述する独自のアクションのセットを定義します。

単一のステートメントに複数のアクションを指定するには次のようにコンマで区切ります。

```
"Action": [
      "xray:action1",
      "xray:action2"
```

ワイルドカード (*) を使用して複数アクションを指定できます。たとえば、Get という単語で始まるすべてのアクションを指定するには、次のアクションを含めます。

```
"Action": "xray:Get*"
```

X-Ray アクションのリストを表示するには、IAM ユーザーガイドの「AWS X-Rayによって定義されたアクション」を参照してください。

リソース

管理者は JSON AWS ポリシーを使用して、誰が何にアクセスできるかを指定できます。つまり、どのプリンシパルがどのリソースに対してどのような条件下でアクションを実行できるかということです。

Resource JSON ポリシー要素はアクションが適用されるオブジェクトを指定します。ステートメントにはResource または NotResource 要素を含める必要があります。ベストプラクティスとして、Amazon リソースネーム (ARN) を使用してリソースを指定します。これは、リソースレベルの許可と呼ばれる特定のリソースタイプをサポートするアクションに対して実行できます。

オペレーションのリスト化など、リソースレベルの権限をサポートしないアクションの場合は、ステートメントがすべてのリソースに適用されることを示すために、ワイルドカード (*) を使用します。

```
"Resource": "*"
```

IAM ポリシーを使用してリソースへのアクセスを制御できます。リソースレベルのアクセス許可をサポートするアクションの場合は、Amazon リソースネーム (ARN) を使用して、ポリシーが適用されるリソースを識別します。

X-Ray ポリシーではすべての IAM アクションを使用して、そのアクションを使用するアクセス許可をユーザーに付与または拒否できます。ただし、すべての X-Ray アクションが、アクションを実行

することができるリソースを指定できる、リソースレベルのアクセス許可をサポートしているわけではありません。

リソースレベルの権限をサポートしていないアクションの場合、「*」をリソースとして使用する必要があります。

次の X-Ray アクションは、リソースレベルのアクセス許可をサポートします。

- `CreateGroup`
- `GetGroup`
- `UpdateGroup`
- `DeleteGroup`
- `CreateSamplingRule`
- `UpdateSamplingRule`
- `DeleteSamplingRule`

以下は、`CreateGroup` アクションのアイデンティティベースのアクセス許可ポリシーの例です。この例では、グループ名 `local-users` に関連する ARN を使用する一意の ID をワイルドカードとして使用します。グループが作成されたときに一意の ID が生成されるため、事前にポリシーで予測することはできません。`GetGroup`、`UpdateGroup`、または `DeleteGroup` を使用する場合、ワイルドカードとして、または ID を含む正確な ARN として定義できます。

> **Note**
>
> サンプリングルールの ARN は、名前によって定義されます。グループ ARN とは異なり、サンプリングルールには一意に生成された ID がありません。

X-Ray リソースタイプとその ARN のリストを表示するには、IAM ユーザーガイドの「AWS X-Ray で定義されるリソース」を参照してください。どのアクションで各リソースの ARN を指定できるかについては、AWS X-Rayで定義されるアクションを参照してください。

条件キー

X-Ray にはサービス固有条件キーがありませんが、いくつかのグローバル条件キーの使用がサポートされています。すべての AWS グローバル条件キーを確認するには、IAM ユーザーガイドのAWS「グローバル条件コンテキストキー」を参照してください。

例

X-Ray アイデンティティベースのポリシーの例を表示するには、「AWS X-Ray アイデンティティベースのポリシーの例」を参照してください。

X-Ray リソースベースのポリシー

X-Ray は、Amazon SNS アクティブトレースなど、現在および将来の AWS のサービス 統合のためのリソースベースのポリシーをサポートしています。X-Ray リソースベースのポリシーは、他の AWS Management Consoleによって、または AWS SDK または CLI を使用して更新できます。例えば、Amazon SNS コンソールは、X-Ray にトレースを送信するためのリソースベースのポリシーを自動的に設定しようとします。次のポリシードキュメントは、X-Ray リソースベースのポリシーを手動で設定する例を示しています。

Example Amazon SNS アクティブトレース用の X-Ray リソースベースのポリシーの例

このポリシードキュメントの例では、Amazon SNS がトレースデータを X-Ray に送信するために必要なアクセス許可を指定します。

```
{
    Version: "2012-10-17",
    Statement: [
      {
        Sid: "SNSAccess",
        Effect: Allow,
        Principal: {
          Service: "sns.amazonaws.com",
        },
        Action: [
          "xray:PutTraceSegments",
          "xray:GetSamplingRules",
          "xray:GetSamplingTargets"
        ],
        Resource: "*",
        Condition: {
          StringEquals: {
            "aws:SourceAccount": "account-id"
          },
          StringLike: {
            "aws:SourceArn": "arn:partition:sns:region:account-id:topic-name"
          }
        }
```

```
      }
    ]
  }
```

CLI を使用して、Amazon SNS アクセス許可を付与するリソースベースのポリシーを作成して、トレースデータを X-Ray に送信します。

```
aws xray put-resource-policy --policy-name MyResourcePolicy --policy-document
 '{ "Version": "2012-10-17", "Statement": [ { "Sid": "SNSAccess", "Effect": "Allow",
 "Principal": { "Service": "sns.amazonaws.com" }, "Action": [ "xray:PutTraceSegments",
 "xray:GetSamplingRules", "xray:GetSamplingTargets" ], "Resource": "*",
 "Condition": { "StringEquals": { "aws:SourceAccount": "account-id" }, "StringLike":
 { "aws:SourceArn": "arn:partition:sns:region:account-id:topic-name" } } } ] }'
```

これらの例を使用するには、*partition*、、、 *topic-name*を特定の AWS パーティション*regionaccount-id*、リージョン、アカウント ID、Amazon SNS トピック名に置き換えます。すべての Amazon SNS トピックに、トレースデータを X-Ray に送信するアクセス許可を付与するには、トピック名を * に置き換えます。

X-Ray タグに基づいた承認

X-Ray グループまたはサンプリングルールにタグをアタッチしたり、リクエストでタグを X-Ray に渡すことができます。タグに基づいてアクセスを制御するには`xray:ResourceTag/`*key-name*、`aws:RequestTag/`*key-name*、または `aws:TagKeys` の条件キーを使用して、ポリシーの条件要素でタグ情報を提供します。X-Ray リソースのタグ付けの詳細については、「X-Ray のサンプリングルールとグループのタグ付け」を参照してください。

リソースのタグに基づいてリソースへのアクセスを制限するためのアイデンティティベースポリシーの例を表示するには、「タグに基づいて X-Ray グループおよびサンプリングルールへのアクセスを管理する」を参照してください。

アプリケーションをローカルで実行する

実装したアプリケーションから X-Ray デーモンがトレースデータに送信されます。デーモンは、セグメントドキュメントをバッファし、バッチ処理で X-Ray サービスにアップロードします。デーモンは、X-Ray サービスにトレースデータおよびテレメトリをアップロードする書き込みのアクセス許可が必要です。

デーモンをローカルで実行する場合は、IAM ロールを作成し、そのロールを引き受けて、一時的な認証情報を環境変数に保存するか、ユーザーフォルダー内の `.aws` フォルダーの `credentials`

ファイルに保存します。詳細については、「AWS CLIでの一時的なセキュリティ認証情報の使用」を参照してください。

Example ~/.aws/credentials

```
[default]
aws_access_key_id={access key ID}
aws_secret_access_key={access key}
aws_session_token={AWS session token}
```

AWS SDK または で使用する認証情報をすでに設定している場合 AWS CLI、デーモンはそれらを使用できます。複数のプロファイルが利用可能な場合はデフォルトのプロファイルが使用されます。

でアプリケーションを実行する AWS

でアプリケーションを実行するときは AWS、 ロールを使用して、デーモンを実行する Amazon EC2 インスタンスまたは Lambda 関数にアクセス許可を付与します。

- Amazon Elastic Compute Cloud (Amazon EC2) – IAM ロールを作成し、「インスタンスプロファイル」として EC2 インスタンスにアタッチします。
- Amazon Elastic Container Service (Amazon ECS) – IAM ロールを作成し、コンテナインスタンスの IAM ロールとしてコンテナインスタンスにアタッチします。
- AWS Elastic Beanstalk (Elastic Beanstalk) – Elastic Beanstalk には、デフォルトのインスタンスプロファイルに X-Ray アクセス許可が含まれます。デフォルトのインスタンスプロファイルを使用するか、カスタムのインスタンスプロファイルに書き込みのアクセス許可を追加できます。
- AWS Lambda (Lambda) – 関数の実行ロールに書き込みアクセス許可を追加します。

X-Ray で使用するためのロールを作成するには

1. [IAM コンソール] を開きます。
2. [ロール] を選択します。
3. [Create New Role (新しいロールを作成)] を選択します。
4. [Role Name (ロール名)] に **`xray-application`** を入力します。[Next Step (次のステップ)] をクリックします。
5. [Role Type (ロールタイプ)] で、[Amazon EC2] を選択します。
6. 次の管理ポリシーをアタッチして AWS のサービスへのアクセス権限をアプリケーションに付与します。

- AWSXRayDaemonWriteAccess - トレースデータを X-Ray にアップロードするためのアクセス許可を付与します。

アプリケーションが AWS SDK を使用して他の サービスにアクセスする場合は、それらのサービスへのアクセスを許可するポリシーを追加します。

7. [Next Step] (次のステップ) をクリックします。
8. [Create Role (ロールを作成)] を選択します。

暗号化のユーザーアクセス許可

X-Ray は、すべてのトレースデータを暗号化します。デフォルトでは、管理するキーを使用するように設定できます。 AWS Key Management Service カスタマーマネージドキーを選択した場合は、キーのアクセスポリシーで、暗号化に使用するアクセス許可を X-Ray に付与できることを確認する必要があります。また、アカウントの他のユーザーがキーにアクセスし、X-Ray コンソールで暗号化されたトレースデータを確認できるようにする必要があります。

カスタマー管理キーに関しては、以下のアクションが可能なアクセスポリシーを使用してキーを設定します。

- X-Ray でキーを設定するユーザーには、`kms:CreateGrant` と `kms:DescribeKey` を呼び出すためのアクセス許可があります。
- 暗号化されたトレースデータにアクセスできるユーザーには、`kms:Decrypt` を呼び出すためのアクセス許可があります。

IAM コンソールのキー設定セクション内の主要ユーザーグループにユーザーを追加する場合、彼らにはこれらの両方のオペレーションに対するアクセス許可があります。アクセス許可はキーポリシーにのみ設定する必要があるため、ユーザー、グループ、またはロールに対する AWS KMS アクセス許可は必要ありません。詳細については、「 AWS KMS デベロッパーガイド」の「キーポリシーの使用」を参照してください。

デフォルトの暗号化の場合、または AWS マネージド CMK (aws/xray) を選択した場合、アクセス許可は X-Ray APIs。`AWSXrayFullAccess` に含まれる `PutEncryptionConfig`にアクセスできるユーザーはすべて、暗号化の設定を変更することが可能です。ユーザーが暗号化キーを変更できないようにする場合は、`PutEncryptionConfig` を使用するためのアクセス許可を付与しないようにしてください。

AWS X-Ray アイデンティティベースのポリシーの例

デフォルトでは、ユーザーおよびロールには、X-Ray リソースを作成または変更するアクセス許可はありません。また、 AWS Management Console、 AWS CLI、または AWS API を使用してタスクを実行することはできません。IAM 管理者は、指定されたリソースで特定の API 操作を実行するための許可をユーザーとロールに付与する IAM ポリシーを作成する必要があります。続いて、管理者はそれらのアクセス許可が必要なユーザーまたはグループにそのポリシーをアタッチします。

JSON ポリシードキュメントのこれらの例を使用して、IAM アイデンティティベースのポリシーを作成する方法については、「IAM ユーザーガイド」の「JSON タブでのポリシーの作成」を参照してください。

トピック

- ポリシーに関するベストプラクティス
- X-Ray コンソールの使用
- ユーザーが自分の許可を表示できるようにする
- タグに基づいて X-Ray グループおよびサンプリングルールへのアクセスを管理する
- X-Ray の IAM マネージドポリシー
- AWS 管理ポリシーの X-Ray 更新
- IAM ポリシーでリソースを指定する

ポリシーに関するベストプラクティス

ID ベースのポリシーは、ユーザーのアカウントで誰が X-Ray リソースを作成、アクセス、削除できるかを決定します。これらのアクションを実行すると、 AWS アカウントに料金が発生する可能性があります。アイデンティティベースポリシーを作成したり編集したりする際には、以下のガイドラインと推奨事項に従ってください:

- AWS 管理ポリシーを開始し、最小特権のアクセス許可に移行する – ユーザーとワークロードにアクセス許可の付与を開始するには、多くの一般的なユースケースにアクセス許可を付与するAWS 管理ポリシーを使用します。これらは で使用できます AWS アカウント。ユースケースに固有の AWS カスタマー管理ポリシーを定義することで、アクセス許可をさらに減らすことをお勧めします。詳細については、「IAM ユーザーガイド」の「AWS マネージドポリシー」または「ジョブ機能のAWS マネージドポリシー」を参照してください。
- 最小特権を適用する – IAM ポリシーで許可を設定する場合は、タスクの実行に必要な許可のみを付与します。これを行うには、特定の条件下で特定のリソースに対して実行できるアクションを定

義します。これは、最小特権アクセス許可とも呼ばれています。IAM を使用して許可を適用する方法の詳細については、「IAM ユーザーガイド」の「IAM でのポリシーとアクセス許可」を参照してください。

- IAM ポリシーで条件を使用してアクセスをさらに制限する - ポリシーに条件を追加して、アクションやリソースへのアクセスを制限できます。例えば、ポリシー条件を記述して、すべてのリクエストを SSL を使用して送信するように指定できます。条件を使用して、サービスアクションが などの特定の を通じて使用されている場合に AWS のサービス、サービスアクションへのアクセスを許可することもできます AWS CloudFormation。詳細については、「IAM ユーザーガイド」の「IAM JSON ポリシー要素:条件」を参照してください。
- IAM Access Analyzer を使用して IAM ポリシーを検証し、安全で機能的な権限を確保する - IAM Access Analyzer は、新規および既存のポリシーを検証して、ポリシーが IAM ポリシー言語 (JSON) および IAM のベストプラクティスに準拠するようにします。IAM アクセスアナライザーは 100 を超えるポリシーチェックと実用的な推奨事項を提供し、安全で機能的なポリシーの作成をサポートします。詳細については、「IAM ユーザーガイド」の「IAM Access Analyzer でポリシーを検証する」を参照してください。
- 多要素認証 (MFA) を要求する – で IAM ユーザーまたはルートユーザーを必要とするシナリオがある場合は AWS アカウント、MFA をオンにしてセキュリティを強化します。API オペレーションが呼び出されるときに MFA を必須にするには、ポリシーに MFA 条件を追加します。詳細については、「IAM ユーザーガイド」の「MFA を使用した安全な API アクセス」を参照してください。

IAM でのベストプラクティスの詳細については、「IAM ユーザーガイド」の「IAM でのセキュリティのベストプラクティス」を参照してください。

X-Ray コンソールの使用

AWS X-Ray コンソールにアクセスするには、最小限のアクセス許可のセットが必要です。これらのアクセス許可により、 の X-Ray リソースの詳細を一覧表示および表示できます AWS アカウント。最小限必要な許可よりも制限が厳しいアイデンティティベースのポリシーを作成すると、そのポリシーを持つエンティティ (ユーザーまたはロール) に対してコンソールが意図したとおりに機能しません。

これらのエンティティが X-Ray コンソールを引き続き使用できるようにするには、エンティティに `AWSXRayReadOnlyAccess` AWS 管理ポリシーをアタッチします。このポリシーについては、X-Ray の IAM マネージドポリシーで詳しく説明されています。詳細については、「IAM ユーザーガイド」の「ユーザーへのアクセス許可の追加」を参照してください。

AWS CLI または AWS API のみを呼び出すユーザーには、最小限のコンソールアクセス許可を付与する必要はありません。代わりに、実行しようとしている API オペレーションに一致するアクションのみへのアクセスが許可されます。

ユーザーが自分の許可を表示できるようにする

この例では、ユーザーアイデンティティにアタッチされたインラインおよびマネージドポリシーの表示を IAM ユーザーに許可するポリシーの作成方法を示します。このポリシーには、コンソールで、または AWS CLI または AWS API を使用してプログラムでこのアクションを実行するアクセス許可が含まれています。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
```

```
}
```

タグに基づいて X-Ray グループおよびサンプリングルールへのアクセスを管理する

アイデンティティベースのポリシーの条件を使用して、タグに基づいて X-Ray グループやタグに基づいたサンプリングルールへのアクセスを制御できます。次の例のポリシーはタグ `stage:prod` または `stage:preprod` 付きのグループを作成、削除、または更新するアクセス権限をユーザーロールで拒否するために使用できます。X-Ray サンプリングルールとグループのタグ付けの詳細については、「X-Ray のサンプリングルールとグループのタグ付け」を参照してください。

サンプリングルールの作成を拒否するには、`aws:RequestTag` を使って作成リクエストの一部として渡すことができないタグを示します。サンプリングルールの更新または削除を拒否するには、`aws:ResourceTag` を使って、それらのリソースのタグに基づくアクションを拒否します。

これらのポリシーをアカウントのユーザーにアタッチする (または、単一のポリシーに結合してからポリシーをアタッチする) ことができます。ユーザーがグループまたはサンプリングルールに変更を加えるには、グループまたはサンプリングルールにタグ `stage=prepod` または `stage=prod` を付けないでください。条件キー名では大文字と小文字が区別されないため、条件タグキー `Stage` は `Stage` と `stage` の両方に一致します。条件ブロックの詳細については、IAM ユーザーガイドの「IAM JSON ポリシーの要素: 条件」を参照してください。

次のポリシーがアタッチされているロールを持つユーザーは、タグ `role:admin` を追加してリソースにアクセスしたり、`role:admin` に関連付けられたリソースからタグを削除したりすることはできません。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowAllXRay",
            "Effect": "Allow",
            "Action": "xray:*",
            "Resource": "*"
        },
        {
            "Sid": "DenyRequestTagAdmin",
            "Effect": "Deny",
            "Action": "xray:TagResource",
```

```
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:RequestTag/role": "admin"
                }
            }
        },
        {
            "Sid": "DenyResourceTagAdmin",
            "Effect": "Deny",
            "Action": "xray:UntagResource",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/role": "admin"
                }
            }
        }
    ]
}
```

X-Ray の IAM マネージドポリシー

アクセス許可を簡単に付与するために、IAM は各サービスのマネージドポリシーに対応しています。サービスは、新しい APIs をリリースするときに、これらの管理ポリシーを新しいアクセス許可で更新できます。 は、読み取り専用、書き込み専用、管理者のユースケース用の管理ポリシー AWS X-Ray を提供します。

- `AWSXrayReadOnlyAccess` – X-Ray コンソール AWS CLI、または AWS SDK を使用して、X-Ray API からトレースデータ、トレースマップ、インサイト、X-Ray 設定を取得するための読み取りアクセス許可。Observability Access Manager (OAM) `oam:ListSinks` と、CloudWatch のクロスアカウントオブザーバビリティの一環として、ソースアカウントから共有されたトレースをコンソールで表示できるようにする `oam:ListAttachedSinks` アクセス許可が含まれています。`BatchGetTraceSummaryById` および `GetDistinctTraceGraphs` API アクションは、コードによって呼び出されることを意図しておらず、 AWS CLI および AWS SDKs には含まれていません。

```
{
    "Version": "2012-10-17",
    "Statement": [
```

```
        {
            "Effect": "Allow",
            "Action": [
                "xray:GetSamplingRules",
                "xray:GetSamplingTargets",
                "xray:GetSamplingStatisticSummaries",
                "xray:BatchGetTraces",
                "xray:BatchGetTraceSummaryById",
                "xray:GetDistinctTraceGraphs",
                "xray:GetServiceGraph",
                "xray:GetTraceGraph",
                "xray:GetTraceSummaries",
                "xray:GetGroups",
                "xray:GetGroup",
                "xray:ListTagsForResource",
                "xray:ListResourcePolicies",
                "xray:GetTimeSeriesServiceStatistics",
                "xray:GetInsightSummaries",
                "xray:GetInsight",
                "xray:GetInsightEvents",
                "xray:GetInsightImpactGraph",
                "oam:ListSinks"
            ],
            "Resource": [
                "*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "oam:ListAttachedLinks"
            ],
            "Resource": "arn:aws:oam:*:*:sink/*"
        }

}
```

- `AWSXRayDaemonWriteAccess` – X-Ray デーモン AWS CLI、または AWS SDK を使用してセグメントドキュメントとテレメトリを X-Ray API にアップロードするための書き込みアクセス許可。サンプリングルールを取得してサンプリング結果を報告するための読み取りアクセス許可が含まれています。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "xray:PutTraceSegments",
                "xray:PutTelemetryRecords",
                "xray:GetSamplingRules",
                "xray:GetSamplingTargets",
                "xray:GetSamplingStatisticSummaries"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

- `AWSXrayCrossAccountSharingConfiguration` – アカウント間で X-Ray リソースを共有するための Observability Access Manager のリンクを作成、管理、表示するためのアクセス許可を付与します。ソースアカウントとモニタリングアカウント間の CloudWatch クロスアカウントオブザーバビリティを有効にするために使用されます。

  JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "xray:Link",
                "oam:ListLinks"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
```

```
                "oam:DeleteLink",
                "oam:GetLink",
                "oam:TagResource"
            ],
            "Resource": "arn:aws:oam:*:*:link/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "oam:CreateLink",
                "oam:UpdateLink"
            ],
            "Resource": [
                "arn:aws:oam:*:*:link/*",
                "arn:aws:oam:*:*:sink/*"
            ]
        }
    ]

}
```

- `AWSXrayFullAccess` – 読み取りアクセス許可、書き込みアクセス許可、および暗号化キー設定とサンプリングルールを指定するためのアクセス許可を含む、すべての X-Ray API を使用するためのアクセス許可。Observability Access Manager (OAM) `oam:ListSinks` と、CloudWatch のクロスアカウントオブザーバビリティの一環として、ソースアカウントから共有されたトレースをコンソールで表示できるようにする `oam:ListAttachedSinks` アクセス許可が含まれています。

  JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "xray:*",
                "oam:ListSinks"
            ],
            "Resource": [
                "*"
            ]
        },
        {
```

```
            "Effect": "Allow",
            "Action": [
                "oam:ListAttachedLinks"
            ],
            "Resource": "arn:aws:oam:*:*:sink/*"
        }
    ]
}
```

マネージドポリシーを IAM ユーザー、グループ、ロールに追加するには

1. [IAM コンソール] を開きます。
2. インスタンスプロファイル、IAM ユーザー、または IAM グループに関連付けられたロールを開きます。
3. [アクセス許可] で、管理ポリシーをアタッチします。

AWS 管理ポリシーの X-Ray 更新

このサービスがこれらの変更の追跡を開始してからの X-Ray の AWS マネージドポリシーの更新に関する詳細を表示します。このページの変更に関する自動通知については、X-Ray ドキュメントの履歴ページの RSS フィードをサブスクライブしてください。

変更	説明	日付
X-Ray 用の IAM マネージドポリシー – 新しいポリシー `AWSXrayCrossAccountSharingConfiguration` を追加、`AWSXrayReadOnlyAccess` と `AWSXrayFullAccess` を更新	X-Ray で Observability Access Manager (OAM) のアクセス許可 `oam:ListSinks` と `oam:ListAttachedSinks` をポリシーに追加し、CloudWatch のクロスアカウントオブザーバビリティの一環として、ソースアカウントから共有されたトレースをコンソールで表示できるようにしました。	2022 年 11 月 27 日

変更	説明	日付
X-Ray の IAM マネージドポリシー – AWSXrayReadOnlyAccess ポリシーの更新	X-Ray に API アクション ListResourcePolicies が追加されました。	2022 年 11 月 15 日
X-Ray コンソールの使用 – AWSXrayReadOnlyAccess ポリシーの更新	X-Ray に BatchGetTraceSummaryById と GetDistinctTraceGraphs の 2 つの新しい API アクションが追加されました。 これらのアクションは、コードで呼び出すものではありません。したがって、これらの API アクションは AWS CLI および AWS SDKs に含まれません。	2022 年 11 月 11 日

IAM ポリシーでリソースを指定する

IAM ポリシーを使用してリソースへのアクセスを制御できます。リソースレベルのアクセス許可をサポートするアクションの場合は、Amazon リソースネーム (ARN) を使用して、ポリシーが適用されるリソースを識別します。

X-Ray ポリシーではすべての IAM アクションを使用して、そのアクションを使用するアクセス許可をユーザーに付与または拒否できます。ただし、すべての X-Ray アクションが、アクションを実行することができるリソースを指定できる、リソースレベルのアクセス許可をサポートしているわけではありません。

リソースレベルの権限をサポートしていないアクションの場合、「*」をリソースとして使用する必要があります。

次の X-Ray アクションは、リソースレベルのアクセス許可をサポートします。

- CreateGroup
- GetGroup

- UpdateGroup
- DeleteGroup
- CreateSamplingRule
- UpdateSamplingRule
- DeleteSamplingRule

以下は、CreateGroup アクションのアイデンティティベースのアクセス許可ポリシーの例です。この例では、グループ名 local-users に関連する ARN を使用する一意の ID をワイルドカードとして使用します。グループが作成されたときに一意の ID が生成されるため、事前にポリシーで予測することはできません。GetGroup、UpdateGroup、または DeleteGroup を使用する場合、ワイルドカードとして、または ID を含む正確な ARN として定義できます。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "xray:CreateGroup"
            ],
            "Resource": [
                "arn:aws:xray:eu-west-1:123456789012:group/local-users/*"
            ]
        }
    ]
}
```

以下は、CreateSamplingRule アクションのアイデンティティベースのアクセス許可ポリシーの例です。

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
```

```
        {
            "Effect": "Allow",
            "Action": [
                "xray:CreateSamplingRule"
            ],
            "Resource": [
                "arn:aws:xray:eu-west-1:123456789012:sampling-rule/base-
scorekeep"
            ]
        }
    ]
}
```

> **Note**
>
> サンプリングルールの ARN は、名前によって定義されます。グループ ARN とは異なり、サンプリングルールには一意に生成された ID がありません。

AWS X-Ray ID とアクセスのトラブルシューティング

次の情報は、X-Ray と IAM の使用に伴って発生する可能性がある一般的な問題の診断や修復に役立ちます。

トピック

- X-Ray でアクションを実行する権限がない
- iam:PassRole を実行する認可がない
- 管理者として X-Ray へのアクセスを他のユーザーに許可したい
- 自分の 以外のユーザーに X-Ray リソース AWS アカウント へのアクセスを許可したい

X-Ray でアクションを実行する権限がない

でアクションを実行する権限がないと AWS Management Console 通知された場合は、管理者に連絡してサポートを依頼する必要があります。管理者とは、サインイン認証情報を提供した担当者です。

次の例のエラーは、`mateojackson` ユーザーがコンソールを使用してサンプリングルールの詳細を表示する際に、`xray:GetSamplingRules` アクセス許可を持っていない場合に発生します。

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to
 perform: xray:GetSamplingRules on resource: arn:${Partition}:xray:${Region}:
${Account}:sampling-rule/${SamplingRuleName}
```

この場合、Mateo は管理者に依頼し、`xray:GetSamplingRules` アクションを使用してサンプリングルールリソースにアクセスできるようにポリシーを更新してもらいます。

iam:PassRole を実行する認可がない

`iam:PassRole` アクションを実行する権限がないというエラーが表示された場合は、ポリシーを更新して X-Ray にロールを渡すことができるようにする必要があります。

一部の AWS のサービス では、新しいサービスロールまたはサービスにリンクされたロールを作成する代わりに、そのサービスに既存のロールを渡すことができます。そのためには、サービスにロールを渡す権限が必要です。

以下の例のエラーは、`marymajor` という IAM ユーザーがコンソールを使用して X-Ray でアクションを実行しようする場合に発生します。ただし、このアクションをサービスが実行するには、サービスロールから付与された権限が必要です。メアリーには、ロールをサービスに渡す許可がありません。

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

この場合、Mary のポリシーを更新してメアリーに `iam:PassRole` アクションの実行を許可する必要があります。

サポートが必要な場合は、 AWS 管理者にお問い合わせください。サインイン認証情報を提供した担当者が管理者です。

管理者として X-Ray へのアクセスを他のユーザーに許可したい

他のユーザーが X-Ray にアクセスすることを許可するには、アクセスが必要なユーザーまたはアプリケーションにアクセス許可を付与する必要があります。 AWS IAM Identity Center を使用してユーザーとアプリケーションを管理する場合は、アクセスレベルを定義するアクセス許可セットをユーザーまたはグループに割り当てます。アクセス許可セットは、ユーザーまたはアプリケーションに関連付けられている IAM ロールに自動的に IAM ポリシーを作成して割り当てます。詳細については、「AWS IAM Identity Center ユーザーガイド」の「アクセス許可セット」を参照してください。

IAM アイデンティティセンターを使用していない場合は、アクセスを必要としているユーザーまたはアプリケーションの IAM エンティティ (ユーザーまたはロール) を作成する必要があります。次に、X-Ray の適切なアクセス許可を付与するポリシーを、そのエンティティにアタッチする必要があります アクセス許可が付与されたら、ユーザーまたはアプリケーション開発者に認証情報を提供します。これらの認証情報を使用して AWSにアクセスします。IAM ユーザー、グループ、ポリシー、アクセス許可の作成の詳細については、「IAM ユーザーガイド」の「IAM アイデンティティ」と「IAM のポリシーとアクセス許可」を参照してください。

自分の 以外のユーザーに X-Ray リソース AWS アカウント へのアクセスを許可したい

他のアカウントのユーザーや組織外の人が、リソースにアクセスするために使用できるロールを作成できます。ロールの引き受けを委託するユーザーを指定できます。リソースベースのポリシーまたはアクセスコントロールリスト (ACL) をサポートするサービスの場合、それらのポリシーを使用して、リソースへのアクセスを付与できます。

詳細については、以下を参照してください:

- X-Ray がこれらの機能をサポートしているかどうかを確認するには、「AWS X-Ray が IAM と連携する方法」を参照してください。
- 所有 AWS アカウント している のリソースへのアクセスを提供する方法については、IAM ユーザーガイドの「所有 AWS アカウント している別の の IAM ユーザーへのアクセスを提供する」を参照してください。
- リソースへのアクセスをサードパーティーに提供する方法については AWS アカウント、IAM ユーザーガイドの「サードパーティー AWS アカウント が所有する へのアクセスを提供する」を参照してください。
- ID フェデレーションを介してアクセスを提供する方法については、「IAM ユーザーガイド」の「外部で認証されたユーザー (ID フェデレーション) へのアクセスの許可」を参照してください。
- クロスアカウントアクセスにおけるロールとリソースベースのポリシーの使用方法の違いについては、「IAM ユーザーガイド」の「IAM でのクロスアカウントのリソースへのアクセス」を参照してください。

でのログ記録とモニタリング AWS X-Ray

モニタリングは、 AWS ソリューションの信頼性、可用性、および性能を維持するうえで重要な部分です。マルチポイント障害が発生した場合は、その障害をより簡単にデバッグできるように、 AWS

ソリューションのすべての部分からモニタリングデータを収集する必要があります。 には、X-Ray リソースをモニタリングし、潜在的なインシデントに対応するための複数のツール AWS が用意されています。

AWS CloudTrail ログ

AWS X-Ray は と統合 AWS CloudTrail して、ユーザー、ロール、または のサービスによって実行された API アクションを AWS X-Ray に記録します。CloudTrailを使用して、X-Ray API依頼を即時にモニタリングし、Amazon S3、Amazon CloudWatch Logs、Amazon CloudWatch Events にログを保存することができます。詳細については、「を使用した X-Ray API コールのログ記録 AWS CloudTrail」を参照してください。

AWS Config 追跡

AWS X-Ray は と統合 AWS Config して、X-Ray 暗号化リソースに加えられた設定変更を記録します。 AWS Config を使用して、X-Ray 暗号化リソースのインベントリ、X-Ray 設定履歴の監査、リソースの変更に基づいた通知の送信を行うことができます。詳細については、「を使用した X-Ray 暗号化設定の変更の追跡 AWS Config」を参照してください。

Amazon CloudWatchのモニタリング

X-Ray SDK for Java を使用して、収集した X-Rayセグメントからサンプリングされずに Amazon CloudWatch メトリクスを公開することができます。これらのメトリクスは、セグメントの開始時間と終了時間、さらにエラー、障害およびスロットル状態を示すフラグから取得されます。これらの追跡メトリクスを使用して、サブセグメント内の再試行と依存関係の問題を公開します。詳細については、「AWS X-Ray X-Ray SDK for Java の メトリクス」を参照してください。

のコンプライアンス検証 AWS X-Ray

AWS のサービス が特定のコンプライアンスプログラムの範囲内にあるかどうかを確認するには、AWS のサービス「コンプライアンスプログラムによる範囲内」を参照して、関心のあるコンプライアンスプログラムを選択します。一般的な情報については、AWS 「コンプライアンスプログラム」を参照してください。

を使用して、サードパーティーの監査レポートをダウンロードできます AWS Artifact。詳細については、「Downloading Reports in AWS Artifact」を参照してください。

を使用する際のお客様のコンプライアンス責任 AWS のサービス は、お客様のデータの機密性、貴社のコンプライアンス目的、適用される法律および規制によって決まります。 は、コンプライアンスに役立つ以下のリソース AWS を提供します。

- セキュリティのコンプライアンスとガバナンス – これらのソリューション実装ガイドでは、アーキテクチャ上の考慮事項について説明し、セキュリティとコンプライアンスの機能をデプロイする手順を示します。
- HIPAA 対応サービスのリファレンス – HIPAA 対応サービスの一覧が提供されています。すべて AWS のサービス HIPAA の対象となるわけではありません。
- AWS コンプライアンスリソース – このワークブックとガイドのコレクションは、お客様の業界と地域に適用される場合があります。
- AWS カスタマーコンプライアンスガイド – コンプライアンスの観点から責任共有モデルを理解します。このガイドでは、複数のフレームワーク (米国国立標準技術研究所 (NIST)、Payment Card Industry Security Standards Council (PCI)、国際標準化機構 (ISO) など) にわたるセキュリティコントロールを保護し、そのガイダンスに AWS のサービス マッピングするためのベストプラクティスをまとめています。
- 「 デベロッパーガイド」の「ルールによるリソースの評価」 – この AWS Config サービスは、リソース設定が内部プラクティス、業界ガイドライン、および規制にどの程度準拠しているかを評価します。 AWS Config
- AWS Security Hub – これにより AWS のサービス 、 内のセキュリティ状態を包括的に把握できます AWS。Security Hub では、セキュリティコントロールを使用して AWS リソースを評価し、セキュリティ業界標準とベストプラクティスに対するコンプライアンスをチェックします。サポートされているサービスとコントロールの一覧については、Security Hub のコントロールリファレンスを参照してください。
- Amazon GuardDuty – 環境をモニタリングして AWS アカウント、疑わしいアクティビティや悪意のあるアクティビティがないか調べることで、、ワークロード、コンテナ、データに対する潜在的な脅威 AWS のサービス を検出します。GuardDuty を使用すると、特定のコンプライアンスフレームワークで義務付けられている侵入検知要件を満たすことで、PCI DSS などのさまざまなコンプライアンス要件に対応できます。
- AWS Audit Manager – これにより AWS のサービス 、 AWS 使用状況を継続的に監査し、リスクの管理方法と規制や業界標準への準拠を簡素化できます。

の耐障害性 AWS X-Ray

AWS グローバルインフラストラクチャは、 AWS リージョン およびアベイラビリティーゾーンを中心に構築されています。 は、低レイテンシー、高スループット、冗長性の高いネットワークで接続された、物理的に分離および分離された複数のアベイラビリティーゾーン AWS リージョン を提供します。アベイラビリティーゾーンでは、アベイラビリティーゾーン間で中断せずに、自動的にフェ

イルオーバーするアプリケーションとデータベースを設計および運用することができます。アベイラビリティーゾーンは、従来の単一または複数のデータセンターインフラストラクチャよりも可用性、耐障害性、およびスケーラビリティが優れています。

AWS リージョン およびアベイラビリティーゾーンの詳細については、AWS 「 グローバルインフラストラクチャ」を参照してください。

のインフラストラクチャセキュリティ AWS X-Ray

マネージドサービスである AWS X-Ray は、 AWS グローバルネットワークセキュリティで保護されています。 AWS セキュリティサービスと がインフラストラクチャ AWS を保護する方法については、AWS 「 クラウドセキュリティ」を参照してください。インフラストラクチャセキュリティのベストプラクティスを使用して AWS 環境を設計するには、「Security Pillar AWS Well-Architected Framework」の「Infrastructure Protection」を参照してください。

AWS 公開された API コールを使用して、ネットワーク経由で X-Ray にアクセスします。クライアントは以下をサポートする必要があります。

- Transport Layer Security (TLS)。TLS 1.2 が必須で、TLS 1.3 をお勧めします。
- DHE (楕円ディフィー・ヘルマン鍵共有) や ECDHE (楕円曲線ディフィー・ヘルマン鍵共有) などの完全前方秘匿性 (PFS) による暗号スイート。これらのモードはJava 7 以降など、ほとんどの最新システムでサポートされています。

また、リクエストにはアクセスキー ID と、IAM プリンシパルに関連付けられているシークレットアクセスキーを使用して署名する必要があります。または、AWS Security Token Service (AWS STS) を使用して、一時的なセキュリティ認証情報を生成し、リクエストに署名することもできます。

VPC エンドポイント AWS X-Ray での の使用

Amazon Virtual Private Cloud (Amazon VPC) を使用して AWS リソースをホストする場合は、VPC と X-Ray の間にプライベート接続を確立できます。これによりAmazon VPC内のリソースが公衆インターネットを経由せずに、X-Rayサービスと通信できるようになります。

Amazon VPC AWS のサービス は、定義した仮想ネットワークで AWS リソースを起動するために使用できる です。VPC を使用すると、IP アドレス範囲、サブネット、ルートテーブル、ネットワークゲートウェイなどのネットワーク設定を制御できます。VPCをX-Rayに接続するには、インターフェイスのVPC評価項目を定義します。この評価項目は、インターネットゲートウェイ、ネットワーク

アドレス変換(NAT)の場合、またはVPN接続を必要とせず、X-Rayへの信頼性が高く拡張性のある接続性を提供します。詳細については、『Amazon VPC ユーザーガイド』の「Amazon VPCとは何か」を参照してください。

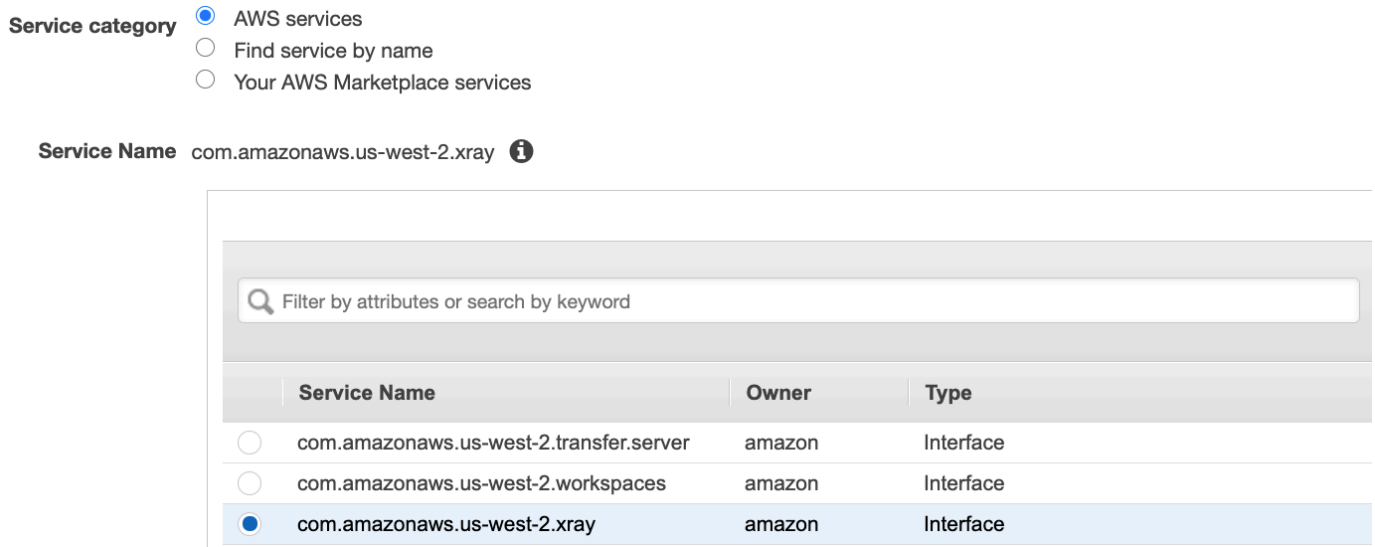
インターフェイス VPC エンドポイントは、プライベート IP アドレスを持つ Elastic Network Interface AWS のサービス を使用して 間のプライベート通信を可能にする AWS テクノロジーである AWS PrivateLink を利用しています。詳細については、ブログ記事の「New – AWS PrivateLink for AWS のサービス blog」および「Amazon VPC ユーザーガイド」の「開始方法https://docs.aws.amazon.com/vpc/latest/userguide/GetStarted.html」を参照してください。

選択した で X-Ray の VPC エンドポイントを作成できるようにするには AWS リージョン、「」を参照してくださいサポートされている地域。

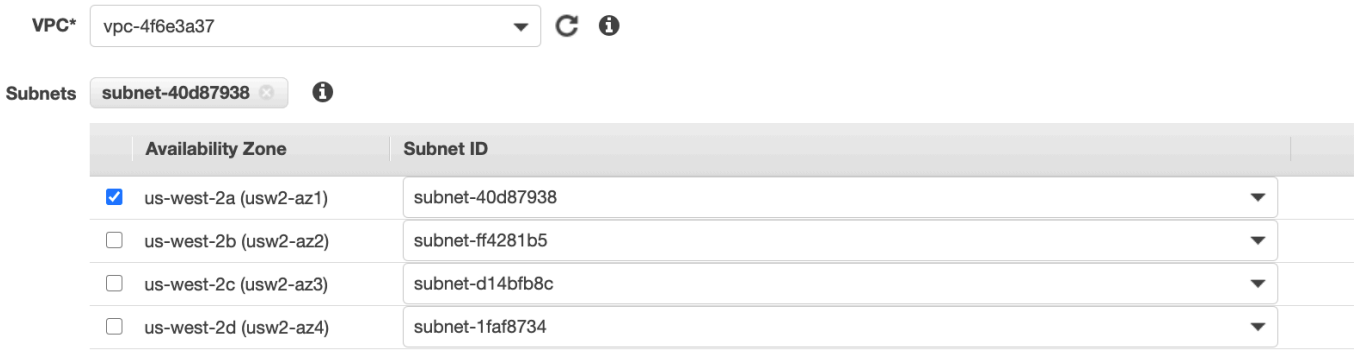
X-Ray用のVPC評価項目の作成

VPCでX-Rayの使用を開始するには、X-Rey用のインターフェイスVPC評価項目を作成します。

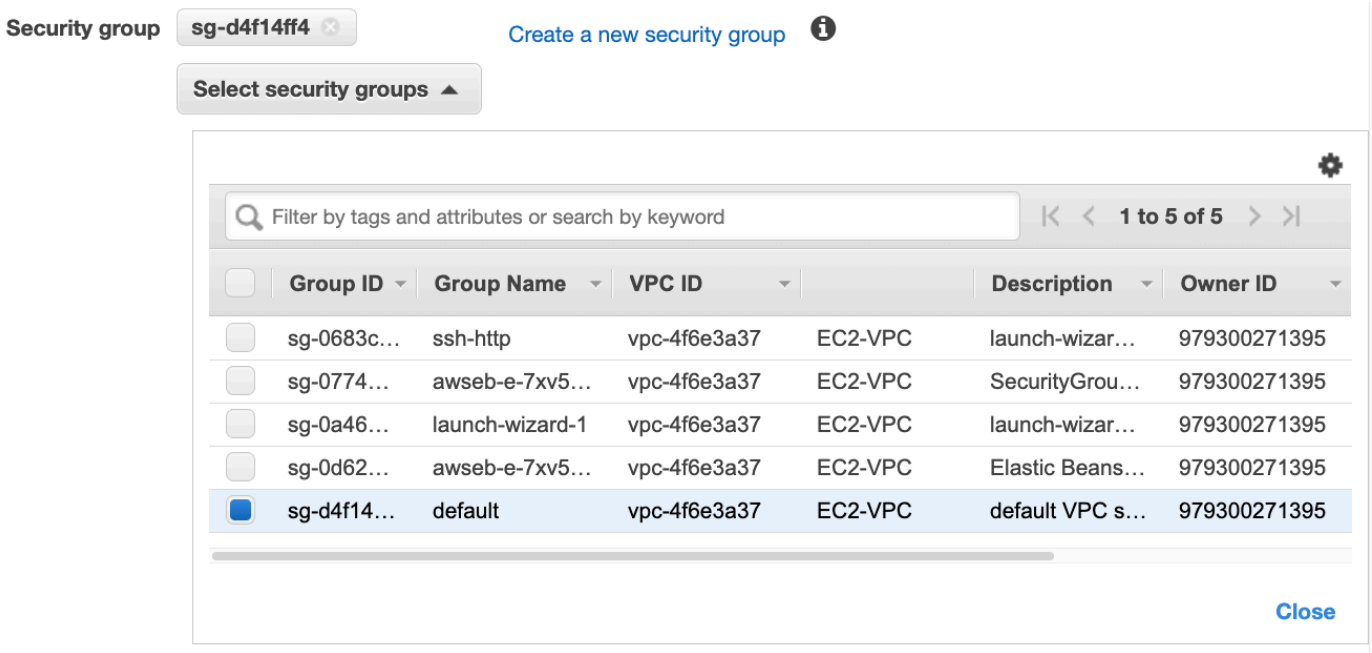
1. (https://console.aws.amazon.com/vpc/)Amazon VPC制御盤を開きます。
2. ナビゲーションペイン内の、「評価項目」に移動して評価項目の作成を選択します。
3. AWS X-Ray サービスの名前 を検索して選択しますcom.amazonaws.*region*.xray。



4. インターフェイス評価項目を使用して目的のVPCを選択し、VPC内のサブネットを選択します。選択したサブネットに評価項目のネットワークインターフェイスが作成されます。サービスでサポートされている異なる可用性の範囲に複数のサブネットを指定することで、可用性の範囲の障害に対応したインターフェイス評価項目を、より確実に回復できます。この操作を行うと、指定した各サブネットでインターフェイスのネットワークインターフェイスが作成されます。



5. (オプション)初期設定の情報システムのDNSホスト名を使用してX-Rayのリクエストを行うには、評価項目のDNSプライベート情報システムが初期設定で有効になっています。無効にすることも可能です。
6. 評価項目ネットワークインターフェイスに関連付けるセキュリティグループを指定します。



7. （オプション）X-Rayサービスへのアクセス許可を制御するカスタムポリシーを指定します。初期設定では、数多くのアクセスが許可されています。

X-RayVPCの情報システム評価項目へのアクセスの制御

VPC評価項目ポリシーは、評価項目の作成時または変更時に評価項目に加える国際機械技術者協会（IAM）のリソースポリシーです。評価項目の作成時にポリシーを加えない場合、サービスへの数多くのアクセスを許可する初期設定のポリシーがAmazon VPCによって自動的に接続されます。評価項目ポリシーは、IAMユーザーポリシーやサービス固有のポリシーを上書き、または置き換えたりする

ものではありません。これは、評価項目から指定されたサービスへのアクセスを制御するための別のポリシーです。評価項目のポリシーは、JSON形式で記載する必要があります。詳細については、「Amazon VPCユーザーガイド」の「VPC評価項目によるサービスのアクセス制御」を参照してください。

VPC評価項目ポリシーを使用すると、さまざまなX-Ray行為に対するアクセス許可を制御することができます。たとえば、putTraceSegmentのみを許可し、その他のすべての行為を拒否するポリシーを作成することができます。これにより、VPC内のワークロードおよびサービスは、X-Rayにトレースデータのみを送信し、データの取得、暗号化設定の変更、グループの作成/更新など、その他の行為を拒否するように制限されます。

X-Rayの評価項目ポリシーの例を以下に示します。このポリシーは、VPCを介してX-Rayに接続するユーザーに対して、セグメントデータをX-Reyに送信することを許可し、また他のX-Rey行為を実行することを禁止します。

```
{"Statement": [
    {"Sid": "Allow PutTraceSegments",
      "Principal": "*",
      "Action": [
        "xray:PutTraceSegments"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

X-RayのVPC評価項目ポリシーを編集するには

1. Amazon VPCコンソール(https://console.aws.amazon.com/vpc/)を開きます。
2. ナビゲーションペインで、[評価項目]を選択します。
3. X-Ray用の評価項目を、まだ作成していない場合は、X-Ray用のVPC評価項目の作成の手順に従います。
4. [com.amazonaws.##.xray]評価項目を選択し、[ポリシー]タブを選択します。
5. [ポリシーの編集]を選択し、変更を加えます。

サポートされている地域

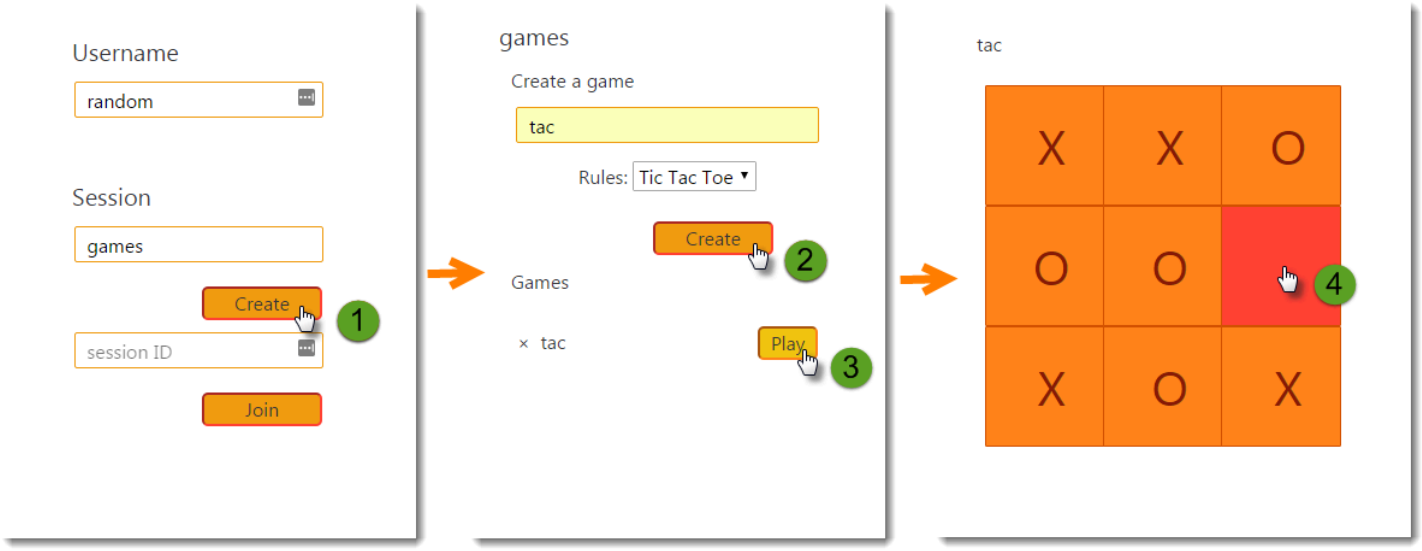
X-Ray は現在、以下の VPC エンドポイントをサポートしています AWS リージョン。

- 米国東部(オハイオ)
- 米国東部 (バージニア北部)
- 米国西部 (北カリフォルニア)
- 米国西部 (オレゴン)
- アフリカ (ケープタウン)
- アジアパシフィック (香港)
- アジアパシフィック (ムンバイ)
- アジアパシフィック (大阪)
- アジアパシフィック (ソウル)
- アジアパシフィック (シンガポール)
- アジアパシフィック (シドニー)
- アジアパシフィック (東京)
- カナダ (中部)
- 欧州 (フランクフルト)
- 欧州 (アイルランド)
- 欧州 (ロンドン)
- 欧州 (ミラノ)
- ヨーロッパ (パリ)
- ヨーロッパ (ストックホルム)
- 中東 (バーレーン)
- 南米 (サンパウロ)
- AWS GovCloud (米国東部)
- AWS GovCloud (米国西部)

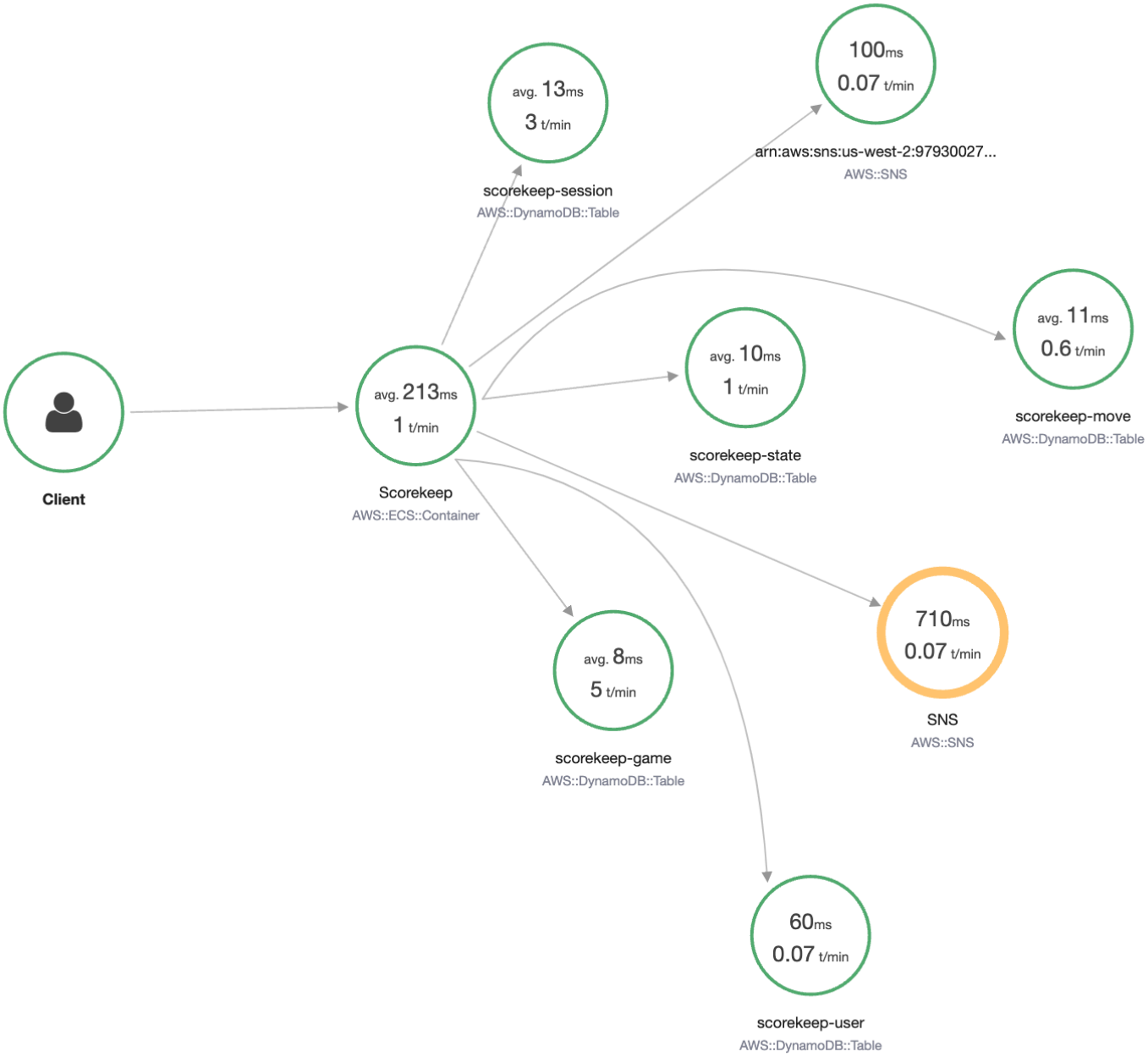
AWS X-Ray サンプルアプリケーション

GitHub で利用可能な AWS X-Ray eb-java-scorekeep サンプルアプリケーションは、 AWS X-Ray SDK を使用して受信 HTTP コール、DynamoDB SDK クライアント、および HTTP クライアントを計測する方法を示しています。サンプルアプリケーションは AWS CloudFormation を使用して DynamoDB テーブルを作成し、インスタンスで Java コードをコンパイルし、追加の設定なしで X-Ray デーモンを実行します。

Scorekeep チュートリアルを参照して、 AWS Management Console または を使用して、計測されたサンプルアプリケーションのインストールと使用を開始します AWS CLI。



サンプルには、フロントエンドのウェブアプリ、それが呼び出す API、データの保存に使用する DynamoDB テーブルが含まれています。フィルター、プラグイン、および計測された AWS SDK クライアントを使用した基本的な計測は、プロジェクトの`xray-gettingstarted`ブランチに表示されます。これは、入門ガイドチュートリアルでデプロイするブランチです。このブランチには基本情報しか含まれていないので、`master` ブランチと比較すると、基本をすばやく理解できます。



同じアプリケーションで、次のファイルの基本計測も説明します。

- HTTP リクエストフィルタ–[WebConfig.java]
- AWS SDK クライアントの計測 – [build.gradle]

アプリケーションの `xray` ブランチには、HTTPClient、注釈、SQL クエリ、カスタムサブセグメント、実装された AWS Lambda 関数、および実装された初期化コードとスクリプトの使用が含まれています。

ユーザーログインとブラウザでの AWS SDK for JavaScript の使用をサポートするために、`xray-cognito`ブランチはユーザーの認証と認可をサポートする Amazon Cognito を追加します。また、Amazon Cognitoから認証情報を取得すると、ウェブアプリケーションはトレースデータをX-Rayに送信してクライアントの観点からリクエスト情報を記録します。ブラウザクライアントは、トレースマップ上に独自のノードとして表示され、ユーザーが表示しているページの URL やユーザーの ID などの追加情報を記録します。

最後に、 `xray-worker`ブランチは、個別に実行され、Amazon SQS キューの項目を処理する、実装された Python Lambda 関数を追加します。Scorekeep は、ゲームが終了するたびに項目をキューに追加します。CloudWatch Events によりトリガーされるLambda ワーカーは、数分ごとにキューから項目を取得し、それらの項目を処理して分析のためにゲームレコードをAmazon S3 に保存します。

トピック

- Scorekeep サンプルアプリケーションの開始方法
- AWS SDK クライアントの手動計測
- 追加のサブセグメントを作成する
- 注釈、メタデータ、およびユーザー ID を記録する
- 送信 HTTP 呼び出しの計測
- PostgreSQL データベースに対する呼び出しの計測
- AWS Lambda 関数の計測
- スタートアップコードの作成
- 実装スクリプト
- ウェブアプリケーションクライアントの実装
- 実装されたクライアントをワーカースレッドで使用する

Scorekeep サンプルアプリケーションの開始方法

このチュートリアルでは、Scorekeep サンプルアプリケーションの `xray-gettingstarted`ブランチを使用します。このブランチは、 AWS CloudFormation を使用して、Amazon ECS でサンプルア

プリケーションと X-Ray デーモンを実行するリソースを作成および設定します。アプリケーションは Spring フレームワークを使用して JSON ウェブ API を実装し、 AWS SDK for Java を使用してデータを Amazon DynamoDB に保持します。アプリケーションのサーブレットフィルターは、アプリケーションによって処理されるすべての受信リクエストを計測し、 AWS SDK クライアントのリクエストハンドラーは DynamoDB へのダウンストリーム呼び出しを計測します。

このチュートリアルは、 AWS Management Console または を使用して実行できます AWS CLI。

セクション

前提条件

このチュートリアルでは AWS CloudFormation 、 を使用して、サンプルアプリケーションと X-Ray デーモンを実行するリソースを作成および設定します。チュートリアルでインストールと実行をする前提条件として以下が必要です。

1. アクセス許可が限定された IAM ユーザーを使用する場合は、IAM コンソールに次のユーザーポリシーを追加してください。

   - `AWSCloudFormationFullAccess` – CloudFormation にアクセスして使用
   - `AmazonS3FullAccess` – を使用してテンプレートファイルを CloudFormation にアップロードするには AWS Management Console
   - `IAMFullAccess` – Amazon ECS インスタンスロールと Amazon EC2 インスタンスロールを作成
   - `AmazonEC2FullAccess` – Amazon EC2 リソースを作成
   - `AmazonDynamoDBFullAccess` – DynamoDB テーブルを作成

- `AmazonECS_FullAccess` – Amazon ECS リソースを作成
- `AmazonSNSFullAccess` – Amazon SNS トピックを作成
- `AWSXrayReadOnlyAccess` - X-Ray コンソールでトレースマップとトレースを表示するアクセス許可

2. を使用してチュートリアルを実行するには AWS CLI、CLI バージョン 2.7.9 以降をインストールし、前のステップの ユーザーを使用して CLI を設定します。ユーザー AWS CLI で を設定するときは、リージョンが設定されていることを確認してください。リージョンが設定されていない場合は、すべての CLI コマンドに `--region` *`AWS-REGION`* を追加する必要があります。
3. サンプルアプリケーションリポジトリを複製ために、Git がインストールされていることを確認してください。
4. 次のコード例を使用して、Scorekeep リポジトリの `xray-gettingstarted` ブランチをクローンします。

```
git clone https://github.com/aws-samples/eb-java-scorekeep.git xray-scorekeep -b
 xray-gettingstarted
```

CloudFormation を使用した Scorekeep アプリケーションのインストール

AWS Management Console

を使用してサンプルアプリケーションをインストールする AWS Management Console

1. CloudFormation コンソールを開きます。
2. [スタックの作成] を選択し、ドロップダウンメニューから [新しいリソースを使用] を選択します。
3. [テンプレートの指定] セクションで、[テンプレートファイルのアップロード] を選択します。
4. [ファイルの選択] を選択し、git リポジトリをクローンしたときに作成された `xray-scorekeep/cloudformation` フォルダーに移動して、`cf-resources.yaml` ファイルを選択します。
5. [次へ] を選択して続行します。
6. [スタック名] テキストボックスに `scorekeep` と入力し、ページ下部の [次へ] を選択して続行します。このチュートリアルのこれ以降の部分ではスタックの名前を `scorekeep` とします。

7. [スタックオプションの設定] ページの一番下までスクロールし、[次へ] を選択して続行します。
8. [レビュー] ページの一番下までスクロールし、[CloudFormation によってカスタム名がついた IAM リソースが作成される場合があることを承認します] チェックボックスをオンにし、[スタックの作成] を選択します。
9. CloudFormation スタックが作成中になります。スタックのステータスは約 5 分間 CREATE_COMPLETE で、その後 CREATE_IN_PROGRESS に変わります。ステータスは定期的に更新されます。ページを更新して再表示することもできます。

AWS CLI

を使用してサンプルアプリケーションをインストールする AWS CLI

1. このチュートリアルの前の部分でクローンを作成した xray-scorekeep リポジトリの cloudformation フォルダーに移動します。

```
cd xray-scorekeep/cloudformation/
```

2. 次の AWS CLI コマンドを入力して CloudFormation スタックを作成します。

```
aws cloudformation create-stack --stack-name scorekeep --capabilities
 "CAPABILITY_NAMED_IAM" --template-body file://cf-resources.yaml
```

3. CloudFormation スタックのステータスが CREATE_COMPLETE になるまで約 5 分待ってください。ステータスを確認するには、次の AWS CLI コマンドを使用します。

```
aws cloudformation describe-stacks --stack-name scorekeep --query
 "Stacks[0].StackStatus"
```

トレースデータの生成

サンプルアプリケーションには、フロントエンドのウェブアプリケーションが含まれています。ウェブアプリケーションを使用して API へのトラフィックを生成し、トレースデータを X-Ray に送信します。まず、 AWS Management Console または AWS CLIを使用してウェブアプリの URL を取得します。

AWS Management Console

を使用してアプリケーション URL を検索する AWS Management Console

1. CloudFormation コンソールを開きます。
2. リストから scorekeep スタックを選択します。
3. scorekeep スタックページの [出力] タブを選択し、LoadBalancerUrl URL リンクを選択してウェブアプリケーションを開きます。

AWS CLI

を使用してアプリケーション URL を検索する AWS CLI

1. 次のコマンドを使用して、ウェブアプリケーションの URL を表示します。

```
aws cloudformation describe-stacks --stack-name scorekeep --query
 "Stacks[0].Outputs[0].OutputValue"
```

2. この URL をコピーしてブラウザで開き、Scorekeep ウェブアプリケーションを表示します。

ウェブアプリケーションを使用してトレースデータを生成する

1. [Create] を選択して、ユーザーとセッションを作成します。
2. [game name] を入力し、[Rules] を [Tic Tac Toe] に設定したら、[Create] を選択して、ゲームを作成します。
3. [Play] を選択してゲームを開始します。
4. ゲームの状態を移行および変更するには、タイルを選択します。

これらの各ステップで、API への HTTP リクエスト、および DynamoDB へのダウンストリーム呼び出しが生成され、ユーザー、セッション、ゲーム、移動、および状態データが読み書きされます。

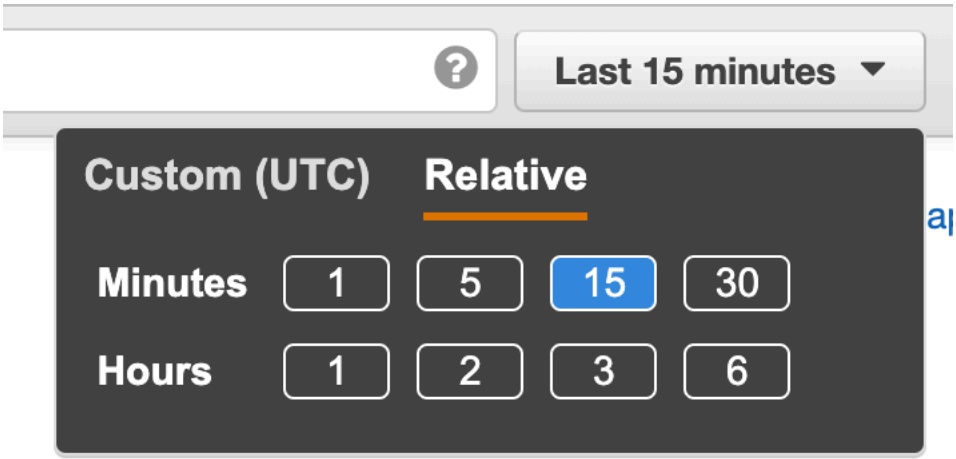
でトレースマップを表示する AWS Management Console

X-Ray コンソールと CloudWatch コンソールで、サンプルアプリケーションによって生成されたトレースマップとトレースを確認できます。

X-Ray console

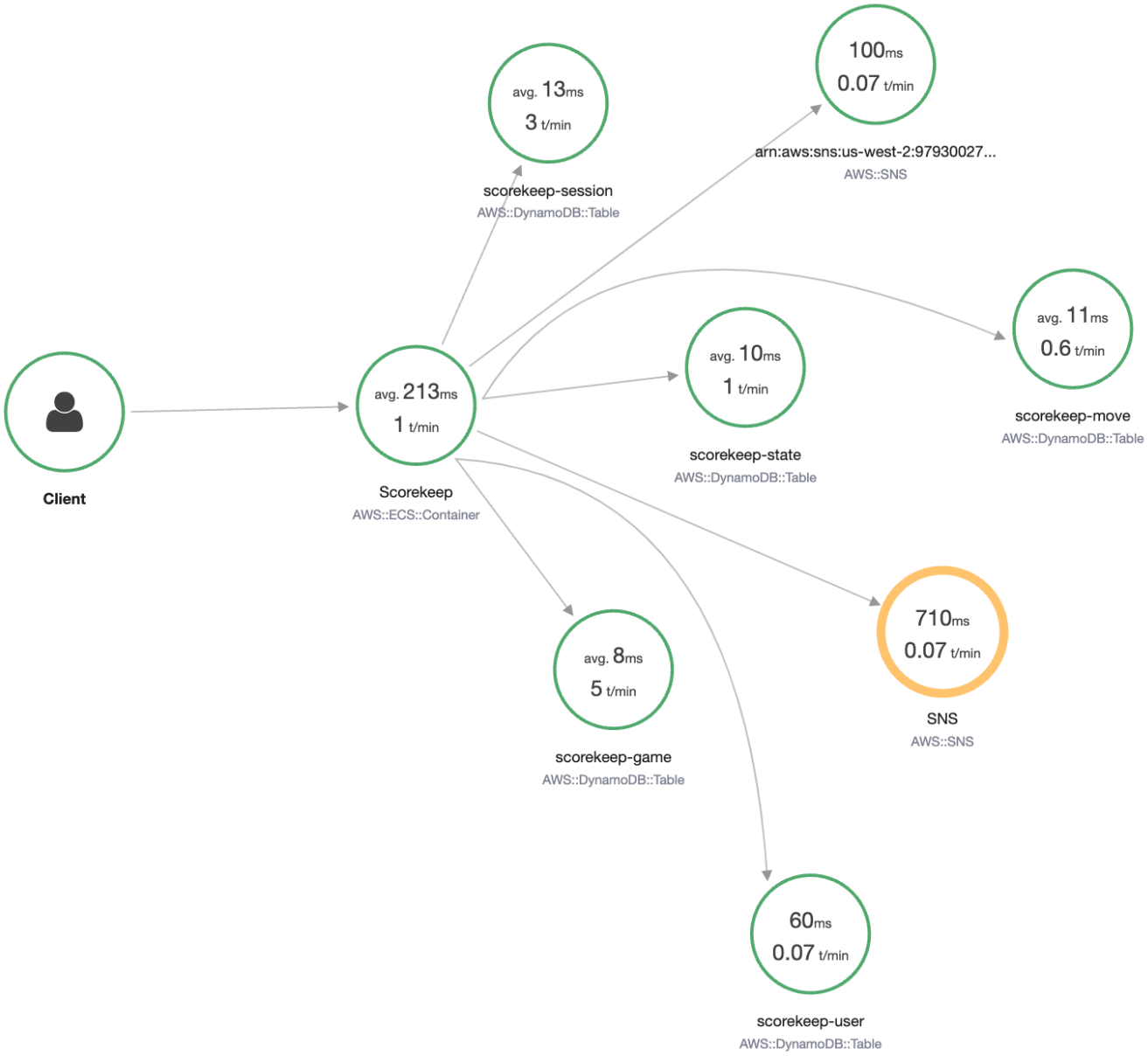
X-Ray コンソールを使用する

1. X-Ray コンソールのトレースマップページを開きます。
2. コンソールには、X-Ray によってアプリケーションから送信されたトレースデータから生成されたサービスグラフの表現が表示されます。ウェブアプリケーションを最初に起動した時点からのすべてのトレースが表示されるように、必要に応じてトレースマップの期間を調整してください。



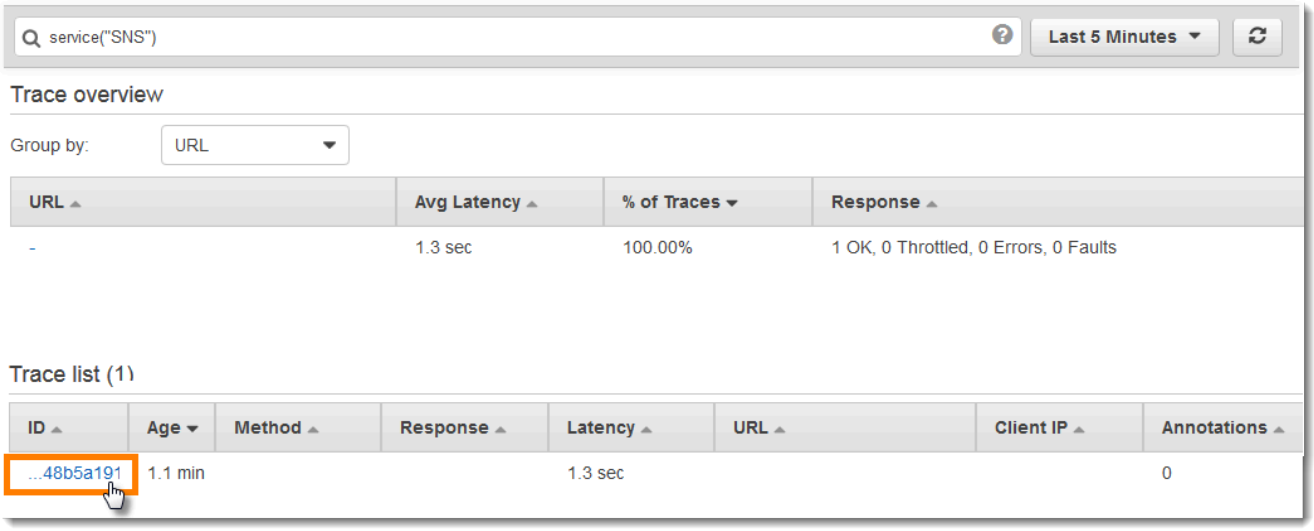
トレースマップに表示されるのは、ウェブアプリケーションクライアント、Amazon ECS で実行されている API、アプリケーションで使用される各 DynamoDB テーブルです。アプリケーションに対するすべてのリクエストは、1 秒あたりのリクエストの設定可能な最大数まで、API にヒットした際にトレースされ、ダウンストリームサービスへのリクエストを生成して、完了します。

サービスグラフの任意のノードを選択すると、そのノードに対してトラフィックを生成したリクエストのトレースを表示できます。現在、Amazon SNS ノードは黄色になっています。理由を調べるために掘り下げます。

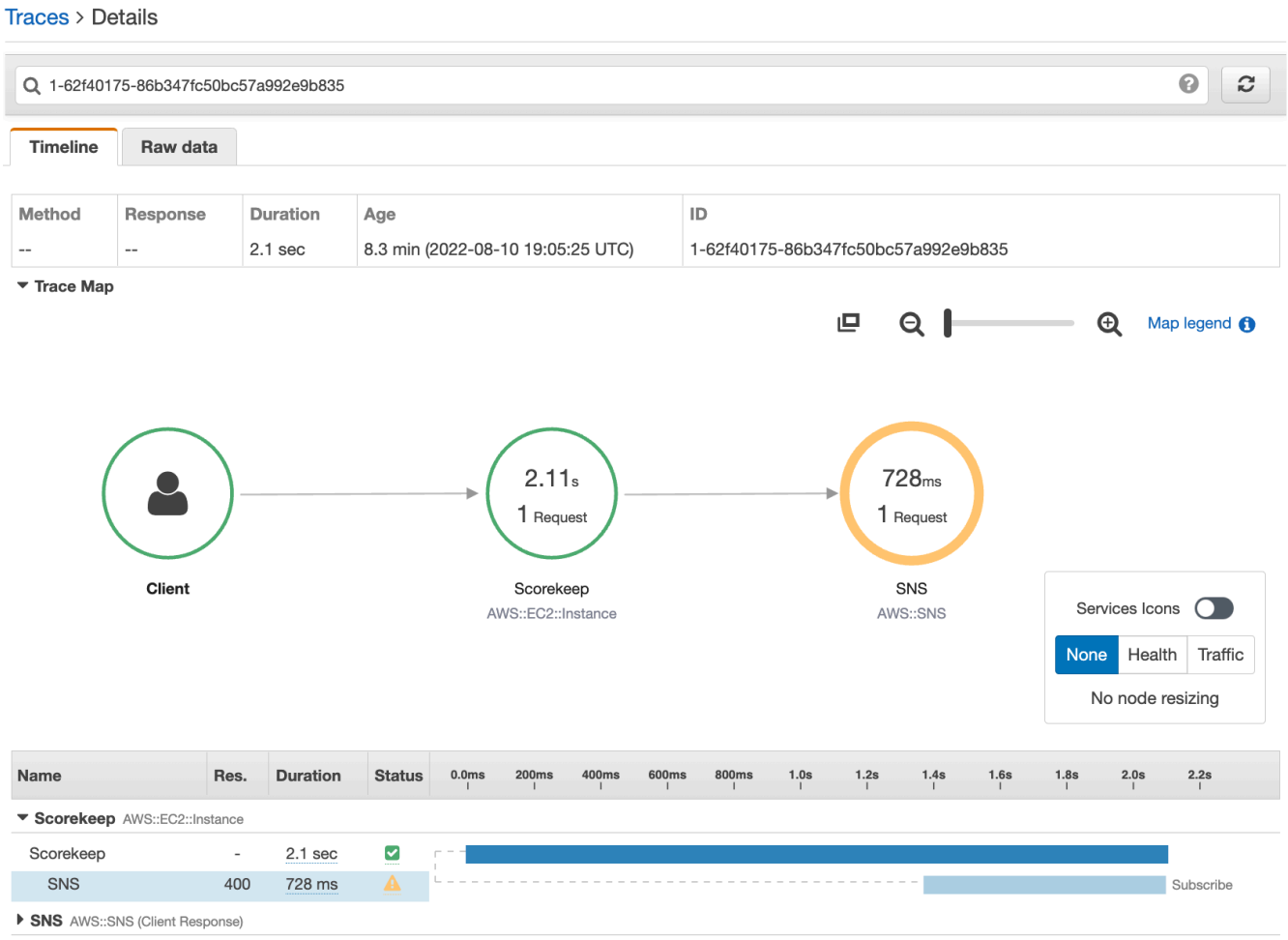


エラーの原因を見つけるには

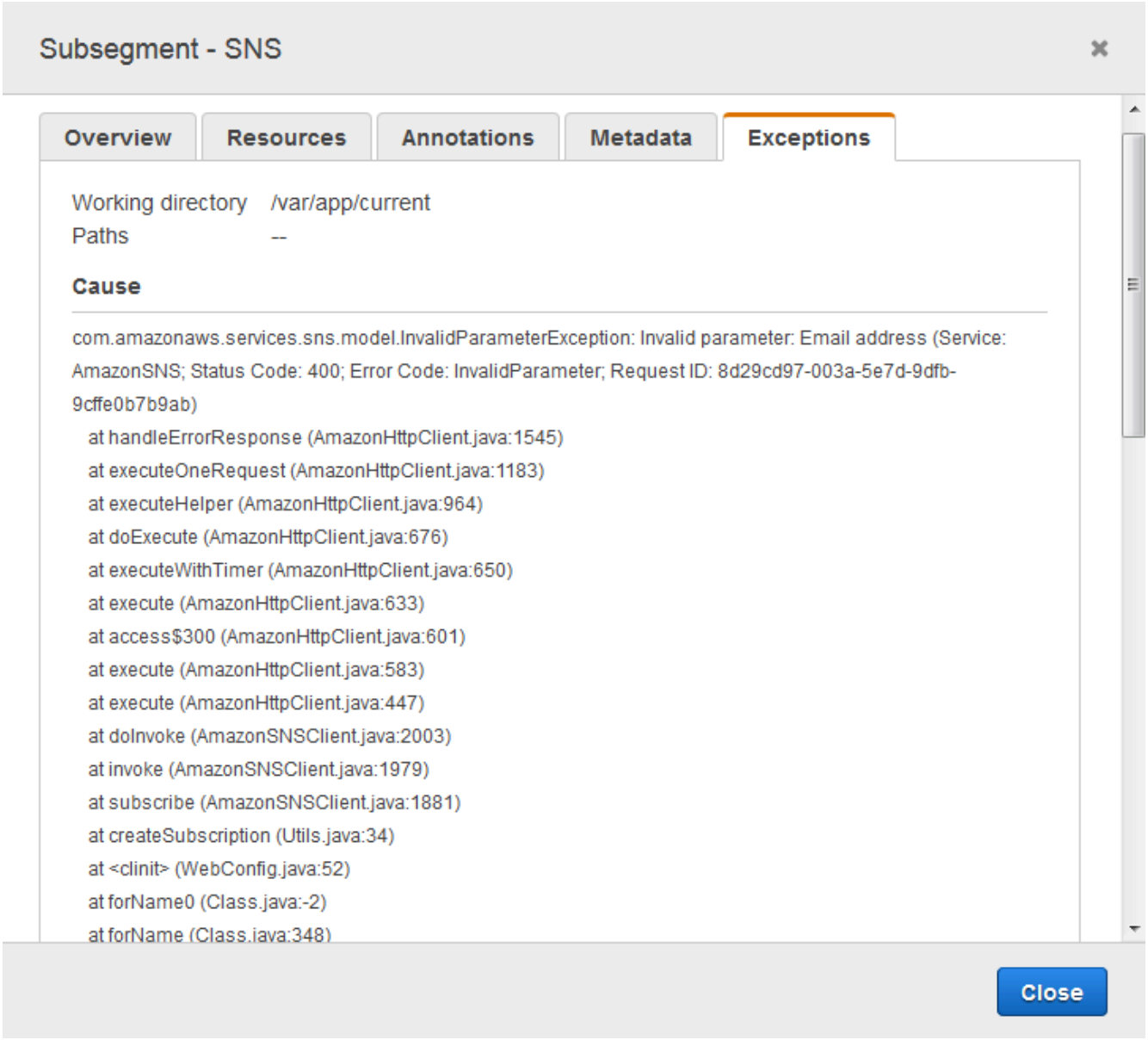
1. [SNS] という名前のノードを選択します。ノードの詳細パネルが表示されます。
2. [トレースの表示] を選択して、[トレースの概要] 画面にアクセスします。
3. [Trace のリスト] からトレースを選択します。受信リクエストに応答するのではなく起動時に記録されているため、トレースには、メソッドまたは URL はありません。



4. ページ下部の Amazon SNS セグメント内のエラーステータスアイコンを選択し、SNS サブセグメントの [例外] ページを開きます。



5. X-Ray SDK は、計測された AWS SDK クライアントによってスローされた例外を自動的にキャプチャし、スタックトレースを記録します。

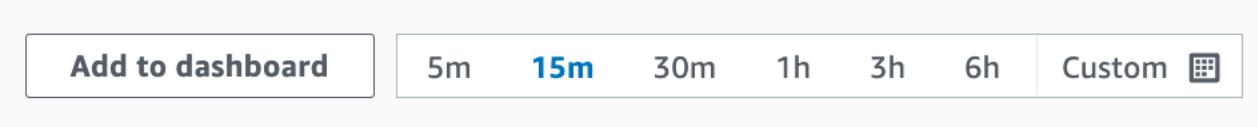


CloudWatch console

CloudWatch コンソールを使用する

1. CloudWatch コンソールの X-Ray トレースマップページを開きます。
2. コンソールには、X-Ray によってアプリケーションから送信されたトレースデータから生成されたサービスグラフの表現が表示されます。ウェブアプリケーションを最初に起動した時

点からのすべてのトレースが表示されるように、必要に応じてトレースマップの期間を調整してください。



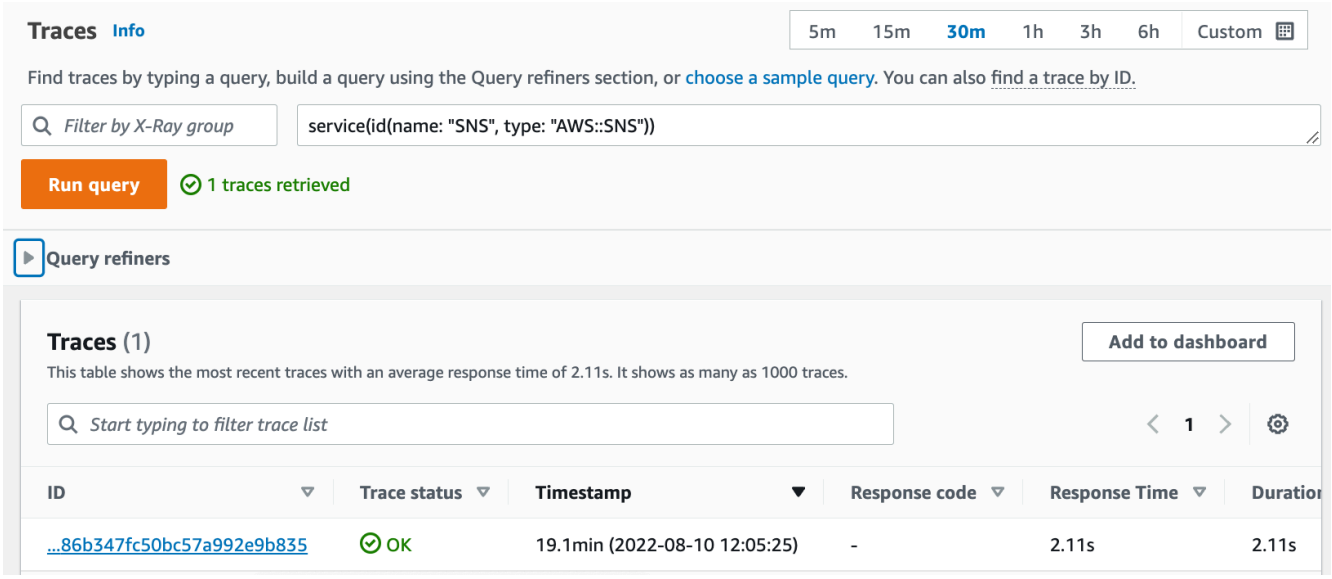
トレースマップに表示されるのは、ウェブアプリケーションクライアント、Amazon EC2 で実行されている API、アプリケーションで使用される各 DynamoDB テーブルです。アプリケーションに対するすべてのリクエストは、1 秒あたりのリクエストの設定可能な最大数まで、API にヒットした際にトレースされ、ダウンストリームサービスへのリクエストを生成して、完了します。

サービスグラフの任意のノードを選択すると、そのノードに対してトラフィックを生成したリクエストのトレースを表示できます。現在、Amazon SNS ノードはオレンジ色になっています。理由を調べるために掘り下げます。

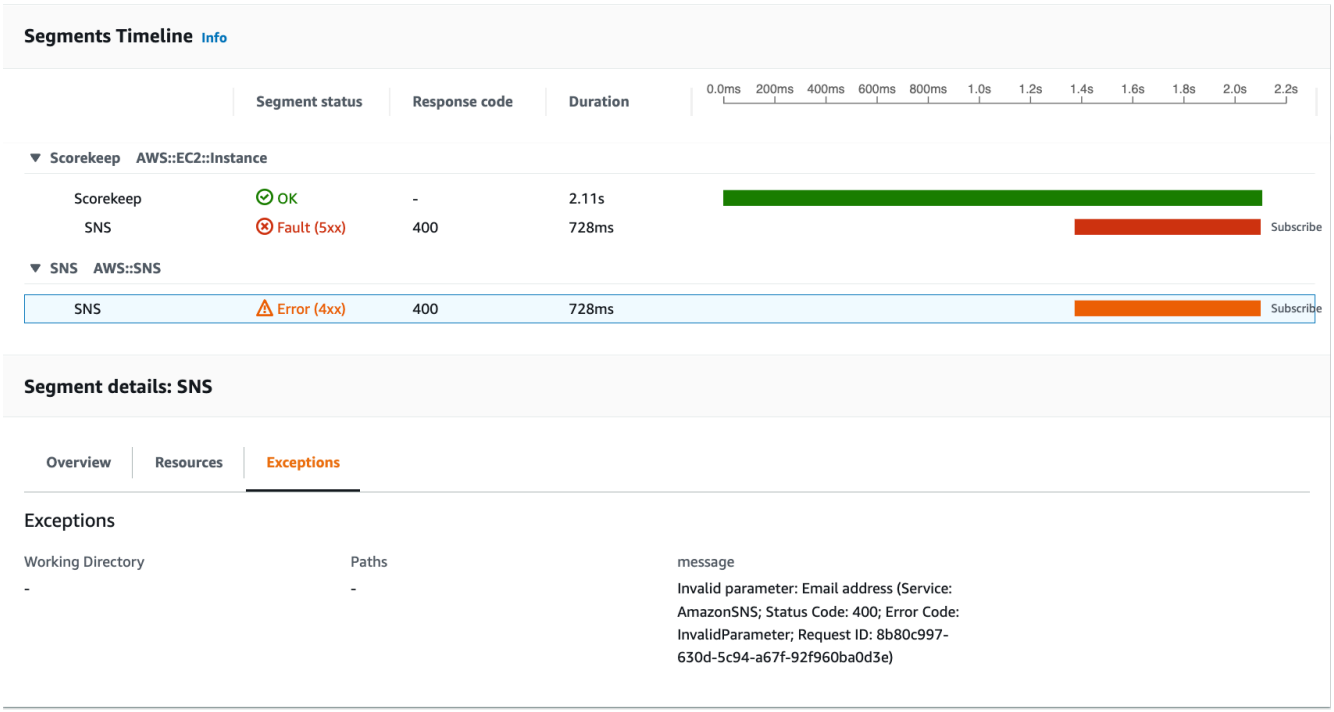

scorekeep-state
DynamoDB Table
scorekeep-game
DynamoDB Table
scorekeep-move
DynamoDB Table
Client
Scorekeep
ECS Container
SNS
SNS
scorekeep-session
DynamoDB Table

エラーの原因を見つけるには

1. [SNS] という名前のノードを選択します。SNS ノードの詳細パネルがマップの下に表示されます。
2. [トレースを表示] を選択して [トレース] ページにアクセスします。
3. ページの下部で、[トレース] リストからトレースを選択します。受信リクエストに応答するのではなく起動時に記録されているため、トレースには、メソッドまたは URL はありません。



4. セグメントタイムラインの下部にある Amazon SNS サブセグメントを選択し、SNS サブセグメントの [例外] タブを選択すると、例外の詳細が表示されます。



この原因は、`WebConfig` クラスで行われた `createSubscription` の呼び出しで指定された E メールアドレスが無効であることを意味します。次のセクションで、これを修正します。

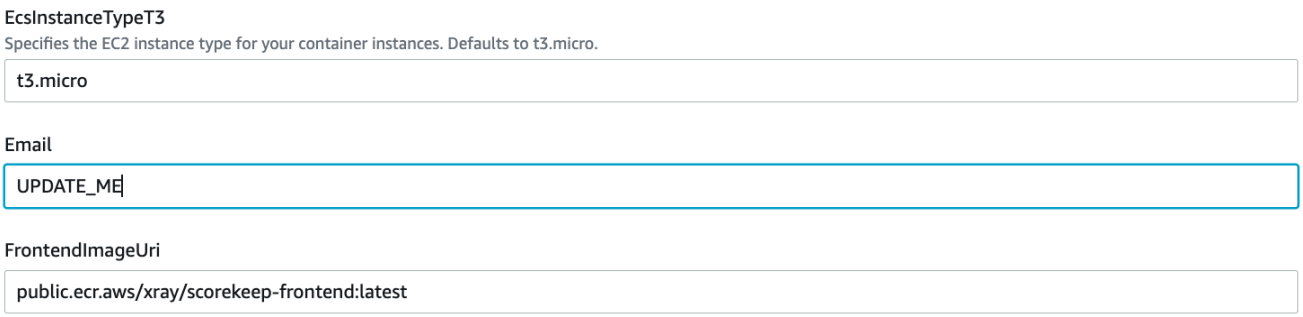
Amazon SNS 通知の設定

Scorekeep では、ユーザーがゲームを達成すると、Amazon SNS で通知が送信されます。アプリケーション起動時、CloudFormation のスタックパラメータで定義された E メールアドレスのサブスクリプションが作成されます。現在その呼び出しに失敗しています。通知を有効にするように通知 E メールを設定し、トレースマップで強調表示されている障害を解決します。

AWS Management Console

を使用して Amazon SNS 通知を設定するには AWS Management Console

1. CloudFormation コンソールを開きます。
2. リストの `scorekeep` スタック名の横にあるラジオボタンを選択して、[更新] を選択します。
3. [現在のテンプレートの使用] が選択されていることを確認し、[スタックの更新] ページで [次へ] をクリックします。
4. リストから [E メール] パラメータを探し、デフォルト値を有効な E メールアドレスに置き換えます。



5. ページの下部にスクロールし、[次へ] を選択します。
6. [レビュー] ページの一番下までスクロールし、[CloudFormation によってカスタム名がついた IAM リソースが作成される場合があることを承認します] チェックボックスをオンにし、[スタックの更新] を選択します。
7. CloudFormation スタックが更新中になります。スタックのステータスは約 5 分間 UPDATE_COMPLETE で、その後 UPDATE_IN_PROGRESS に変わります。ステータスは定期的に更新されます。ページを更新して再表示することもできます。

AWS CLI

を使用して Amazon SNS 通知を設定するには AWS CLI

1. 以前に作成した `xray-scorekeep/cloudformation/` フォルダーに移動し、`cf-resources.yaml` ファイルをテキストエディタで開きます。
2. [E メール] パラメータ内の `Default` 値を検索し、*`UPDATE_ME`* から有効な E メールアドレスに変更します。

```
Parameters:
  Email:
    Type: String
    Default: UPDATE_ME # <- change to a valid abc@def.xyz email address
```

3. `cloudformation` フォルダから、次の AWS CLI コマンドを使用して CloudFormation スタックを更新します。

```
aws cloudformation update-stack --stack-name scorekeep --capabilities
 "CAPABILITY_NAMED_IAM" --template-body file://cf-resources.yaml
```

4. CloudFormation スタックのステータスが UPDATE_COMPLETE になるまで数分待ってください。ステータスを確認するには、次の AWS CLI コマンドを使用します。

```
aws cloudformation describe-stacks --stack-name scorekeep --query
 "Stacks[0].StackStatus"
```

更新が完了すると、Scorekeep が再起動し、SNS トピックへのサブスクリプションが作成されます。E メールとサブスクリプションを確認して、ゲーム達成時にアップデートの有無を確認します。トレースマップを開いて、SNS への呼び出しが失敗しなくなったことを確認します。

サンプルアプリケーションの詳細

サンプルアプリケーションは、X-Ray SDK for Java を使用するように設定された Java の HTTP ウェブ API です。CloudFormation テンプレートを使用してアプリケーションをデプロイすると、ECS で Scorekeep を実行するために必要な DynamoDB テーブル、Amazon ECS クラスター、およびその他のサービスが作成されます。ECS のタスク定義ファイルは、CloudFormation によって作成されます。このファイルは ECS クラスター内のタスクごとに使用されるコンテナイメージを定義します。これらのイメージは、公式の X-Ray パブリック ECR から取得されます。Scorekeep API コンテナイメージには Gradle でコンパイルされた API が含まれています。Scorekeep フロントエンドコンテナのコンテナイメージは、nginx プロキシサーバーを使用するフロントエンドに対応します。このサーバーは /api で始まるパスにリクエストをルーティングして API に送信します。

受信 HTTP リクエストを測定するには、アプリケーションで SDK によって提供された `TracingFilter` を追加します。

Example src/main/java/scorekeep/WebConfig.java - サーブレットフィルタ

```
import javax.servlet.Filter;
import com.amazonaws.xray.javax.servlet.AWSXRayServletFilter;
...

@Configuration
public class WebConfig {

  @Bean
  public Filter TracingFilter() {
    return new AWSXRayServletFilter("Scorekeep");
  }
...
```

このフィルタは、アプリケーションが処理するすべての受信リクエストに関するトレースデータを送信します。リクエスト URL、メソッド、レスポンスステータス、開始時間、終了時間が含まれます。

また、アプリケーションは AWS SDK for Javaを使用して DynamoDB に対するダウンストリーム呼び出しを行います。これらの呼び出しを計測するために、アプリケーションは単に AWS SDK 関連のサブモジュールを依存関係として受け取り、X-Ray SDK for Java はすべての AWS SDK クライアントを自動的に計測します。

アプリケーションは `Docker` を使用して、インスタンス上で `Gradle Docker Image` と `Scorekeep API Dockerfile` ファイルを使用するソースコードを構築し、Gradle の `ENTRYPOINT` で生成する実行可能 JAR を実行します。

Example Docker を使用して Gradle Docker イメージ経由で構築する

```
docker run --rm -v /PATH/TO/SCOREKEEP_REPO/home/gradle/project -w /home/gradle/project
 gradle:4.3 gradle build
```

Example Dockerファイル ENTRYPOINT

```
ENTRYPOINT [ "sh", "-c", "java -Dserver.port=5000 -jar scorekeep-api-1.0.0.jar" ]
```

SDK サブモジュールを依存関係として宣言することで、コンパイル中に `build.gradle` ファイルによって SDK サブモジュールが Maven からダウンロードされます。

Example build.gradle -- 依存関係

```
...
dependencies {
    compile("org.springframework.boot:spring-boot-starter-web")
    testCompile('org.springframework.boot:spring-boot-starter-test')
    compile('com.amazonaws:aws-java-sdk-dynamodb')
    compile("com.amazonaws:aws-xray-recorder-sdk-core")
    compile("com.amazonaws:aws-xray-recorder-sdk-aws-sdk")
    compile("com.amazonaws:aws-xray-recorder-sdk-aws-sdk-instrumentor")
    ...
}
dependencyManagement {
    imports {
        mavenBom("com.amazonaws:aws-java-sdk-bom:1.11.67")
        mavenBom("com.amazonaws:aws-xray-recorder-sdk-bom:2.11.0")
    }
```

```
}
```

コア、 AWS SDK、および AWS SDK Instrumentor サブモジュールは、 AWS SDK で行われたダウンストリーム呼び出しを自動的に計測するために必要なすべてです。

未加工のセグメントデータを X-Ray API に中継するには、X-Ray デーモンが UDP ポート 2000 でトラフィックを受信する必要があります。そのため、アプリケーションでは、ECS で Scorekeep アプリケーションとともにサイドカーコンテナとしてデプロイされるコンテナでX-Ray デーモンを実行します。詳細については、X-Ray デーモンのトピックを参照してください。

Example ECS タスク定義内の X-Ray デーモンコンテナ定義

```
...
Resources:
  ScorekeepTaskDefinition:
    Type: AWS::ECS::TaskDefinition
    Properties:
      ContainerDefinitions:
      ...

      - Cpu: '256'
        Essential: true
        Image: amazon/aws-xray-daemon
        MemoryReservation: '128'
        Name: xray-daemon
        PortMappings:
          - ContainerPort: '2000'
            HostPort: '2000'
            Protocol: udp
      ...
```

X-Ray SDK for Java は、AWSXRay という名前のクラスを提供します。これはコードを計測するために使用する TracingHandler というグローバルレコーダーを提供します。グローバルレコーダーを設定して、受信 HTTP 呼び出しのセグメントを作成する AWSXRayServletFilter をカスタマイズできます。サンプルには、プラグインとサンプリングルールでグローバルレコーダーを設定する WebConfig クラスの静的ブロックが含まれています。

Example src/main/java/scorekeep/WebConfig.java - レコーダー

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.AWSXRayRecorderBuilder;
import com.amazonaws.xray.javax.servlet.AWSXRayServletFilter;
```

```
import com.amazonaws.xray.plugins.ECSPlugin;
import com.amazonaws.xray.plugins.EC2Plugin;
import com.amazonaws.xray.strategy.sampling.LocalizedSamplingStrategy;
...

@Configuration
public class WebConfig {
  ...

  static {
    AWSXRayRecorderBuilder builder = AWSXRayRecorderBuilder.standard().withPlugin(new
 ECSPlugin()).withPlugin(new EC2Plugin());

    URL ruleFile = WebConfig.class.getResource("/sampling-rules.json");
    builder.withSamplingStrategy(new LocalizedSamplingStrategy(ruleFile));

    AWSXRay.setGlobalRecorder(builder.build());
    ...

  }
}
```

この例では、ビルダーを使用して sampling-rules.json という名前のファイルからサンプリングルールをロードします。サンプリングルールは、SDK が受信リクエストのセグメントを記録するレートを決定します。

Example src/main/java/resources/sampling-rules.json

```
{
  "version": 1,
  "rules": [
    {
      "description": "Resource creation.",
      "service_name": "*",
      "http_method": "POST",
      "url_path": "/api/*",
      "fixed_target": 1,
      "rate": 1.0
    },
    {
      "description": "Session polling.",
      "service_name": "*",
      "http_method": "GET",
```

```
      "url_path": "/api/session/*",
      "fixed_target": 0,
      "rate": 0.05
    },
    {
      "description": "Game polling.",
      "service_name": "*",
      "http_method": "GET",
      "url_path": "/api/game/*/*",
      "fixed_target": 0,
      "rate": 0.05
    },
    {
      "description": "State polling.",
      "service_name": "*",
      "http_method": "GET",
      "url_path": "/api/state/*/*/*",
      "fixed_target": 0,
      "rate": 0.05
    }
  ],
  "default": {
    "fixed_target": 1,
    "rate": 0.1
  }
}
```

サンプリングルールファイルでは、4 つのカスタムサンプリングルールおよびデフォルトルールが定義されます。各受信リクエスト用に、SDK は定義された順にカスタムルールを適用します。SDK は、リクエストのメソッド、パス、サービス名に一致する最初のルールを適用します。Scorekeep の場合、最初のルールは、1 秒あたり 1 リクエストの固定ターゲット、および固定ターゲットが満たされた後の 1.0 または 100% のリクエストのレートを適用して、すべての POST リクエスト (リソース作成呼び出し) をキャッチします。

他の 3 つのカスタムルールでは、固定ターゲットなしで、セッション、ゲーム、および状態の読み取り (GET リクエスト) に 5% のレートを適用します。これにより、フロントエンドが、コンテンツが最新であることを確認するために数秒ごとに自動的に行う周期的呼び出しのトレース数を最小限に抑えることができます。他のすべてのリクエストの場合は、ファイルは、1 秒あたり 1 リクエストのデフォルトレートおよび 10% のレートを定義します。

また、サンプルアプリケーションでは、手動 SDK クライアント計測、追加サブセグメントの作成、HTTP 呼び出しの出力など、高度な機能の使用方法も説明します。詳細については、「AWS X-Ray サンプルアプリケーション」を参照してください。

オプション: 最小特権ポリシー

Scorekeep ECS コンテナは、`AmazonSNSFullAccess` や `AmazonDynamoDBFullAccess` などのフルアクセスポリシーを使用してリソースにアクセスします。フルアクセスポリシーの使用は、本稼働アプリケーションではベストプラクティスではありません。次の例では、DynamoDB IAM ポリシーを更新してアプリケーションのセキュリティを向上させます。IAM ポリシーのセキュリティのベストプラクティスの詳細については、AWS 「X-Ray の Identity and Access Management」を参照してください。

Example cf-resources.yaml テンプレート ECSTaskRole 定義

```
ECSTaskRole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Version: "2012-10-17"
        Statement:
          -
            Effect: "Allow"
            Principal:
              Service:
                - "ecs-tasks.amazonaws.com"
            Action:
              - "sts:AssumeRole"
      ManagedPolicyArns:
        - "arn:aws:iam::aws:policy/AmazonDynamoDBFullAccess"
        - "arn:aws:iam::aws:policy/AmazonSNSFullAccess"
        - "arn:aws:iam::aws:policy/AWSXrayFullAccess"
      RoleName: "scorekeepRole"
```

ポリシーを更新するには、最初に DynamoDB リソースの ARN を特定します。次に、カスタム IAM ポリシーで ARN を使用します。最後に、そのポリシーをインスタンスプロファイルに適用します。

DynamoDB リソースの ARN を識別するには :

1. DynamoDB コンソールを開きます。
2. 左側のナビゲーションバーから [テーブル] を選択します。

3. scorekeep-* のいずれかを選択すると、テーブルの詳細ページが表示されます。
4. [概要] タブで [追加情報] を選択してセクションを展開し、Amazon リソースネーム (ARN) を表示します。この値をコピーします。
5. AWS_REGION および AWS_ACCOUNT_ID の値を特定のリージョンとアカウント ID に置き換えて、ARN を次の IAM ポリシーに挿入します。この新しいポリシーは、すべてのアクションを許可する AmazonDynamoDBFullAccess ポリシーではなく、指定されたアクションのみ許可します。

   Example

   JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ScorekeepDynamoDB",
            "Effect": "Allow",
            "Action": [
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:Scan",
                "dynamodb:Query"
            ],
            "Resource":
 "arn:aws:dynamodb:<AWS_REGION>:<AWS_ACCOUNT_ID>:table/scorekeep-*"
        }
    ]
}
```

   アプリケーションが作成するテーブルは、一貫した命名規則に従います。scorekeep-* 形式を使用して、すべての Scorekeep テーブルを指定できます。

IAM ポリシーを変更する

1. IAM コンソールから Scorekeep タスクロール (scorekeepRole) を開きます。

2. AmazonDynamoDBFullAccess ポリシーの横にあるチェックボックスを選択して [削除] を選択し、このポリシーを削除します。
3. [アクセス許可の追加]、[ポリシーのアタッチ]、[ポリシーの作成] の順に選択します。
4. [JSON] タブを選択し、上で作成したポリシーを貼り付けます。
5. ページ下部の [次へ: タグ] を選択します。
6. ページ下部の [次へ: 確認] を選択します。
7. [名前] に、ポリシーの名前を割り当てます。
8. ページの下部の [ポリシーの作成] を選択します。
9. 新しく作成したポリシーを scorekeepRole ロールにアタッチします。アタッチしたポリシーが適用されるまで数分かかることがあります。

新しいポリシーを scorekeepRole ロールにアタッチした場合は、CloudFormation スタックを削除する前にデタッチする必要があります。このアタッチしたポリシーによってスタックの削除がブロックされるためです。ポリシーを削除すると、ポリシーを自動的にデタッチできます。

カスタム IAM ポリシーを削除する

1. [IAM コンソール] を開きます。
2. 左側のナビゲーションバーから [ポリシー] を選択します。
3. このセクションで先ほど作成したカスタムポリシー名を検索し、ポリシー名の横にあるラジオボタンを選択して強調表示します。
4. [アクション] ドロップダウンを選択してから、[削除] を選択します。
5. カスタムポリシーの名前を入力して [削除] を選択し、削除を確定します。これにより、ポリシーが scorekeepRole ロールから自動的にデタッチされます。

クリーンアップ

Scorekeep アプリケーションのリソースを削除するには、次の手順を従います。

> **Note**
>
> このチュートリアルの前のセクションでカスタムポリシーを作成してアタッチした場合は、CloudFormation スタックを削除する前に、scorekeepRole からポリシーを削除する必要があります。

AWS Management Console

を使用してサンプルアプリケーションを削除する AWS Management Console

1. CloudFormation コンソールを開きます。
2. リストの scorekeep スタック名の横にあるラジオボタンを選択して、[削除] を選択します。
3. CloudFormation スタックが削除中になります。スタックのステータスは、すべてのリソースが削除されるまで数分間 DELETE_IN_PROGRESS になります。ステータスは定期的に更新されます。ページを更新して再表示することもできます。

AWS CLI

を使用してサンプルアプリケーションを削除する AWS CLI

1. CloudFormation スタック AWS CLI を削除するには、次のコマンドを入力します。

```
aws cloudformation delete-stack --stack-name scorekeep
```

2. CloudFormation スタックが存在しなくなるまで約 5 分待ってください。ステータスを確認するには、次の AWS CLI コマンドを使用します。

```
aws cloudformation describe-stacks --stack-name scorekeep --query
 "Stacks[0].StackStatus"
```

次のステップ

次の章「AWS X-Ray の概念」で X-Ray の詳細をご覧ください。。

独自のアプリケーションを測定するには、X-Ray SDK for Javaまたは他の X-Ray SDK のいずれかの詳細をご覧ください。

- X-Ray SDK for Java – AWS X-Ray SDK for Java
- X-Ray SDK for Node.js – AWS Node.js 用 X-Ray SDK
- X-Ray SDK for .NET – AWS X-Ray SDK for .NET

X-Ray デーモンをローカルまたは で実行するには AWS、「」を参照してくださいAWS X-Ray デーモン。

サンプルアプリケーションを GitHub に投稿するには、eb-java-scorekeepを参照してください。

AWS SDK クライアントの手動計測

X-Ray SDK for Java は、ビルドの依存関係に AWS SDK Instrumentor サブモジュールを含めると、すべての SDK クライアントを自動的に計測します。 AWS

クライアントの自動実装は、Instrumentor サブモジュールを削除することで無効にできます。これにより、他を無視しながら一部のクライアントを手動で実装するか、異なるクライアントで異なるトレースハンドラーを使用できるようになります。

特定の AWS SDK クライアントの計測のサポートを説明するために、アプリケーションはユーザー、ゲーム、セッションモデルのリクエストハンドラーAmazonDynamoDBClientBuilderとして にトレースハンドラーを渡します。このコード変更により、これらのクライアントを使用する DynamoDB に対するすべての呼び出しを実装するように SDK に指示します。

Example **src/main/java/scorekeep/SessionModel.java** – AWS SDK クライアントの手動実装

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.handlers.TracingHandler;

public class SessionModel {
  private AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
        .withRegion(Constants.REGION)
        .withRequestHandlers(new TracingHandler(AWSXRay.getGlobalRecorder()))
        .build();
  private DynamoDBMapper mapper = new DynamoDBMapper(client);
```

プロジェクトの依存関係から AWS SDK Instrumentor サブモジュールを削除すると、手動で計測された AWS SDK クライアントのみがトレースマップに表示されます。

追加のサブセグメントを作成する

ユーザーモデルクラスでは、アプリケーションがサブセグメントを手動で作成して、saveUser 関数内で行われるすべてのダウンストリーム呼び出しをグループ化し、メタデータを追加します。

Example **src/main/java/scorekeep/UserModel.java** - カスタムサブセグメント

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.entities.Subsegment;
...
    public void saveUser(User user) {
    // Wrap in subsegment
    Subsegment subsegment = AWSXRay.beginSubsegment("## UserModel.saveUser");
    try {
      mapper.save(user);
    } catch (Exception e) {
      subsegment.addException(e);
      throw e;
    } finally {
      AWSXRay.endSubsegment();
    }
  }
```

注釈、メタデータ、およびユーザー ID を記録する

ゲームモデルクラスでは、アプリケーションはDynamoDB にゲームを保存するたびにメタデータ ブロックに Game オブジェクトを記録します。アプリケーションは個別にゲーム ID を、 フィルタ式 で使用できるように、注釈 に記録します。

Example **src/main/java/scorekeep/GameModel.java**–注釈とメタデータ

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.entities.Segment;
import com.amazonaws.xray.entities.Subsegment;
...
  public void saveGame(Game game) throws SessionNotFoundException {
    // wrap in subsegment
    Subsegment subsegment = AWSXRay.beginSubsegment("## GameModel.saveGame");
    try {
      // check session
      String sessionId = game.getSession();
      if (sessionModel.loadSession(sessionId) == null ) {
        throw new SessionNotFoundException(sessionId);
      }
      Segment segment = AWSXRay.getCurrentSegment();
      subsegment.putMetadata("resources", "game", game);
```

```
      segment.putAnnotation("gameid", game.getId());
      mapper.save(game);
    } catch (Exception e) {
      subsegment.addException(e);
      throw e;
    } finally {
      AWSXRay.endSubsegment();
    }
  }
```

移動コントローラーでは、アプリケーションはユーザー ID を setUser を使用して記録します。ユーザー ID はセグメントの個別のフィールドに記録され、検索用にインデックスが作成されます。

Example src/main/java/scorekeep/MoveController.java – ユーザー ID

```
import com.amazonaws.xray.AWSXRay;
...
  @RequestMapping(value="/{userId}", method=RequestMethod.POST)
  public Move newMove(@PathVariable String sessionId, @PathVariable String
 gameId, @PathVariable String userId, @RequestBody String move) throws
 SessionNotFoundException, GameNotFoundException, StateNotFoundException,
 RulesException {
    AWSXRay.getCurrentSegment().setUser(userId);
    return moveFactory.newMove(sessionId, gameId, userId, move);
  }
```

送信 HTTP 呼び出しの計測

ユーザーファクトリクラスは、アプリケーションがX-Ray SDK for Javaの HTTPClientBuilder バージョンを使用して送信 HTTP 呼び出しを計測する方法を示します。

Example **src/main/java/scorekeep/UserFactory.java**–HTTP クライアント計測

```
import com.amazonaws.xray.proxies.apache.http.HttpClientBuilder;

  public String randomName() throws IOException {
    CloseableHttpClient httpclient = HttpClientBuilder.create().build();
    HttpGet httpGet = new HttpGet("http://uinames.com/api/");
    CloseableHttpResponse response = httpclient.execute(httpGet);
    try {
      HttpEntity entity = response.getEntity();
```

```
      InputStream inputStream = entity.getContent();
      ObjectMapper mapper = new ObjectMapper();
      Map<String, String> jsonMap = mapper.readValue(inputStream, Map.class);
      String name = jsonMap.get("name");
      EntityUtils.consume(entity);
      return name;
    } finally {
      response.close();
    }
  }
```

現在 `org.apache.http.impl.client.HttpClientBuilder` を使用している場合は、そのクラスのインポートステートメントを `com.amazonaws.xray.proxies.apache.http.HttpClientBuilder` のものと交換するだけです。

PostgreSQL データベースに対する呼び出しの計測

`application-pgsql.properties` ファイルは X-Ray PostgreSQL トレースインターセプターを `RdsWebConfig.java` で作成されたデータソースに追加します。

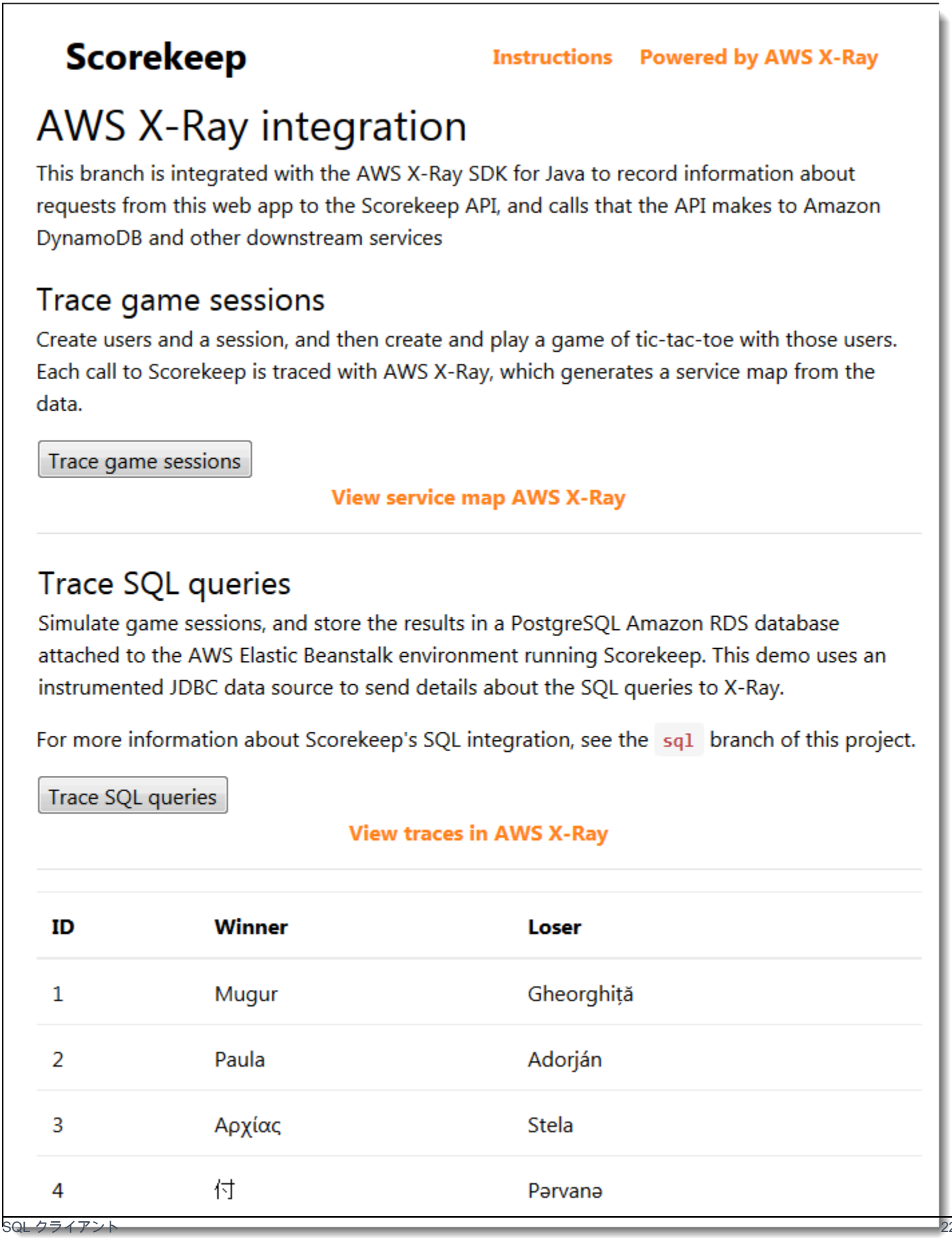
Example **`application-pgsql.properties`** – PostgreSQL データベース実装

```
spring.datasource.continue-on-error=true
spring.jpa.show-sql=false
spring.jpa.hibernate.ddl-auto=create-drop
spring.datasource.jdbc-interceptors=com.amazonaws.xray.sql.postgres.TracingInterceptor
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQL94Dialect
```

> **Note**
>
> アプリケーション環境に PostgreSQL データベースを追加する方法に関する詳細については、『https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/using-features.managing.db.html 開発者ガイド』の「AWS Elastic Beanstalk Elastic Beanstalk を使用してデータベースを設定する」 を参照してください。

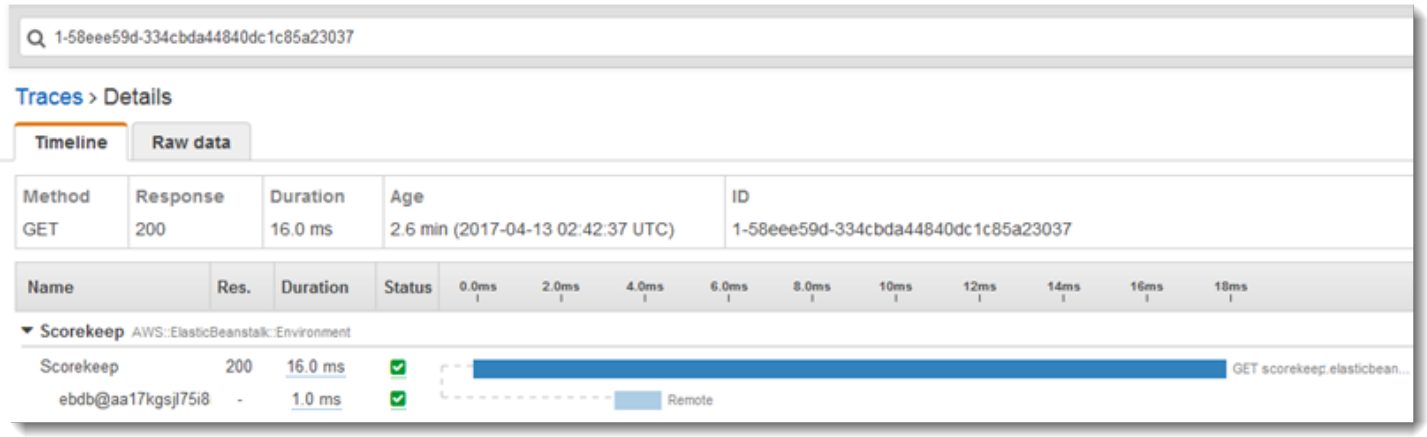
`xray` ブランチの X-Rayデモページは、実装済みデータソースを使用して、生成した SQL クエリに関する情報を示すトレースを生成するデモを含みます。実行中のアプリケーションの `/#/xray` パス

に移動するか、ナビゲーションバーの [Powered by AWS X-Ray] を選択してデモページを表示します。


Scorekeep
Instructions
Powered by AWS X-Ray
AWS X-Ray integration
This branch is integrated with the AWS X-Ray SDK for Java to record information about requests from this web app to the Scorekeep API, and calls that the API makes to Amazon DynamoDB and other downstream services
Trace game sessions
Create users and a session, and then create and play a game of tic-tac-toe with those users. Each call to Scorekeep is traced with AWS X-Ray, which generates a service map from the data.
Trace game sessions
View service map AWS X-Ray
Trace SQL queries
Simulate game sessions, and store the results in a PostgreSQL Amazon RDS database attached to the AWS Elastic Beanstalk environment running Scorekeep. This demo uses an instrumented JDBC data source to send details about the SQL queries to X-Ray.
For more information about Scorekeep's SQL integration, see the sql branch of this project.
Trace SQL queries
View traces in AWS X-Ray
ID
Winner
Loser
1
Mugur
Gheorghiță
2
Paula
Adorján
3
Αρχίας
Stela
4
付
Pərvanə

[Trace SQL queries] を選択して、ゲームセッションをシミュレートし、アタッチされたデータベースに結果を保存します。次に、 AWS X-Ray でトレースを表示する を選択して、API の`/api/history`ルートにヒットしたトレースのフィルタリングされたリストを表示します。

SQL クエリを含め、リストからトレースのいずれかを選択して、タイムラインを表示します。



AWS Lambda 関数の計測

Scorekeep は 2 つの AWS Lambda 関数を使用します。1 つ目は、新しいユーザー用にランダムな名前を生成する `lambda` ブランチからの Node.js 関数です。ユーザーが名前を入力せずにセッションを作成すると、アプリケーションは、 AWS SDK for Javaで `random-name` という名前の関数を呼び出します。X-Ray SDK for Java は、計測された AWS SDK クライアントで行われた他の呼び出しと同様に、サブセグメント内の Lambda への呼び出しに関する情報を記録します。

> **Note**
>
> `random-name` Lambda 関数を実装するには、Elastic Beanstalk 環境の外でその他のリソースを作成する必要があります。詳細については readme、手順については「AWS Lambda との統合」を参照してください。

2 つ目の関数 `scorekeep-worker` は、Scorekeep API と関係なく実行される Python 関数です。ゲームが終了すると、API はセッション ID とゲーム ID を SQS キューに書き込みます。ワーカー関数はキューから項目を読み取り、Scorekeep API を呼び出してAmazon S3 内のストレージのゲームセッションごとに完全なレコードを構築します。

Scorekeep には、両方の関数を作成するための AWS CloudFormation テンプレートとスクリプトが含まれています。X-Ray SDK を関数コードとバンドルする必要があるため、テンプレートはコード

なしで関数を作成します。Scorekeep をデプロイすると、`.ebextensions` フォルダに含まれている設定ファイルにより、SDK を含むソースバンドルが作成され、 AWS Command Line Interfaceによって関数コードと設定が更新されます。

関数

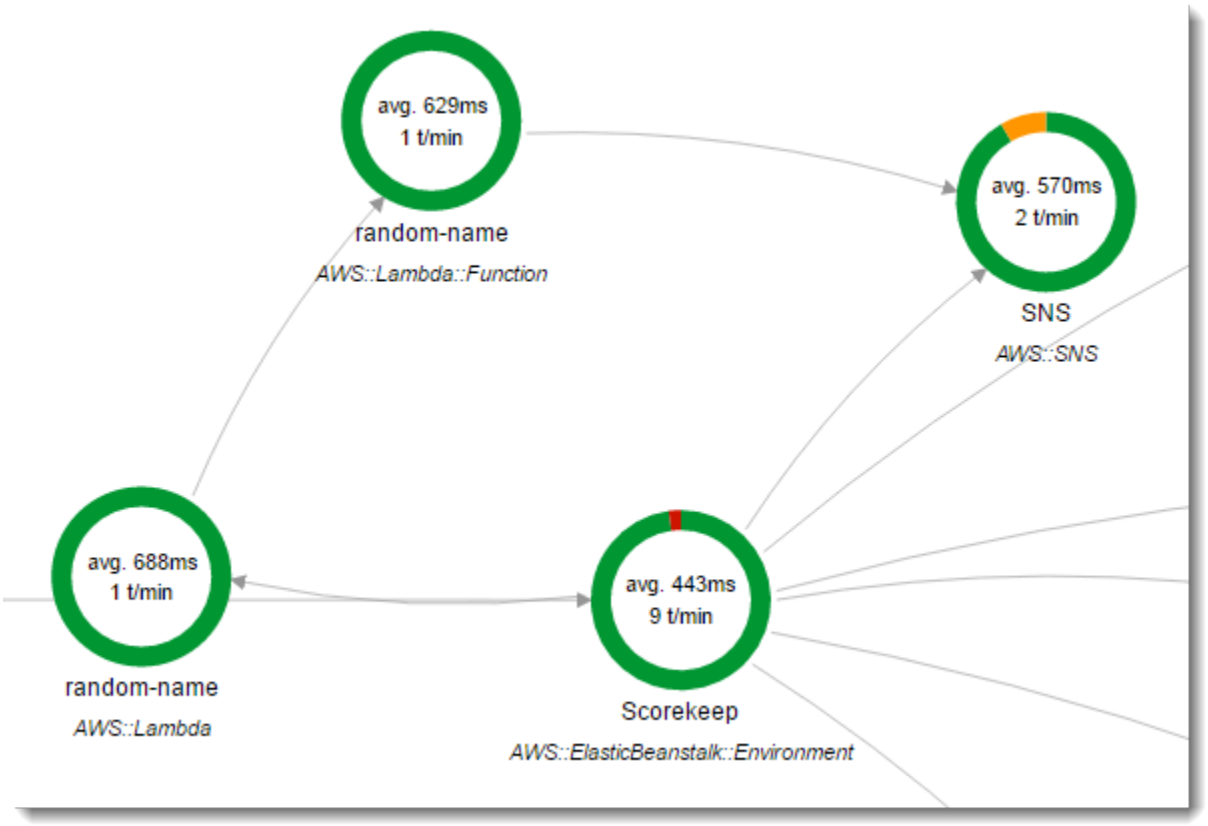
- ランダム名
- ワーカー

ランダム名

Scorekeep は、ユーザーがサインインしたりユーザー名を指定したりせずにゲームセッションを開始するとランダム名関数を呼び出します。`random-name` の呼び出しが Lambda で処理されると、「トレースヘッダー」が読み出されます。これには、X-Ray SDK for Javaによって書き込まれるトレース ID とサンプリングデシジョンを含みます。

Lambda は、サンプリングされたリクエストごとに X-Ray デーモンを実行し、2 つのセグメントを書き込みます。最初のセグメントでは、関数を呼び出す Lambda の呼び出しに関する情報を記録します。このセグメントには、Scorekeep によって記録されるサブセグメントと同じ情報が含まれますが、Lambda の視点からという点で異なります。2 番目のセグメントは、関数の動作を表します。

Lambda は、関数コンテキストを通じて X-Ray SDK に関数セグメントを渡します。Lambda 関数を実装した場合、受信リクエストのセグメントを作成するために SDK は使用しません。 Lambdaにセグメントが提供され、SDK を使用することでクライアントを実装してサブセグメントを書き込みます。



`random-name` 関数は、Node.js で実装されています。Node.js の SDK for JavaScript を使用して Amazon SNS で通知を送信し、X-Ray SDK for Node.js を使用して AWS SDK クライアントを計測します。注釈を書き込むため、関数は `AWSXRay.captureFunc` を使用してカスタムサブセグメントを作成し、実装された関数に注釈を書き込みます。 Lambdaでは、関数セグメントに直接注釈を書き込むことはできません。作成したサブセグメントにのみ書き込むことができます。

Example **`function/index.js`** - Random 名 Lambda 関数

```
var AWSXRay = require('aws-xray-sdk-core');
var AWS = AWSXRay.captureAWS(require('aws-sdk'));

AWS.config.update({region: process.env.AWS_REGION});
var Chance = require('chance');

var myFunction = function(event, context, callback) {
  var sns = new AWS.SNS();
  var chance = new Chance();
  var userid = event.userid;
  var name = chance.first();

  AWSXRay.captureFunc('annotations', function(subsegment){
```

```
    subsegment.addAnnotation('Name', name);
    subsegment.addAnnotation('UserID', event.userid);
  });

  // Notify
  var params = {
    Message: 'Created randon name "' + name + '"" for user "' + userid + '".',
    Subject: 'New user: ' + name,
    TopicArn: process.env.TOPIC_ARN
  };
  sns.publish(params, function(err, data) {
    if (err) {
      console.log(err, err.stack);
      callback(err);
    }
    else {
      console.log(data);
      callback(null, {"name": name});
    }
  });
};

exports.handler = myFunction;
```

この関数は、サンプルアプリケーションを Elastic Beanstalk にデプロイするときに自動的に作成されます。xray ブランチには、空白のLambda 関数を作成するスクリプトが含まれています。.ebextensions フォルダの設定ファイルは、デプロイnpm install中に で関数パッケージを構築し、 CLI で Lambda AWS 関数を更新します。

ワーカー

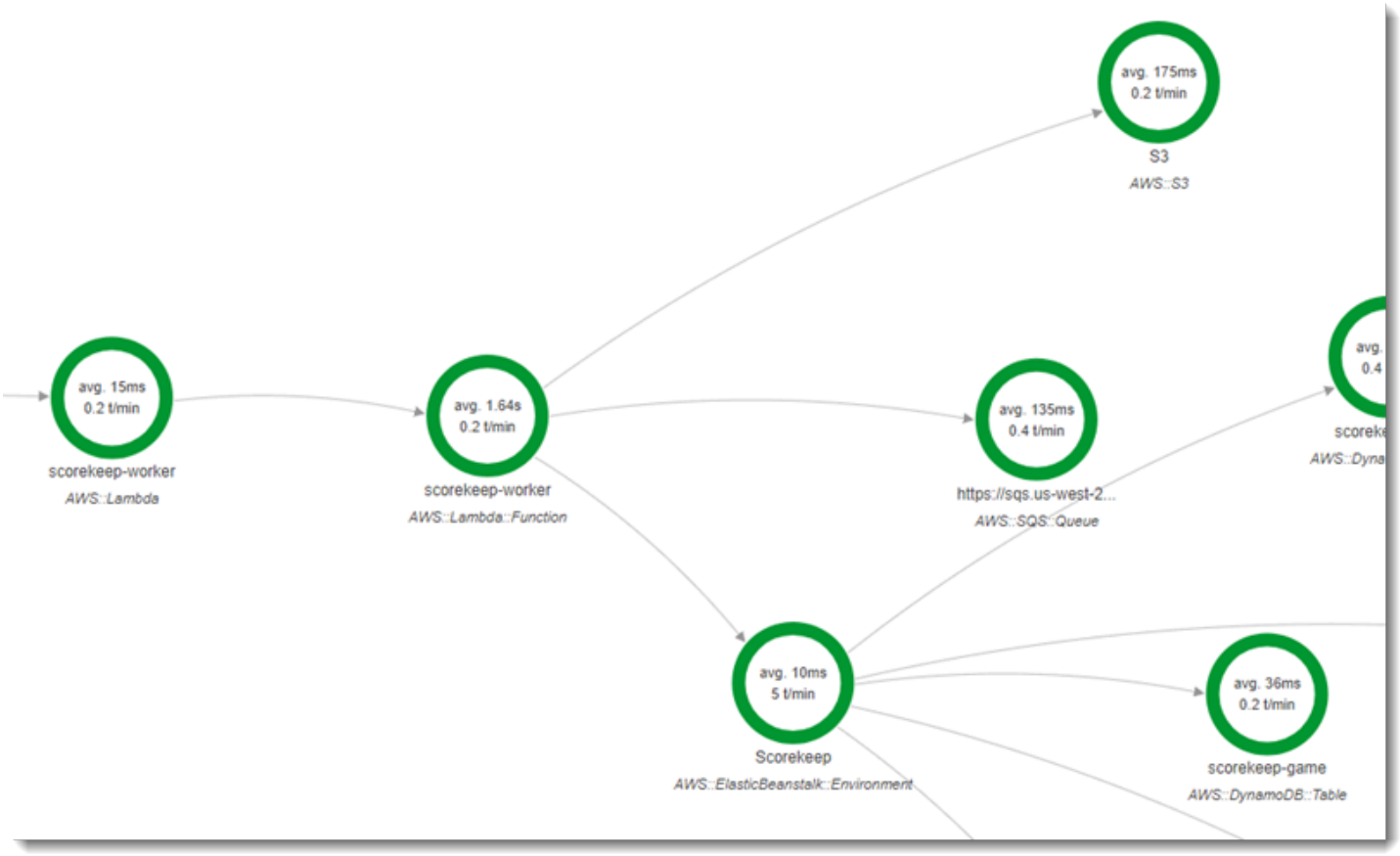
実装されたワーカー関数は、ワーカー関数と関連リソースを作成してからでないと実行できないため、自身のブランチ xray-worker で提供されます。手順については、ブランチの readme を参照してください。

この関数は、5 分ごとにバンドルされているAmazon CloudWatch Events イベントによってトリガーされます。実行されると、この関数は Scorekeep によって管理されるAmazon SQS キューから項目を取得します。各メッセージには、完了したゲームに関する情報が含まれています。

ワーカーは、ゲームレコードが参照する他のテーブルからゲームレコードとドキュメントを取得します。たとえば、 DynamoDBのゲームレコードには、ゲーム中に実行されたムービーのリストが含ま

れています。リストには、ムービー自体は含まれておらず、別個のテーブルに格納されたムービーの ID が含まれています。

セッションと状態は、参照としても保存されます。これにより、ゲームテーブル内のエントリが大きくなりすぎることはありませんが、ゲームに関するすべての情報を取得するには追加の呼び出しが必要です。このワーカーは、これらのすべてのエントリを間接参照し、ゲームの完全なレコードをAmazon S3 に単一のドキュメントとして構築します。データで分析を行う場合、読み取り量の多いデータ移行を実行してデータを DynamoDBから取得しなくても、Amazon Athena を使用して Amazon S3 で直接クエリを実行できます。



ワーカー関数では、 AWS Lambdaのその設定においてアクティブトレースが有効になっています。ランダム名関数とは異なり、ワーカーは計測されたアプリケーションからリクエストを受信しないため、 AWS Lambda はトレースヘッダーを受信しません。アクティブトレースを使用すると、Lambdaはトレース ID を作成してサンプリングの決定を行います。

X-Ray SDK for Python は、SDK をインポートし、そのpatch_all関数を実行して、Amazon SQS と Amazon S3 の呼び出しに使用する AWS SDK for Python (Boto) と HTTclientsにパッチを適用する関数の上部にある数行です。ワーカーが API を呼び出すと、SDK はトレースヘッダーをリクエストに追加し、API を通じて呼び出しをトレースします。

Example **_lambda/scorekeep-worker/scorekeep-worker.py** -- ワーカー Lambda 関数

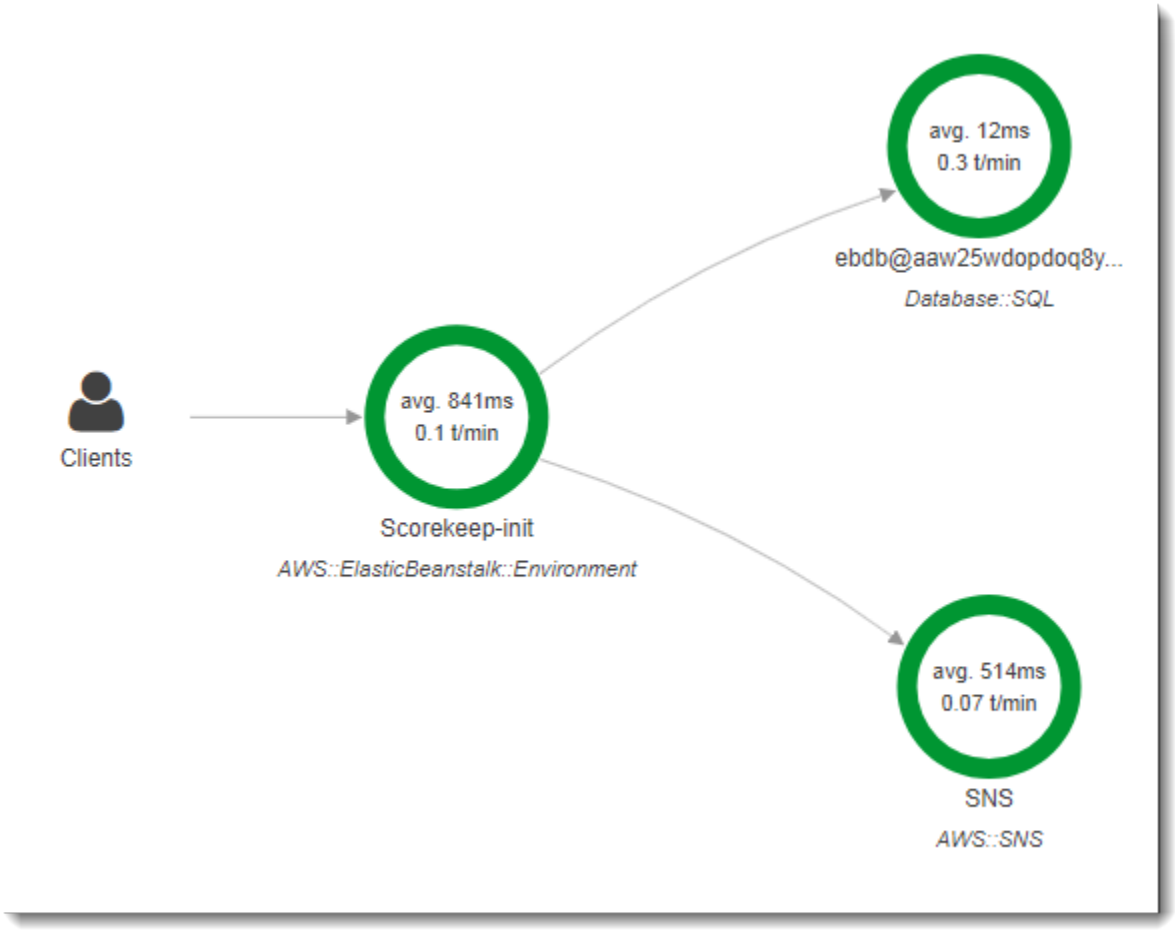
```
import os
import boto3
import json
import requests
import time
from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch_all

patch_all()
queue_url = os.environ['WORKER_QUEUE']

def lambda_handler(event, context):
    # Create SQS client
    sqs = boto3.client('sqs')
    s3client = boto3.client('s3')

    # Receive message from SQS queue
    response = sqs.receive_message(
        QueueUrl=queue_url,
        AttributeNames=[
            'SentTimestamp'
        ],
        MaxNumberOfMessages=1,
        MessageAttributeNames=[
            'All'
        ],
        VisibilityTimeout=0,
        WaitTimeSeconds=0
    )
   ...
```

スタートアップコードの作成

X-Ray SDK for Java は、着信リクエストのセグメントを自動的に作成します。リクエストが範囲内にある限り、計測クライアントを使用して問題なしでサブセグメントを記録できます。しかし、計測クライアントをスタートアップコードで使用しようとすると、SegmentNotFoundException が発生します。

スタートアップコードは、ウェブアプリケーションの標準的なリクエスト/レスポンスフローの外側で実行されるため、手動でセグメントを作成して計測する必要があります。Scorekeep はスタート

アップコードのインストルメンテーションをWebConfig ファイルに表示します。Scorekeep はスタートアップ時に SQL データベースと Amazon SNS を呼び出します。



デフォルトの WebConfig クラスは、通知のための Amazon SNS サブスクリプションを作成します。Amazon SNSクライアントの使用時に X-Ray SDK が書き込むセグメントを提供するために、Scorekeep はグローバルレコーダー上で beginSegment と endSegment を呼び出します。

Example **src/main/java/scorekeep/WebConfig.java** – スタートアップコードの計測 AWS SDK クライアント

```
AWSXRay.beginSegment("Scorekeep-init");
if ( System.getenv("NOTIFICATION_EMAIL") != null ){
  try { Sns.createSubscription(); }
  catch (Exception e ) {
    logger.warn("Failed to create subscription for email "+
 System.getenv("NOTIFICATION_EMAIL"));
  }
}
```

```
AWSXRay.endSegment();
```

Amazon RDS データベースが接続されているときに Scorekeep が使用する `RdsWebConfig` では、スタートアップ時にデータベーススキーマを適用するときに Hibernate が使用する SQL クライアントのセグメントも作成されます。

Example **src/main/java/scorekeep/RdsWebConfig.java** – スタートアップコードの実装 SQL データベースクライアント

```
@PostConstruct
public void schemaExport() {
  EntityManagerFactoryImpl entityManagerFactoryImpl = (EntityManagerFactoryImpl)
 localContainerEntityManagerFactoryBean.getNativeEntityManagerFactory();
  SessionFactoryImplementor sessionFactoryImplementor =
 entityManagerFactoryImpl.getSessionFactory();
  StandardServiceRegistry standardServiceRegistry =
 sessionFactoryImplementor.getSessionFactoryOptions().getServiceRegistry();
  MetadataSources metadataSources = new MetadataSources(new
 BootstrapServiceRegistryBuilder().build());
  metadataSources.addAnnotatedClass(GameHistory.class);
  MetadataImplementor metadataImplementor = (MetadataImplementor)
 metadataSources.buildMetadata(standardServiceRegistry);
  SchemaExport schemaExport = new SchemaExport(standardServiceRegistry,
 metadataImplementor);

  AWSXRay.beginSegment("Scorekeep-init");
  schemaExport.create(true, true);
  AWSXRay.endSegment();
}
```

`SchemaExport` は自動的に実行され、SQL クライアントを使用します。クライアントが計測されているため、Scorekeep はデフォルトの実装をオーバーライドし、SDK がクライアントの呼び出し時に使用するセグメントを提供する必要があります。

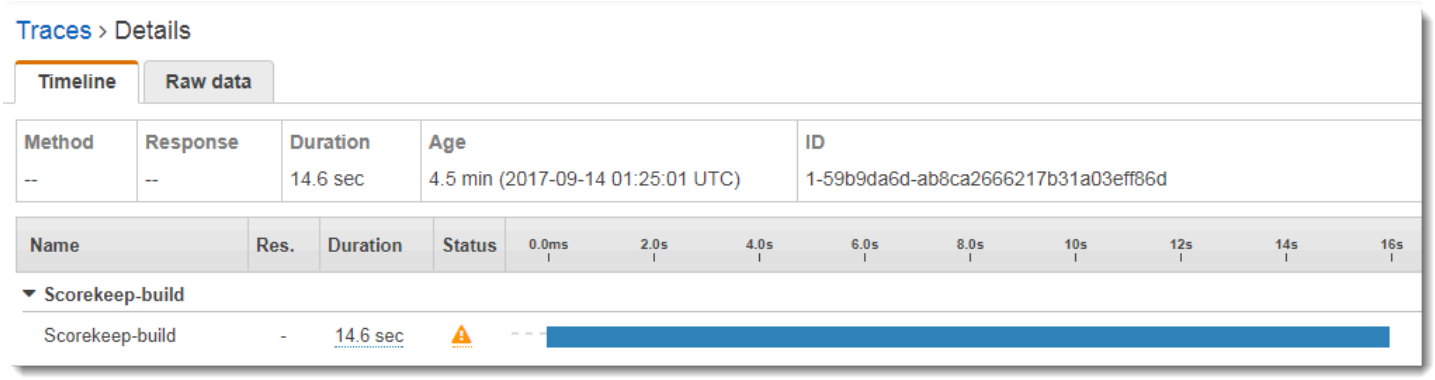
実装スクリプト

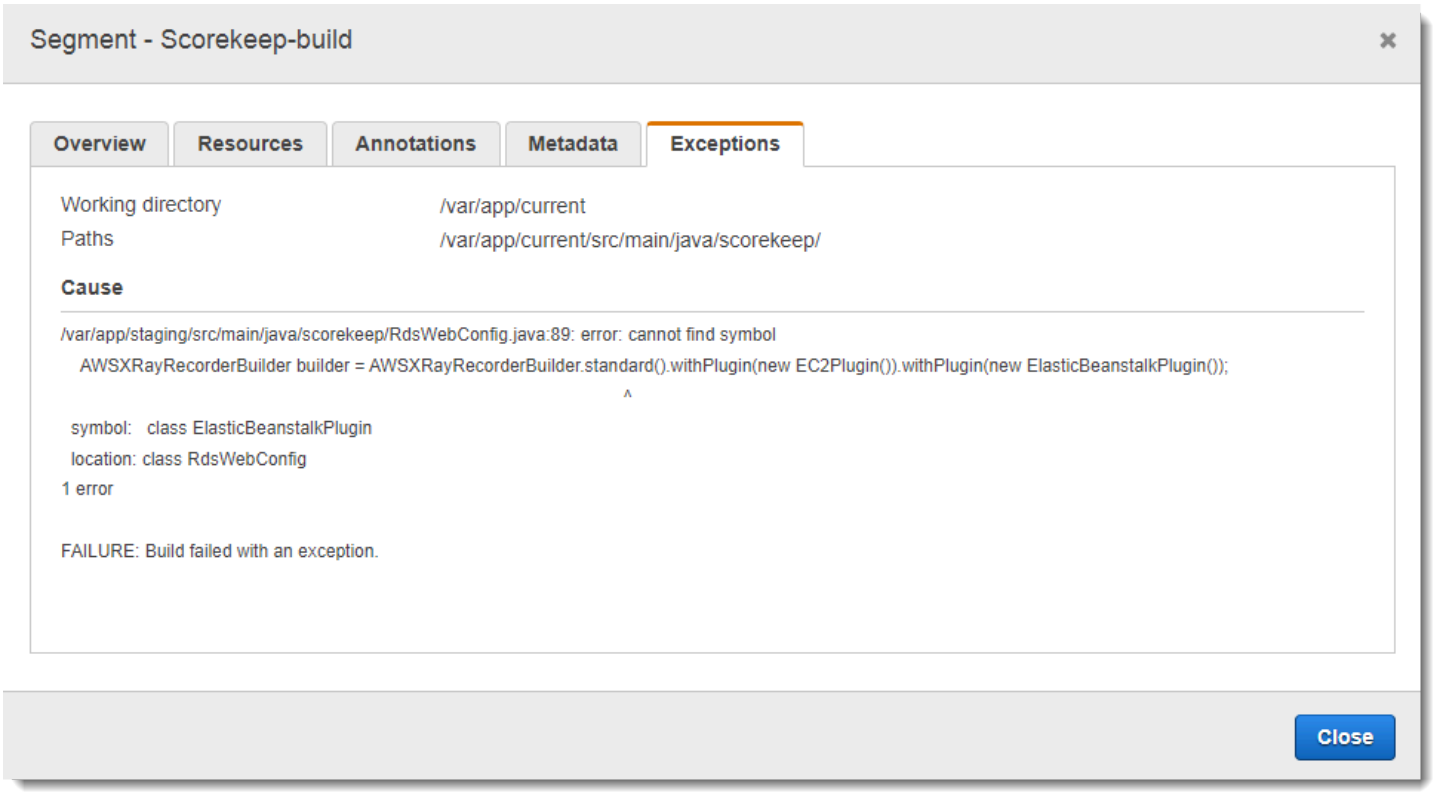
また、アプリケーションの一部ではないコードを計測することもできます。X-Ray デーモンが実行されている場合、X-Ray SDK によって生成されない場合でも、デーモンは受信したすべてのセグメントを X-Ray に中継します。Scorekeep は独自のスクリプトを使用して、展開中にアプリケーションをコンパイルするビルドを実装します。

Example **bin/build.sh** – 実装されたビルドスクリプト

```
SEGMENT=$(python bin/xray_start.py)
gradle build --quiet --stacktrace &> /var/log/gradle.log; GRADLE_RETURN=$?
if (( GRADLE_RETURN != 0 )); then
  echo "Gradle failed with exit status $GRADLE_RETURN" >&2
  python bin/xray_error.py "$SEGMENT" "$(cat /var/log/gradle.log)"
  exit 1
fi
python bin/xray_success.py "$SEGMENT"
```

xray_start.py、xray_error.py、および xray_success.py は、セグメントオブジェクトを構築し、JSON 文書に変換し、UDP 経由でデーモンに送信する単純な Python スクリプトです。Gradle のビルドに失敗した場合は、X-Ray コンソールトレースマップの [scorekeep-build] ノードをクリックして、エラーメッセージを見つけることができます。





ウェブアプリケーションクライアントの実装

xray-cognito ブランチで、Scorekeep は Amazon Cognito を使用して、ユーザーがアカウントを作成し、それを使用してサインインして Amazon Cognito ユーザープールからユーザー情報を取得することができるようにします。ユーザーがサインインすると、Scorekeep は Amazon Cognito ID プールを使用して、 で使用する一時的な AWS 認証情報を取得します AWS SDK for JavaScript。

ID プールは、サインインしたユーザーがトレースデータを AWS X-Rayに書き込むことができるように設定されています。ウェブアプリケーションは、これらの認証情報を使用して、ログインしたユーザーの ID、ブラウザパス、および Scorekeep API への呼び出しのクライアントビューを記録します。

ほとんどの作業は `xray` というサービスクラスで行われます。このサービスクラスは、必要な識別子の生成、進行中のセグメントの作成、セグメントのファイナライズ、セグメントドキュメントの X-Ray API への送信手段を提供します。

Example **public/xray.js** – セグメントの記録とアップロード

```
...
  service.beginSegment = function() {
```

```
    var segment = {};
    var traceId = '1-' + service.getHexTime() + '-' + service.getHexId(24);

    var id = service.getHexId(16);
    var startTime = service.getEpochTime();

    segment.trace_id = traceId;
    segment.id = id;
    segment.start_time = startTime;
    segment.name = 'Scorekeep-client';
    segment.in_progress = true;
    segment.user =  sessionStorage['userid'];
    segment.http = {
      request: {
        url: window.location.href
      }
    };

    var documents = [];
    documents[0] = JSON.stringify(segment);
    service.putDocuments(documents);
    return segment;
  }

  service.endSegment = function(segment) {
    var endTime = service.getEpochTime();
    segment.end_time = endTime;
    segment.in_progress = false;
    var documents = [];
    documents[0] = JSON.stringify(segment);
    service.putDocuments(documents);
  }

  service.putDocuments = function(documents) {
    var xray = new AWS.XRay();
    var params = {
      TraceSegmentDocuments: documents
    };
    xray.putTraceSegments(params, function(err, data) {
      if (err) {
        console.log(err, err.stack);
      } else {
        console.log(data);
      }
```

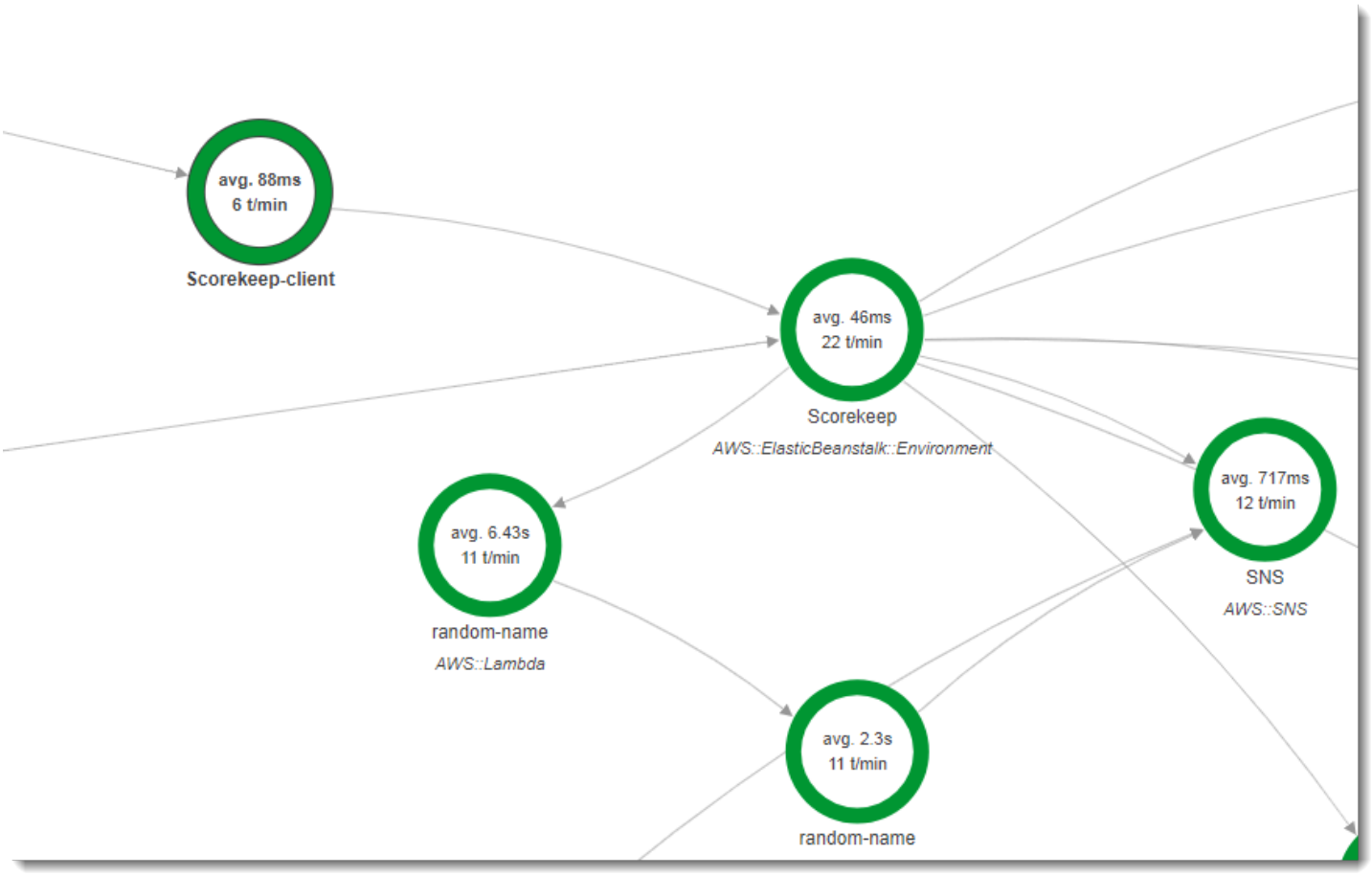
```
    })
  }
```

これらのメソッドは、ウェブアプリケーションが Scorekeep API を呼び出すために使用するリソースサービスのヘッダーおよび `transformResponse` 関数で呼び出されます。API が生成するセグメントと同じトレースにクライアントセグメントを含めるには、ウェブアプリケーションは、X-Ray SDK が読み取り可能なトレースヘッダー (`X-Amzn-Trace-Id`) にトレース ID とセグメント ID を含める必要があります。実装された Java アプリケーションがこのヘッダーでリクエストを受け取ると、X-Ray SDK for Java は同じトレース ID を使用し、ウェブアプリケーションクライアントからのセグメントをセグメントの親にします。

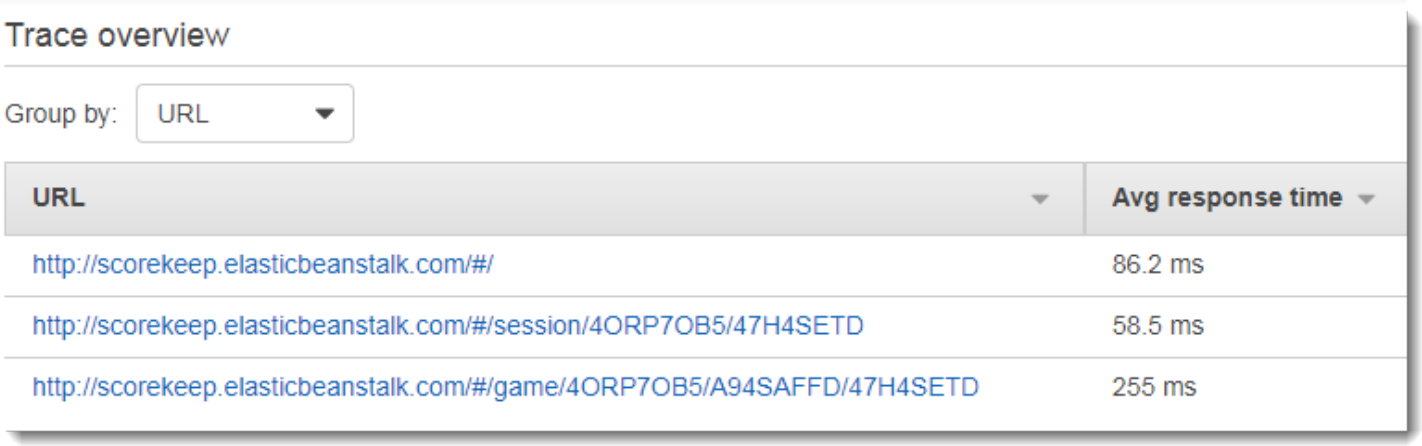
Example **`public/app/services.js`** – 角度リソースコールとトレースヘッダーの書き込みセグメントの記録

```
var module = angular.module('scorekeep');
module.factory('SessionService', function($resource, api, XRay) {
  return $resource(api + 'session/:id', { id: '@_id' }, {
    segment: {},
    get: {
      method: 'GET',
      headers: {
        'X-Amzn-Trace-Id': function(config) {
          segment = XRay.beginSegment();
          return XRay.getTraceHeader(segment);
        }
      },
      transformResponse: function(data) {
        XRay.endSegment(segment);
        return angular.fromJson(data);
      },
    },
...
```

表示されるトレースマップには、ウェブアプリケーションクライアントのノードが含まれます。



ウェブアプリケーションからのセグメントを含むトレースには、ユーザーがブラウザに表示する URL (/#/ で始まるパス) が表示されます。クライアント実装機能がなければ、ウェブアプリケーションが呼び出す API リソース (/api/ で始まるパス) の URL のみを取得します。



実装されたクライアントをワーカースレッドで使用する

Scorekeep は、ユーザーがゲームに勝利したときにワーカースレッドを使用して Amazon SNS に通知を発行します。通知の発行は、残りのリクエスト操作の組み合わせよりも時間がかかり、クライアントまたはユーザーには影響しません。したがって、タスクを非同期で実行することは、応答時間を改善するための良い方法です。

ただし、X-Ray SDK for Java は、スレッドが作成されたときにどのセグメントがアクティブであったかを認識しません。その結果、スレッド内で計測された AWS SDK for Java クライアントを使用しようとすると、 がスローされSegmentNotFoundException、スレッドがクラッシュします。

Example web-1.error.log

```
Exception in thread "Thread-2" com.amazonaws.xray.exceptions.SegmentNotFoundException:
 Failed to begin subsegment named 'AmazonSNS': segment cannot be found.
        at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
        at
 sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:62)
        at
 sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.ja
...
```

この問題を解決するために、アプリケーションは GetTraceEntity を使用してメインスレッドのセグメントへの参照を取得し、Entity.run() を使用してセグメントのコンテキストにアクセスし、ワーカースレッドコードを安全に実行します。

Example **src/main/java/scorekeep/MoveFactory.java** – ワーカースレッドにトレースコンテキストを渡す

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.AWSXRayRecorder;
import com.amazonaws.xray.entities.Entity;
import com.amazonaws.xray.entities.Segment;
import com.amazonaws.xray.entities.Subsegment;
...
      Entity segment = recorder.getTraceEntity();
      Thread comm = new Thread() {
        public void run() {
         segment.run(() -> {
            Subsegment subsegment = AWSXRay.beginSubsegment("## Send notification");
            Sns.sendNotification("Scorekeep game completed", "Winner: " + userId);
```

```
            AWSXRay.endSubsegment();
        }
    }
```

Amazon SNS の呼び出しの前に要求が解決されるため、アプリケーションはスレッド用に別のサブセグメントを作成します。これにより、Amazon SNS からの応答を記録する前に X-Ray SDK がセグメントを閉じるのを防ぎます。Scorekeep がリクエストを解決したときにサブセグメントが開いていない場合、Amazon SNS からの応答は失われる可能性があります。

マルチスレッドの詳細については、「マルチスレッドアプリケーションでのスレッド間のセグメントコンテキストの受け渡し」を参照してください。

AWS X-Ray デーモン

> **Note**
>
> CloudWatch エージェントを使用して Amazon EC2 インスタンスとオンプレミスサーバーからメトリクス、ログ、トレースを収集できます。CloudWatch エージェントバージョン 1.300025.0 以降では、OpenTelemetry または X-Ray クライアント SDK からトレースを収集し、それらを X-Ray に送信できます。 AWS Distro for OpenTelemetry (ADOT) コレクターまたは X-Ray デーモンの代わりに CloudWatch エージェントを使用してトレースを収集することで、管理するエージェントの数を減らすことができます。詳細については、「CloudWatch ユーザーガイド」の「CloudWatch エージェント」のトピックを参照してください。

AWS X-Ray デーモンは、UDP ポート 2000 でトラフィックをリッスンし、raw セグメントデータを収集して AWS X-Ray API に中継するソフトウェアアプリケーションです。デーモンは AWS X-Ray SDKs と連携して動作し、SDKs によって送信されたデータが X-Ray サービスに到達できるように実行されている必要があります。X-Ray デーモンは、オープンソースプロジェクトです。プロジェクトに従って、GitHub github.com/aws/aws-xray-daemon で問題とプルリクエストを送信できます。

AWS Lambda および では AWS Elastic Beanstalk、これらのサービスの X-Ray との統合を使用してデーモンを実行します。Lambda は、サンプルリクエスト用に関数が呼び出される度に自動的にデーモンを実行します。Elastic Beanstalk では、`XRayEnabled`設定オプションを使用して、環境のインスタンスでデーモンを実行します。詳細については「」を参照してください。

X-Ray デーモンをローカル、オンプレミス、またはその他の場所で実行するには AWS のサービス、ダウンロードして実行し、セグメントドキュメントを X-Ray にアップロードするアクセス許可を付与します。

デーモンのダウンロード

デーモンは、Amazon S3、Amazon ECR、または Docker Hub からダウンロードしてローカルで実行するか、起動時に Amazon EC2 インスタンスにインストールします。

Amazon S3

X-Ray デーモンのインストーラおよび実行ファイル

- Linux (実行可能ファイル) – `aws-xray-daemon-linux-3.x.zip` (sig)

- Linux (RPM インストーラ) – aws-xray-daemon-3.x.rpm
- Linux (DEB インストーラ) – aws-xray-daemon-3.x.deb
- Linux (ARM64、実行可能ファイル) – aws-xray-daemon-linux-arm64-3.x.zip (sig)
- Linux (ARM64、RPM インストーラ) – aws-xray-daemon-arm64-3.x.rpm
- Linux (ARM64、DEB インストーラ) – aws-xray-daemon-arm64-3.x.deb
- OS X (実行可能ファイル) – aws-xray-daemon-macos-3.x.zip (sig)
- Windows (実行可能ファイル) – aws-xray-daemon-windows-process-3.x.zip (sig)
- Windows (サービス) – aws-xray-daemon-windows-service-3.x.zip (sig)

これらのリンクは、常にデーモンの最新の3.xリリースを指しています。特定のリリースをダウンロードするには、以下を実行します。

- バージョン 3.3.0 より前のリリースをダウンロードする場合は、3.x をそのバージョン番号に置き換えます。例えば、2.1.0 と指定します。3.3.0 より前のバージョンでは、使用可能なアーキテクチャは arm64 のみです。例えば、2.1.0 と arm64 です。
- バージョン 3.3.0 より後のリリースをダウンロードする場合は、3.x をそのバージョン番号に、arch をアーキテクチャタイプに置き換えます。

X-Ray アセットは、サポートされている各リージョンのバケットにレプリケートされます。お客様またはお客様の AWS リソースに最も近いバケットを使用するには、上記のリンクのリージョンをお客様のリージョンに置き換えます。

```
https://s3.us-west-2.amazonaws.com/aws-xray-assets.us-west-2/xray-daemon/aws-xray-daemon-3.x.rpm
```

Amazon ECR

バージョン 3.2.0 以降は、デーモンはAmazon ECRに掲載されています。イメージを引っ張る前にAmazon ECR パブリックレジストリにDocker クライアントを認証する必要があります。

次のコマンドを実行して、最新のリリース 3.x バージョンタグを引き出します。

```
docker pull public.ecr.aws/xray/aws-xray-daemon:3.x
```

以前のリリースまたはアルファ版は、3.xとalphaまたは特定のバージョン番号に置き換えてダウンロードできます。本番環境では、アルファタグ付きのデーモンイメージを使用することは推奨されません。

Docker Hub

デーモンは、Docker Hubで見ることができます。次のコマンドを実行して、最新リリースの3.xバージョンをダウンロードします。

```
docker pull amazon/aws-xray-daemon:3.x
```

デーモンの以前のリリースは、3.x希望のバージョンに置き換えてリリースすることができます。

デーモンアーカイブの署名の確認

GPG 署名ファイルは、ZIP アーカイブで圧縮されたデーモンアセットで使用するために含まれています。ホストのパブリックキーは、次の場所にあります。aws-xray.gpg

公開鍵を使用して、デーモンの ZIP アーカイブがオリジナルで変更されていないことを確認できます。まず、GnuPG で公開鍵をインポートします。

パブリックキーをインポートするには

1. 公開鍵をダウンロードします。

```
$ BUCKETURL=https://s3.us-east-2.amazonaws.com/aws-xray-assets.us-east-2
$ wget $BUCKETURL/xray-daemon/aws-xray.gpg
```

2. 公開鍵をキーリングにインポートします。

```
$ gpg --import aws-xray.gpg
gpg: /Users/me/.gnupg/trustdb.gpg: trustdb created
gpg: key 7BFE036BFE6157D3: public key "AWS X-Ray <aws-xray@amazon.com>" imported
gpg: Total number processed: 1
gpg:               imported: 1
```

インポートされたキーを使用してデーモンの ZIP アーカイブの署名を確認します。

アーカイブの署名を確認するには

1. アーカイブおよび署名ファイルをダウンロードします。

```
$ BUCKETURL=https://s3.us-east-2.amazonaws.com/aws-xray-assets.us-east-2
$ wget $BUCKETURL/xray-daemon/aws-xray-daemon-linux-3.x.zip
$ wget $BUCKETURL/xray-daemon/aws-xray-daemon-linux-3.x.zip.sig
```

2. `gpg --verify` を実行して署名を確認します。

```
$ gpg --verify aws-xray-daemon-linux-3.x.zip.sig aws-xray-daemon-linux-3.x.zip
gpg: Signature made Wed 19 Apr 2017 05:06:31 AM UTC using RSA key ID FE6157D3
gpg: Good signature from "AWS X-Ray <aws-xray@amazon.com>"
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: EA6D 9271 FBF3 6990 277F  4B87 7BFE 036B FE61 57D3
```

信頼性に関する警告に注意します。自分や信頼する人が署名した場合、鍵は信頼されます。これは、署名が無効であることを意味するものではなく、公開鍵を確認していないことを意味します。

デーモンを実行する

コマンドラインからローカルでデーモンを実行します。ローカルモードで実行するには `-o` オプションを、リージョンを設定するには `-n` を使用します。

```
~/Downloads$ ./xray -o -n us-east-2
```

プラットフォーム固有の詳細な手順については、以下のトピックを参照してください。

- Linux (ローカル) – Linux で X-Ray デーモンを実行する
- Windows (ローカル) – Windows で X-Ray デーモンを実行する
- Elastic Beanstalk – での X-Ray デーモンの実行 AWS Elastic Beanstalk
- Amazon EC2 – Amazon EC2 での X-Ray デーモンの実行
- Amazon ECS – Amazon ECS での X-Ray デーモンの実行

コマンドラインオプションまたは設定ファイルを使用して、デーモンの動作をさらにカスタマイズできます。詳細については、「AWS X-Ray デーモンの設定」を参照してください。

X-Rayにデータを送信するアクセス権限をデーモンに付与する

X-Ray デーモンは AWS SDK を使用してトレースデータを X-Ray にアップロードするため、そのためのアクセス許可を持つ認証情報が必要です AWS 。

Amazon EC2 では、デーモンはインスタンスのインスタンスプロファイルのロールを自動的に使用します。デーモンをローカルで実行するために必要な認証情報については、「アプリケーションをローカルで実行する」を参照してください。

1 つ以上の場所 (認証情報ファイル、インスタンスプロファイル、または環境変数) で認証情報を指定する場合、SDK プロバイダーチェーンは、使用される認証情報を決定します。SDK に認証情報を提供する方法の詳細については、 SDK for Go 開発者ガイドのAWS 認証情報の指定を参照してください。

デーモンの認証情報が属している IAM ロールまたはユーザーには、サービスにデータを書き込むアクセス権限が必要です。

- Amazon EC2 でデーモンを使用するには、新しいインスタンスプロファイルのロールを作成するか、既存のロールに管理ポリシーを追加します。
- Elastic Beanstalk でデーモンを使用するには、管理ポリシーを Elastic Beanstalk のデフォルトのインスタンスプロファイルのロールに追加します。
- デーモンをローカルで実行するには、「アプリケーションをローカルで実行する」を参照してください。

詳細については、「の ID とアクセスの管理 AWS X-Ray」を参照してください。

X-Ray デーモンログ

デーモンは、送信先の現在の設定とセグメントに関する情報を出力します AWS X-Ray。

```
2016-11-24T06:07:06Z [Info] Initializing AWS X-Ray daemon 2.1.0
2016-11-24T06:07:06Z [Info] Using memory limit of 49 MB
2016-11-24T06:07:06Z [Info] 313 segment buffers allocated
2016-11-24T06:07:08Z [Info] Successfully sent batch of 1 segments (0.123 seconds)
2016-11-24T06:07:09Z [Info] Successfully sent batch of 1 segments (0.006 seconds)
```

デフォルトでは、デーモンはログを STDOUT に出力します。デーモンをバックグラウンドで実行する場合は、`--log-file` コマンドラインオプションまたは設定ファイルを使用してログファイルパ

スを設定します。ログレベルを設定し、ログローテーションを無効にすることもできます。手順については「AWS X-Ray デーモンの設定」を参照してください。

Elastic Beanstalk では、プラットフォームはデーモンログの場所を設定します。詳細については、「での X-Ray デーモンの実行 AWS Elastic Beanstalk」を参照してください。

AWS X-Ray デーモンの設定

コマンドラインオプションまたは設定ファイルを使用して、X-Ray デーモンの動作をカスタマイズできます。ほとんどのオプションは両方の方法を使用して利用できますが、一部は設定ファイルのみを使用でき、一部はコマンドラインのみを使用できます。

開始するには、-n または--regionのみが必要です。デーモンがトレースデータをX-Ray に送信するために使用するリージョンを設定するために使用します。

```
~/xray-daemon$ ./xray -n us-east-2
```

デーモンを、Amazon EC2 ではなく、ローカルで実行している場合、-o オプションを追加すると、デーモンをより迅速に開始できるように、インスタンスプロファイルの認証情報のチェックをスキップできます。

```
~/xray-daemon$ ./xray -o -n us-east-2
```

残りのコマンドラインオプションを使用して、ロギングの設定、別のポートでのリッスン、デーモンが使用できるメモリ量の制限、別のアカウントにトレースデータを送信するロールの割り当てができます。

設定ファイルをデーモンに渡して、詳細な設定オプションにアクセスしたり、X-Ray への同時呼び出し数を制限したり、ログローテーションを無効にしたり、プロキシにトラフィックを送信したりすることができます。

セクション

- サポートされている環境変数
- コマンドラインオプションを使用する
- 設定ファイルを使用する

サポートされている環境変数

X-Ray デーモンは次の環境変数をサポートしています。

- AWS_REGION – X-Ray サービスエンドポイントの AWS リージョン を指定します。
- HTTPS_PROXY セグメントをアップロードするデーモンのプロキシアドレスを指定します。これは、DNS ドメイン名または IP アドレス、あるいはプロキシサーバーで使用されているポート番号のいずれかです。

コマンドラインオプションを使用する

ローカルで実行するか、またはユーザーデータスクリプトを使用して、これらのオプションをデーモンに渡します。

コマンドラインオプション

- -b、--bind 別の UDP ポートでセグメントドキュメントをリッスンします。

  ```
  --bind "127.0.0.1:3000"
  ```

  デフォルト – 2000
- -t、 --bind-tcp 別の TCP ポートで X-Ray サービスへの呼び出しをリッスンします。

  ```
  -bind-tcp "127.0.0.1:3000"
  ```

  デフォルト – 2000
- -c、 --config 指定されたパスから設定ファイルをロードします。

  ```
  --config "/home/ec2-user/xray-daemon.yaml"
  ```
- -f、 --log-file 指定されたファイルパスにログを出力します。

  ```
  --log-file "/var/log/xray-daemon.log"
  ```
- -l、 --log-levelログレベルを dev、debug、info、warn、error、prod (詳細な順) から指定します。

  ```
  --log-level warn
  ```

デフォルト – prod

- -m、 --buffer-memoryバッファが使用できるメモリの量をメガバイト単位で変更します (最小値 3)。

```
--buffer-memory 50
```

デフォルト – 使用可能なメモリ総量の 1%

- -o、 --local-mode– EC2 インスタンスのメタデータをチェックしません。
- -r、 --role-arn– 指定した IAM ロールで、別のアカウントにセグメントをアップロードできるようにします。

```
--role-arn "arn:aws:iam::123456789012:role/xray-cross-account"
```

- -a、 --resource-arn – デーモンを実行するリソースの Amazon AWS リソースネーム (ARN)。
- -p、 --proxy-address — プロキシ AWS X-Ray を介してセグメントを にアップロードします。プロキシサーバーのプロトコルを指定する必要があります。

```
--proxy-address "http://192.0.2.0:3000"
```

- -n、--region 特定のリージョンの X-Ray サービスにセグメントを送信します。
- -v、 --version – AWS X-Ray デーモンのバージョンを表示します。
- -h、--help – ヘルプ画面を表示します。

設定ファイルを使用する

YAML 形式のファイルを使用して、デーモンを設定することもできます。-c オプションを使用して、設定ファイルをデーモンに渡します。

```
~$ ./xray -c ~/xray-daemon.yaml
```

設定ファイルのオプション

- TotalBufferSizeMB – 最大バッファサイズ (MB) (最小値 3)。ホストメモリの 1% を使用するには、0 を選択します。
- Concurrency – セグメントドキュメントをアップロード AWS X-Ray するための への同時呼び出しの最大数。

- Region – 特定のリージョン AWS X-Ray のサービスにセグメントを送信します。
- Socket – デーモンのバインディングを設定します。
  - UDPAddress – デーモンがリッスンするポートを変更します。
  - TCPAddress – 別の TCP ポートで X-Ray サービスへの呼び出しをリッスンします。
- Logging – ログ記録の動作を設定します。
  - LogRotation – falseに設定してログローテーションを無効にします。
  - LogLevel - ログレベルを変更します。詳細なものから順に、dev、debug、info、または prod、warn、error、prod です。デフォルトは です。prodと等価である。info。
  - LogPath – 指定されたファイルパスにログを出力します。
- LocalMode–EC2 インスタンスのメタデータのチェックをスキップするには true に設定します。
- ResourceARN – デーモンを実行するリソースの Amazon AWS リソースネーム (ARN)。
- RoleARN – 指定した IAM ロールで、別のアカウントにセグメントをアップロードできるようにします。
- ProxyAddress – プロキシ AWS X-Ray を介してセグメントを にアップロードします。
- Endpoint – デーモンがセグメントドキュメントを送信する X-Ray サービスエンドポイントを変更します。
- NoVerifySSL – TLS 証明書認証を無効にします。
- Version – デーモンの設定ファイル形式のバージョン。ファイル形式のバージョンは必須フィールド。

Example xray-daemon.yaml

この設定ファイルでは、デーモンのリスニングポートを 3000 に変更し、インスタンスメタデータの確認をオフにして、セグメントをアップロードするために使用するロールを設定し、リージョンおよびログ作成オプションを変更します。

```
Socket:
  UDPAddress: "127.0.0.1:3000"
  TCPAddress: "127.0.0.1:3000"
Region: "us-west-2"
Logging:
  LogLevel: "warn"
  LogPath: "/var/log/xray-daemon.log"
LocalMode: true
```

```
RoleARN: "arn:aws:iam::123456789012:role/xray-cross-account"
Version: 2
```

ローカルで X-Ray; デーモンを実行する

AWS X-Ray デーモンは、Linux、MacOS、Windows、または Docker コンテナでローカルに実行できます。実装されたアプリケーションを開発してテストするときに、デーモンを実行してトレースデータを X-Ray に中継します。ここに示す手順を使用して、デーモンをダウンロードして解凍します。

ローカルで実行する場合、デーモンは AWS SDK 認証情報ファイル (`.aws/credentials` ユーザーディレクトリ) または環境変数から認証情報を読み取ることができます。詳細については、「X-Ray にデータを送信するアクセス権限をデーモンに付与する」を参照してください。

デーモンはポート 2000 の UDP データをリッスンします。ポートおよびその他のオプションは、設定ファイルとコマンドラインオプションを使用して変更できます。詳細については、「AWS X-Ray デーモンの設定」を参照してください。

Linux で X-Ray デーモンを実行する

コマンドラインからデーモンの実行可能ファイルを実行できます。ローカルモードで実行するには `-o` オプションを、リージョンを設定するには `-n` を使用します。

```
~/xray-daemon$ ./xray -o -n us-east-2
```

バックグラウンドでデーモンを実行するには、`&` を使用します。

```
~/xray-daemon$ ./xray -o -n us-east-2 &
```

`pkill` を使用してバックグラウンドで実行されるデーモンプロセスを終了します。

```
~$ pkill xray
```

Docker コンテナで X-Ray デーモンを実行する

Docker コンテナでデーモンをローカルで実行するには、`Dockerfile` という名前のファイルに次のテキストを保存します。Amazon ECR で、完全なイメージ例をダウンロードします。詳細については、「デーモンのダウンロード」を参照してください。

Example Dockerfile — Amazon Linux

```
FROM amazonlinux
RUN yum install -y unzip
RUN curl -o daemon.zip https://s3.us-east-2.amazonaws.com/aws-xray-assets.us-east-2/
xray-daemon/aws-xray-daemon-linux-3.x.zip
RUN unzip daemon.zip && cp xray /usr/bin/xray
ENTRYPOINT ["/usr/bin/xray", "-t", "0.0.0.0:2000", "-b", "0.0.0.0:2000"]
EXPOSE 2000/udp
EXPOSE 2000/tcp
```

docker build を使用してコンテナイメージを構築します。

```
~/xray-daemon$ docker build -t xray-daemon .
```

docker run を使用してコンテナでイメージを実行します。

```
~/xray-daemon$ docker run \
      --attach STDOUT \
      -v ~/.aws/:/root/.aws/:ro \
      --net=host \
      -e AWS_REGION=us-east-2 \
      --name xray-daemon \
      -p 2000:2000/udp \
      xray-daemon -o
```

このコマンドでは、以下のオプションを使用します。

- --attach STDOUT – ターミナルにデーモンからの出力を表示します。
- -v ~/.aws/:/root/.aws/:ro – .aws ディレクトリへの読み取り専用アクセスをコンテナに付与し、 AWS SDK 認証情報を読み取らせます。
- AWS_REGION=*us-east-2* – AWS_REGION環境変数を設定して、使用するリージョンをデーモンに指定します。
- --net=host – コンテナを host ネットワークにアタッチします。ホストネットワーク上のコンテナは、ポートを公開しなくても相互に通信できます。
- -p 2000:2000/udp – マシン上の UDP ポート 2000 をコンテナの同じポートにマップします。これは、同じネットワークのコンテナが通信するために必要ではありませんが、これにより、コマンドラインまたは Docker で実行されていないアプリケーションからセグメントをデーモンに送信できます。

- `--name xray-daemon` – ランダムな名前を生成する代わりに、コンテナに `xray-daemon` という名前を付けます。
- `-o` (イメージ名の後ろ) – コンテナ内でデーモンを実行するエントリポイントに `-o` オプションを追加します。このオプションは、Amazon EC2インスタンスのメタデータを読み取らないようにするため、ローカルモードで実行するようにデーモンに指示します。

デーモンを停止するには、`docker stop` を使用します。Dockerfile を変更して新しいイメージを作成する場合は、同じ名前の別のコンテナを作成する前に、既存のコンテナを削除する必要があります。`docker rm` を使用してコンテナを削除します。

```
$ docker stop xray-daemon
$ docker rm xray-daemon
```

Windows で X-Ray デーモンを実行する

コマンドラインからデーモンの実行可能ファイルを実行できます。ローカルモードで実行するには `-o` オプションを、リージョンを設定するには `-n` を使用します。

```
> .\xray_windows.exe -o -n us-east-2
```

PowerShell スクリプトを使用してデーモンのサービスを作成および実行します。

Example PowerShell スクリプト - Windows

```
if ( Get-Service "AWSXRayDaemon" -ErrorAction SilentlyContinue ){
    sc.exe stop AWSXRayDaemon
    sc.exe delete AWSXRayDaemon
}
if ( Get-Item -path aws-xray-daemon -ErrorAction SilentlyContinue ) {
    Remove-Item -Recurse -Force aws-xray-daemon
}

$currentLocation = Get-Location
$zipFileName = "aws-xray-daemon-windows-service-3.x.zip"
$zipPath = "$currentLocation\$zipFileName"
$destPath = "$currentLocation\aws-xray-daemon"
$daemonPath = "$destPath\xray.exe"
$daemonLogPath = "C:\inetpub\wwwroot\xray-daemon.log"
$url = "https://s3.dualstack.us-west-2.amazonaws.com/aws-xray-assets.us-west-2/xray-
daemon/aws-xray-daemon-windows-service-3.x.zip"
```

```
Invoke-WebRequest -Uri $url -OutFile $zipPath
Add-Type -Assembly "System.IO.Compression.Filesystem"
[io.compression.zipfile]::ExtractToDirectory($zipPath, $destPath)

sc.exe create AWSXRayDaemon binPath= "$daemonPath -f $daemonLogPath"
sc.exe start AWSXRayDaemon
```

OS X で X-Ray デーモンを実行する

コマンドラインからデーモンの実行可能ファイルを実行できます。ローカルモードで実行するには -o オプションを、リージョンを設定するには -n を使用します。

```
~/xray-daemon$ ./xray_mac -o -n us-east-2
```

バックグラウンドでデーモンを実行するには、& を使用します。

```
~/xray-daemon$ ./xray_mac -o -n us-east-2 &
```

ターミナル終了時にデーモンが終了されるのを防止するには、nohup を使用します。

```
~/xray-daemon$ nohup ./xray_mac &
```

での X-Ray デーモンの実行 AWS Elastic Beanstalk

アプリケーションから にトレースデータを中継するには AWS X-Ray、Elastic Beanstalk 環境の Amazon EC2 インスタンスで X-Ray デーモンを実行します。サポートされているプラットフォームの一覧は、[AWS Elastic Beanstalk Developer Guide] (開発者ガイド) の [Configuring AWS X-Ray Debugging] (デバッグ設定)を参照してください。

> **Note**
>
> デーモンは、環境のインスタンスプロファイルのアクセス権限を使用します。Elastic Beanstalk インスタンスプロファイルにアクセス権限を追加する方法については、X-Rayにデータを送信するアクセス権限をデーモンに付与する を参照してください。

Elastic Beanstalk プラットフォームには、デーモンを自動的に実行する設定オプションがあります。ソースコードの設定ファイル内で、または Elastic Beanstalk コンソールでオプションを選択して、

デーモンを有効にできます。設定オプションを有効にすると、デーモンはインスタンスにインストールされ、サービスとして実行されます。

Elastic Beanstalk プラットフォームに含まれているバージョンは最新バージョンではない場合があります。サポートされているプラットフォームのトピックを参照して、プラットフォーム設定で利用できるデーモンのバージョンを確認してください。

Elastic Beanstalk は、複数コンテナの Docker (Amazon ECS) プラットフォームに X-Ray デーモンを提供しません。

Elastic Beanstalk X-Ray 統合を使用して X-Ray デーモンの実行します。

コンソールを使用して X-Ray 統合をオンにするか、設定ファイルを使用してアプリケーションソースコードで設定します。

Elastic Beanstalk コンソールで X-Ray デーモンを有効にするには

1. [Elastic Beanstalk console] (Elastic Beanstalk コンソール) を開いてください。
2. 環境に対応するマネジメントコンソールに移動します。
3. [設定] を選択します。
4. [Software Settings] を選択します。
5. [X-Ray daemon] で、[Enabled] を選択します。
6. [Apply] を選択します。

ソースコードに設定ファイルを含めて、設定を環境間で移植可能にできます。

Example .ebextensions/xray-daemon.config

```
option_settings:
  aws:elasticbeanstalk:xray:
    XRayEnabled: true
```

Elastic Beanstalkは設定ファイルをデーモンに渡し、ログを標準の場所に出力します。

Windows Server プラットフォーム

- [Configuration file] (設定ファイル) – `C:\Program Files\Amazon\XRay\cfg.yaml`
- [Logs] (ログ) – `c:\Program Files\Amazon\XRay\logs\xray-service.log`

Linux プラットフォーム

- [Configuration file] (設定ファイル) – `/etc/amazon/xray/cfg.yaml`
- [Logs] (ログ) – `/var/log/xray/xray.log`

Elastic Beanstalk には、 AWS Management Console または コマンドラインからインスタンスログをプルするためのツールが用意されています。設定ファイルでタスクを追加して、X-Ray デーモンログを含めるように Elastic Beanstalk に伝えることができます。

Example .ebextensions/xray-logs.config - Linux

```
files:
  "/opt/elasticbeanstalk/tasks/taillogs.d/xray-daemon.conf" :
    mode: "000644"
    owner: root
    group: root
    content: |
      /var/log/xray/xray.log
```

Example .ebextensions/xray-logs.config - Windows Server

```
files:
  "c:/Program Files/Amazon/ElasticBeanstalk/config/taillogs.d/xray-daemon.conf" :
    mode: "000644"
    owner: root
    group: root
    content: |
      c:\Progam Files\Amazon\XRay\logs\xray-service.log
```

詳細については、[AWS Elastic Beanstalk Developer Guide] (開発者ガイド)の[Viewing Logs from Your Elastic Beanstalk Environment's Amazon EC2 Instances] (Elastic Beanstalk 環境のAmazon EC2 インスタンスからのログの表示) を参照してください。

X-Ray デーモンを手動でダウンロードして実行します (上級編)

X-Ray デーモンをプラットフォーム設定で使用できない場合は、Amazon S3 からダウンロードして、設定ファイルを使用して実行できます。

Elastic Beanstalk 設定 ファイルを使用して、デーモンをダウンロードして実行します。

Example .ebextensions/xray.config - Linux

```
commands:
  01-stop-tracing:
    command: yum remove -y xray
    ignoreErrors: true
  02-copy-tracing:
    command: curl https://s3.us-east-2.amazonaws.com/aws-xray-assets.us-east-2/xray-
daemon/aws-xray-daemon-3.x.rpm -o /home/ec2-user/xray.rpm
  03-start-tracing:
    command: yum install -y /home/ec2-user/xray.rpm

files:
  "/opt/elasticbeanstalk/tasks/taillogs.d/xray-daemon.conf" :
    mode: "000644"
    owner: root
    group: root
    content: |
      /var/log/xray/xray.log
  "/etc/amazon/xray/cfg.yaml" :
    mode: "000644"
    owner: root
    group: root
    content: |
      Logging:
        LogLevel: "debug"
      Version: 2
```

Example .ebextensions/xray.config - Windows server

```
container_commands:
  01-execute-config-script:
    command: Powershell.exe -ExecutionPolicy Bypass -File c:\\temp\\installDaemon.ps1
    waitAfterCompletion: 0

files:
  "c:/temp/installDaemon.ps1":
    content: |
      if ( Get-Service "AWSXRayDaemon" -ErrorAction SilentlyContinue ) {
          sc.exe stop AWSXRayDaemon
          sc.exe delete AWSXRayDaemon
      }
```

```
      $targetLocation = "C:\Program Files\Amazon\XRay"
      if ((Test-Path $targetLocation) -eq 0) {
          mkdir $targetLocation
      }

      $zipFileName = "aws-xray-daemon-windows-service-3.x.zip"
      $zipPath = "$targetLocation\$zipFileName"
      $destPath = "$targetLocation\aws-xray-daemon"
      if ((Test-Path $destPath) -eq 1) {
          Remove-Item -Recurse -Force $destPath
      }

      $daemonPath = "$destPath\xray.exe"
      $daemonLogPath = "$targetLocation\xray-daemon.log"
      $url = "https://s3.dualstack.us-west-2.amazonaws.com/aws-xray-assets.us-west-2/
xray-daemon/aws-xray-daemon-windows-service-3.x.zip"

      Invoke-WebRequest -Uri $url -OutFile $zipPath
      Add-Type -Assembly "System.IO.Compression.Filesystem"
      [io.compression.zipfile]::ExtractToDirectory($zipPath, $destPath)

      New-Service -Name "AWSXRayDaemon" -StartupType Automatic -BinaryPathName
 "`"$daemonPath`" -f `"$daemonLogPath`""
      sc.exe start AWSXRayDaemon
    encoding: plain
  "c:/Program Files/Amazon/ElasticBeanstalk/config/taillogs.d/xray-daemon.conf" :
    mode: "000644"
    owner: root
    group: root
    content: |
      C:\Program Files\Amazon\XRay\xray-daemon.log
```

これらの例ではまた、デーモンのログファイルを のログ末尾タスクに追加し、コンソールまたは Elastic Beanstalk コマンドラインインターフェイス (EB CLI) を使用したログのリクエスト時にその内容を含めるようにしています。

Amazon EC2 での X-Ray デーモンの実行

Amazon EC2 上の次のオペレーティングシステムで X-Ray デーモンを実行できます。

- Amazon Linux
- Ubuntu

- Windows Server (2012 R2 以降)

インスタンスプロファイルを使用して、トレースデータを X-Ray にアップロードするアクセス権限をデーモンに付与します。詳細については、「X-Rayにデータを送信するアクセス権限をデーモンに付与する」を参照してください。

インスタンスを起動するときに、ユーザーデータのスクリプトを使用して自動的にデーモンを実行します。

Example ユーザーデータスクリプト - Linux

```
#!/bin/bash
curl https://s3.us-east-2.amazonaws.com/aws-xray-assets.us-east-2/xray-daemon/aws-xray-daemon-3.x.rpm -o /home/ec2-user/xray.rpm
yum install -y /home/ec2-user/xray.rpm
```

Example ユーザーデータスクリプト - Windows Server

```
<powershell>
if ( Get-Service "AWSXRayDaemon" -ErrorAction SilentlyContinue ) {
    sc.exe stop AWSXRayDaemon
    sc.exe delete AWSXRayDaemon
}

$targetLocation = "C:\Program Files\Amazon\XRay"
if ((Test-Path $targetLocation) -eq 0) {
    mkdir $targetLocation
}

$zipFileName = "aws-xray-daemon-windows-service-3.x.zip"
$zipPath = "$targetLocation\$zipFileName"
$destPath = "$targetLocation\aws-xray-daemon"
if ((Test-Path $destPath) -eq 1) {
    Remove-Item -Recurse -Force $destPath
}

$daemonPath = "$destPath\xray.exe"
$daemonLogPath = "$targetLocation\xray-daemon.log"
$url = "https://s3.dualstack.us-west-2.amazonaws.com/aws-xray-assets.us-west-2/xray-daemon/aws-xray-daemon-windows-service-3.x.zip"

Invoke-WebRequest -Uri $url -OutFile $zipPath
```

```
Add-Type -Assembly "System.IO.Compression.Filesystem"
[io.compression.zipfile]::ExtractToDirectory($zipPath, $destPath)

New-Service -Name "AWSXRayDaemon" -StartupType Automatic -BinaryPathName
 "`"$daemonPath`" -f `"$daemonLogPath`""
sc.exe start AWSXRayDaemon
</powershell>
```

Amazon ECS での X-Ray デーモンの実行

Amazon ECS で、X-Ray デーモンを実行する Docker イメージを作成し、それを Docker イメージリポジトリにアップロードして、 クラスターにデプロイできます。タスク定義ファイルでポートマッピングとネットワークモード設定を使用すると、アプリケーションがデーモンコンテナと通信できるようになります。

公式 Docker イメージの使用

X-Ray は、アプリケーションと一緒にデプロイできる Amazon ECRのDocker コンテナイメージを提供します。詳細については、「デーモンのダウンロード」を参照してください。

Example タスク定義

```
{
  "name": "xray-daemon",
  "image": "amazon/aws-xray-daemon",
  "cpu": 32,
  "memoryReservation": 256,
  "portMappings" : [
      {
          "hostPort": 0,
          "containerPort": 2000,
          "protocol": "udp"
      }
   ]
}
```

Docker イメージの作成と構築

カスタム設定では、独自の Docker イメージの定義が必要になる場合があります。

タスクロールに管理ポリシーを追加して、デーモンにトレースデータを X-Ray にアップロードするアクセス許可を与えます。詳細については、「X-Rayにデータを送信するアクセス権限をデーモンに付与する」を参照してください。

次のいずれかの Dockerfiles を使用して、デーモンを実行するイメージを作成します。

Example Dockerfile — Amazon Linux

```
FROM amazonlinux
RUN yum install -y unzip
RUN curl -o daemon.zip https://s3.us-east-2.amazonaws.com/aws-xray-assets.us-east-2/
xray-daemon/aws-xray-daemon-linux-3.x.zip
RUN unzip daemon.zip && cp xray /usr/bin/xray
ENTRYPOINT ["/usr/bin/xray", "-t", "0.0.0.0:2000", "-b", "0.0.0.0:2000"]
EXPOSE 2000/udp
EXPOSE 2000/tcp
```

> **Note**
>
> マルチコンテナ環境のループバックをリッスンするためのバインディングアドレスを指定するには、フラグ `-t` および `-b` が必要です。

Example Dockerfile - Ubuntu

Debian から派生した OS では、認証機関 (CA) の証明書をインストールして、インストーラのダウンロード時の問題を回避します。

```
FROM ubuntu:16.04
RUN apt-get update && apt-get install -y --force-yes --no-install-recommends apt-
transport-https curl ca-certificates wget && apt-get clean && apt-get autoremove && rm
 -rf /var/lib/apt/lists/*
RUN wget https://s3.us-east-2.amazonaws.com/aws-xray-assets.us-east-2/xray-daemon/aws-
xray-daemon-3.x.deb
RUN dpkg -i aws-xray-daemon-3.x.deb
ENTRYPOINT ["/usr/bin/xray", "--bind=0.0.0.0:2000", "--bind-tcp=0.0.0.0:2000"]
EXPOSE 2000/udp
EXPOSE 2000/tcp
```

タスク定義では、使用するネットワーキングモードによって設定が異なります。デフォルトはブリッジネットワーキングで、デフォルトの VPC で使用できます。ブリッジネット

ワークで、X-Ray SDK に参照先のコンテナポートを指示し、ホストポートを設定するように AWS_XRAY_DAEMON_ADDRESS 環境変数を設定します。たとえば、UDP ポート 2000 を発行し、アプリケーションコンテナからデーモンコンテナへのリンクを作成します。

Example タスク定義

```
    {
      "name": "xray-daemon",
      "image": "123456789012.dkr.ecr.us-east-2.amazonaws.com/xray-daemon",
      "cpu": 32,
      "memoryReservation": 256,
      "portMappings" : [
          {
              "hostPort": 0,
              "containerPort": 2000,
              "protocol": "udp"
          }
       ]
    },
    {
      "name": "scorekeep-api",
      "image": "123456789012.dkr.ecr.us-east-2.amazonaws.com/scorekeep-api",
      "cpu": 192,
      "memoryReservation": 512,
      "environment": [
          { "name" : "AWS_REGION", "value" : "us-east-2" },
          { "name" : "NOTIFICATION_TOPIC", "value" : "arn:aws:sns:us-
east-2:123456789012:scorekeep-notifications" },
          { "name" : "AWS_XRAY_DAEMON_ADDRESS", "value" : "xray-daemon:2000" }
      ],
      "portMappings" : [
          {
              "hostPort": 5000,
              "containerPort": 5000
          }
      ],
      "links": [
        "xray-daemon"
      ]
    }
```

VPC のプライベートサブネットでクラスターを実行する場合は、awsvpc ネットワークモードを使用して、Elastic Network Interface (ENI) をコンテナにアタッチできます。これにより、リンクの使用を避けることができます。ポートマッピング、リンク、および AWS_XRAY_DAEMON_ADDRESS 環境変数でホストポートを省略します。

Example VPC タスク定義

```
{
    "family": "scorekeep",
    "networkMode":"awsvpc",
    "containerDefinitions": [
        {
          "name": "xray-daemon",
          "image": "123456789012.dkr.ecr.us-east-2.amazonaws.com/xray-daemon",
          "cpu": 32,
          "memoryReservation": 256,
          "portMappings" : [
              {
                  "containerPort": 2000,
                  "protocol": "udp"
              }
          ]
        },
        {
            "name": "scorekeep-api",
            "image": "123456789012.dkr.ecr.us-east-2.amazonaws.com/scorekeep-api",
            "cpu": 192,
            "memoryReservation": 512,
            "environment": [
                { "name" : "AWS_REGION", "value" : "us-east-2" },
                { "name" : "NOTIFICATION_TOPIC", "value" : "arn:aws:sns:us-
east-2:123456789012:scorekeep-notifications" }
            ],
            "portMappings" : [
                {
                    "containerPort": 5000
                }
            ]
        }
    ]
}
```

Amazon ECS コンソールでのコマンドラインオプションの構成

コマンドラインオプションは、イメージの設定ファイル内の競合する値を上書きします。コマンドラインオプションは、一般的にローカルテストに使用されますが、環境変数を設定する際の便宜上、または起動プロセスを制御するために使用することもできます。

コマンドラインオプションを追加することで、コンテナに渡される Docker CMD が更新されます。詳細については、Docker run リファレンスを参照してください。

コマンドラインオプションを設定するには

1. Amazon ECS クラシックコンソール (https://console.aws.amazon.com/ecs/) を開きます。
2. ナビゲーションバーから、タスク定義を含むリージョンを選択します。
3. ナビゲーションペインで、[タスク定義] を選択します。
4. [Task Definitions] ページで、変更するタスク定義の左側にあるボックスをオンにし、[Create new revision] を選択します。
5. [Create new revision of Task Definition (タスク定義の新しいリビジョンを作成)] ページで、コンテナを選択します。
6. [ENVIRONMENT (環境)] セクションで、コマンドラインオプションのカンマ区切りリストを [Command (コマンド)] フィールドに追加します。
7. [Update] (更新) を選択します。
8. 情報を確認し、[Create] を選択します。

次の例は、RoleARN オプションのコマンドラインオプションをカンマで区切って記述する方法を示しています。RoleARN オプションは、セグメントを別のアカウントにアップロードするために、指定された IAM ロール を引き受けます。

Example

```
--role-arn, arn:aws:iam::123456789012:role/xray-cross-account
```

X-Ray で使用できるコマンドラインオプションの詳細については、AWS X-Ray 「 デーモンの設定」を参照してください。

他の AWS X-Ray との統合 AWS のサービス

多くの AWS のサービス は、受信リクエストへのヘッダーのサンプリングと追加、X-Ray デーモンの実行、X-Ray へのトレースデータの自動送信など、さまざまなレベルの X-Ray 統合を提供します。X-Rayとの統合には、次のようなものがあります:

- アクティブ計測 – 受信リクエストをサンプリングして計測します。
- パッシブ計測 – 別のサービスで既にサンプリングされているリクエストを計測します。
- リクエストのトレース – すべての受信リクエストにトレースヘッダーを追加してダウンストリームに伝達します。
- ツール – X-Ray デーモンを実行して X-Ray SDK からセグメントを受信します。

> **Note**
>
> X-Ray SDKs には、 との追加統合用のプラグインが含まれています AWS のサービス。たとえば、X-Ray SDK for Java Elastic Beanstalk プラグイン を使用して、アプリケーション、を実行する Elastic Beanstalk 環境に関する情報 (環境名と ID を含む) を追加できます。

X-Ray と統合 AWS のサービス されている の例をいくつか示します。

- AWS Distro for OpenTelemetry (ADOT) – ADOT を使用すると、エンジニアはアプリケーションを一度計測し、相関メトリクスとトレースを Amazon CloudWatch、Amazon OpenSearch Service AWS X-Ray、Amazon Managed Service for Prometheus などの複数の AWS モニタリングソリューションに送信できます。
- AWS Lambda – すべてのランタイムで受信リクエストをアクティブおよびパッシブに計測します。 は、トレースマップに 2 つのノード AWS Lambda を追加します。1 つは AWS Lambda サービス用、もう 1 つは 関数用です。計測を有効にすると、 は X-Ray SDK で使用する X-Ray デーモンを Java および Node.js ランタイム AWS Lambda でも実行します。
- Amazon API Gateway — アクティブおよびパッシブ計測。API Gateway は、サンプリングルールを使用して記録するリクエストを決定し、ゲートウェイステージ用のノードをサービスマップに追加します。
- AWS Elastic Beanstalk – ツール。Elastic Beanstalk には次のプラットフォームで X-Ray デーモンが含まれています。
  - Java SE – 2.3.0 以降の設定

- Tomcat – 2.4.0 以降の設定
- Node.js – 3.2.0 以降の設定
- Windows Server – Windows Server Core を除く、2016 年 12 月 9 日以降にリリースされたすべての設定

Elastic Beanstalk コンソールを使用するか、`aws:elasticbeanstalk:xray` 名前空間で `XRayEnabled` オプションを使用して、これらのプラットフォームでデーモンを実行するように Elastic Beanstalk を設定できます。

- Elastic Load Balancing — Application Load Balancerでトレースを要求します。Application Load Balancerはトレース ID をリクエストヘッダーに追加してからターゲットグループに送信します。
- Amazon EventBridge — パッシブ計測。EventBridge にイベントを発行するサービスが X-Ray SDK で計測されている場合、イベントターゲットはトレースヘッダーを受け取り、元のトレース ID を継続して伝達できます。
- Amazon Simple Notification Service — パッシブ計測。Amazon SNS パブリッシャーが X-Ray SDK クライアントを使用してクライアントをトレースする場合、サブスクライバーはトレースヘッダーを取得し、同じトレース ID を使用して、パブリッシャーからの元のトレースを継続して伝達できます。
- Amazon Simple Queue Service — パッシブ計測。サービスが X-Ray SDK を使用してリクエストをトレースする場合、Amazon SQS はトレースヘッダーを送信し、整合性のあるトレース ID を持つコンシューマーに、送信者から元のトレースを伝達し続けます。
- Amazon Bedrock AgentCore – AgentCore は X-Ray 統合による分散トレースをサポートしているため、エージェントアプリケーションを通過するリクエストを追跡できます。AgentCore リソースのオブザーバビリティを有効にすると、トレースコンテキストをサービス境界全体に伝播し、AI エージェントとツールのパフォーマンスを可視化できます。

以下のトピックから選択して、統合された の完全なセットをご覧ください AWS のサービス。

トピック

- Amazon Bedrock AgentCore と AWS X-Ray
- Amazon Elastic Compute Cloud と AWS X-Ray
- Amazon SNS と AWS X-Ray
- Amazon SQS と AWS X-Ray
- Amazon S3 と AWS X-Ray
- AWS Distro for OpenTelemetry と AWS X-Ray

- を使用した X-Ray 暗号化設定の変更の追跡 AWS Config
- AWS AppSync and AWS X-Ray
- の Amazon API Gateway アクティブトレースのサポート AWS X-Ray
- Amazon EC2 と AWS App Mesh
- AWS App Runner と X-Ray
- を使用した X-Ray API コールのログ記録 AWS CloudTrail
- X-Ray と CloudWatch の統合
- AWS Elastic Beanstalk and AWS X-Ray
- Elastic Load Balancing と AWS X-Ray
- Amazon EventBridge と AWS X-Ray
- AWS Lambda and AWS X-Ray
- AWS Step Functions and AWS X-Ray

Amazon Bedrock AgentCore と AWS X-Ray

Amazon Bedrock AgentCore は と統合 AWS X-Ray され、AI エージェントとツールに分散トレース機能を提供します。この統合により、エージェントアプリケーションを通過するリクエストを追跡できるため、パフォーマンスのボトルネックを特定し、問題をトラブルシューティングできます。

AgentCore は X-Ray 統合による分散トレースをサポートしているため、AI エージェントとツールのパフォーマンスをモニタリングできます。AgentCore リソースのオブザーバビリティを有効にすると、トレースコンテキストをサービス境界全体に伝達し、エージェントが他の AWS サービスとやり取りする方法を可視化できます。詳細については、「Amazon Bedrock AgentCore」を参照してください。

AgentCore は、次の X-Ray 機能をサポートしています。

- ダウンストリームサービスへのトレースコンテキストの伝播
- AWS Distro for OpenTelemetry (ADOT) SDK を使用したカスタム計測

AgentCore を使用した X-Ray のセットアップ

AgentCore で X-Ray を使用するには、 AWS アカウントで CloudWatch トランザクション検索を有効にする必要があります。これは、AgentCore がトレースデータを X-Ray に送信できるようにする

1 回限りのセットアップです。詳細については、「トランザクション検索を有効にする」を参照してください。

AgentCore のオブザーバビリティの設定の詳細については、「Amazon Bedrock AgentCore エージェントまたはツールにオブザーバビリティを追加する」を参照してください。

AgentCore でのトレースヘッダーの使用

AgentCore は、分散トレース用の X-Ray トレースヘッダー形式をサポートしています。AgentCore へのリクエストに `X-Amzn-Trace-Id`ヘッダーを含めると、サービス境界全体でトレースコンテキストを維持できます。

Amazon Elastic Compute Cloud と AWS X-Ray

AmazonEC2インスタンスにX-Rayデーモンをインストールし、ユーザーデータスクリプトを使用して実行することができます。手順については「Amazon EC2 での X-Ray デーモンの実行」を参照してください。

インスタンスプロファイルを使用して、デーモンにX-Rayのトレースデータをアップロードするアクセス権限を付与します。詳細については、「X-Rayにデータを送信するアクセス権限をデーモンに付与する」を参照してください。

Amazon SNS と AWS X-Ray

Amazon Simple Notification Service (Amazon SNS) AWS X-Ray で を使用すると、リクエストが SNS トピックを介して SNS がサポートするサブスクリプションサービスに移動するときに、リクエストを追跡および分析できます。Amazon SNS と X-Ray トレースを併用して、リクエストがトピックに費やされる時間や、トピックの各サブスクリプションにメッセージを配信するのにかかった時間など、メッセージとそのバックエンドサービスのレイテンシーを分析できます。Amazon SNS は、標準トピックと FIFO トピックの両方で X-Ray トレースをサポートしています。

X-Ray で既に計測されているサービスから Amazon SNS トピックに発行すると、Amazon SNS はトレースコンテキストをパブリッシャーからサブスクライバーに渡します。さらに、アクティブトレースを有効にして、計測された SNS クライアントから発行されたメッセージの Amazon SNS サブスクリプションに関するセグメントデータを X-Ray に送信できます。Amazon SNS コンソール、または Amazon SNS API か CLI を使用して、Amazon SNS トピックのアクティブトレースを有効にします。SNS クライアントの計測の詳細については、「アプリケーションの計測」を参照してください。

Amazon SNS アクティブトレースの設定

Amazon SNS コンソール、CLI、または SDK AWS を使用して、Amazon SNS アクティブトレースを設定できます。

Amazon SNS コンソールを使用する場合、Amazon SNS は SNS が X-Ray を呼び出すために必要なアクセス許可の作成を試みます。X-Ray リソースポリシーを変更するための十分なアクセス許可がない場合、この試行は拒否されることがあります。これらのアクセス許可の詳細については、「Amazon Simple Notification Service デベロッパーガイド」の「Amazon SNS での Identity and Access Management」および「Amazon SNS アクセスコントロールのケース例」を参照してください。Amazon SNS コンソールを使用してアクティブトレースを有効にする方法の詳細については、「Amazon Simple Notification Service デベロッパーガイド」の「Amazon SNS トピックでアクティブトレースを有効にする」を参照してください。

CLI または SDK AWS を使用してアクティブなトレースを有効にする場合は、リソースベースのポリシーを使用してアクセス許可を手動で設定する必要があります。`PutResourcePolicy` を使用して、Amazon SNS が X-Ray にトレースを送信できるようにするために必要なリソースベースのポリシーを使用して X-Ray を設定します。

Example Amazon SNS アクティブトレース用の X-Ray リソースベースのポリシーの例

このポリシードキュメントの例では、Amazon SNS がトレースデータを X-Ray に送信するために必要なアクセス許可を指定します。

```
{
    Version: "2012-10-17",
    Statement: [
      {
        Sid: "SNSAccess",
        Effect: Allow,
        Principal: {
          Service: "sns.amazonaws.com",
        },
        Action: [
          "xray:PutTraceSegments",
          "xray:GetSamplingRules",
          "xray:GetSamplingTargets"
        ],
        Resource: "*",
        Condition: {
          StringEquals: {
            "aws:SourceAccount": "account-id"
```

```
        },
        StringLike: {
          "aws:SourceArn": "arn:partition:sns:region:account-id:topic-name"
        }
      }
    }
  ]
}
```

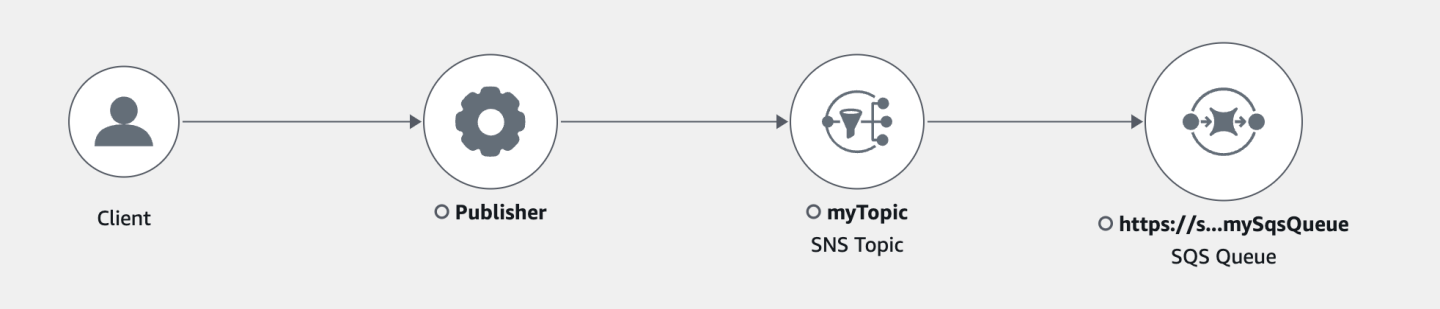
CLI を使用して、Amazon SNS アクセス許可を付与するリソースベースのポリシーを作成して、トレースデータを X-Ray に送信します。

```
aws xray put-resource-policy --policy-name MyResourcePolicy --policy-document
 '{ "Version": "2012-10-17", "Statement": [ { "Sid": "SNSAccess", "Effect": "Allow",
 "Principal": { "Service": "sns.amazonaws.com" }, "Action": [ "xray:PutTraceSegments",
 "xray:GetSamplingRules", "xray:GetSamplingTargets" ], "Resource": "*",
 "Condition": { "StringEquals": { "aws:SourceAccount": "account-id" }, "StringLike":
 { "aws:SourceArn": "arn:partition:sns:region:account-id:topic-name" } } } ] }'
```

これらの例を使用するには、*partition*、*region*、*account-id*、 *topic-name*を特定の AWS パーティション、リージョン、アカウント ID、Amazon SNS トピック名に置き換えます。すべての Amazon SNS トピックに、トレースデータを X-Ray に送信するアクセス許可を付与するには、トピック名を * に置き換えます。

X-Ray コンソールで Amazon SNS パブリッシャートレースとサブスクライバーのトレースを表示する

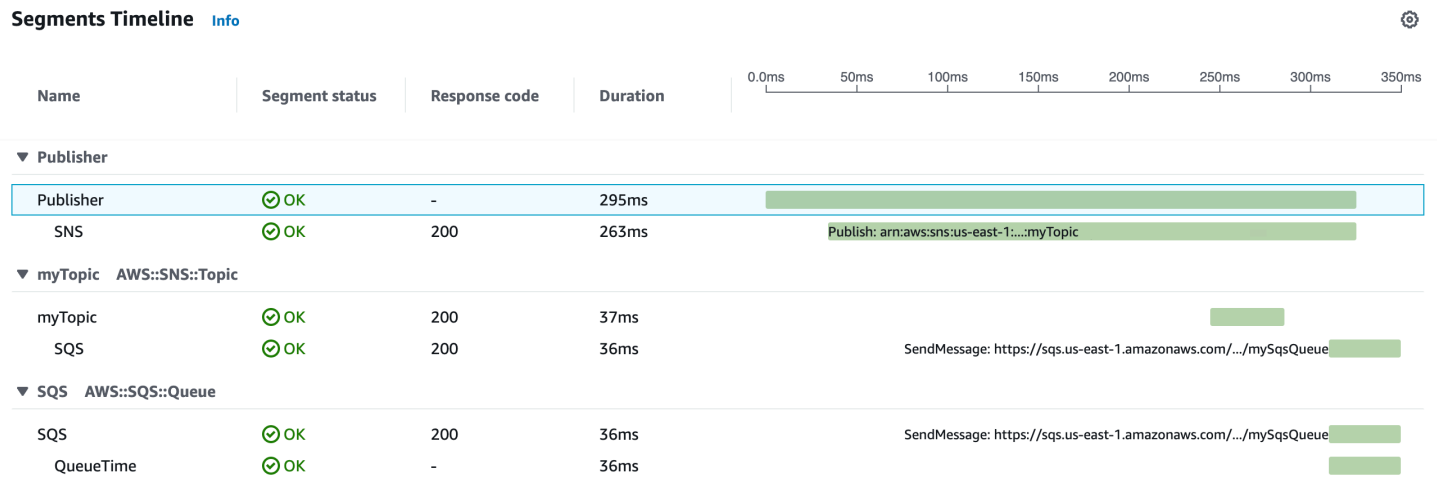
X-Ray コンソールを使用して、Amazon SNS のパブリッシャーとサブスクライバーの接続ビューを表示するトレースマップおよびトレースの詳細を表示します。トピックの Amazon SNS アクティブトレースを有効にすると、X-Ray トレースマップとトレース詳細マップに Amazon SNS パブリッシャー、Amazon SNS トピック、およびダウンストリームサブスクライバーの接続ノードが表示されます。



Amazon SNS パブリッシャーとサブスクライバーにまたがるトレースを選択すると、X-Ray のトレースの詳細ページにトレース詳細マップとセグメントタイムラインが表示されます。

Example Amazon SNS パブリッシャーとサブスクライバーのタイムラインの例

この例は、Amazon SNS トピックにメッセージを送信する Amazon SNS パブリッシャーを含むタイムラインを示しています。このメッセージは Amazon SQS サブスクライバーによって処理されます。

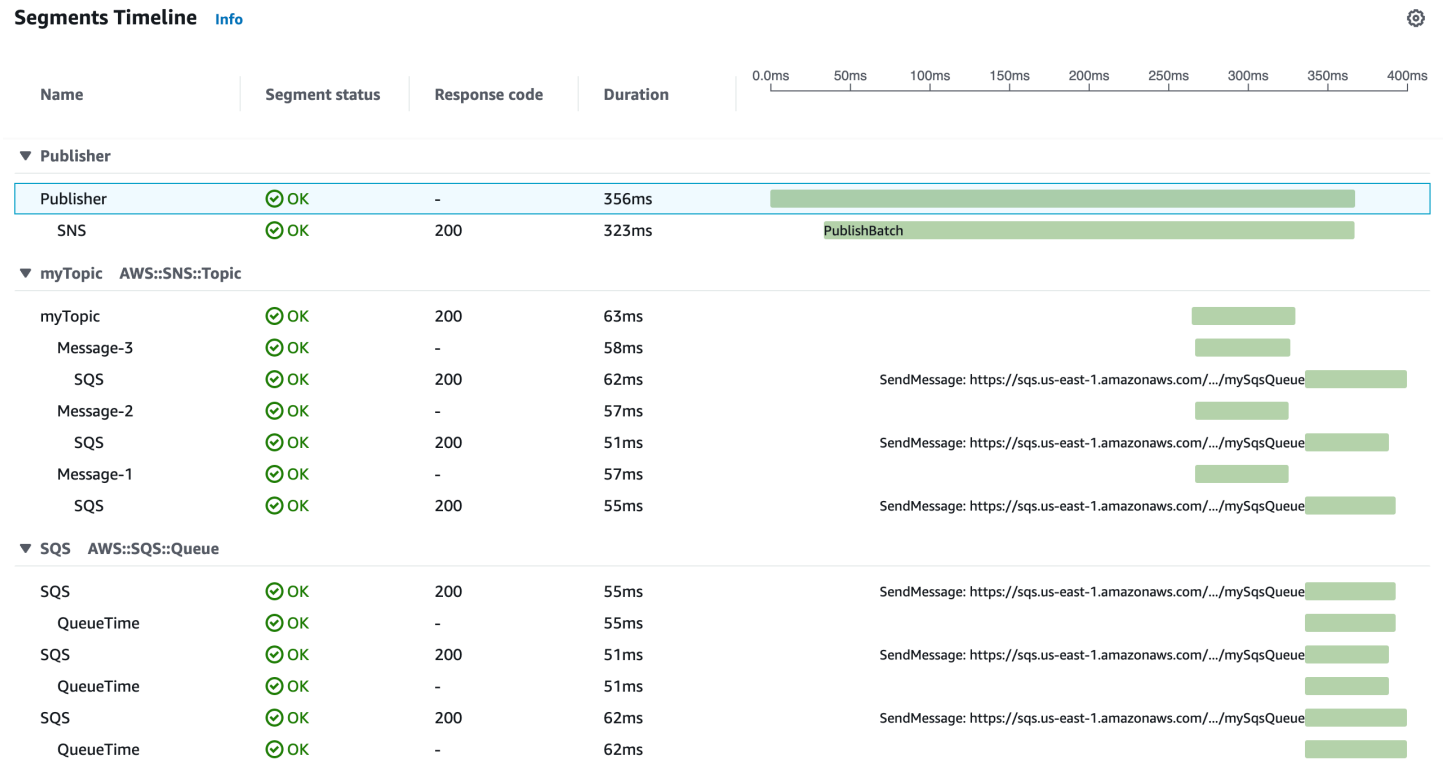


上記のタイムラインの例は、Amazon SNS メッセージフローの詳細を示しています。

- SNS セグメントは、クライアントからの `Publish` API 呼び出しの往復所要時間を表します。
- myTopic セグメントは、発行リクエストに対する Amazon SNS レスポンスのレイテンシーを表します。
- SQS サブセグメントは、Amazon SNS が Amazon SQS キューにメッセージを発行するのにかかる往復時間を表します。
- myTopic セグメントと SQS サブセグメントの間隔は、メッセージが Amazon SNS システムで費やす時間を表します。

Example Amazon SNS メッセージのバッチ処理を含むタイムラインの例

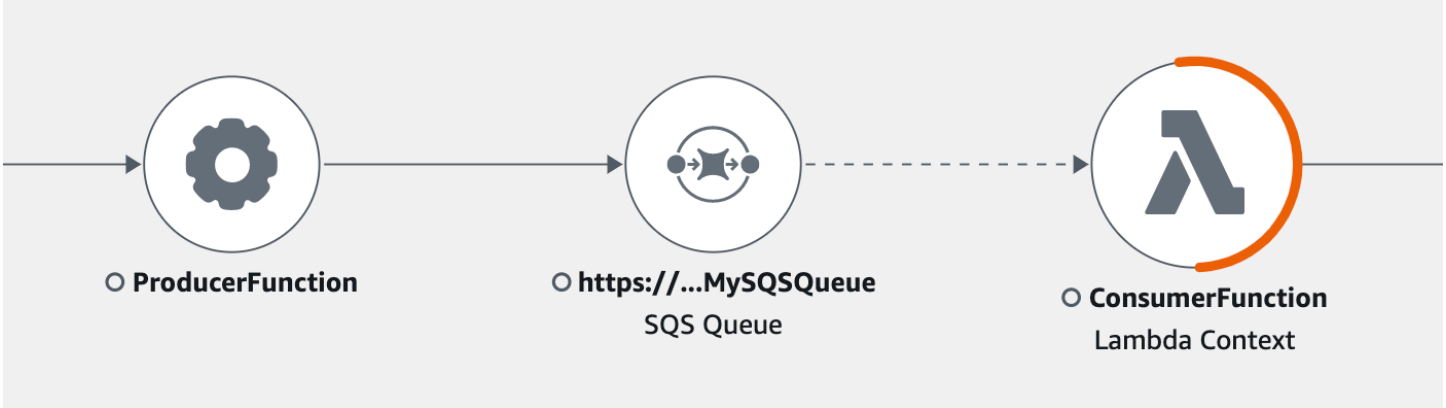
複数の Amazon SNS メッセージが 1 つのトレース内でバッチ処理される場合、セグメントタイムラインには、処理された各メッセージを表すセグメントが表示されます。



Amazon SQS と AWS X-Ray

AWS X-Ray は Amazon Simple Queue Service (Amazon SQS) と統合して、Amazon SQS キューを通過するメッセージをトレースします。サービスが X-Ray SDK を使用してリクエストをトレースする場合、Amazon SQS はトレースヘッダーを送信し、整合性のあるトレース ID を持つコンシューマーに、送信者から元のトレースを伝達し続けます。トレースの継続性により、ユーザーはダウンストリームサービス全体でトレース、分析、およびデバッグを実行できます。

AWS X-Ray は、Amazon SQS と を使用したイベント駆動型アプリケーションのトレースをサポートします AWS Lambda。CloudWatch コンソールを使用すると、各リクエストが Amazon SQS のキューに入れられ、ダウンストリーム Lambda 関数によって処理される過程を各リクエストの接続されたビューで確認できます。アップストリームメッセージプロデューサーからのトレースは、ダウンストリーム Lambda コンシューマーノードからのトレースに自動的にリンクされるため、アプリケーションのエンドツーエンドのビューが作成されます。詳細については、「イベント駆動型アプリケーションのトレース」を参照してください。



Amazon SQS では、次のトレースヘッダー計測がサポートされています。

- デフォルトの HTTP ヘッダー – SDK を介して Amazon SQS を呼び出すと、X-Ray AWS SDK はトレースヘッダーを HTTP ヘッダーとして自動的に入力します。デフォルトのトレースヘッダーは `X-Amzn-Trace-Id` によって転送され、`SendMessage` または `SendMessageBatch` リクエストに含まれるすべてのメッセージに対応します。デフォルトの HTTP ヘッダーの詳細については、「トレースヘッダー」を参照してください。
- **`AWSTraceHeader`** システム属性 — `AWSTraceHeader` は、Amazon SQS によって予約されたメッセージシステム属性で、X-Ray トレースヘッダーをキューのメッセージとともに渡します。`AWSTraceHeader` は、新しい言語のトレース SDK を構築する場合など、X-Ray SDK による自動計測ができない場合でも使用できます。両方のヘッダー計測が設定されている場合、メッセージシステム属性が HTTP トレースヘッダーを上書きします。

Amazon EC2 で実行した場合、Amazon SQS は一度に 1 つのメッセージの処理をサポートします。これは、オンプレミスホストで実行されている場合、および AWS Fargate Amazon ECS や などのコンテナサービスを使用する場合に適用されます AWS App Mesh。

トレースヘッダーは、Amazon SQS メッセージサイズとメッセージ属性のクォータの両方から除外されます。X-Ray トレースを有効にしても、Amazon SQS クォータを超えることはありません。AWS クォータの詳細については、Amazon SQSクォータ」を参照してください。

HTTP トレースヘッダーの送信

Amazon SQS の送信者コンポーネントは、`SendMessageBatch` または `SendMessage` 呼び出しを通じて、自動的にトレースヘッダーを送信できます。 AWS SDK クライアントが計測されると、X-Ray SDK でサポートされているすべての言語を通じて自動的に追跡できます。これらのサービス (Amazon S3 バケットや Amazon SQS キューなど) 内でアクセスするトレースされた AWS のサービ

ス および リソースは、X-Ray コンソールのトレースマップにダウンストリームノードとして表示されます。

任意の言語で AWS SDK 呼び出しをトレースする方法については、サポートされている SDKs の以下のトピックを参照してください。

- Go – X-Ray AWS SDK for Go を使用した SDK 呼び出しのトレース
- Java – X-Ray AWS SDK for Java を使用した SDK 呼び出しのトレース
- Node.js – X-Ray AWS SDK for Node.js を使用した SDK 呼び出しのトレース
- Python – X-Ray AWS SDK for Python を使用した SDK 呼び出しのトレース
- Ruby – X-Ray AWS SDK for Ruby を使用した SDK 呼び出しのトレース
- .NET – X-Ray AWS SDK for .NET を使用した SDK 呼び出しのトレース

トレースヘッダーを取得し、トレースコンテキストを復元する

Lambda ダウンストリームコンシューマーを使用している場合、トレースコンテキストの伝達は自動的に行われます。他の Amazon SQS コンシューマーでコンテキストの伝達を続行するには、レシーバーコンポーネントへのハンドオフを手動で計測する必要があります。

トレースコンテキストを復元するには、主に 3 つのステップがあります。

- `AWSTraceHeader` APIを呼び出して `ReceiveMessage` 属性のキューからメッセージを受信します。
- 属性からトレースヘッダーを取得します。
- ヘッダーからトレース ID を復元します。必要に応じて、セグメントにメトリクスを追加します。

以下は、X-Ray SDK for Java で記述された実装の例です。

Example : トレースヘッダーを取得し、トレースコンテキストを復元する

```
// Receive the message from the queue, specifying the "AWSTraceHeader"
ReceiveMessageRequest receiveMessageRequest = new ReceiveMessageRequest()
        .withQueueUrl(QUEUE_URL)
        .withAttributeNames("AWSTraceHeader");
List<Message> messages = sqs.receiveMessage(receiveMessageRequest).getMessages();
```

```
if (!messages.isEmpty()) {
    Message message = messages.get(0);

    // Retrieve the trace header from the AWSTraceHeader message system attribute
    String traceHeaderStr = message.getAttributes().get("AWSTraceHeader");
    if (traceHeaderStr != null) {
        TraceHeader traceHeader = TraceHeader.fromString(traceHeaderStr);

        // Recover the trace context from the trace header
        Segment segment = AWSXRay.getCurrentSegment();
        segment.setTraceId(traceHeader.getRootTraceId());
        segment.setParentId(traceHeader.getParentId());

 segment.setSampled(traceHeader.getSampled().equals(TraceHeader.SampleDecision.SAMPLED));
    }
}
```

Amazon S3 と AWS X-Ray

AWS X-Ray は Amazon S3 と統合してアップストリームリクエストをトレースし、アプリケーションの S3 バケットを更新します。サービスが X-Ray SDK を使用してリクエストをトレースする場合、Amazon S3 は AWS Lambda、Amazon SQS、Amazon SNS などのダウンストリームイベントサブスクライバーにトレースヘッダーを送信できます。X-Ray は Amazon S3 イベント通知のトレースメッセージを有効にします。

X-Ray トレースマップを使用して、Amazon S3 およびアプリケーションが使用する他のサービス間の接続を表示できます。コンソールを使用して、平均レイテンシーや障害発生率などのメトリクスを表示することもできます。X-Ray コンソールの詳細については、「X-Ray コンソールを使用する」を参照してください。

Amazon S3 は、デフォルトの HTTP ヘッダーの計測をサポートしています。X-Ray SDK は、 AWS SDK を介して Amazon S3 を呼び出すと、トレースヘッダーを HTTP ヘッダーとして自動的に入力します。デフォルトのトレースヘッダーは、`X-Amzn-Trace-Id` によって伝送されます。トレースヘッダーの詳細については、コンセプトページの「トレースヘッダー」を参照してください。Amazon S3 トレースコンテキストの伝播では、Lambda、SQS、および SNS のサブスクライバーがサポートされます。SQS と SNS はそれ自体でセグメントデータを送信しないため、S3 でトリガーされた場合、トレースヘッダーはダウンストリームのサービスに伝達されるものの、トレースやトレースマップには表示されません。

Amazon S3 イベント通知を設定する

Amazon S3 通知機能で、バケット内で の特定のイベントが発生したときに、通知を受けることができます。これらの通知は、アプリケーション内の次の宛先に伝播できます。

- Amazon Simple Notification Service (Amazon SNS)
- Amazon Simple Queue Service (Amazon SQS)
- AWS Lambda

サポートされているイベントのリストについては、Amazon S3 開発者ガイドでサポートされているイベントタイプを参照してください。。

Amazon SNS と Amazon SQS

SNS トピックや SQS キューに通知を発行するには、まず Amazon S3 のアクセス許可を付与する必要があります。これらのアクセス許可を付与するには、送信先 SNS トピックまたは SQS キューに AWS Identity and Access Management (IAM) ポリシーをアタッチします。必要な IAM ポリシーの詳細については、SNS トピックまたは SQS キューにメッセージを発行するアクセス許可の付与を参照してください。

SNS および SQS の X-Ray との統合については、Amazon SNS と AWS X-Ray および Amazon SQS と AWS X-Ray を参照してください。

AWS Lambda

Amazon S3 コンソールを使用して、Lambda 関数で S3 バケットのイベント通知を設定する場合、コンソールは Lambda 関数で必要なアクセス許可を設定し、Amazon S3 がバケットから関数を呼び出すアクセス許可を持つようにします。詳細については、Amazon Simple Storage Service コンソールユーザーガイドの「S3 バケットのイベント通知を有効化および設定する方法」を参照してください。

から Amazon S3 に Lambda 関数 AWS Lambda を呼び出すアクセス許可を付与することもできます。詳細については、AWS Lambda デベロッパーガイドの「チュートリアル: Amazon S3 での AWS Lambda の使用」を参照してください。

Lambda と X-Ray の統合の詳細については、AWS 「Lambda での Java コードの計測」を参照してください。

AWS Distro for OpenTelemetry と AWS X-Ray

AWS Distro for OpenTelemetry (ADOT) を使用して、メトリクスとトレースを収集し、Amazon CloudWatch、Amazon OpenSearch Service、Amazon Managed Service for Prometheus などの AWS X-Ray 他のモニタリングソリューションに送信します。

AWS Distro for OpenTelemetry

AWS Distro for OpenTelemetry (ADOT) は、Cloud Native Computing Foundation (CNCF) OpenTelemetry プロジェクトに基づく AWS ディストリビューションです。OpenTelemetry は、分散トレースとメトリクスを収集するためのオープンソース API、ライブラリ、およびエージェントの単一セットを提供します。このツールキットは、SDK、自動計測エージェント、およびコレクタを含むアップストリームの OpenTelemetry コンポーネントのディストリビューションであり、によってテスト、最適化、保護、およびサポートされます。 AWS。

ADOT を使用すると、エンジニアはアプリケーションを一度計測し、相関メトリクスとトレースを Amazon CloudWatch、Amazon OpenSearch Service AWS X-Ray、Amazon Managed Service for Prometheus などの複数の AWS モニタリングソリューションに送信できます。

ADOT は、X-Ray などのモニタリングソリューションへのトレースとメトリクスの送信を簡素化 AWS のサービス するために、増え続ける と統合されています。ADOT と統合されたサービスの例として、次のようなものがあります。

- AWS Lambda ADOT 用 AWS マネージド Lambda レイヤーは、Lambda 関数を自動的に計測し、OpenTelemetry をout-of-the-box使える および AWS Lambda X-Ray の設定とともに、セットアップしやすいレイヤーにパッケージ化することで、plug-and-playのユーザーエクスペリエンスを提供します。ユーザーは、コードを変更せずに Lambda 関数の OpenTelemetry を有効または無効にできます。詳細については、「AWS Distro for OpenTelemetry Lambda」を参照してください。
- Amazon Elastic Container Service (ECS) – AWS Distro for OpenTelemetry Collector を使用して Amazon ECS アプリケーションからメトリクスとトレースを収集し、X-Ray やその他のモニタリングソリューションに送信します。詳細については、Amazon ECS デベロッパーガイドの「アプリケーショントレースデータの収集」を参照してください。
- AWS App Runner – App Runner は AWS 、 Distro for OpenTelemetry (ADOT) を使用した X-Ray へのトレースの送信をサポートしています。ADOT SDK を使用してコンテナ化されたアプリケーションのトレースデータを収集し、X-Ray を使用して計測したアプリケーションを分析し、インサイトを得ます。詳細については、「AWS App Runner と X-Ray」を参照してください。

AWS Distro for OpenTelemetry の詳細については、AWS 「 Distro for OpenTelemetry ドキュメント AWS のサービス」を参照してください。

AWS Distro for OpenTelemetry と X-Ray を使用したアプリケーションの計測の詳細については、AWS 「 Distro for OpenTelemetry を使用したアプリケーションの計測」を参照してください。

を使用した X-Ray 暗号化設定の変更の追跡 AWS Config

AWS X-Ray は と統合 AWS Config して、X-Ray 暗号化リソースに加えられた設定変更を記録します。 AWS Config を使用して、X-Ray 暗号化リソースのインベントリ、X-Ray 設定履歴の監査、リソースの変更に基づいた通知の送信を行うことができます。

AWS Config では、次の X-Ray 暗号化リソースの変更をイベントとして ログ記録できます。

- 設定変更 暗号化キーの変更および追加、もしくは暗号化X-Ray設定をデフォルトに戻すことができます。

X-Ray と の間に基本的な接続を作成する方法については、次の手順を参照してください AWS Config。

Lambda関数のトリガーの作成

カスタム AWS Config ルールを生成する前に、カスタム AWS Lambda 関数の ARN が必要です。以下の手順に従い、 AWS Config リソースの状態に基づいて、準拠している値または準拠しない値を`XrayEncryptionConfig`返す基本的な関数をNode.jsで作成します。

AWS::XrayEncryptionConfigの変更トリガーを持つLambda関数を作成するには

1. Lambdaのコンソールを開きます。[機能の作成]を選択します。
2. [設計図]を選択し、設計図ライブラリをフィルタリングして、トリガー変更のルール設定で設計図を探します。設計図名のリンクをクリックするか、[設定]を選択して続行してください。
3. 以下のフィールドを定義して設計図を設定します：
   - [Name] には、名前を入力します。
   - [役割]では、[テンプレートから新しい役割の作成]を選択します。
   - [役割名]に名前を入力します。
   - [ポリシーテンプレート]では、[AWS Config ルールのアクセス権限]を選択します。

4. 関数の作成 を選択して、 AWS Lambda コンソールで関数を作成して表示します。
5. 関数コードを編集して、AWS::EC2::InstanceAWS::XrayEncryptionConfigに置き換えます。また、説明欄を更新して、この変更を反映させることもできます。

   初期設定コード

```
    if (configurationItem.resourceType !== 'AWS::EC2::Instance') {
        return 'NOT_APPLICABLE';
    } else if (ruleParameters.desiredInstanceType ===
configurationItem.configuration.instanceType) {
        return 'COMPLIANT';
    }
        return 'NON_COMPLIANT';
```

   更新されたコード

```
    if (configurationItem.resourceType !== 'AWS::XRay::EncryptionConfig') {
        return 'NOT_APPLICABLE';
    } else if (ruleParameters.desiredInstanceType ===
configurationItem.configuration.instanceType) {
        return 'COMPLIANT';
    }
        return 'NON_COMPLIANT';
```

6. X-Rayにアクセスするために、IAMの実行の役割に以下を追加します。これらのアクセス権限により、X-Rayリソースへの読み取り専用のアクセスを許可します。適切なリソースへのアクセスを許可しない場合、ルールに関連付けられた Lambda 関数を評価する AWS Config と、 からの範囲外のメッセージが発生します。

```
    {
        "Sid": "Stmt1529350291539",
        "Action": [
            "xray:GetEncryptionConfig"
        ],
        "Effect": "Allow",
        "Resource": "*"
     }
```

X 線のカスタム AWS Config ルールの作成

Lambda 関数が作成されたら、関数の ARN を書き留め、 AWS Config コンソールに移動してカスタムルールを作成します。

X-Ray の AWS Config ルールを作成するには

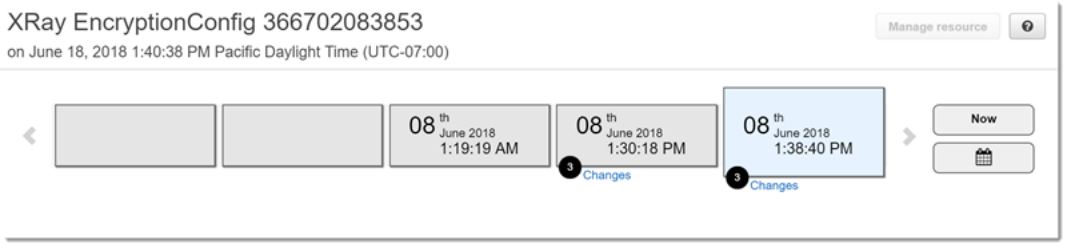
1. コンソールの[ルール AWS Config]ページを開きます。
2. [ルールの追加]を選択し、[カスタムルールを追加]を選択します。
3. AWS Lambda [関数のARN]には、使用するLambda関数に関連付けられた ARNを挿入します。
4. 設定するトリガーの種類を選択します：
   - 設定の変更 – ルールのスコープに一致するリソースが設定で変更されたときに評価を AWS Config トリガーします。評価は、 が設定項目の変更通知 AWS Config を送信した後に実行されます。
   - 定期 – 選択した頻度でルールの評価 AWS Config を実行します (24 時間ごとなど)。
5. [リソースタイプ]は、EncryptionConfigX-Rayの項目で選択します。
6. [Save] を選択します。

AWS Config コンソールは、ルールのコンプライアンスをすぐに評価し始めます。評価は完了までに数分かかることがあります。

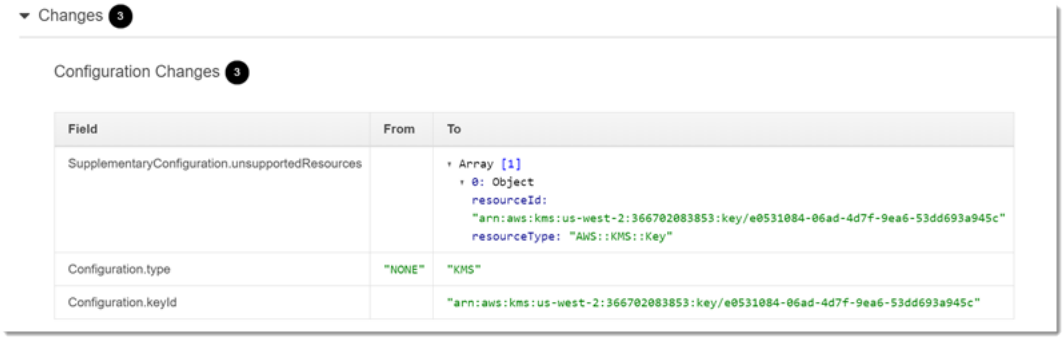
このルールが準拠したので、 は監査履歴のコンパイルを開始 AWS Config できます。リソースの変更をタイムラインの形式で AWS Config 記録します。イベントのタイムラインの変更ごとに、 はテーブルを from/to 形式で AWS Config 生成し、暗号化キーの JSON 表現で何が変更されたかを表示します。EncryptionConfigに関連付けられた2つのフィールドの変更は`Configuration.type`及び`Configuration.keyID`。

結果の例

以下は、特定の日時に加えられた変更を示す AWS Config タイムラインの例です。



以下は、 AWS Config 変更エントリの例です。変更前/変更後の形式で変更内容を示します。この例では、デフォルトのX-Ray暗号化設定を、定義済みの暗号化キーに変更されたことを示しています。



Amazon SNSの通知

設定変更の通知を受け取るには、 を設定 AWS Config して Amazon SNS 通知を発行します。詳細については、「E メールによる AWS Config リソースの変更のモニタリング」を参照してください。

AWS AppSync and AWS X-Ray

AWS AppSync のリクエストを有効にしてトレースできます。詳細については、AWS 「X-Ray を使用したトレース」を参照してください。

AWS AppSync API で X-Ray トレースを有効にすると、 AWS Identity and Access Management サービスにリンクされたロールが、適切なアクセス許可を持つアカウントに自動的に作成されます。これにより、 AWS AppSync は安全な方法で X-Ray にトレースを送信できます。

の Amazon API Gateway アクティブトレースのサポート AWS X-Ray

X-Rayを使用して、ユーザーリクエストがAmazon API Gateway APIsを経由し、基礎となるサービスへの流れをトレースして分析することができます。API Gatewayは、すべてのAPI Gateway評価項目タイプでX - Rayトレースがサポートしています：地域、エッジの最適化、プライベートです。X-Ray が利用可能なすべての AWS リージョン でAmazon API Gateway で X-Ray を使用できます。詳細については、『Amazon API Gateway開発者ガイド』 AWS X-Rayの「API Gateway APIの実行の追跡」を参照してください。

> **Note**
>
> X-Rayは、 API Gatewayを介したREST APIの追跡のみをサポートしています。

Amazon API Gateway は、 のアクティブなトレースサポートを提供します AWS X-Ray。APIステージの追跡機能を有効にすると、受信リクエストをサンプリングし、X-Rayにトレースを送信することができます。

APIステージで追跡機能を有効にするには

1. API Gateway(https://console.aws.amazon.com/apigateway)コンソールを開きます。
2. APIを選択します。
3. ステージを選択します。
4. 「ログ/トレース」タブで、以下を選択します、「X-Rayトレースを有効にする」を選択し、変更の保存。
5. 左側のナビゲーションパネルで、「リソース」を選択します。
6. 新しい設定でAPIを再配置するには、[Actions]ドロップダウン方式を選択し[Deploy API]の順に選択します。

API Gatewayは、X-Rayコンソールで定義したサンプリングルールを使用し、記録するリクエストを決定します。APIのみに適用されるルール、または特定のヘッダーを含むリクエストにのみ適用されるルールを作成することができます。API Gatewayは、ステージおよびリクエストの詳細と共に、セグメント上の属性にヘッダーを記録します。詳細については、「サンプリングルールの設定」を参照してください。

> **Note**
>
> API Gateway HTTP 統合で REST API をトレースする場合、各セグメントのサービス名は API Gateway から HTTP 統合エンドポイントまでのリクエスト URL パスに設定され、一意の URL パスごとに X-Ray トレースマップ上にサービスノードが作成されます。URL パスの数が多いと、トレースマップが 10,000 ノードの制限を超え、エラーが発生する可能性があります。
>
> API Gateway によって作成されるサービスノードの数を最小限に抑えるには、パラメータを URL クエリ文字列内、またはリクエスト本文に入れて POST 経由で渡すことを検討してください。いずれの方法でも、パラメータが URL パスに含まれることはないため、個別の URL パスやサービスノードの数が少なくなる可能性があります。

API Gatewayは、HTTP（すべての受信ハイパーテキスト転送プロトコル）リクエストに対し、まだトレースヘッダーを持たない受信リクエストにトレースヘッダーを追加します。

```
X-Amzn-Trace-Id: Root=1-5759e988-bd862e3fe1be46a994272793
```

X-Ray のトレース ID 形式

X-Ray `trace_id` は、ハイフンで区切られた 3 つの数字で構成されています。例えば、`1-58406520-a006649127e371903a2de979` と指定します。これには、以下のものが含まれます：

- バージョン番号、すなわち、1。
- 元のリクエストの時刻。ユニックスエポックタイムで、16 進数 8 桁で表示されます。

  例えば、エポックタイムで 2016 年 12 月 1 日 10:00AM PST (太平洋標準時刻) は `1480615200` 秒、または 16 進数で `58406520` と表示されます。
- グローバルに一意なトレースの 96 ビットの識別子で、24 桁の 16 進数で表示されます。

アクティブなトレースが無効になっている場合でも、リクエストがサンプリングされ、かつトレースが開始されているサービスからのリクエストが送信された場合は、ステージによりセグメントが記録されます。たとえば、搭載されたウェブアプリケーションがHTTPクライアントを使用してAPI Gateway APIを呼び出すことができます。X-Ray SDKを使用してHTTPクライアントを導入すると、サンプリング判定を含む発信リクエストにトレースヘッダーが追加されます。API Gatewayは、トレースヘッダーを読み取り、サンプリングされたリクエストのセグメントを作成します。

API Gateway を使用して API 用の Java SDK を生成する場合、SDK クライアントを手動で計測するのと同じ方法で、クライアントビルダーにリクエストハンドラーを追加して AWS SDK クライアントを計測できます。手順については、「X-Ray AWS SDK for Java を使用した SDK 呼び出しのトレース」を参照してください。

Amazon EC2 と AWS App Mesh

AWS X-Ray は と統合AWS App Meshして、マイクロサービスの Envoy プロキシを管理します。App Meshでは、同じタスクまたはポッドのコンテナ内で実行されているX-Rayデーモンにトレースデータを送信するように設定できるEnvoyバージョンを提供しています。X-Rayは、以下のApp Mesh対応サービスでのトレースをサポートしています:

- Amazon Elastic Container Service (Amazon ECS)
- Amazon Elastic Kubernetes Service (Amazon EKS)

- Amazon Elastic Compute Cloud (Amazon EC2)

App MeshでX-Rayトレースを有効にする方法については、以下の説明を参照にしてください。

Envoy proxyがX-Rayにデータを送信するように設定するには、ENABLE_ENVOY_XRAY_TRACING コンテナの定義で環境変数を設定します。

> **Note**
>
> App MeshバージョンのEnvoyは現在、サンプリングのルール設定に基づいてトレースを送信しません。代わりに、Envoyバージョン1.16.3 以降では5%の固定サンプリングレートを使用し、Envoyバージョン1.16.3以前では50%のサンプリングレートを使用します。

Example Amazon ECSのためのEnvoyコンテナの定義

```
{
      "name": "envoy",
      "image": "public.ecr.aws/appmesh/aws-appmesh-envoy:envoy-version",
      "essential": true,
      "environment": [
        {
          "name": "APPMESH_VIRTUAL_NODE_NAME",
          "value": "mesh/myMesh/virtualNode/myNode"
        },
        {
          "name": "ENABLE_ENVOY_XRAY_TRACING",
          "value": "1"
        }
      ],
      "healthCheck": {
        "command": [
          "CMD-SHELL",
          "curl -s http://localhost:9901/server_info | cut -d' ' -f3 | grep -q live"
        ],
        "startPeriod": 10,
        "interval": 5,
        "timeout": 2,
        "retries": 3
      }
```

> **Note**
>
> 使用可能なEnvoy リージョンのアドレスの詳細については、「ユーザーガイド」のの「 AWS App Mesh Envoyイメージ」を参照してください。

X-Rayデーモンをコンテナ内で実行する方法について参照してくださいAmazon ECS での X-Rayデーモンの実行。サービスメッシュ、マイクロサービス、Envoy proxy、および X-Ray デーモンを含むサンプルアプリケーションについては、colorappApp Mesh Examples GitHub リポジトリのサンプルを配置してください。

詳細はこちら

- AWS App Meshの開始方法
- AWS App Mesh と Amazon ECS の開始方法

AWS App Runner と X-Ray

AWS App Runner は、ソースコードまたはコンテナイメージから のスケーラブルで安全なウェブアプリケーションに直接デプロイするための、高速でシンプルで費用対効果の高い方法 AWS のサービス を提供する です AWS クラウド。新しいテクノロジーを学習したり、使用するコンピューティングサービスを決定したり、 AWS リソースをプロビジョニングして設定する方法を知っている必要はありません。詳細については、AWS 「App Runner とは」を参照してください。

AWS App Runner はAWS 、 Distro for OpenTelemetry (ADOT) と統合することで、トレースを X-Ray に送信します。ADOT SDK を使用してコンテナ化されたアプリケーションのトレースデータを収集し、X-Ray を使用して計測したアプリケーションを分析し、インサイトを得ます。詳細については、「X-Ray を使用した App Runner アプリケーションのトレース」を参照してください。

を使用した X-Ray API コールのログ記録 AWS CloudTrail

AWS X-Ray は、ユーザーAWS CloudTrail、ロール、または によって実行されたアクションを記録するサービスである と統合されています AWS のサービス。CloudTrail は、X-Ray のすべての API コールをイベントとしてキャプチャします。キャプチャされたコールには、X-Ray コンソールからの呼び出しと、X-Ray API オペレーションへのコード呼び出しが含まれます。CloudTrail で収集された情報を使用して、X-Ray に対するリクエスト、リクエスト元の IP アドレス、リクエストの作成日時、その他の詳細を確認できます。

各イベントまたはログエントリには、誰がリクエストを生成したかという情報が含まれます。アイデンティティ情報は、以下を判別するのに役立ちます。

- ルートユーザーまたはユーザー認証情報のどちらを使用してリクエストが送信されたか。
- リクエストが IAM Identity Center ユーザーに代わって行われたかどうか。

- リクエストがロールまたはフェデレーションユーザーのテンポラリなセキュリティ認証情報を使用して行われたかどうか。
- リクエストが、別の AWS のサービスによって送信されたかどうか。

アカウント AWS アカウント を作成するとCloudTrail は でアクティブになり、CloudTrail イベント履歴に自動的にアクセスできます。CloudTrail の [イベント履歴] では、 AWS リージョンで過去 90 日間に記録された 管理イベントの表示、検索、およびダウンロードが可能で、変更不可能な記録を確認できます。詳細については、「AWS CloudTrail ユーザーガイド」の「CloudTrail イベント履歴の使用」を参照してください。[イベント履歴] の閲覧には CloudTrail の料金はかかりません。

AWS アカウント 過去 90 日間のイベントの継続的な記録については、証跡または CloudTrail Lake イベントデータストアを作成します。

CloudTrail 証跡

追跡により、CloudTrail はログファイルを Amazon S3 バケットに配信できます。を使用して作成されたすべての証跡 AWS Management Console はマルチリージョンです。 AWS CLIを使用する際は、単一リージョンまたは複数リージョンの証跡を作成できます。アカウント AWS リージョン 内のすべての でアクティビティをキャプチャするため、マルチリージョン証跡を作成することをお勧めします。単一リージョンの証跡を作成する場合、証跡の AWS リージョンに記録されたイベントのみを表示できます。証跡の詳細については、「AWS CloudTrail ユーザーガイド」の「AWS アカウントの証跡の作成」および「組織の証跡の作成」を参照してください。

証跡を作成すると、進行中の管理イベントのコピーを 1 つ無料で CloudTrail から Amazon S3 バケットに配信できますが、Amazon S3 ストレージには料金がかかります。CloudTrail の料金の詳細については、「AWS CloudTrail の料金」を参照してください。Amazon S3 の料金に関する詳細については、「Amazon S3 の料金」を参照してください。

CloudTrail Lake イベントデータストア

[CloudTrail Lake] を使用すると、イベントに対して SQL ベースのクエリを実行できます。CloudTrail Lake は、行ベースの JSON 形式の既存のイベントを Apache ORC 形式に変換します。ORC は、データを高速に取得するために最適化された単票ストレージ形式です。イベントは、イベントデータストアに集約されます。イベントデータストアは、高度なイベントセレクタを適用することによって選択する条件に基づいた、イベントのイミュータブルなコレクションです。どのイベントが存続し、クエリに使用できるかは、イベントデータストアに適用するセレクタが制御します。CloudTrail Lake の詳細については、 AWS CloudTrail ユーザーガイドのAWS CloudTrail 「Lake の使用」を参照してください。

CloudTrail Lake のイベントデータストアとクエリにはコストがかかります。イベントデータストアを作成する際に、イベントデータストアに使用する料金オプションを選択します。料金オプションによって、イベントの取り込みと保存にかかる料金、および、そのイベントデータストアのデフォルトと最長の保持期間が決まります。CloudTrail の料金の詳細については、「AWS CloudTrail の料金」を参照してください。

トピック

- CloudTrail の X-Ray 管理イベント
- CloudTrail の X-Ray データイベント
- X-Ray イベントの例

CloudTrail の X-Ray 管理イベント

AWS X-Ray は と統合 AWS CloudTrail して、ユーザー、ロール、または によって実行された API アクションを X-Ray AWS のサービス に記録します。CloudTrailを使用して、X-RayのAPIリクエストをリアルタイムでモニタリングし、Amazon S3、Amazon CloudWatch Logs、またはAmazon CloudWatch Eventsにログを保存することができます。X-Rayは、CloudTrail ログファイルのイベントとして以下のアクションのログの記録をサポートしています：

サポートされているAPIアクション

- PutEncryptionConfig
- GetEncryptionConfig
- CreateGroup
- UpdateGroup
- DeleteGroup
- GetGroup
- GetGroups
- GetInsight
- GetInsightEvents
- GetInsightImpactGraph
- GetInsightSummaries
- GetSamplingStatisticSummaries

CloudTrail の X-Ray データイベント

データイベントは、リソースに対して実行、またはリソース内で実行されるリソースオペレーションに関する情報を提供します (セグメントドキュメントを X-Ray にアップロードする PutTraceSegments など)。

これらのイベントは、データプレーンオペレーションとも呼ばれます。データイベントは、多くの場合、高ボリュームのアクティビティです。デフォルトでは、CloudTrail はデータイベントを口グ記録しません。CloudTrail [イベント履歴] にはデータイベントは記録されません。

追加の変更がイベントデータに適用されます。CloudTrail の料金の詳細については、「AWS CloudTrail の料金」を参照してください。

CloudTrail コンソール、または CloudTrail CloudTrail API オペレーションを使用して AWS CLI、X-Ray リソースタイプのデータイベントを口グに記録できます。データイベントを口グに記録する方法の詳細については、「AWS CloudTrail ユーザーガイド」の「AWS Management Consoleを使用したデータイベントの口グ記録」および「AWS Command Line Interfaceを使用したデータイベントの口グ記録」を参照してください。

次の表に、データイベントを口グに記録できる X-Ray リソースタイプを示します。データイベントタイプ (コンソール) 列には、CloudTrail コンソールの [データイベントタイプ] リストから選択する値が表示されます。resources.type 値列には、 AWS CLI または CloudTrail APIs を使用して高度なイベントセレクタを設定するときに指定する `resources.type`値が表示されます。CloudTrail に記録されたデータ API 列には、リソース タイプの CloudTrail に口グ記録された API コールが表示されます。

データイベントタイプ (コンソール)	resources.type 値	CloudTrail に口グ記録されたデータ API
[X-Ray トレース]	`AWS::XRay::Trace`	• PutTraceSegments • GetTraceSummaries • GetTraceGraph • GetServiceGraph • BatchGetTraces • GetTimeSeriesServiceStatistics • PutTelemetryRecords

データイベントタイプ (コンソール)	resources.type 値	CloudTrail にログ記録されたデータ API
		• GetSamplingTargets

eventName および readOnly フィールドでフィルタリングして、自分にとって重要なイベントのみを口グに記録するように高度なイベントセレクタを設定できます。ただし、X-Ray トレースには ARN がないため、resources.ARN フィールドセレクタを追加したイベントの選択はできません。オブジェクトの詳細については、「AWS CloudTrail API リファレンス」の「AdvancedFieldSelector」を参照してください。以下は、 put-event-selectors AWS CLI コマンドを実行して CloudTrail 証跡のデータイベントを口グに記録する方法の例です。証跡が作成されたリージョンでコマンドを実行、またはそのリージョンを指定する必要があります。そうしないと、オペレーションは InvalidHomeRegionException 例外を返します。

```
aws cloudtrail put-event-selectors --trail-name myTrail --advanced-event-selectors \
'{
   "AdvancedEventSelectors": [
      {
         "FieldSelectors": [
            { "Field": "eventCategory", "Equals": ["Data"] },
            { "Field": "resources.type", "Equals": ["AWS::XRay::Trace"] },
            { "Field": "eventName", "Equals":
 ["PutTraceSegments","GetSamplingTargets"] }
         ],
         "Name": "Log X-Ray PutTraceSegments and GetSamplingTargets data events"
      }
   ]
}'
```

X-Ray イベントの例

管理イベントの例、**GetEncryptionConfig**

以下は、CloudTrail の X-Ray GetEncryptionConfig の口グエントリの例です。

Example

```
{
    "eventVersion"=>"1.05",
    "userIdentity"=>{
```

```
        "type"=>"AssumedRole",
        "principalId"=>"AROAJVHBZWD3DN6CI2MHM:MyName",
        "arn"=>"arn:aws:sts::123456789012:assumed-role/MyRole/MyName",
        "accountId"=>"123456789012",
        "accessKeyId"=>"AKIAIOSFODNN7EXAMPLE",
        "sessionContext"=>{
            "attributes"=>{
                "mfaAuthenticated"=>"false",
                "creationDate"=>"2023-7-01T00:24:36Z"
            },
            "sessionIssuer"=>{
                "type"=>"Role",
                "principalId"=>"AROAJVHBZWD3DN6CI2MHM",
                "arn"=>"arn:aws:iam::123456789012:role/MyRole",
                "accountId"=>"123456789012",
                "userName"=>"MyRole"
            }
        }
    },
    "eventTime"=>"2023-7-01T00:24:36Z",
    "eventSource"=>"xray.amazonaws.com",
    "eventName"=>"GetEncryptionConfig",
    "awsRegion"=>"us-east-2",
    "sourceIPAddress"=>"33.255.33.255",
    "userAgent"=>"aws-sdk-ruby2/2.11.19 ruby/2.3.1 x86_64-linux",
    "requestParameters"=>nil,
    "responseElements"=>nil,
    "requestID"=>"3fda699a-32e7-4c20-37af-edc2be5acbdb",
    "eventID"=>"039c3d45-6baa-11e3-2f3e-e5a036343c9f",
    "eventType"=>"AwsApiCall",
    "recipientAccountId"=>"123456789012"
}
```

データイベントの例、**PutTraceSegments**

以下は、CloudTrail の X-Ray PutTraceSegments データイベントのログエントリの例です。

Example

```
{
  "eventVersion": "1.09",
  "userIdentity": {
    "type": "AssumedRole",
```

```
    "principalId": "AROAWYXPW54Y4NEXAMPLE:i-0dzz2ac111c83zz0z",
    "arn": "arn:aws:sts::012345678910:assumed-role/my-service-role/
i-0dzz2ac111c83zz0z",
    "accountId": "012345678910",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AROAWYXPW54Y4NEXAMPLE",
        "arn": "arn:aws:iam::012345678910:role/service-role/my-service-role",
        "accountId": "012345678910",
        "userName": "my-service-role"
      },
      "attributes": {
        "creationDate": "2024-01-22T17:34:11Z",
        "mfaAuthenticated": "false"
      },
      "ec2RoleDelivery": "2.0"
    }
  },
  "eventTime": "2024-01-22T18:22:05Z",
  "eventSource": "xray.amazonaws.com",
  "eventName": "PutTraceSegments",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "198.51.100.0",
  "userAgent": "aws-sdk-ruby3/3.190.0 md/internal ua/2.0 api/xray#1.0.0 os/linux md/
x86_64 lang/ruby#2.7.8 md/2.7.8 cfg/retry-mode#legacy",
  "requestParameters": {
    "traceSegmentDocuments": [
      "trace_id:1-00zzz24z-EXAMPLE4f4e41754c77d0000",
      "trace_id:1-00zzz24z-EXAMPLE4f4e41754c77d0000",
      "trace_id:1-00zzz24z-EXAMPLE4f4e41754c77d0001",
      "trace_id:1-00zzz24z-EXAMPLE4f4e41754c77d0002"
    ]
  },
  "responseElements": {
    "unprocessedTraceSegments": []
  },
  "requestID": "5zzzzz64-acbd-46ff-z544-451a3ebcb2f8",
  "eventID": "4zz51z7z-77f9-44zz-9bd7-6c8327740f2e",
  "readOnly": false,
  "resources": [
    {
      "type": "AWS::XRay::Trace"
```

```
    }
  ],
  "eventType": "AwsApiCall",
  "managementEvent": false,
  "recipientAccountId": "012345678910",
  "eventCategory": "Data",
  "tlsDetails": {
    "tlsVersion": "TLSv1.2",
    "cipherSuite": "ZZZZZ-RSA-AAA128-GCM-SHA256",
    "clientProvidedHostHeader": "example.us-west-2.xray.cloudwatch.aws.dev"
  }
}
```

X-Ray と CloudWatch の統合

AWS X-Ray は CloudWatch Application Signals、CloudWatch RUM、および CloudWatch Synthetics と統合され、アプリケーションのヘルスを簡単にモニタリングできます。Application Signals のアプリケーションを有効にして、サービス、クライアントページ、Synthetics canary、およびサービス依存関係の運用健全性をモニタリングおよびトラブルシューティングします。

CloudWatch メトリクス、ログ、X-Ray トレースを相互に関連付けることにより、X-Ray トレースマップは、サービスの端末相互間の視点を提供し、パフォーマンスのボトルネックをすばやく特定し、影響を受けたユーザーを特定するのに役立ちます。

CloudWatch RUM を使用すると、現実的なユーザーモニタリングを実行できます。ウェブアプリケーションのパフォーマンスに関するクライアント側のデータを、ほぼリアルタイムで実際のユーザーセッションから収集し、表示できます。 AWS X-Ray と CloudWatch RUM を使用すると、ダウンストリームの AWS マネージドサービスを通じて、アプリケーションのエンドユーザーから開始するリクエストパスを分析およびデバッグできます。これにより、エンドユーザーに影響を与えるレイテンシーの傾向やエラーを特定できます。

トピック

- CloudWatch RUM と AWS X-Ray
- X-Rayを使用した CloudWatchの合成カナリアのデバッグ

CloudWatch RUM と AWS X-Ray

Amazon CloudWatch RUM を使用すると、現実的なユーザーモニタリングを実行できます。ウェブアプリケーションのパフォーマンスに関するクライアント側のデータを、ほぼリアルタイムで実際の

ユーザーセッションから収集し、表示できます。 AWS X-Ray と CloudWatch RUM を使用すると、ダウンストリームの AWS マネージドサービスを通じて、アプリケーションのエンドユーザーから開始するリクエストパスを分析およびデバッグできます。これにより、エンドユーザーに影響を与えるレイテンシーの傾向やエラーを特定できます。

ユーザーセッションの X-Ray トレースを有効にすると、CloudWatch RUM は許可された HTTP リクエストに X-Ray トレースヘッダーを追加し、許可された HTTP リクエストの X-Ray セグメントを記録します。その後、X-Ray トレースマップなど、これらのユーザーセッションについてのトレースやセグメントを X-Ray および CloudWatch コンソールに表示できるようになります。

> **Note**
>
> CloudWatch RUM は X-Ray のサンプリングルールとは統合されません。代わりに、CloudWatch RUM を使用するようにアプリケーションを設定するときに、サンプリング率を選択してください。CloudWatch RUM から送信されるトレースには、追加のコストが発生する可能性があります。詳細については、AWS X-Ray の料金を参照してください。

デフォルトでは、CloudWatch RUM から送信されるクライアント側トレースは、サーバー側のトレースと接続されません。クライアント側のトレースをサーバー側のトレースと接続するには、CloudWatch RUM のウェブクライアントで HTTP リクエストに X-Ray トレースヘッダーを追加するように設定します。

> **Warning**
>
> HTTP リクエストに X-Ray トレースヘッダーを追加するための設定を CloudWatch RUM のウェブクライアントで行うことで、クロスオリジンリソース共有 (CORS) に失敗する場合があります。これを回避するには、ダウンストリームサービスの CORS 設定で、許可されるヘッダーのリストに `X-Amzn-Trace-Id` HTTP ヘッダーを追加します。API Gateway をダウンストリームとして使用している場合は、「REST API リソースの CORS を有効にする」を参照してください。本番環境でクライアント側の X-Ray トレースヘッダーの追加を行う前に、アプリケーションのテストを実施することを強くお勧めします。詳細については、「CloudWatch RUM web client documentation」を参照してください。

CloudWatch での実際のユーザーモニタリングの詳細については、「CloudWatch RUMを使用する」を参照してください。X-Ray によるユーザーセッションのトレースなど、CloudWatch RUMを使用

するようにアプリケーションを設定するには、「CloudWatch RUM を使用するためにアプリケーションをセットアップする」を参照してください。

X-Rayを使用した CloudWatchの合成カナリアのデバッグ

CloudWatch Syntheticsは1日24時間有効になり1分間に1回起動されるスクリプト化されたカナリアを使用して、評価項目およびAPIをモニタリングできるフルマネージド型のサービスです。

以下の変更を確認して、canaryスクリプトをカスタマイズできます：

- 可用性
- レイテンシー
- トランザクション
- リンク切れまたはデットリンク
- タスクのステップごとの完了
- ページロードエラー
- UI アセットのロードレイテンシー
- 複雑なウィザードフロー
- アプリケーションのチェックアウトフロー

カナリアは、お客様と同じルートをたどり、同じアクションと動作を実行して、お客様の満足体験を継続的に検証します。

Syntheticsテストの設定の詳細については、「Syntheticsを使用してカナリアを作成および管理する」を参照してください。

以下の例では、Syntheticsのカナリアで発生する問題をデバッグするための一般的な使用例を示しています。各例は、トレースマップまたは X-Ray Analytics コンソールのいずれかを使用してデバッグするための重要な戦略を示しています。

トレースマップの読み方と操作方法の詳細については、「サービスマップの表示」を参照してください。

X-Ray Analytics コンソールの読み取りと操作方法の詳細については、AWS X-Ray 「分析コンソールの操作」を参照してください。

トピック

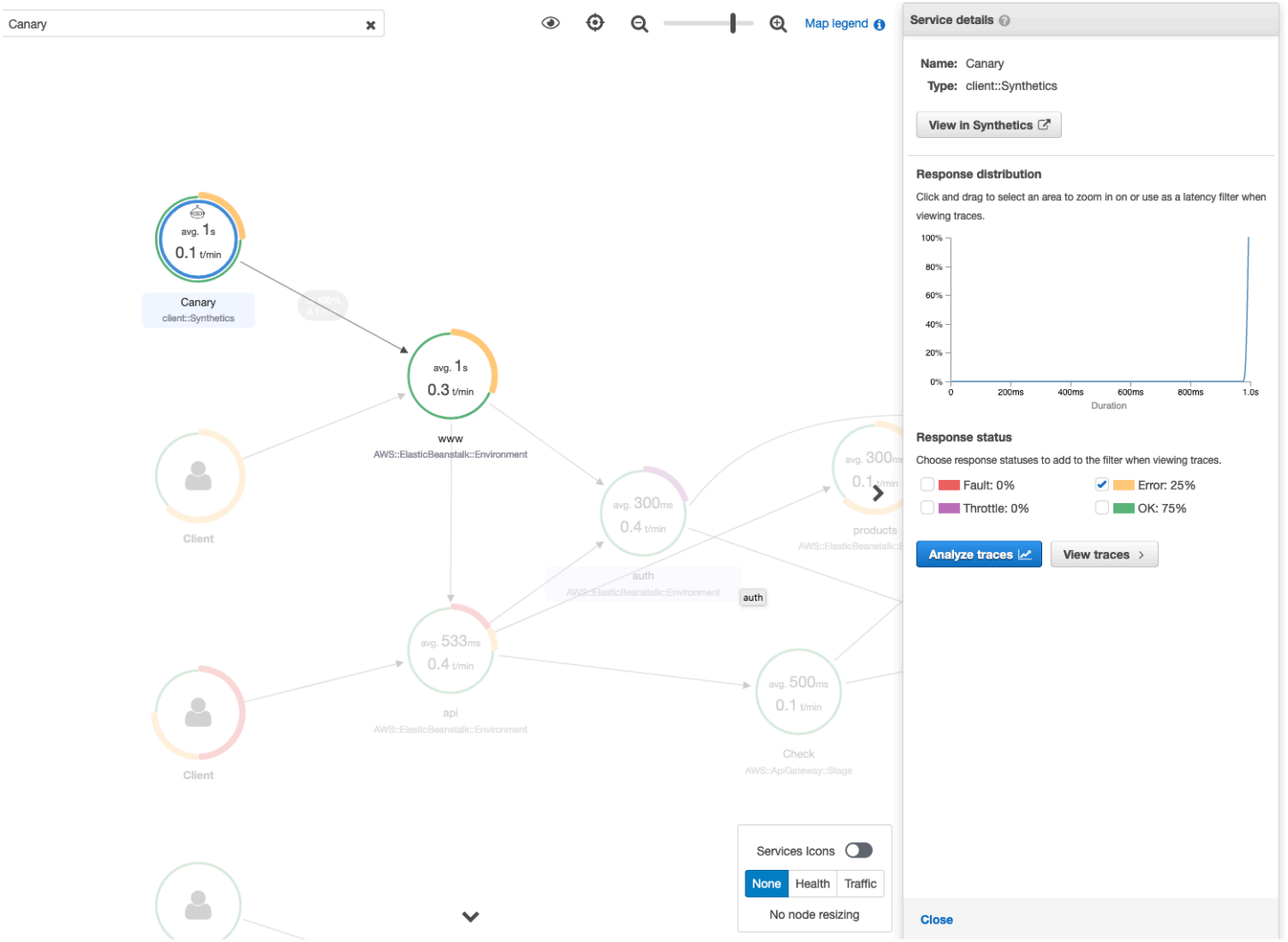
- トレースマップでエラーレポートが増加した canary を検証する
- 個々のトレースのトレース詳細マップを使用して、各リクエストを詳細に確認する

- アップストリームおよびダウンストリームサービスで継続的に発生している障害の根本原因を特定する
- パフォーマンスのボトルネックとトレンドを特定する
- 変更前と変更後で待ち時間およびエラー・障害率を比較する
- すべてのAPIとURLに必要なcanaryの受信可能範囲エリアを特定する
- グループを使用してSyntheticsテストに焦点を合わせる

トレースマップでエラーレポートが増加した canary を検証する

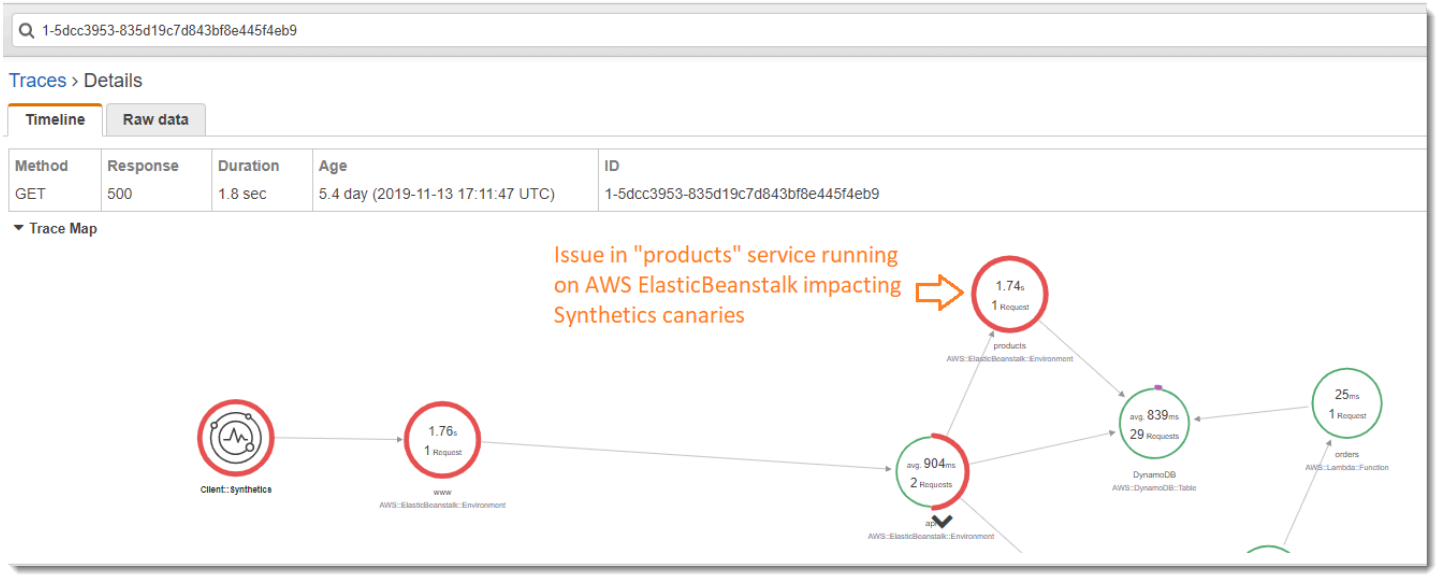
X-Ray トレースマップ内で、どの canary でエラーや障害、スロットリング率、または応答時間が増加しているか確認するには、`Client::Synthetic` フィルターを使用して Synthetics canary のクライアントノードを強調表示できます。ノードをクリックすると、リクエスト全体の応答時間の分布が表示されます。2つのノード間のエッジをクリックすると、その接続を経由したリクエストの詳細が表示されます。また、トレースマップの関連するダウンストリームサービスの「リモート」推定ノードを表示できます。

Syntheticsノードをクリックすると、サイドパネルに「View in Synthetics」ボタンが表示され、Syntheticsコンソールにリダイレクトされ、canaryの詳細を確認することができます。



個々のトレースのトレース詳細マップを使用して、各リクエストを詳細に確認する

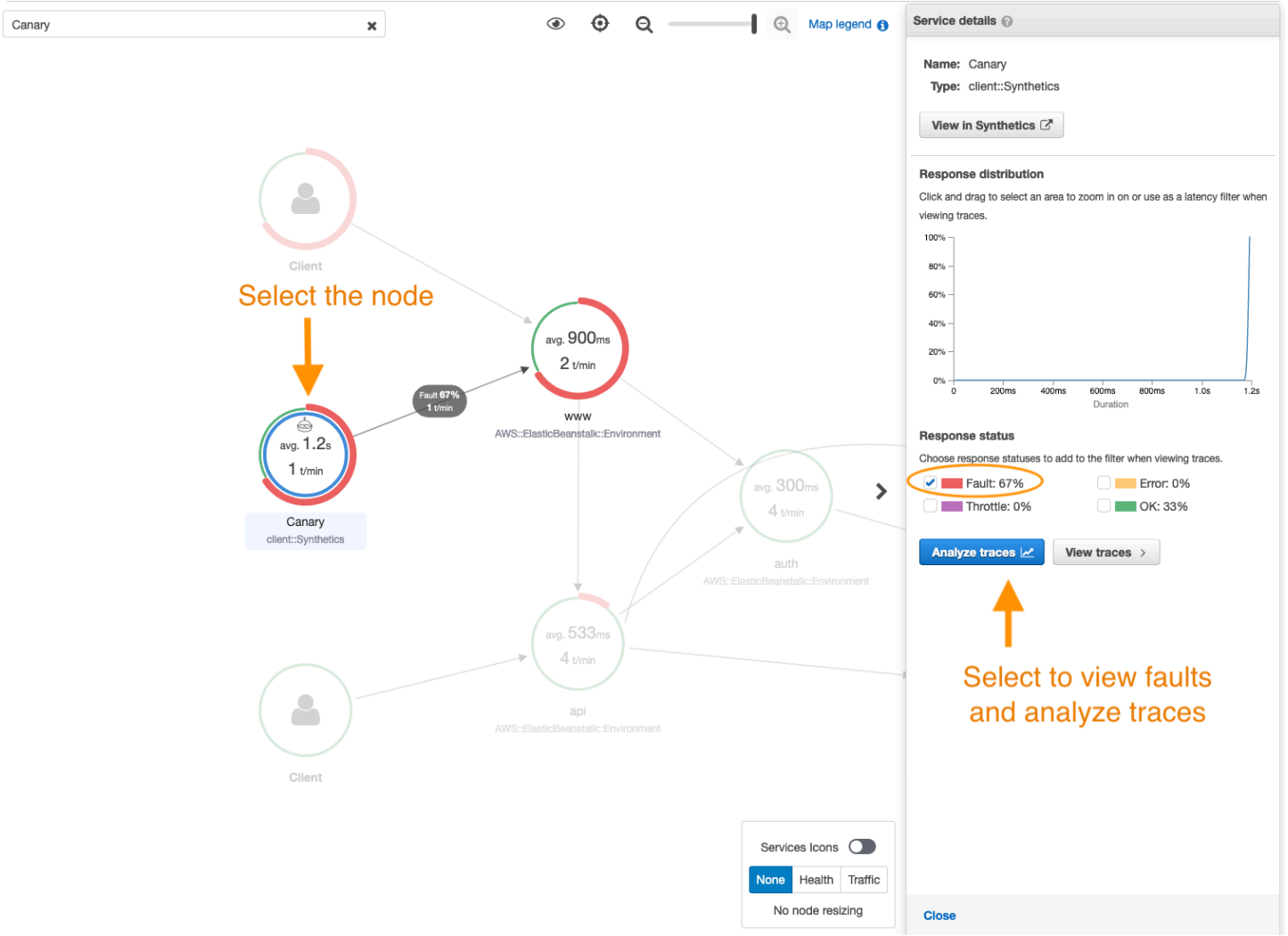
どのサービスで、最も待ち時間が発生しているか、またはエラーが発生しているかを判断するには、トレースマップのトレースを選択してトレース詳細マップを呼び出します。個々のトレース詳細マップには、1 つのリクエストの端末相互間でパスが表示されます。このパスを使用して、起動するサービスを把握し、アップストリームおよびダウンストリームサービスを可視化します。

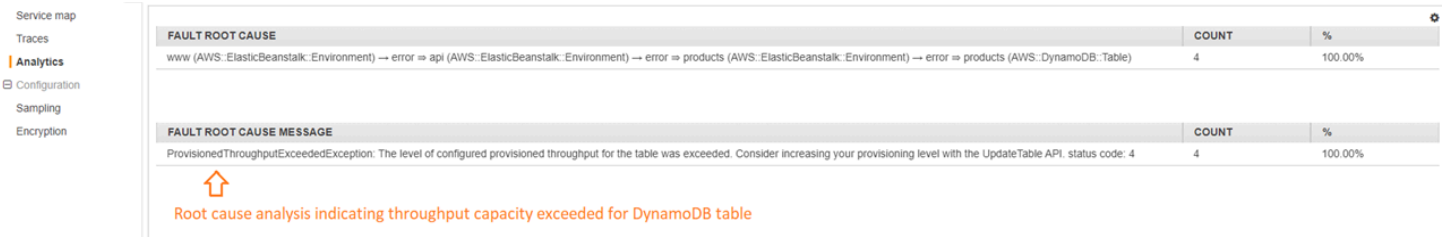


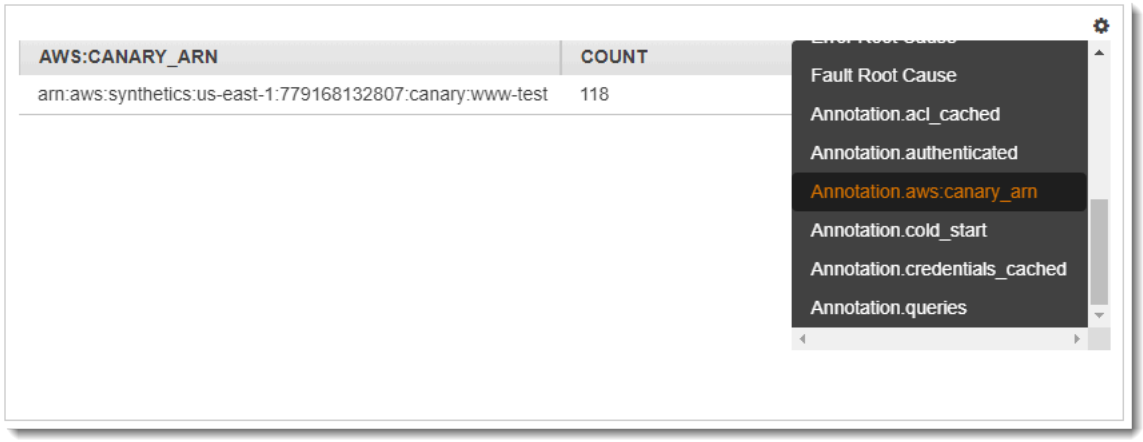
アップストリームおよびダウンストリームサービスで継続的に発生している障害の根本原因を特定する

Synthetics canaryの障害に関するCloudWatchアラームを受診したら、X-Rayのトレースデータの統計的モデリングを使用して、X-Ray Analyticsコンソール内で問題が推定される根本原因を特定します。Analyticsコンソールno、応答時間の根本原因表には、記録されたエンティティパスを表示します。X-Rayは、トレース内の、どのパスが応答時間の最大の原因であるかを判断します。この形式は、検出されたエンティティの階層を示し、最後に応答時間の根本原因を示します。

以下の例では、API Gateway上で実行されている API 「XXX」のSyntheticsテストが、Amazon DynamoDB表からのスループット容量の例外により障害になっていることを示しています。

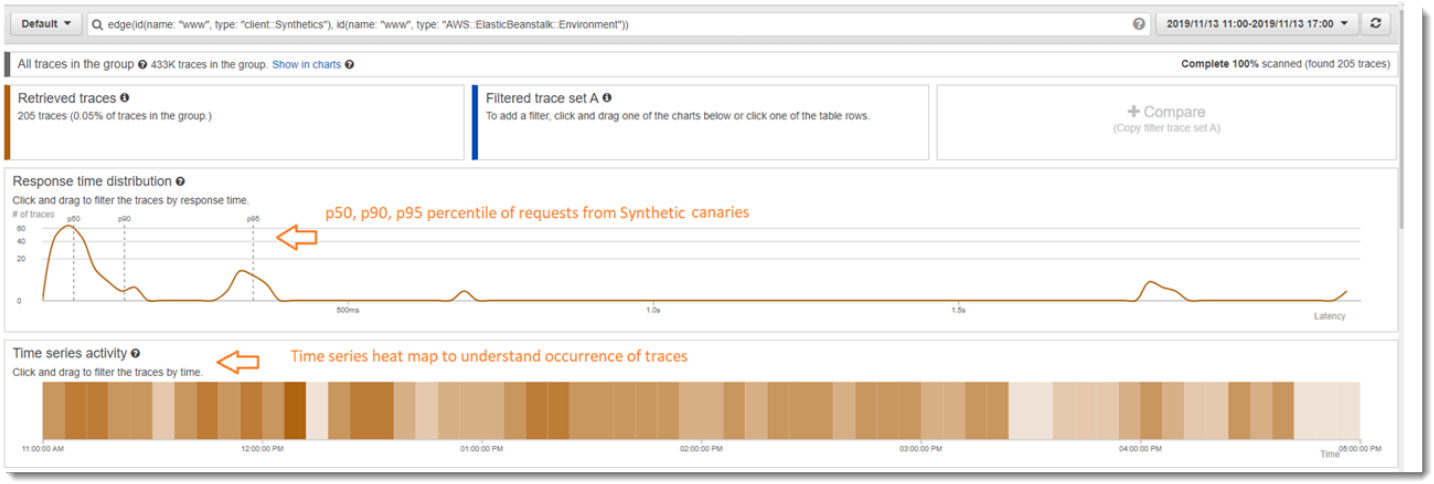

Canary
Map legend
Service details
Name: Canary
Type: client::Synthetics
View in Synthetics
Response distribution
Click and drag to select an area to zoom in on or use as a latency filter when viewing traces.
100%
80%
60%
40%
20%
0%
0
200ms
400ms
600ms
800ms
1.0s
1.2s
Duration
Response status
Choose response statuses to add to the filter when viewing traces.
Fault: 67%
Error: 0%
Throttle: 0%
OK: 33%
Analyze traces
View traces
Select to view faults and analyze traces
Client
Select the node
avg. 900ms
2 t/min
Fault 67%
1 t/min
www
AWS::ElasticBeanstalk::Environment
avg. 1.2s
1 t/min
Canary
client::Synthetics
avg. 300ms
4 t/min
auth
AWS::ElasticBeanstalk::Environment
avg. 533ms
4 t/min
api
AWS::ElasticBeanstalk::Environment
Client
Services Icons
None
Health
Traffic
No node resizing
Close


Service map
Traces
Analytics
Configuration
Sampling
Encryption
FAULT ROOT CAUSE
COUNT
%
www (AWS::ElasticBeanstalk::Environment) ⟶ error ⇒ api (AWS::ElasticBeanstalk::Environment) ⟶ error ⇒ products (AWS::ElasticBeanstalk::Environment) ⟶ error ⇒ products (AWS::DynamoDB::Table)
4
100.00%
FAULT ROOT CAUSE MESSAGE
COUNT
%
ProvisionedThroughputExceededException: The level of configured provisioned throughput for the table was exceeded. Consider increasing your provisioning level with the UpdateTable API. status code: 4
4
100.00%
Root cause analysis indicating throughput capacity exceeded for DynamoDB table



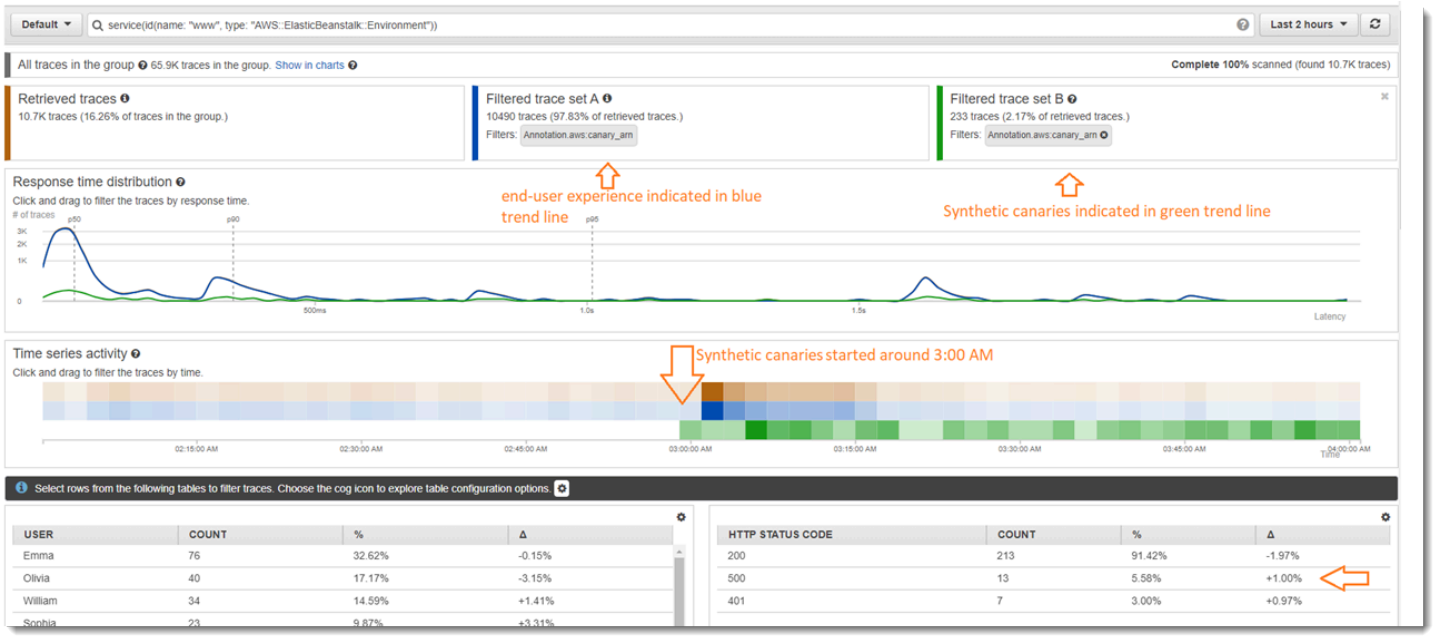
パフォーマンスのボトルネックとトレンドを特定する

Synthetics canary からの継続的なトラフィックを使用して、一定期間にわたってトレース詳細マップを作成し、エンドポイントのパフォーマンスのトレンドを経時的に表示できます。



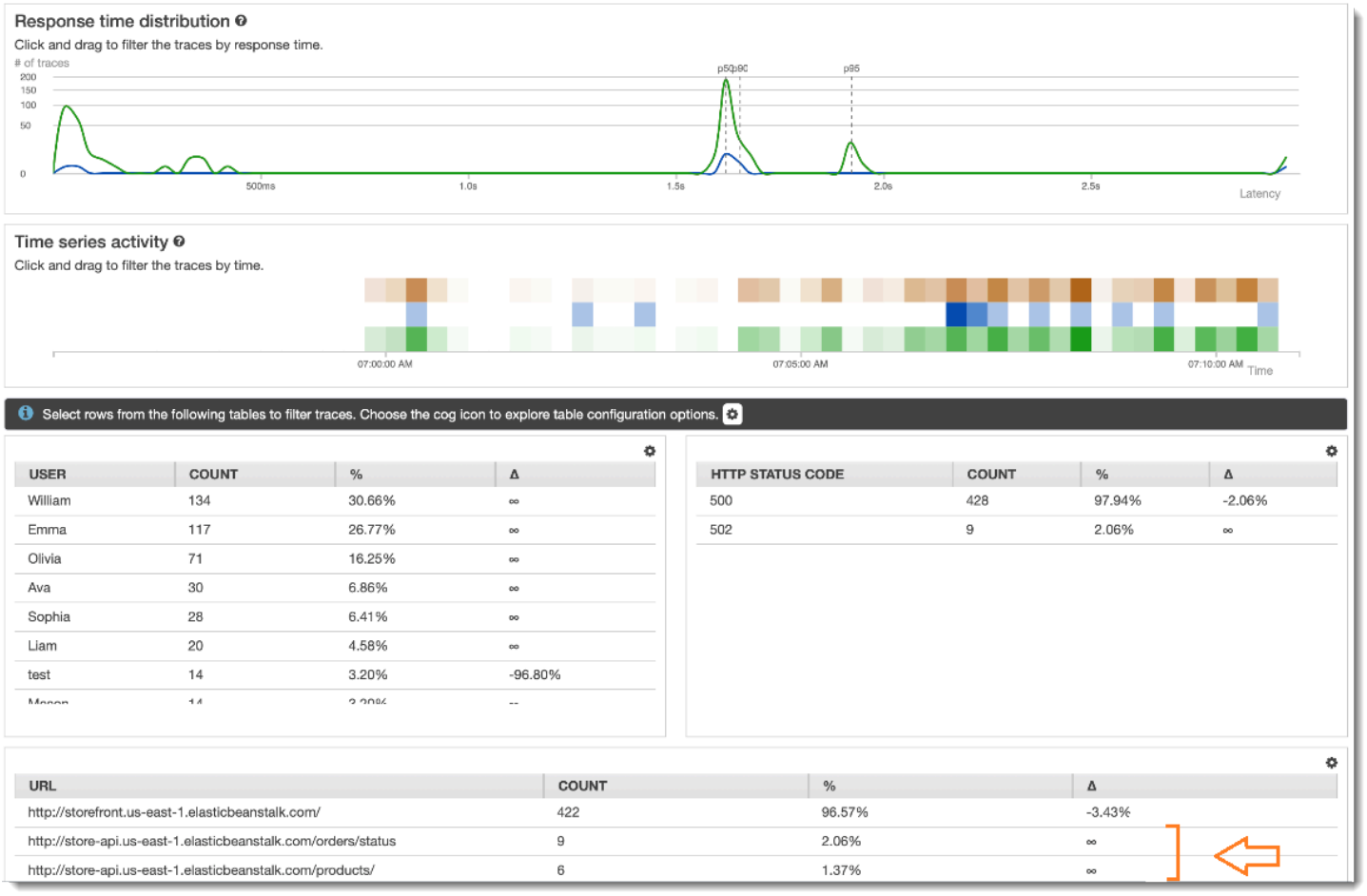
変更前と変更後で待ち時間およびエラー・障害率を比較する

変更が発生した時間を特定して把握し、その変更を canary による問題の発見数の増加と関連させます。X-Ray Analyticsコンソールを使用して、前後の時間範囲を異なるトレースセットとして定義し、応答時間分布に視覚的な差異が生じます。



すべてのAPIとURLに必要なcanaryの受信可能範囲エリアを特定する

X-Ray Analyticsを使用して、ユーザー間でカナリアの満足体験を比較します。以下のUIは、カナリアが青いトレンドラインおよびユーザーが緑のトレンドラインを示しています。また、3つのURLのうち2つにcanaryテストがないことを確認できます。

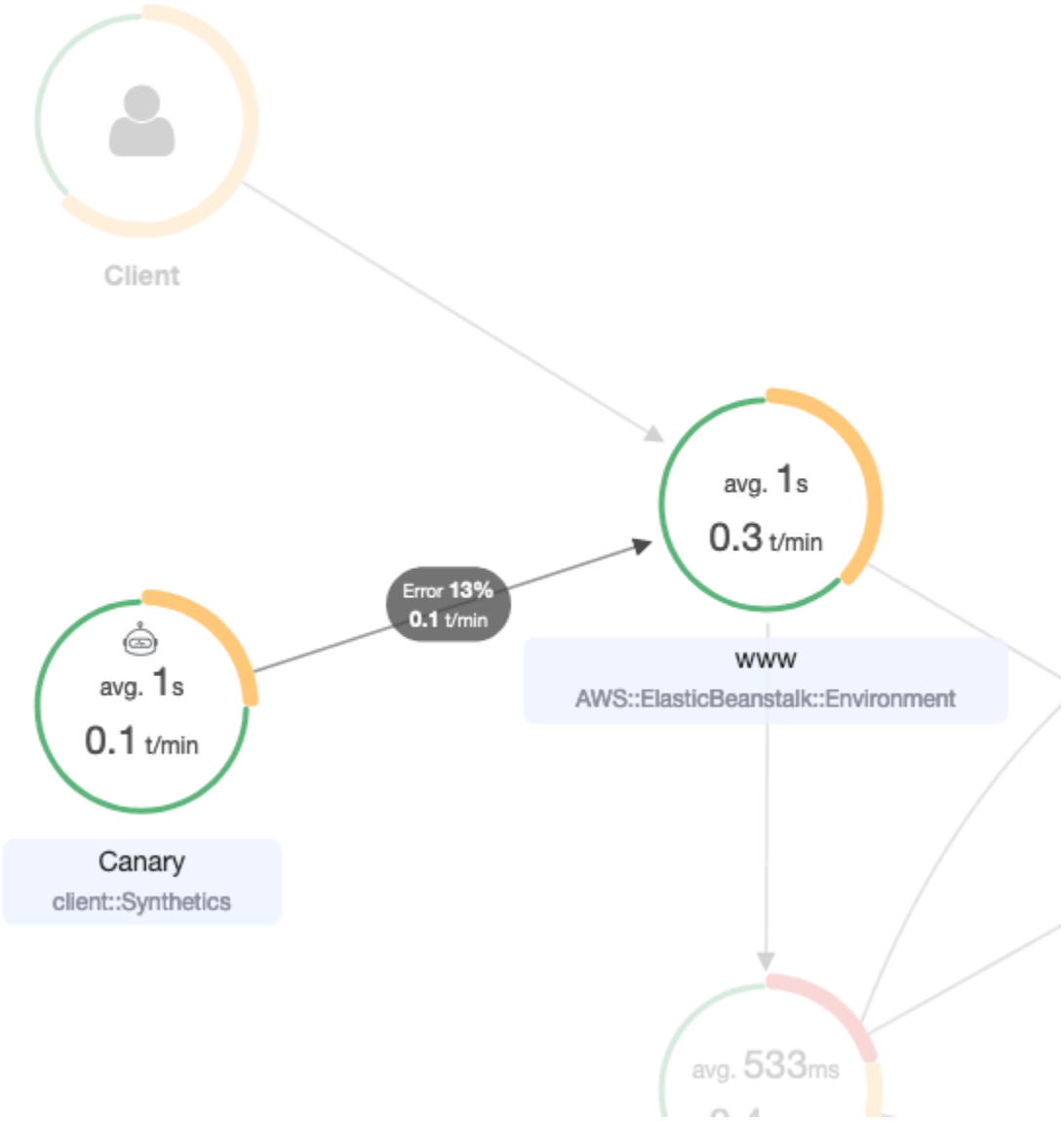


グループを使用してSyntheticsテストに焦点を合わせる

フィルター式を使用してX-Rayグループを作成し、特定のワークフローセットに焦点を合わせることができ、 AWS Elastic Beanstalk例えばNETで実行されているアプリケーション「www」のSyntheticsテストなどです。複合型キーワードを使用して、`service()`そして`edge()`サービスとエッジをフィルタリングします。

Example グループフィルタ式

```
"edge(id(name: "www", type: "client::Synthetics"), id(name: "www", type:
 "AWS::ElasticBeanstalk::Environment"))"
```



AWS Elastic Beanstalk and AWS X-Ray

AWS Elastic Beanstalk プラットフォームには X-Ray デーモンが含まれています。デーモンは、Elastic Beanstalkのコンソールで、オプションを設定するか、または環境設定ファイルで実行することができます。

Java SEプラットフォームでは、Buildfileファイルを使用して、MavenまたはGradleをオンインスタンスで使用するアプリケーションを構築することができます。X-Ray SDK for Java および AWS SDK for Java は Maven から利用できるため、アプリケーションコードのみをデプロイし、インスタンス上で構築することで、すべての依存関係のバンドルやアップロードを回避できます。

Elastic Beanstalkの環境プロパティを使用して、X-Ray SDKを設定できます。Elastic Beanstalkが環境プロパティをアプリケーションに渡すために使用する方法は、プラットフォームによって異なります。お使いのプラットフォームに応じて、X-Ray SDK'sの環境変数または、異なるシステムプロパティを使用してください。

- Node.jsプラットフォーム・環境変数を使用する
- Java SEプラットフォーム・環境変数を使用する
- Tomcatプラットフォーム・システムプロパティを使用する

詳細については、「 AWS Elastic Beanstalk デベロッパーガイド」の AWS X-Ray 「デバッグの設定」を参照してください。

Elastic Load Balancing と AWS X-Ray

Elastic Load Balancingアプリケーションロードバランサーは、受信するHTTPリクエストに、ヘッダーにトレースIDを追加します。`X-Amzn-Trace-Id`

```
X-Amzn-Trace-Id: Root=1-5759e988-bd862e3fe1be46a994272793
```

X-Ray のトレース ID 形式

X-Ray `trace_id` は、ハイフンで区切られた 3 つの数字で構成されています。例えば、`1-58406520-a006649127e371903a2de979` と指定します。これには、以下のものが含まれます：

- バージョン番号、すなわち、1。
- 元のリクエストの時刻。ユニックスエポックタイムで、16 進数 8 桁で表示されます。

  例えば、エポックタイムで 2016 年 12 月 1 日 10:00AM PST (太平洋標準時刻) は `1480615200` 秒、または 16 進数で `58406520` と表示されます。
- グローバルに一意なトレースの 96 ビットの識別子で、24 桁の 16 進数で表示されます。

ロードバランサーはX-Rayにデータを送信しない場合、サービスマップ上にノードとして表示されません。

詳細については、『Elastic Load Balancing Developer Guide』の「Application Load Balancerに関するリクエストのトレース」を参照してください。

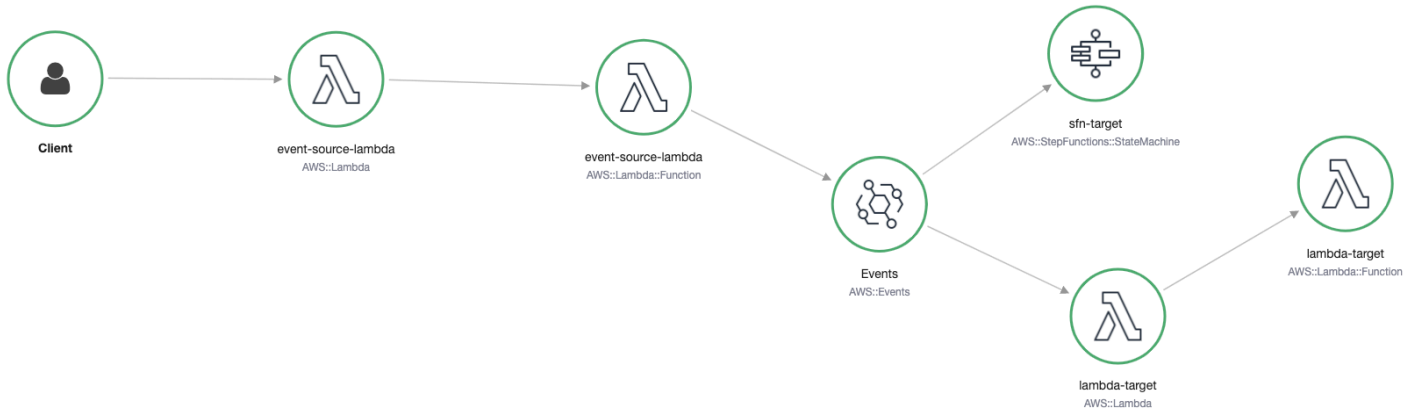
Amazon EventBridge と AWS X-Ray

AWS X-Ray は Amazon EventBridge と統合してEventBridge を通過するイベントをトレースします。X-Ray SDKでインストルメントされたサービスがEventBridgeにイベントを送信すると、トレースコンテキストはトレースヘッダー内でダウンストリームイベントターゲットに伝播されます。???X-Ray SDKは、自動的にトレースヘッダーを取得し、後続のインストルメンテーションに適用します。この継続性により、ユーザーはダウンストリームサービス全体でトレース、分析、およびデバッグを実行できます。また、システムの全体像を把握できるようになります。

詳細については、『EventBridgeユーザーガイド』の「EventBridge X-Rayターゲット」を参照してください。

X-Ray サービスマップでのソースおよびターゲットの表示

X-Ray トレースマップには、以下の例のように、ソースサービスとターゲットサービスを接続するEventBridge イベントノードが表示されます:



トレースコンテキストをイベントターゲットに伝播する

X-Ray SDKを使用すると、EventBridgeイベントソースがダウンストリームイベントターゲットにトレースコンテキストを伝播することを可能にします。以下の言語固有の例では、アクティブトレースが有効になっている Lambda 関数から EventBridge を呼び出す例を示しています。

Java

X-Rayに必要な依存関係を追加します：

- AWS X-Ray SDK for Java
- AWS X-Ray Recorder SDK for Java

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.services.eventbridge.AmazonEventBridge;
import com.amazonaws.services.eventbridge.AmazonEventBridgeClientBuilder;
import com.amazonaws.services.eventbridge.model.PutEventsRequest;
import com.amazonaws.services.eventbridge.model.PutEventsRequestEntry;
import com.amazonaws.services.eventbridge.model.PutEventsResult;
import com.amazonaws.services.eventbridge.model.PutEventsResultEntry;
import com.amazonaws.xray.handlers.TracingHandler;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.lang.StringBuilder;
import java.util.Map;
import java.util.List;
import java.util.Date;
import java.util.Collections;

/*
   Add the necessary dependencies for XRay:
   https://mvnrepository.com/artifact/com.amazonaws/aws-java-sdk-xray
   https://mvnrepository.com/artifact/com.amazonaws/aws-xray-recorder-sdk-aws-sdk
*/
public class Handler implements RequestHandler<SQSEvent, String>{
  private static final Logger logger = LoggerFactory.getLogger(Handler.class);

  /*
    build EventBridge client
  */
  private static final AmazonEventBridge eventsClient =
 AmazonEventBridgeClientBuilder
          .standard()
          // instrument the EventBridge client with the XRay Tracing Handler.
          // the AWSXRay globalRecorder will retrieve the tracing-context
          // from the lambda function and inject it into the HTTP header.
          // be sure to enable 'active tracing' on the lambda function.
          .withRequestHandlers(new TracingHandler(AWSXRay.getGlobalRecorder()))
          .build();
```

```
  @Override
  public String handleRequest(SQSEvent event, Context context)
  {
    PutEventsRequestEntry putEventsRequestEntry0 = new PutEventsRequestEntry();
    putEventsRequestEntry0.setTime(new Date());
    putEventsRequestEntry0.setSource("my-lambda-function");
    putEventsRequestEntry0.setDetailType("my-lambda-event");
    putEventsRequestEntry0.setDetail("{\"lambda-source\":\"sqs\"}");
    PutEventsRequest putEventsRequest = new PutEventsRequest();
    putEventsRequest.setEntries(Collections.singletonList(putEventsRequestEntry0));
    // send the event(s) to EventBridge
    PutEventsResult putEventsResult = eventsClient.putEvents(putEventsRequest);
    try {
      logger.info("Put Events Result: {}", putEventsResult);
    } catch(Exception e) {
      e.getStackTrace();
    }
    return "success";
  }
}
```

Python

requirements.txt ファイルに以下の依存関係を追加します：

```
aws-xray-sdk==2.4.3
```

```
import boto3
from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch_all

# apply the XRay handler to all clients.
patch_all()

client = boto3.client('events')

def lambda_handler(event, context):
    response = client.put_events(
        Entries=[
            {
                'Source': 'foo',
                'DetailType': 'foo',
```

```
                'Detail': '{\"foo\": \"foo\"}'
            },
        ]
    )
    return response
```

Go

```
package main

import (
  "context"
  "github.com/aws/aws-lambda-go/lambda"
  "github.com/aws/aws-lambda-go/events"
  "github.com/aws/aws-sdk-go/aws/session"
  "github.com/aws/aws-xray-sdk-go/xray"
  "github.com/aws/aws-sdk-go/service/eventbridge"
  "fmt"
)

var client = eventbridge.New(session.New())


func main() {
 //Wrap the eventbridge client in the AWS XRay tracer
  xray.AWS(client.Client)
  lambda.Start(handleRequest)
}

func handleRequest(ctx context.Context, event events.SQSEvent) (string, error) {
  _, err := callEventBridge(ctx)
  if err != nil {
    return "ERROR", err
  }
  return "success", nil
}


func callEventBridge(ctx context.Context) (string, error) {
    entries := make([]*eventbridge.PutEventsRequestEntry, 1)
    detail := "{ \"foo\": \"foo\"}"
    detailType := "foo"
    source := "foo"
```

```
    entries[0] = &eventbridge.PutEventsRequestEntry{
        Detail: &detail,
        DetailType: &detailType,
        Source: &source,
    }

  input := &eventbridge.PutEventsInput{
     Entries: entries,
  }

  // Example sending a request using the PutEventsRequest method.
  resp, err := client.PutEventsWithContext(ctx, input)

  success := "yes"
  if err == nil { // resp is now filled
      success = "no"
      fmt.Println(resp)
  }
  return success, err
}
```

Node.js

```
const AWSXRay = require('aws-xray-sdk')
//Wrap the aws-sdk client in the AWS XRay tracer
const AWS = AWSXRay.captureAWS(require('aws-sdk'))
const eventBridge = new AWS.EventBridge()

exports.handler = async (event) => {

  let myDetail = { "name": "Alice" }

  const myEvent = {
    Entries: [{
      Detail: JSON.stringify({ myDetail }),
      DetailType: 'myDetailType',
      Source: 'myApplication',
      Time: new Date
    }]
  }

  // Send to EventBridge
  const result = await eventBridge.putEvents(myEvent).promise()
```

```
  // Log the result
  console.log('Result: ', JSON.stringify(result, null, 2))

}
```

C#

以下のX-RayパッケージをC#の依存関係に追加します：

```
<PackageReference Include="AWSXRayRecorder.Core" Version="2.6.2" />
<PackageReference Include="AWSXRayRecorder.Handlers.AwsSdk" Version="2.7.2" />
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Amazon;
using Amazon.Util;
using Amazon.Lambda;
using Amazon.Lambda.Model;
using Amazon.Lambda.Core;
using Amazon.EventBridge;
using Amazon.EventBridge.Model;
using Amazon.Lambda.SQSEvents;
using Amazon.XRay.Recorder.Core;
using Amazon.XRay.Recorder.Handlers.AwsSdk;
using Newtonsoft.Json;
using Newtonsoft.Json.Serialization;

[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.Json.JsonSerializer))]

namespace blankCsharp
{
  public class Function
  {
    private static AmazonEventBridgeClient eventClient;

    static Function() {
      initialize();
    }
```

```
    static async void initialize() {
      //Wrap the AWS SDK clients in the AWS XRay tracer
      AWSSDKHandler.RegisterXRayForAllServices();
      eventClient = new AmazonEventBridgeClient();
    }

    public async Task<PutEventsResponse> FunctionHandler(SQSEvent invocationEvent,
 ILambdaContext context)
    {
      PutEventsResponse response;
      try
      {
        response = await callEventBridge();
      }
      catch (AmazonLambdaException ex)
      {
        throw ex;
      }

      return response;
    }

    public static async Task<PutEventsResponse> callEventBridge()
    {
      var request = new PutEventsRequest();
      var entry = new PutEventsRequestEntry();
      entry.DetailType = "foo";
      entry.Source = "foo";
      entry.Detail = "{\"instance_id\":\"A\"}";
      List<PutEventsRequestEntry> entries = new List<PutEventsRequestEntry>();
      entries.Add(entry);
      request.Entries = entries;
      var response = await eventClient.PutEventsAsync(request);
      return response;
    }
  }
}
```

AWS Lambda and AWS X-Ray

を使用して関数 AWS X-Ray をトレースできます AWS Lambda 。Lambda は X-Ray デーモンを実行し、関数の呼び出しと実行に関する詳細でセグメントを記録します。さらに計測するには、X-Ray SDK を関数にバンドルして送信呼び出しを記録し、注釈とメタデータを追加できます。

Lambda 関数が他の計測サービスより呼び出されると、Lambda は、追加の設定なしで、すでにサンプルを作成したリクエストをトレースします。アップストリームサービスは、計測されたウェブアプリケーションや別の Lambda 関数とすることができます。サービスは、計測された AWS SDK クライアントを使用して関数を直接呼び出すか、計測された HTTP クライアントを使用して API Gateway API を呼び出すことができます。

AWS X-Ray は、 AWS Lambda および Amazon SQS を使用したイベント駆動型アプリケーションのトレースをサポートします。CloudWatch コンソールを使用すると、各リクエストが Amazon SQS のキューに入れられ、ダウンストリーム Lambda 関数によって処理される過程を各リクエストの接続されたビューで確認できます。アップストリームメッセージプロデューサーからのトレースは、ダウンストリーム Lambda コンシューマーノードからのトレースに自動的にリンクされるため、アプリケーションのエンドツーエンドのビューが作成されます。詳細については、「イベント駆動型アプリケーションのトレース」を参照してください。

> **Note**
>
> ダウンストリーム Lambda 関数でトレースを有効にした場合は、ダウンストリーム関数がトレースを生成するために、ダウンストリーム関数を呼び出すルート Lambda 関数でもトレースを有効にする必要があります。

Lambda 関数がスケジュールで実行される場合や、計測されていないサービスによって呼び出される場合は、アクティブトレースで呼び出しをサンプルおよび記録するように Lambda を設定できます。

AWS Lambda 関数で X-Ray 統合を設定するには

1. AWS Lambda コンソールを開きます。
2. 左のナビゲーションペインから [関数] を選択します。
3. 関数を選択します。

4. [設定] タブで、[その他の監視ツール] カードが表示されるまで下へスクロールします。このカードは、左側のナビゲーションペインで [モニタリングおよび運用ツール] を選択して見つけることもできます。
5. [Edit] (編集) を選択します。
6. [AWS X-Ray] で、[アクティブトレース] を有効にします。

対応する X-Ray SDK、 Lambda があるランタイムでも、X-Ray デーモンが実行されます。

Lambda の X-Ray SDK

- Go X-Ray SDK for Go – Go 1.7 以降のランタイム
- X-Ray SDK for Java – Java 8 ランタイム
- X-Ray SDK for Node.js – Node.js 4.3 以降のランタイム
- X-Ray SDK for Python – Python 2.7、Python 3.6 以降のランタイム
- X-Ray SDK for .NET – .NET Core 2.0 以降のランタイム

Lambda で X-Ray SDK を使用するには、新しいバージョンを作成するたびに、関数コードでバンドルします。Lambda 関数は、他のサービスで実行されているアプリケーションを計測する場合と同じメソッドを使用して計測することができます。主な違いは、SDK を使用して、受信リクエストの計測、サンプリングの決定、セグメントの作成を行わないことです。

Lambda 関数とウェブアプリケーションの計測のもう 1 つの違いは、Lambda が作成して X-Ray に送信するセグメントは、関数コードで変更できないことです。サブセグメントを作成し、そこに注釈とメタデータを記録できますが、親セグメントに注釈とメタデータを追加することはできません。

詳細については、AWS 開発者ガイドの「AWS Lambda X-Ray を使用する」を参照してください。

AWS Step Functions and AWS X-Ray

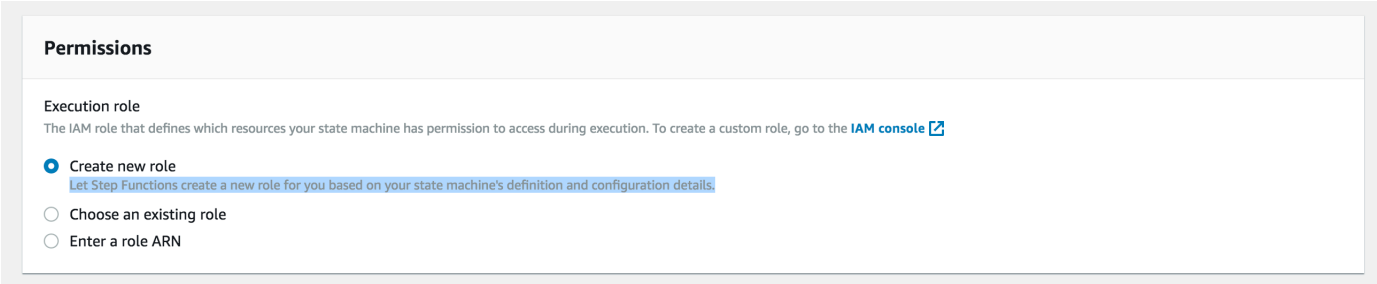
AWS X-Ray は と統合 AWS Step Functions して、Step Functions のリクエストを追跡および分析します。ステートマシンのコンポーネントの視覚化、パフォーマンスのボトルネックの特定、およびエラーの原因となったリクエストのトラブルシューティングを行うことができます。詳細については、AWS Step Functions デベロッパーガイドのAWS X-Ray 「」および「Step Functions」を参照してください。

新しいステートマシンの作成時に X-Ray トレースを有効にするには

1. https://console.aws.amazon.com/states/ で Step Functions コンソールを開きます。
2. [ステートマシンの作成] を選択します。
3. [ステートマシンを定義する] ページで、[コードスニペットで作成] および [テンプレートで開始する] を選択します。サンプルプロジェクトを実行することを選択した場合、作成中に X-Ray トレースを有効にすることはできません。の代わりに、ステートマシンを作成した後で X-Ray トレースを有効にします。
4. [Next (次へ)] を選択します。
5. [詳細の指定] ページで、ステートマシンを設定します。
6. [X-Ray トレースを有効にする] を選択します。

既存のステートマシンで、X-Ray トレースを有効にするには

1. Step Functions コンソールで、トレースを有効にするステートマシンを選択します。
2. [編集] を選択します。
3. [X-Ray トレースを有効にする] を選択します。
4. （オプション）[Permissions] ウィンドウから [Create new role] を選択して、X-Ray のアクセス許可を含むステートマシンの新しいロールを自動生成します。



5. [Save] を選択します。

> **Note**
>
> 新しいステートマシンを作成すると、リクエストがサンプリングされ、Amazon API Gateway や などのアップストリームサービスでトレースが有効になっている場合、自動的にトレースされます AWS Lambda。テンプレートなど、コンソールで設定されていない既存のステートマシンについては AWS CloudFormation 、X-Ray トレースを有効にするのに十分なアクセス許可を付与する IAM ポリシーがあることを確認します。

のアプリケーションの計測 AWS X-Ray

アプリケーションを計測するには、アプリケーション内の受信と送信リクエストおよびその他のイベントのトレースデータを、各リクエストに関するメタデータと一緒に送信する必要があります。特定の要件に基づいて、選択または組み合わせることができる計測オプションがいくつかあります。

- 自動計測 – 通常、設定の変更または自動計測エージェントや他のメカニズムを追加して、コードをゼロに変更してアプリケーションを計測します。
- ライブラリ計測 – アプリケーションコードの変更を最小限に抑えて、 AWS SDK、Apache HTTP クライアント、SQL クライアントなどの特定のライブラリやフレームワークを対象とした構築済みの計測を追加します。
- 手動計測 – トレース情報を送信する各場所で、アプリケーションに計測コードを追加します。

X-Ray トレース用のアプリケーションの計測に使用できる SDK、エージェント、およびツールがいくつかあります。

トピック

- AWS Distro for OpenTelemetry を使用したアプリケーションの計測
- AWS X-Ray SDKs を使用したアプリケーションの計測
- Distro for OpenTelemetry AWS と X-Ray SDKs の選択

AWS Distro for OpenTelemetry を使用したアプリケーションの計測

AWS Distro for OpenTelemetry (ADOT) は、Cloud Native Computing Foundation (CNCF) OpenTelemetry プロジェクトに基づく AWS ディストリビューションです。OpenTelemetry は、分散トレースとメトリクスを収集するためのオープンソース API、ライブラリ、およびエージェントの単一セットを提供します。このツールキットは、SDK、自動計測エージェント、およびコレクタを含むアップストリームの OpenTelemetry コンポーネントのディストリビューションであり、によってテスト、最適化、保護、およびサポートされます。 AWS。

ADOT を使用すると、エンジニアはアプリケーションを一度計測し、相関メトリクスとトレースを Amazon CloudWatch AWS X-Ray、Amazon OpenSearch Service などの複数の AWS モニタリングソリューションに送信できます。

X-Ray を ADOT で使用するには、X-Ray で使用できる OpenTelemetry SDK と、X-Ray で使用できる AWS Distro for OpenTelemetry Collector の 2 つのコンポーネントが必要です。 AWS Distro for OpenTelemetry を AWS X-Ray およびその他の で使用する方法の詳細については AWS のサービス、 AWS Distro for OpenTelemetry ドキュメントを参照してください。

言語サポートと使用方法の詳細については、「AWS Observability on Github」を参照してください。

> **Note**
>
> CloudWatch エージェントを使用して Amazon EC2 インスタンスとオンプレミスサーバーからメトリクス、ログ、トレースを収集できます。CloudWatch エージェントバージョン 1.300025.0 以降では、OpenTelemetry または X-Ray クライアント SDK からトレースを収集し、それらを X-Ray に送信できます。 AWS Distro for OpenTelemetry (ADOT) コレクターまたは X-Ray デーモンの代わりに CloudWatch エージェントを使用してトレースを収集することで、管理するエージェントの数を減らすことができます。詳細については、「CloudWatch ユーザーガイド」の「CloudWatch エージェント」のトピックを参照してください。

ADOT には以下が含まれます。

- AWS Distro for OpenTelemetry Go
- AWS Distro for OpenTelemetry Java
- AWS Distro for OpenTelemetry JavaScript
- AWS Distro for OpenTelemetry Python
- AWS Distro for OpenTelemetry .NET

ADOT は現在、JavaおよびPython用の自動計測サポートを含んでいます。さらに、ADOT は、ADOT Managed AWS Lambda Layers を介して、Java、Node.js、Python ランタイムを使用して Lambda 関数とそのダウンストリームリクエストの自動計測を有効にします。 https://aws-otel.github.io/docs/getting-started/lambda

Java および Go 用の ADOT SDK は、X-Ray の一元化されたサンプリングルールをサポートしています。他の言語の X-Ray サンプリングルールのサポートが必要な場合は、 AWS X-Ray SDK の使用を検討してください。

> **Note**
>
> W3C トレース ID を X-Ray に送信できるようになりました。デフォルトでは、OpenTelemetry で作成されたトレースには、W3C トレースコンテキスト仕様に基づくトレース ID 形式があります。これは、X-Ray SDK または X-Ray と統合された AWS サービスを使用して作成されたトレース IDs の形式とは異なります。W3C 形式のトレース ID が X-Ray で確実に受け入れられるようにするため、バージョン 0.86.0 以降の AWS X-Ray エクスポーターを使用する必要があります。このエクスポーターはバージョン 0.34.0 以降の ADOT Collector に含まれています。以前のバージョンのエクスポーターはトレース ID のタイムスタンプを検証し、これにより W3C トレース ID が拒否される可能性があります。

AWS X-Ray SDKs を使用したアプリケーションの計測

AWS X-Ray には、X-Ray にトレースを送信するためにアプリケーションを計測するための言語固有の SDKs のセットが含まれています。各 X-Ray SDK は、以下を提供します。

- インターセプター コードに追加して受信 HTTP リクエストをトレースする
- アプリケーションが他の を呼び出すために使用する AWS SDK クライアントを計測するクライアントハンドラー AWS のサービス
- HTTP クライアント 他の内部および外部 HTTP ウェブサービス呼び出しを計測する

X-Ray SDKsSQL データベースへの呼び出しの計測、 AWS SDK クライアントの自動計測、その他の機能もサポートしています。トレースデータを直接 X-Ray に送信する代わりに、SDK は JSON セグメントドキュメントを UDP トラフィックをリッスンしているデーモンプロセスに送信します。X-Ray デーモンはセグメントをキューにバッファし、バッチで X-Ray にアップロードします。

次の言語別の SDK が用意されています。

- AWS X-Ray SDK for Go
- AWS X-Ray SDK for Java
- AWS X-Ray Node.js 用 SDK
- AWS X-Ray SDK for Python
- AWS X-Ray SDK for .NET
- AWS X-Ray SDK for Ruby

X-Ray は現在、Java用の自動計測サポートを含んでいます。

Distro for OpenTelemetry AWS と X-Ray SDKs の選択

X-Ray に含まれる SDK は、 AWSによって提供される緊密に統合された計測ソリューションの一部です。 AWS Distro for OpenTelemetry は、X-Ray が数あるトレースソリューションの 1 つにすぎない、より広範な業界ソリューションの一部です。どちらの方法でも X-Ray でエンドツーエンドのトレースを実装できますが、最も有用なアプローチを決定するには、違いを理解することが重要です。

以下が必要な場合は、 AWS Distro for OpenTelemetry を使用してアプリケーションを計測することをお勧めします。

- コードを再計測することなく複数の異なるトレースバックエンドにトレースを送信する機能
- OpenTelemetry コミュニティによって維持されている、各言語の多数のライブラリ計測のサポート
- Java、Python、Node.js を使用するときにコード変更する必要がない、テレメトリデータの収集に必要なすべてがパッケージ化された完全マネージド型の Lambda レイヤー

> **Note**
>
> AWS Distro for OpenTelemetry は、Lambda 関数を計測するための簡単な入門エクスペリエンスを提供します。ただし、OpenTelemetry に備わる柔軟性が原因で、Lambda 関数が必要とするメモリが増加し、呼び出し時に、コールドスタートのレイテンシーが増加し、追加料金が発生する可能性があります。低レイテンシー向けに最適化していて、動的に設定可能なバックエンド送信先など、OpenTelemetry の高度な機能を必要としない場合は、AWS X-Ray SDK を使用してアプリケーションを計測できます。

以下が必要な場合は、アプリケーションの計測に X-Ray SDK を選択することをお勧めします。

- 緊密に統合されたシングルベンダーソリューション
- Node.js、Python、Ruby、.NET を使用する場合に、X-Ray コンソールからサンプリングルールを設定し、複数のホスト間で自動的に使用する機能を含む、X-Ray の一元化されたサンプリングルールとの統合

トランザクション検索

トランザクション検索は、アプリケーショントランザクションのスパンを完全に可視化するために使用できるインタラクティブな分析エクスペリエンスです。スパンは分散トレースの基本的なオペレーション単位であり、アプリケーションまたはシステム内の特定のアクションまたはタスクを表します。すべてのスパンは、トランザクションの特定のセグメントに関する詳細を記録します。これらの詳細には、開始時刻と終了時刻、期間、関連するメタデータが含まれます。これには、顧客 ID や注文 ID などのビジネス属性が含まれる場合があります。スパンは親子階層に配置されます。この階層は、さまざまなコンポーネントまたはサービスにわたるトランザクションのフローを完全にマッピングするトレースを形成します。

詳細については、「Transaction Search」を参照してください。

OpenTelemetry Protocol (OTLP) エンドポイント

OpenTelemetry は、テレメトリデータを収集してルーティングするための標準化されたプロトコルとツールを IT チームに提供するオープンソースのオブザーバビリティフレームワークです。メトリクス、ログ、トレースなどのアプリケーションテレメトリデータを計測、生成、収集、エクスポートするための統一された形式を、分析とインサイトのためのモニタリングプラットフォームに提供します。OpenTelemetry を使用することで、チームはベンダーのロックインを回避し、オブザーバビリティソリューションの柔軟性を確保できます。

OpenTelemetry を使用すると、OpenTelemetry Protocol (OTLP) エンドポイントにトレースを直接送信し、CloudWatch Application Signals out-of-the使用できるアプリケーションパフォーマンスモニタリングエクスペリエンスを得ることができます。

詳細については、「OpenTelemetry」を参照してください。

Go の使用

X-Ray にトレースを送信する Go アプリケーションを計測するには、次の 2 つの方法があります。

- AWS Distro for OpenTelemetry Go – AWS Distro for OpenTelemetry Collector を介して、相関メトリクスとトレースを Amazon CloudWatch AWS X-Ray、Amazon OpenSearch Service などの複数の AWS モニタリングソリューションに送信するための一連のオープンソースライブラリを提供する AWS ディストリビューション。
- AWS X-Ray SDK for Go – X-Ray デーモンを介してトレースを生成して X-Ray に送信するためのライブラリのセット。

詳細については、「Distro for OpenTelemetry AWS と X-Ray SDKs の選択」を参照してください。

AWS Distro for OpenTelemetry Go

AWS Distro for OpenTelemetry Go を使用すると、アプリケーションを一度計測し、相関メトリクスとトレースを Amazon CloudWatch AWS X-Ray、Amazon OpenSearch Service などの複数の AWS モニタリングソリューションに送信できます。OpenTelemetry 用 AWS Distro で X-Ray を使用するには、2 つのコンポーネントが必要です。X-Ray での使用が有効になっている OpenTelemetry SDK と、X-Ray での使用が有効になっている AWS OpenTelemetry Collector 用 Distro。

開始するには、「」を参照してください。AWS Distro for OpenTelemetry Go ドキュメント。

AWS Distro for OpenTelemetry を AWS X-Ray およびその他の で使用する方法の詳細については AWS のサービス、AWS 「 Distro for OpenTelemetry」またはAWS 「 Distro for OpenTelemetry ドキュメント」を参照してください。

言語サポートと使用方法の詳細については、「AWS Observability on Github」を参照してください。

AWS X-Ray SDK for Go

は、X-Ray SDK for Go アプリケーション用のライブラリのセットです。トレースデータを作成して X-Ray デーモンに送信するためのクラスとメソッドを提供します。トレースデータには、アプリケーションによって提供される受信 HTTP リクエストに関する情報、および AWS SDK、HTTP クライアント、または SQL データベースコネクタを使用してアプリケーションがダウンストリームサービスに対して行う呼び出しが含まれます。セグメントを手動で作成し、注釈およびメタデータにデバッグ情報を追加することもできます。

go get を使用して GitHub リポジトリから SDK をダウンロードします。

```
$ go get -u github.com/aws/aws-xray-sdk-go/...
```

ウェブアプリケーションの場合は、xray.Handler 関数を使用して受信リクエストをトレースします。メッセージハンドラーでは、トレース対象リクエストごとに「セグメント」を作成し、レスポンスが送信されるとセグメントを完了します。セグメントが開いている間、SDK クライアントのメソッドを使用してセグメントに情報を追加し、サブセグメントを作成してダウンストリーム呼び出しをトレースできます。また、SDK では、セグメントが開いている間にアプリケーションがスローする例外を自動的に記録します。

インストルメント済みアプリケーションまたはサービスによって呼び出される Lambda 関数の場合、Lambda は トレースヘッダー を読み取り、サンプリングされたリクエストを自動的にトレースします。その他の関数については、Lambda の設定 から受信リクエストのサンプリングとトレースを行うことができます。いずれの場合も、Lambda はセグメントを作成し、X-Ray SDK に提供します。

> **Note**
>
> Lambda では、X-Ray SDK はオプションです。関数でこれを使用しない場合、サービスマップには Lambda サービスのノードと Lambda 関数ごとに 1 つのノードが含まれます。SDK を追加することで、関数コードをインストルメントして、Lambda で記録された関数セグメントにサブセグメントを追加することができます。詳細については「AWS Lambda and AWS X-Ray」を参照してください。

次に、クライアントをAWS 関数の呼び出しでラップします。このステップでは、X-Ray 計測が任意のクライアントメソッドを呼び出すようにします。 計測が SQL データベースを呼び出すようにすることもできます。

SDK を入手したら、レコーダーとミドルウェアを設定して、その動作をカスタマイズします。プラグインを追加して、アプリケーションを実行しているコンピューティングリソースに関するデータを記録したり、サンプリングルールを定義することでサンプリングの動作をカスタマイズしたり、アプリケーションログに SDK からの情報をより多くあるいは少なく表示するように口グレベルを設定できます。

アプリケーションが注釈やメタデータで行うリクエストや作業に関する追加情報を記録します。注釈は、フィルタ式で使用するためにインデックス化されたシンプルなキーと値のペアで、特定のデータ

が含まれているトレースを検索できます。メタデータのエントリは制約が緩やかで、JSON にシリアル化できるオブジェクトと配列全体を記録できます。

> **注釈とメタデータ**
>
> 注釈およびメタデータとは、X-Ray SDK を使用してセグメントに追加する任意のテキストです。注釈は、フィルタ式用にインデックス付けされます。メタデータはインデックス化されませんが、X-Ray コンソールまたは API を使用して raw セグメントで表示できます。X-Ray への読み取りアクセスを許可した人は誰でも、このデータを表示できます。

コードに多数の計測されたクライアントがある場合、単一のリクエストセグメントには計測されたクライアントで行われた呼び出しごとに 1 個の多数のサブセグメントを含めることができます。カスタムサブセグメントで、クライアント呼び出しをラップすることで、サブセグメントを整理してグループできます。関数全体またはコードの任意のセクションのサブセグメントを作成し、親セグメントにすべてのレコードを記述する代わりにサブセグメントにメタデータと注釈を記録できます。

要件

X-Ray SDK for Go 1.9 以降が必要です。

SDK は、コンパイル時および実行時に次のライブラリに依存します。

- AWS SDK for Go バージョン 1.10.0 以降

これらの依存関係は SDK の `README.md` ファイルで宣言されています。

リファレンスドキュメント

SDK をダウンロードしたら、ドキュメントをビルドしてローカルにホストし、ウェブブラウザで表示します。

リファレンスドキュメントを表示するには

1. `$GOPATH/src/github.com/aws/aws-xray-sdk-go` (Linux または Mac) ディレクトリまたは `%GOPATH%\src\github.com\aws\aws-xray-sdk-go` (Windows) フォルダに移動します。
2. godoc コマンドを実行します。

```
$ godoc -http=:6060
```

3. `http://localhost:6060/pkg/github.com/aws/aws-xray-sdk-go/` でブラウザを開きます。

X-Ray SDK for Goの設定

環境変数を使って X-Ray SDK for Go の設定を指定するには、`Config` オブジェクトで `Configure` を呼び出すか、デフォルト値を使用します。環境変数は `Config` 値よりも優先されます。これはデフォルト値よりも優先されます。

セクション

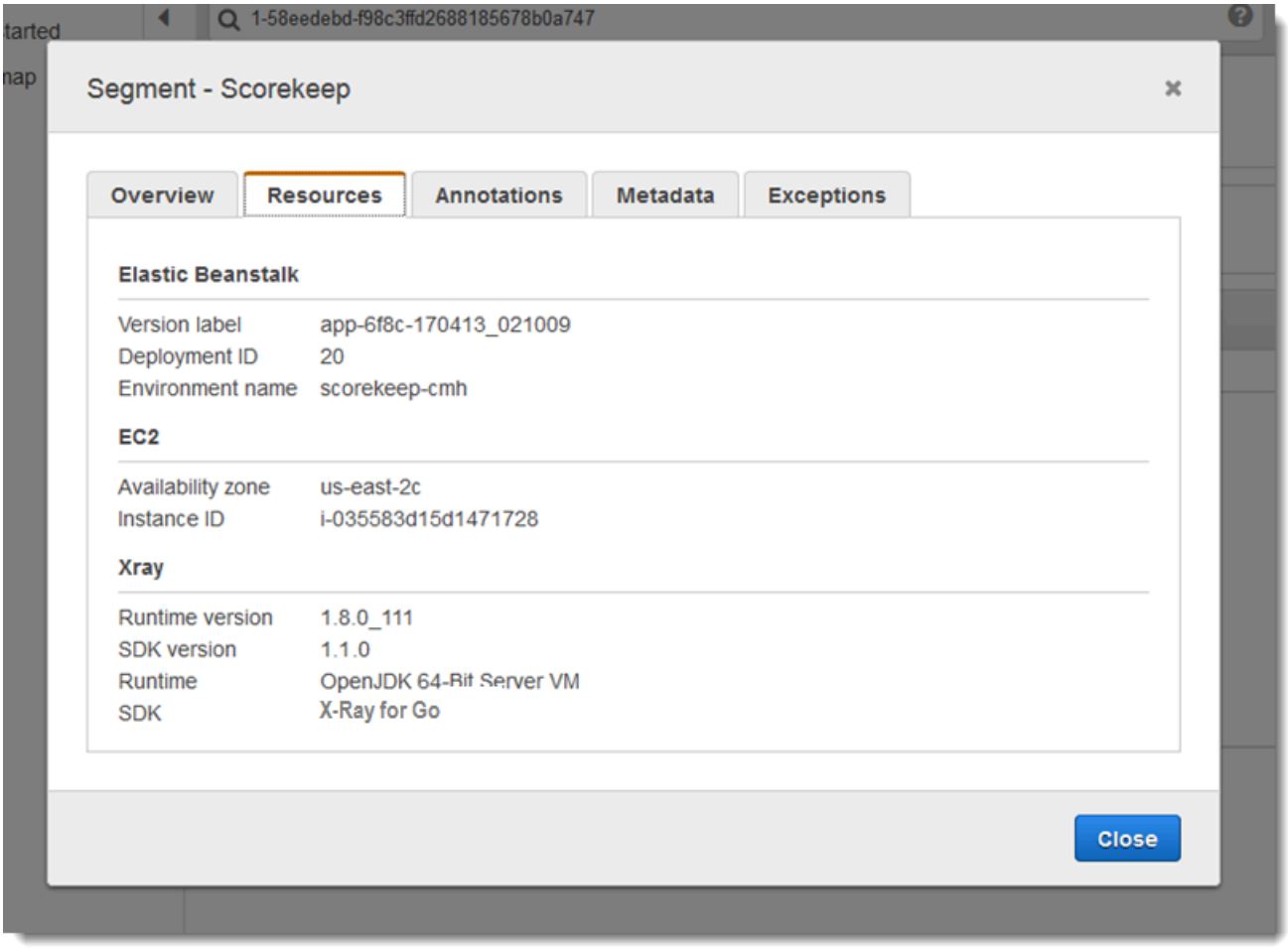
- サービスプラグイン
- サンプリングルール
- ロギング
- 環境変数
- 設定の使用

サービスプラグイン

`plugins`を使用して、アプリケーションをホストしているサービスに関する情報を記録します。

プラグイン

- Amazon EC2 —`EC2Plugin`は、インスタンス ID、アベイラビリティーゾーン、および CloudWatch Logs グループを追加します。
- ElasticBeanstalk– `ElasticBeanstalkPlugin`は、環境名、バージョンラベル、およびデプロイ ID を追加します。
- Amazon ECS —`ECSPlugin`は、コンテナ ID を追加します。



プラグインを使用するには、次のいずれかのパッケージをインポートします。

```
"github.com/aws/aws-xray-sdk-go/awsplugins/ec2"
"github.com/aws/aws-xray-sdk-go/awsplugins/ecs"
"github.com/aws/aws-xray-sdk-go/awsplugins/beanstalk"
```

各プラグインには、プラグインをロードする明示的な `Init()` 関数呼び出しがあります。

Example ec2.Init()

```
import (
 "os"

 "github.com/aws/aws-xray-sdk-go/awsplugins/ec2"
 "github.com/aws/aws-xray-sdk-go/xray"
)

func init() {
```

```
  // conditionally load plugin
  if os.Getenv("ENVIRONMENT") == "production" {
    ec2.Init()
  }

  xray.Configure(xray.Config{
    ServiceVersion: "1.2.3",
  })
}
```

SDK はプラグイン設定を使用して、originセグメントのフィールド。これは、アプリケーションを実行する AWS リソースのタイプを示します。複数のプラグインを使用する場合、SDK は次の解決順序を使用して起点を決定します。ElasticBeanStalk > EKS > ECS > EC2。

サンプリングルール

SDK は X-Ray コンソールで定義したサンプリングルールを使用し、記録するリクエストを決定します。デフォルトルールでは、最初のリクエストを毎秒トレースし、X-Ray にトレースを送信するすべてのサービスで追加のリクエストの 5% をトレースします。X-Ray コンソールに追加のルールを作成するをクリックして、各アプリケーションで記録されるデータ量をカスタマイズします。

SDK は、定義された順序でカスタムルールを適用します。リクエストが複数のカスタムルールと一致する場合、SDK は最初のルールのみを適用します。

> **Note**
>
> SDK が X-Ray に到達してサンプリングルールを取得できない場合、1 秒ごとに最初のリクエストのデフォルトのローカルルールに戻り、ホストあたりの追加リクエストの 5% に戻ります。これは、ホストがサンプリング API を呼び出す権限を持っていない場合や、SDK によって行われる API 呼び出しの TCP プロキシとして機能する X-Ray デーモンに接続できない場合に発生します。

JSON ドキュメントからサンプリングルールをロードするように SDK を設定することもできます。SDK は、X-Ray サンプリングが利用できない場合のバックアップとしてローカルルールを使用することも、ローカルルールを排他的に使用することもできます。

Example sampling-rules.json

```
{
  "version": 2,
```

```
  "rules": [
    {
      "description": "Player moves.",
      "host": "*",
      "http_method": "*",
      "url_path": "/api/move/*",
      "fixed_target": 0,
      "rate": 0.05
    }
  ],
  "default": {
    "fixed_target": 1,
    "rate": 0.1
  }
}
```

この例では、1 つのカスタムルールとデフォルトルールを定義します。カスタムルールでは、5 パーセントのサンプリングレートが適用され、/api/move/以下のパスに対してトレースするリクエストの最小数はありません。デフォルトのルールでは、1秒ごとの最初のリクエストおよび追加リクエストの 10 パーセントをトレースします。

ルールをローカルで定義することの欠点は、固定ターゲットが X-Ray サービスによって管理されるのではなく、レコーダーの各インスタンスによって個別に適用されることです。より多くのホストをデプロイすると、固定レートが乗算され、記録されるデータ量の制御が難しくなります。

オンの場合 AWS Lambda、サンプリングレートを変更することはできません。関数がインストルメント化されたサービスによって呼び出された場合、そのサービスによってサンプリングされたリクエストを生成した呼び出しは Lambda によって記録されます。アクティブなトレースが有効で、トレースヘッダーが存在しない場合、Lambda はサンプリングを決定します。

バックアップルールを規定するには、NewCentralizedStrategyWithFilePath を使用して JSON ファイルのローカルサンプリングを指定します。

Example main.go – ローカルサンプリングルール

```
s, _ := sampling.NewCentralizedStrategyWithFilePath("sampling.json") // path to local
 sampling json
xray.Configure(xray.Config{SamplingStrategy: s})
```

ローカルルールのみを使用するには、NewLocalizedStrategyFromFilePath を使用して JSON ファイルのローカルサンプリングを指定します。

Example main.go – サンプリングを無効にする

```
s, _ := sampling.NewLocalizedStrategyFromFilePath("sampling.json") // path to local
 sampling json
xray.Configure(xray.Config{SamplingStrategy: s})
```

ロギング

> **Note**
>
> xray.Config{} フィールド LogLevel と LogFormat は、バージョン 1.0.0-rc.10 以降では非推奨です。

X-Ray は、ログ記録に次のインターフェイスを使用します。デフォルトのロガーは、LogLevelInfo 以上で stdout に書き込みます。

```
type Logger interface {
 Log(level LogLevel, msg fmt.Stringer)
}

const (
 LogLevelDebug LogLevel = iota + 1
 LogLevelInfo
 LogLevelWarn
 LogLevelError
)
```

Example **io.Writer** に書き込み

```
xray.SetLogger(xraylog.NewDefaultLogger(os.Stderr, xraylog.LogLevelError))
```

環境変数

X-Ray SDK for Goの設定 SDK は次の変数をサポートしています。

- AWS_XRAY_CONTEXT_MISSING – 計測されたコードが、セグメントが開いていないときにデータを記録しようとした場合に例外をスローするには、RUNTIME_ERROR に設定します。

有効な値

- RUNTIME_ERROR— ランタイム例外をスローします。
- LOG_ERROR – エラーをログ記録して続行します (デフォルト)。
- IGNORE_ERROR – エラーを無視して続行します。

オープン状態のリクエストがない場合、または新しいスレッドを発生させるコードで、スタートアップコードに実装されたクライアントを使用しようとした場合に発生する可能性がある、セグメントまたはサブセグメントの欠落に関連するエラー。

- AWS_XRAY_TRACING_NAME – SDK がセグメントに使用するサービス名を設定します。
- AWS_XRAY_DAEMON_ADDRESS – X-Ray デーモン リスナーのホストとポートを設定します。デフォルトでは、SDK は、トレースデータをに送信します127.0.0.1:2000。この変数は、デーモンを次のように構成している場合に使用します。別のポートでリッスンするまたは、別のホストで実行されている場合。
- AWS_XRAY_CONTEXT_MISSING – 値を設定して、SDK がコンテキスト欠落エラーをどのように処理するかを決定します。オープン状態のリクエストがない場合、または新しいスレッドを発生させるコードで、スタートアップコードに実装されたクライアントを使用しようとした場合に発生する可能性がある、セグメントまたはサブセグメントの欠落に関連するエラー。
  - RUNTIME_ERROR – デフォルトでは、SDK はランタイムの例外をスローするように設定されています。
  - LOG_ERROR – エラーを記録して続行します。

環境変数は、コードで設定される同等の値を上書きします。

設定の使用

X-Ray SDK for Go を Configure を使用して設定することもできます。Configureは、1 つの引数、Config オブジェクトと次のオプションフィールドを使用します。

DaemonAddr

この文字列は X-Ray デーモン リスナーのホストとポートを指定します。指定しない場合、X-Ray は AWS_XRAY_DAEMON_ADDRESS 環境変数の値を使用します。この値が設定されていない場合は、「127.0.0.1:2000」を使用します。

ServiceVersion

この文字列は、サービスのバージョンを指定します。指定されていない場合、X-Ray は空の文字列 (「」) を使用します。

SamplingStrategy

この SamplingStrategy オブジェクトは、どのアプリケーションコールをトレースするかを指定します。指定しない場合、X-Ray は LocalizedSamplingStrategy で定義された戦略を取る xray/resources/DefaultSamplingRules.json を使用します。

StreamingStrategy

この StreamingStrategy オブジェクトは、RequiresStreaming が true を返すときにセグメントをストリーミングするかどうかを指定します。指定しない場合、X-Ray は、サブセグメントの数が 20 を超える場合、サンプリングされたセグメントをストリーミングする DefaultStreamingStrategy を使用します。

ExceptionFormattingStrategy

この ExceptionFormattingStrategy オブジェクトは、さまざまな例外を処理する方法を指定します。指定しない場合、X-Ray は、タイプ DefaultExceptionFormattingStrategy の XrayError、エラーメッセージ、およびスタックトレースを持つ error を使用します。

X-Ray SDK for Goを使用して受信する HTTP リクエストの計測を行う

X-Ray SDK を使用して、アプリケーションが Amazon EC2 の EC2 インスタンス AWS Elastic Beanstalk、または Amazon ECS で処理する受信 HTTP リクエストをトレースできます。 Amazon EC2

xray.Handler を使用して受信 HTTP リクエストを計測します。X-Ray SDK for Go は、標準の Go ライブラリ http.Handler インターフェィスを xray.Handler クラスに実装して、ウェブリクエストをインターセプトします。xray.Handler クラスは、リクエストのコンテキストを使って、提供された http.Handler を xray.Capture でラップし、必要に応じてレスポンスヘッダーを設定し、HTTP 固有のトレースフィールドを設定します。

このクラスを使用して HTTP リクエストとレスポンスを処理すると、X-Ray SDK for Go はサンプリングされた要求ごとにセグメントを作成します。このセグメントには、時間、メソッド、HTTP リクエストの処理などが含まれます。追加の計測により、このセグメントでサブセグメントが作成されます。

> **Note**
>
> AWS Lambda 関数の場合、Lambda はサンプリングされたリクエストごとにセグメントを作成します。詳細については「AWS Lambda and AWS X-Ray」を参照してください。

次の例は、ポート8000 でリクエストをインターセプトし、レスポンスとして「Hello!」 を返します。任意のアプリケーションでセグメント myApp と計測呼び出しを作成します。

Example main.go

```
func main() {
  http.Handle("/", xray.Handler(xray.NewFixedSegmentNamer("MyApp"),
 http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Hello!"))
  })))

  http.ListenAndServe(":8000", nil)
}
```

各セグメントには、サービスマップ内のアプリケーションを識別する名前があります。セグメントの名前は静的に指定することも、受信リクエストのホストヘッダーに基づいて動的に名前を付けるように SDK を設定することもできます。動的ネーミングでは、リクエスト内のドメイン名に基づいてトレースをグループ化でき、名前が予想されるパターンと一致しない場合 (たとえば、ホストヘッダーが偽造されている場合) 、デフォルト名を適用できます。

> **転送されたリクエスト**
>
> ロードバランサーまたは他の仲介者がアプリケーションにリクエストを転送する場合、X-Ray は、クライアントの IP をIP パケットの送信元 IP からではなく、リクエストのX-Forwarded-Forヘッダーから取得します。転送されたリクエストについて記録されたクライアント IP は偽造される可能性があるため、信頼されるべきではありません。

リクエストが転送されると、それを示す追加フィールドが SDK によってセグメントに設定されます。セグメントのフィールド x_forwarded_for が true に設定されている場合、クライアント IP が X-Forwarded-For HTTP リクエストのヘッダーから取得されます。

ハンドラーは、次の情報が含まれる http ブロックを使用して、各受信リクエスト用にセグメントを作成します。

- HTTP メソッド – GET、POST、PUT、DELETE、その他。
- クライアントアドレス – リクエストを送信するクライアントの IP アドレス。
- レスポンスコード – 完了したリクエストの HTTP レスポンスコード。
- タイミング – 開始時間 (リクエストが受信された時間) および終了時間 (レスポンスが送信された時間)。
- ユーザーエージェント — リクエストからの`user-agent`
- コンテンツの長さ — レスポンスからの `content-length`。

セグメント命名ルールの設定

AWS X-Ray は、サービス名を使用してアプリケーションを識別し、アプリケーションが使用する他のアプリケーション、データベース、外部 APIs、 AWS リソースと区別します。X-Ray SDK が受信リクエストのセグメントを生成すると、アプリケーションのサービス名がセグメントの名前フィールドに記録されます。

X-Ray SDK では、HTTP リクエストヘッダーのホスト名の後にセグメントの名前を指定できます。ただし、このヘッダーは偽造され、サービスマップに予期しないノードが発生する可能性があります。偽造されたホストヘッダーを持つリクエストによって SDK がセグメントの名前を間違えないようにするには、受信リクエストのデフォルト名を指定する必要があります。

アプリケーションが複数のドメインのリクエストを処理する場合、動的ネーミングストラテジーを使用してセグメント名にこれを反映するように SDK を設定できます。動的ネーミングストラテジーにより、SDK は予想されるパターンに一致するリクエストにホスト名を使用し、そうでないリクエストにデフォルト名を適用できます。

たとえば、3 つのサブドメイン (`www.example.com`,`api.example.com`,および`static.example.com`) に対してリクエストを処理する単一のアプリケーションがあるとします。動的ネーミングストラテジーをパターン `*.example.com` で使用して、異なる名前を持つ各サブドメインのセグメントを識別することができます。結果的にはサービスマップ上に 3 つのサービスノードを作成することになります。アプリケーションがパターンと一致しないホスト名のリクエストを受信すると、指定したフォールバック名を持つ 4 番目のノードがサービスマップに表示されます。

すべてのリクエストセグメントに対して同じ名前を使用するには、前のセクションで示すとおり、ハンドラーを作成するときに、アプリケーションの名前を指定します。

> **Note**
>
> コードで定義したデフォルトのサービス名は、AWS_XRAY_TRACING_NAME 環境変数で上書きできます。

動的命名ルールは、ホスト名と一致するようパターンを定義し、HTTP リクエストのホスト名がパターンと一致しない場合はデフォルトの名前を使用します。セグメントに動的に名前を付けるには、NewDynamicSegmentNamer を使用して、一致させるデフォルトの名前とパターンを設定します。

Example main.go

リクエストのホスト名がパターン *.example.com と一致する場合は、そのホスト名を使用します。それ以外の場合は、MyApp を使用します。

```
func main() {
  http.Handle("/", xray.Handler(xray.NewDynamicSegmentNamer("MyApp", "*.example.com"),
 http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Hello!"))
  })))

  http.ListenAndServe(":8000", nil)
}
```

X-Ray AWS SDK for Go を使用した SDK 呼び出しのトレース

アプリケーションが AWS のサービス を呼び出してデータの保存、キューへの書き込み、または通知の送信を行うと、X-Ray SDK for Go はサブセグメントのダウンストリームの呼び出しを追跡します。これらのサービス (Amazon S3 バケットや Amazon SQS キューなど) 内でアクセスするトレースされた AWS のサービス および リソースは、X-Ray コンソールのトレースマップにダウンストリームノードとして表示されます。

AWS SDK クライアントをトレースするには、次の例に示すように、クライアントオブジェクトを xray.AWS() 呼び出しでラップします。

Example main.go

```
var dynamo *dynamodb.DynamoDB
```

```
func main() {
  dynamo = dynamodb.New(session.Must(session.NewSession()))
  xray.AWS(dynamo.Client)
}
```

次に、AWS SDK クライアントを使用する場合は、呼び出しメソッドの `withContext` バージョンを使用し、それを`context`ハンドラー`http.Request`に渡された オブジェクトから に渡します。

Example main.go – AWS SDK 呼び出し

```
func listTablesWithContext(ctx context.Context) {
  output := dynamo.ListTablesWithContext(ctx, &dynamodb.ListTablesInput{})
  doSomething(output)
}
```

すべてのサービスにおいて、X-Ray コンソールでコールされた API の名前を確認できます。サービスのサブセットの場合、X-Ray SDK はセグメントに情報を追加して、サービスマップでより細かく指定します。

たとえば、実装された DynamoDB クライアントでコールすると、SDK はテーブルをターゲットとするコールのセグメントにテーブル名を追加します。コンソールで、各テーブルはサービスマップ内に個別のノードとして表示され、テーブルをターゲットにしないコール用の汎用の DynamoDB ノードが表示されます。

Example 項目を保存するための DynamoDB に対するコールのサブセグメント

```
{
  "id": "24756640c0d0978a",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "DynamoDB",
  "namespace": "aws",
  "http": {
    "response": {
      "content_length": 60,
      "status": 200
    }
  },
  "aws": {
    "table_name": "scorekeep-user",
    "operation": "UpdateItem",
```

```
    "request_id": "UBQNSO5AEM8T4FDA4RQDEB94OVTDRVV4K4HIRGVJF66Q9ASUAAJG",
  }
}
```

名前付きリソースにアクセスしたとき、次のサービスをコールすると、サービスマップに追加のノードが作成されます。特定のリソースをターゲットとしないコールでは、サービスの汎用ノードが作成されます。

- Amazon DynamoDB – テーブル名
- Amazon Simple Storage Service – バケットとキー名
- Amazon Simple Queue Service – キュー名

X-Ray SDK for Go を使用してダウンストリーム HTTP ウェブサービスの呼び出しをトレースする

アプリケーションがマイクロサービスまたはパブリック HTTP API を呼び出すときは、次の例に示すように、`xray.Client` を使用してこれらの呼び出しを Go アプリケーションのサブセグメントとして計測できます。次の例に示すように、http-client は HTTP クライアントです。

クライアントは、提供された HTTP クライアントのシャローコピーを作成します。これは `http.DefaultClient` のデフォルトの `xray.RoundTripper` でラウンドトリップされます。

Example

<caption>main.go – HTTP クライアント</caption>

```
myClient := xray.Client(http-client)
```

<caption>main.go — ctxhttp ライブラリを使用したダウンストリームの HTTP 呼び出しのトレース</caption>

次の例では、ctxhttp ライブラリを使用して発信 HTTP コールをインストルメントします。`xray.Client`。`ctx`アップストリームコールから渡すことができます。これにより、既存のセグメントコンテキストが使用されることが保証されます。たとえば、X-Ray では Lambda 関数内に新しいセグメントを作成できないので、既存の Lambda セグメントコンテキストを使用する必要があります。

```
resp, err := ctxhttp.Get(ctx, xray.Client(nil), url)
```

X-Ray SDK for Goを使用して SQL クエリをトレースします

PostgreSQL または MySQL への SQL 呼び出しをトレースするには、次の例に示すように、`sql.Open` への `xray.SQLContext` 呼び出しに置き換えます。可能であれば、構成文字列の代わりに URL を使用します。

Example main.go

```
func main() {
  db, err := xray.SQLContext("postgres", "postgres://user:password@host:port/db")
  row, err := db.QueryRowContext(ctx, "SELECT 1") // Use as normal
}
```

X-Ray SDK for Goを使用してカスタムサブセグメントを生成する

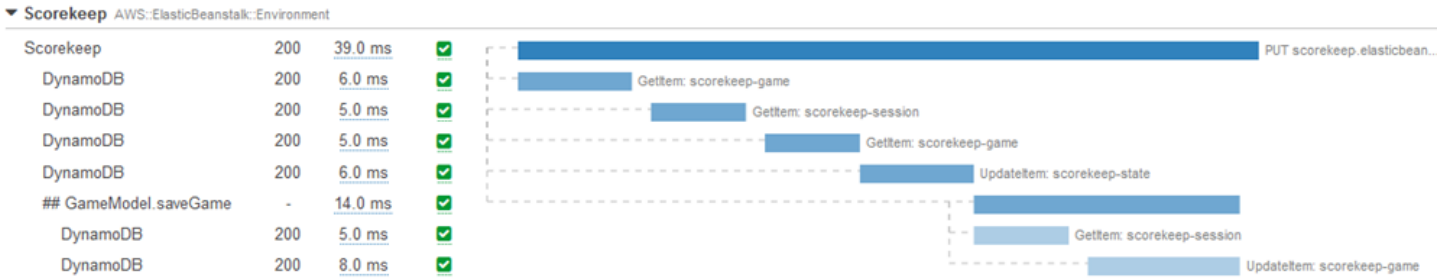
サブセグメントはトレースを拡張するセグメントリクエストを処理するために行われた作業の詳細を含む。計測済みクライアント内で呼び出しを行うたびに、X-Ray SDK によってサブセグメントに生成された情報が記録されます。追加のサブセグメントを作成して、他のサブセグメントをグループ化したり、コードセクションのパフォーマンスを測定したり、注釈とメタデータを記録したりできます。

`Capture` メソッドを使用して、関数の周囲にサブセグメントを作成します。

Example main.go – カスタムサブセグメント

```
func criticalSection(ctx context.Context) {
  //this is an example of a subsegment
  xray.Capture(ctx, "GameModel.saveGame", func(ctx1 context.Context) error {
    var err error

    section.Lock()
    result := someLockedResource.Go()
    section.Unlock()

    xray.AddMetadata(ctx1, "ResourceResult", result)
  })
```

次のスクリーンショットは、アプリケーション `saveGame` のトレースに `Scorekeep` サブセグメントがどのように表示されるかの例を示しています。



X-Ray SDK for Go を使用してセグメントに注釈とメタデータを追加する

注釈やメタデータで行うリクエストや環境やアプリケーションに関する追加情報を記録します。X-Ray SDK が作成するセグメントまたは作成するカスタムサブセグメントに、注釈およびメタデータを追加できます。

注釈は文字列、数値、またはブール値を使用したキーと値のペアです。注釈は、フィルタ式用にインデックス付けされます。注釈を使用して、コンソールでトレースをグループ化するため、または`GetTraceSummaries` API を呼び出すときに使用するデータを記録します。

メタデータは、オブジェクトとリストを含む、任意のタイプの値を持つことができるキーバリューのペアですが、フィルタ式に使用するためにインデックスは作成されません。メタデータを使用してトレースに保存する追加のデータを記録しますが、検索で使用する必要はありません。

注釈とメタデータに加えて、セグメントにユーザー ID 文字列を記録することもできます。ユーザー ID はセグメントの個別のフィールドに記録され、検索用にインデックスが作成されます。

セクション

- X-Ray SDK for Goを使用して注釈を記録する
- X-Ray SDK for Goを使用してメタデータを記録する
- X-Ray SDK for Goを使用してユーザー ID の記録

X-Ray SDK for Goを使用して注釈を記録する

注釈を使用して、検索用にインデックスを作成するセグメントに情報を記録します。

注釈の要件

- キー – X-Ray 注釈のキーには最大 500 文字の英数字を使用できます。スペースまたはドットやピリオド (.) 以外の記号は使用できません。
- 値 – X-Ray 注釈の値には最大 1,000 の Unicode 文字を使用できます。

- 注釈の数 – トレースごとに最大 50 の注釈を使用できます。

注釈を記録するには、セグメントに関連付けるメタデータを含む文字列で AddAnnotation を呼び出します。

```
xray.AddAnnotation(key string, value interface{})
```

SDK は、セグメントドキュメントの annotations オブジェクトにキーと値のペアとして、注釈を記録します。同じキーで AddAnnotation を 2 回呼び出すと、同じセグメントに以前記録された値が上書きされます。

特定の値を持つ注釈のあるトレースを見つけるには、annotation[*key*]フィルタ式の キーワードを使用します。

X-Ray SDK for Goを使用してメタデータを記録する

メタデータを使用して、検索用にインデックスを作成する必要のないセグメントに情報を記録します。

メタデータを記録するには、セグメントに関連付けるメタデータを含む文字列で AddMetadata を呼び出します。

```
xray.AddMetadata(key string, value interface{})
```

X-Ray SDK for Goを使用してユーザー ID の記録

リクエストセグメントにユーザー ID を記録して、リクエストを送信したユーザーを識別します。

ユーザー ID を記録するには

1. AWSXRay から現在のセグメントへの参照を取得します。

   ```
   import (
     "context"
     "github.com/aws/aws-xray-sdk-go/xray"
   )

   mySegment := xray.GetSegment(context)
   ```

2. リクエストを送信したユーザーの文字列 ID を使用して setUser を呼び出します。

```
mySegment.User = "U12345"
```

ユーザー ID のトレースを見つけるには、userフィルタ式で、 キーワードを使用します。

Java の使用

X-Ray にトレースを送信する Java アプリケーションを計測するには、次の 2 つの方法があります。

- AWS Distro for OpenTelemetry Java – AWS Distro for OpenTelemetry Collector を介して、相関メトリクスとトレースを Amazon CloudWatch、 AWS X-Ray Amazon OpenSearch Service などの複数の AWS モニタリングソリューションに送信するための一連のオープンソースライブラリを提供する AWS ディストリビューション。
- AWS X-Ray SDK for Java – X-Ray デーモンを介してトレースを生成して X-Ray に送信するためのライブラリのセット。

詳細については、「Distro for OpenTelemetry AWS と X-Ray SDKs の選択」を参照してください。

AWS Distro for OpenTelemetry Java

AWS Distro for OpenTelemetry (ADOT) Java を使用すると、アプリケーションを一度計測し、相関メトリクスとトレースを Amazon CloudWatch AWS X-Ray、Amazon OpenSearch Service などの複数の AWS モニタリングソリューションに送信できます。X-Ray を ADOT で使用するには、X-Ray で使用できる OpenTelemetry SDK と、X-Ray で使用できる AWS Distro for OpenTelemetry Collector の 2 つのコンポーネントが必要です。ADOT Java には自動計測のサポートが含まれており、アプリケーションでコードを変更せずにトレースを送信できます。

開始するには、AWS Distro for OpenTelemetry Java ドキュメントを参照してください。

AWS Distro for OpenTelemetry を AWS X-Ray およびその他の で使用する方法の詳細については AWS のサービス、AWS 「 Distro for OpenTelemetry」またはAWS 「 Distro for OpenTelemetry ドキュメント」を参照してください。

言語サポートと使用方法の詳細については、「AWS Observability on Github」を参照してください。

AWS X-Ray SDK for Java

X-Ray SDK for Java は、Java ウェブアプリケーション用のライブラリのセットです。トレースデータを作成し X-Ray デーモンに送信するためのクラスおよびメソッドを提供します。トレースデータには、アプリケーションによって提供される受信 HTTP リクエストに関する情報、および AWS SDK、HTTP クライアント、または SQL データベースコネクタを使用してアプリケーションがダウ

ンストリームサービスに対して行う呼び出しが含まれます。セグメントを手動で作成し、注釈およびメタデータにデバッグ情報を追加することもできます。

X-Ray SDK for Java は、オープンソースプロジェクトです。プロジェクトに従って、GitHub github.com/aws/aws-xray-sdk-java で問題とプルリクエストを送信できます。

受信リクエストのトレースは、`AWSXRayServletFilter` をサーブレットフィルタとして追加することから開始します。サーブレットフィルタにより、セグメントが作成されます。セグメントが開いている間、SDK クライアントのメソッドを使用してセグメントに情報を追加し、サブセグメントを作成してダウンストリーム呼び出しをトレースできます。また、SDK では、セグメントが開いている間にアプリケーションがスローする例外を自動的に記録します。

リリース 1.3 以降では、Spring のアスペクト指向プログラミング (AOP) を使用してアプリケーションを計測できます。つまり、アプリケーションのランタイムにコードを追加することなく AWS、実行中にアプリケーションを計測できます。

次に、X-Ray SDK for Java を使用して、SDK Instrumentor サブモジュールをビルド設定に含めることで AWS SDK for Java クライアントを計測します。計測されたクライアントを使用してダウンストリーム AWS のサービス またはリソースを呼び出すたびに、SDK は呼び出しに関する情報をサブセグメントに記録します。 AWS のサービス また、サービス内でアクセスするリソースは、トレースマップにダウンストリームノードとして表示され、個々の接続のエラーやスロットリングの問題を特定するのに役立ちます。

すべてのダウンストリーム呼び出しを計測しない場合は AWS のサービス、Instrumentor サブモジュールを省略し、計測するクライアントを選択できます。 AWS SDK サービスクライアントに `TracingHandler` を追加して、個々のクライアントを計測します。

その他の X-Ray SDK for Java サブモジュールでは、HTTP ウェブ API および SQL データベースに対するダウンストリーム呼び出しを計測できます。Apache HTTP サブモジュールで X-Ray SDK for Java のバージョン `HTTPClient` と `HTTPClientBuilder` を使用する と、Apache HTTP クライアントを計測することができます。SQL クエリの計測には、データソースに SDK のインターセプターを追加します。

SDK を使用し始めたら、レコーダーやサーブレットフィルターを設定して、SDK の動作をカスタマイズしてみましょう。プラグインを追加して、アプリケーションを実行しているコンピューティングリソースに関するデータを記録したり、サンプリングルールを定義することでサンプリングの動作をカスタマイズしたり、アプリケーションログに SDK からの情報をより多くあるいは少なく表示するようにログレベルを設定できます。

アプリケーションが注釈やメタデータで行うリクエストや作業に関する追加情報を記録します。注釈は、フィルタ式で使用するためにインデックス化されたシンプルなキーと値のペアで、特定のデータが含まれているトレースを検索できます。メタデータのエントリは制約が緩やかで、JSON にシリアル化できるオブジェクトと配列全体を記録できます。

> **注釈とメタデータ**
>
> 注釈およびメタデータとは、X-Ray SDK を使用してセグメントに追加する任意のテキストです。注釈は、フィルタ式用にインデックス付けされます。メタデータはインデックス化されませんが、X-Ray コンソールまたは API を使用して raw セグメントで表示できます。X-Ray への読み取りアクセスを許可した人は誰でも、このデータを表示できます。

コードに多数の計測されたクライアントがある場合、単一のリクエストセグメントには計測されたクライアントで行われた呼び出しごとに 1 個の多数のサブセグメントを含めることができます。カスタムサブセグメントで、クライアント呼び出しをラップすることで、サブセグメントを整理してグループできます。関数全体またはコードの任意のセクションのサブセグメントを作成し、親セグメントにすべてのレコードを記述する代わりにサブセグメントにメタデータと注釈を記録できます。

サブモジュール

X-Ray SDK for Java は、Maven からダウンロードできます。X-Ray SDK for Java は、ユースケースごとにサブモジュールに分割され、部品表のバージョン管理に使用されます。

- `aws-xray-recorder-sdk-core` (必須) セグメントを作成して送信するための基本的な機能です。受信リクエストを計測する `AWSXRayServletFilter` が含まれています。
- `aws-xray-recorder-sdk-aws-sdk` – トレース AWS SDK for Java クライアントをリクエストハンドラーとして追加することで、クライアントで AWS のサービス 行われた への呼び出しを計測します。
- `aws-xray-recorder-sdk-aws-sdk-v2` – トレースクライアントをリクエストインターセプターとして追加することで、 AWS SDK for Java 2.2 以降のクライアントで AWS のサービス 行われた への呼び出しを計測します。
- `aws-xray-recorder-sdk-aws-sdk-instrumentor` – では`aws-xray-recorder-sdk-aws-sdk`、 はすべての AWS SDK for Java クライアントを自動的に計測します。
- `aws-xray-recorder-sdk-aws-sdk-v2-instrumentor` – では`aws-xray-recorder-sdk-aws-sdk-v2`、 はすべての AWS SDK for Java 2.2 以降のクライアントを自動的に計測します。

- aws-xray-recorder-sdk-apache-http - Apache HTTP クライアントを使用して行われるアウトバウンド HTTP 呼び出しを計測します。
- aws-xray-recorder-sdk-spring – Spring AOP Framework アプリケーション用のインターセプターを提供します。
- aws-xray-recorder-sdk-sql-postgres - JDBC を使用して PostgreSQL データベースに対して行われるアウトバウンド呼び出しを計測します。
- aws-xray-recorder-sdk-sql-mysql - JDBC を使用して MySQL データベースに対して行われるアウトバウンド呼び出しを計測します。
- aws-xray-recorder-sdk-bom - すべてのサブモジュールで使用するバージョンを指定するための部品表を提供します。
- aws-xray-recorder-sdk-metrics – 収集した X-Ray セグメントからサンプリングされていない Amazon CloudWatch メトリクスを発行します。

Maven または Gradle を使用してアプリケーションを構築する場合は、X-Ray SDK for Java をビルド設定に追加します。

SDK のクラスとメソッドに関するリファレンスドキュメントについては、「AWS X-Ray SDK for Java API リファレンス」を参照してください。

要件

X-Ray SDK for Java には、Java 8 以降、Servlet API 3、 AWS SDK、および Jackson が必要です。

SDK は、コンパイル時および実行時に次のライブラリに依存します。

- AWS SDK for Java バージョン 1.11.398 以降
- Servlet API 3.1.0

これらの依存関係は SDK の `pom.xml` ファイルで宣言され、Maven や Gradle を使用して構築すると自動的に含まれます。

X-Ray SDK for Java に含まれているライブラリを使用する場合、同梱されているバージョンを使用する必要があります。たとえば、すでに実行時に Jackson に依存し、その依存関係のためにデプロイメントに JAR ファイルを含めている場合、SDK JAR には Jackson ライブラリの独自のバージョンが含まれているため、その JAR ファイルを削除する必要があります。

依存関係管理

X-Ray SDK for Javaは、Maven から入手できます。

- グループ – `com.amazonaws`
- Artifact – `aws-xray-recorder-sdk-bom`
- バージョン – `2.11.0`

Maven を使用してアプリケーションを構築する場合は、SDK を依存関係として `pom.xml` ファイルに追加します。

Example pom.xml - 依存関係

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-xray-recorder-sdk-bom</artifactId>
      <version>2.11.0</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
<dependencies>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-xray-recorder-sdk-core</artifactId>
  </dependency>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-xray-recorder-sdk-apache-http</artifactId>
  </dependency>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-xray-recorder-sdk-aws-sdk</artifactId>
  </dependency>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-xray-recorder-sdk-aws-sdk-instrumentor</artifactId>
  </dependency>
```

```
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-xray-recorder-sdk-sql-postgres</artifactId>
  </dependency>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-xray-recorder-sdk-sql-mysql</artifactId>
  </dependency>
</dependencies>
```

Gradle の場合は、SDK をコンパイル時の依存関係として build.gradle ファイルに追加します。

Example build.gradle - 依存関係

```
dependencies {
  compile("org.springframework.boot:spring-boot-starter-web")
  testCompile("org.springframework.boot:spring-boot-starter-test")
  compile("com.amazonaws:aws-java-sdk-dynamodb")
  compile("com.amazonaws:aws-xray-recorder-sdk-core")
  compile("com.amazonaws:aws-xray-recorder-sdk-aws-sdk")
  compile("com.amazonaws:aws-xray-recorder-sdk-aws-sdk-instrumentor")
  compile("com.amazonaws:aws-xray-recorder-sdk-apache-http")
  compile("com.amazonaws:aws-xray-recorder-sdk-sql-postgres")
  compile("com.amazonaws:aws-xray-recorder-sdk-sql-mysql")
  testCompile("junit:junit:4.11")
}
dependencyManagement {
    imports {
        mavenBom('com.amazonaws:aws-java-sdk-bom:1.11.39')
        mavenBom('com.amazonaws:aws-xray-recorder-sdk-bom:2.11.0')
    }
}
```

Elastic Beanstalk を使用してアプリケーションをデプロイする場合、すべての依存関係を含んだ大きなアーカイブを構築してアップロードする代わりに、Maven または Gradle を使用してデプロイするたびにオンインスタンスで構築できます。Gradle の使用例については、サンプルアプリケーションを参照してください。

AWS X-Ray Java 用自動計測エージェント

Java 用 AWS X-Ray 自動計測エージェントは、最小限の開発労力で Java ウェブアプリケーションを計測するトレースソリューションです。エージェントは、サーブレットベースのアプリケーション

と、サポートされているフレームワークおよびライブラリで作成されたエージェントのすべてのダウンストリームリクエストのトレースを可能にします。これには、ダウンストリーム Apache HTTP リクエスト、 AWS SDK リクエスト、および JDBC ドライバーを使用して作成された SQL クエリが含まれます。エージェントは、すべてのアクティブなセグメントとサブセグメントを含む X-Ray コンテキストをスレッド間で伝播します。X-Ray SDK のすべての設定と汎用性は、Java エージェントで引き続き使用できます。エージェントが最小限の労力で動作するように、適切なデフォルトが選択されました。

X-Ray エージェントソリューションは、サーブレットベースの要求応答 Java ウェブアプリケーションサーバーに最適です。アプリケーションが非同期フレームワークを使用している場合、または要求/応答サービスとして適切にモデル化されていない場合は、代わりに SDK を使用した手動計測を検討することをお勧めします。

X-Ray エージェントは、分散システム理解ツールキット (DiSCo) を使用して構築されます。DiSCo は、分散システムで使用できる Java エージェントを構築するためのオープンソースフレームワークです。X-Ray エージェントを使用するために DiSCo を理解する必要はありませんが、GitHub のホームページにアクセスすれば、このプロジェクトについて詳しく知ることができます。X-Ray エージェントも完全にオープンソース化されています。ソースコードの表示、寄付、またはエージェントに関する問題の提起を行うには、GitHub のリポジトリにアクセスします。

サンプルアプリケーション

eb-java-scorekeep サンプルアプリケーションは、X-Ray エージェントによる計測ができるように適応されます。このブランチにはサーブレットフィルターやレコーダー構成は含まれません。これらの機能はエージェントによって行われるためです。ローカルまたは AWS リソースを使用してアプリケーションを実行するには、サンプルアプリケーションの Readme ファイルに記載されている手順に従います。サンプルアプリを使用して X-Ray トレースを生成する方法は、サンプルアプリのチュートリアルに記載されています。

使用開始方法

自分のアプリケーションで X-Ray 自動計測 Java エージェントを使用するには、次の手順を実行します。

1. ご使用の環境で X-Ray デーモンを実行します。詳細については、「AWS X-Ray デーモン」を参照してください。
2. エージェントの最新ディストリビューションをダウンロードします。アーカイブを解凍し、ファイルシステム内の場所を記録します。その内容は次のようになります。

```
disco
### disco-java-agent.jar
### disco-plugins
    ### aws-xray-agent-plugin.jar
    ### disco-java-agent-aws-plugin.jar
    ### disco-java-agent-sql-plugin.jar
    ### disco-java-agent-web-plugin.jar
```

3. 下記を含めるようアプリケーションの JVM 引数を変更します。それにより、エージェントが有効になります。該当する場合は、`-javaagent` 引数が `-jar` 引数の 前 に配置されていることを確認します。JVM 引数を変更するプロセスは、Java サーバーの起動に使用するツールやフレームワークによって異なります。具体的なガイダンスについては、サーバーフレームワークのドキュメントを参照してください。

```
-javaagent:/<path-to-disco>/disco-java-agent.jar=pluginPath=/<path-to-disco>/disco-
plugins
```

4. X-Ray コンソールでのアプリケーション名の表示方法を指定するには、`AWS_XRAY_TRACING_NAME` 環境変数または `com.amazonaws.xray.strategy.tracingName` システムプロパティを設定します。名前が指定されていない場合は、デフォルト名が使用されます。
5. サーバーまたはコンテナを再起動します。着信要求とそのダウンストリーム呼び出しがトレースされるようになりました。期待した結果が表示されない場合は、「the section called “トラブルシューティング”」を参照してください。

設定

X-Ray エージェントは、ユーザー提供の外部の JSON ファイルによって設定されます。デフォルトでは、このファイルはユーザーのクラスパスのルート (たとえば、ユーザーの `resources` ディレクトリ) に `xray-agent.json`という名前で存在します。`com.amazonaws.xray.configFile` システムプロパティに、設定ファイルの絶対ファイルシステムパスを設定することで、設定ファイルのカスタムロケーションを設定できます。

次に、設定ファイルの例を示します。

```
{
    "serviceName": "XRayInstrumentedService",
    "contextMissingStrategy": "LOG_ERROR",
    "daemonAddress": "127.0.0.1:2000",
```

```
    "tracingEnabled": true,
    "samplingStrategy": "CENTRAL",
    "traceIdInjectionPrefix": "prefix",
    "samplingRulesManifest": "/path/to/manifest",
    "awsServiceHandlerManifest": "/path/to/manifest",
    "awsSdkVersion": 2,
    "maxStackTraceLength": 50,
    "streamingThreshold": 100,
    "traceIdInjection": true,
    "pluginsEnabled": true,
    "collectSqlQueries": false
}
```

設定仕様

次の表は、各プロパティの有効な値を説明しています。プロパティ名は大文字と小文字が区別されますが、キーは区別されません。環境変数とシステムプロパティで上書きできるプロパティの場合、優先順位の順序は常に環境変数、システムプロパティ、構成ファイルになります。上書きできるプロパティについては、「環境変数」を参照してください。すべてのフィールドはオプションです。

プロパティ名	タイプ	有効値	説明	環境変数	システムプロパティ	デフォルト値
serviceName	String	任意の文字列	X-Ray コンソールに表示される計測済みサービス名。	AWS_XRAY_TRACING_NAME	com.amazonaws.xray.strategy.tracingName	XRayInstrumentedService
contextMissingStrategy	String	LOG_ERROR、IGNORE_ERROR	エージェントが X-Ray セグメントコンテキストを使用しようとするが、存在しないときに実行するアクション。	AWS_XRAY_CONTEXT_MISSING	com.amazonaws.xray.strategy.contextMissingStrategy	LOG_ERROR

プロパティ名	タイプ	有効値	説明	環境変数	システムプロパティ	デフォルト値
DaemonAddress	String	フォーマットされた IP アドレスとポート、または TCP および UDP アドレスのリスト	X-Ray デーモンとの通信にエージェントが使用するアドレス。	AWS_XRAY_DAEMON_ADDRESS	com.amazonaws.xray.emitter.daemonAddress	127.0.0.1:2000
tracingEnabled	ブール値	True、False	X-Ray エージェントによる計測を有効にします。	AWS_XRAY_TRACING_ENABLED	com.amazonaws.xray.tracingEnabled	正
samplingStrategy	String	CENTRAL、LOCAL、NONE、ALL	エージェントが使用するサンプリング戦略。ALL はすべてのリクエストをキャプチャし、NONE はリクエストをキャプチャしません。サンプリングルールを参照してください。。	該当なし	該当なし	CENTRAL

プロパティ名	タイプ	有効値	説明	環境変数	システムプロパティ	デフォルト値
traceIdInjectionPrefix	String	任意の文字列	ログに注入されたトレース ID の前に指定されたプレフィクスを含めます。	該当なし	該当なし	なし (空の文字列)
samplingRulesManifest	String	絶対ファイルパス	ローカルサンプリング戦略のサンプリングルール、または中央戦略のフォールバックルールのソースとして使用されるカスタムサンプリングルールファイルへのパス。	該当なし	該当なし	DefaultSamplingRules.json
awsServiceHandlerManifest	String	絶対ファイルパス	AWS SDK クライアントから追加情報を取得するためのカスタムパラメータ許可リストへのパス。	該当なし	該当なし	DefaultOperationParameterWhitelist.json

プロパティ名	タイプ	有効値	説明	環境変数	システムプロパティ	デフォルト値
awsSdkVersion	整数	1、2	使用している AWS SDK for Java のバージョン。`awsServiceHandlerManifest` も設定されていない場合は無視されます。	該当なし	該当なし	2
maxStackTraceLength	整数	非負整数	トレースに記録するスタックトレースの最大行数。	該当なし	該当なし	50
streamingThreshold	整数	非負整数	この数以上のサブセグメントが閉じられた後、チャンクが大きすぎるのを避けるために、それらはデーモンアウトオブバンドにストリーミングされます。	該当なし	該当なし	100

プロパティ名	タイプ	有効値	説明	環境変数	システムプロパティ	デフォルト値
traceIdInjection	ブール値	True、False	ログ作成設定 に記述された依存関係や設定も追加されている場合、ログへの X-Ray トレース ID の注入を有効にします。それ以外の場合は、何もしません。	該当なし	該当なし	正
pluginsEnabled	ブール値	True、False	運用している AWS 環境に関するメタデータを記録するプラグインを有効にします。プラグインを参照してください。	該当なし	該当なし	正

プロパティ名	タイプ	有効値	説明	環境変数	システムプロパティ	デフォルト値
collectSqlQueries	ブール値	True、False	SQL クエリ文字列を SQL サブセグメントにベストエフォートベースで記録します。	該当なし	該当なし	誤
contextPropagation	ブール値	True、False	True の場合、スレッド間で X-Ray コンテキストを自動的に伝播します。それ以外の場合、は Thread Local を使用してコンテキストを保存し、スレッド間での手動伝播が必要です。	該当なし	該当なし	正

ログ作成設定

X-Ray エージェントのログレベルは、X-Ray SDK for Java と同じ方法で設定できます。X-Ray SDK for Java でのログ作成設定については、ロギング を参照してください。

手動実装

エージェントの自動計測に加えて手動計測を実行する場合は、プロジェクトへの依存関係として X-Ray SDK を追加します。受信リクエストのトレース に記述した SDK のカスタムサーブレットフィルターは、X-Ray エージェントと互換性がないことに注意してください。

> **Note**
>
> エージェントを使用しながら、手動計測を実行するには、X-Ray SDK の最新バージョンを使用する必要があります。

Maven プロジェクトで作業している場合は、以下の依存関係を `pom.xml` ファイルに追加します。

```
<dependencies>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-xray-recorder-sdk-core</artifactId>
    <version>2.11.0</version>
  </dependency>
  </dependencies>
```

Gradle プロジェクトで作業している場合は、以下の依存関係を `build.gradle` ファイルに追加します。

```
implementation 'com.amazonaws:aws-xray-recorder-sdk-core:2.11.0'
```

通常の SDK と同様に、エージェントを使用しながら、注釈、メタデータ、ユーザー ID に加えて、カスタムサブセグメント を追加することができます。エージェントはスレッド間でコンテキストを自動的に伝播するため、マルチスレッドアプリケーションを操作するときにコンテキストを伝播するための回避策は必要ありません。

トラブルシューティング

エージェントは全自動計測を行うため、問題が発生した場合、根本原因を特定することが困難な場合があります。X-Ray エージェントが期待通りに動作しない場合は、以下の問題点と解決策を確認してください。X-Ray エージェントと SDK は Jakarta Commons Logging (JCL) を使用しています。ログ作成出力を表示するには、次の例 (`log4j-jcl` または `jcl-over-slf4j`) のように、JCL をログ作成バックエンドに接続するブリッジがクラスパスにあることを確認します。

問題: アプリケーションで Java エージェントを有効にしたが、X-Ray コンソールに何も表示されない

X-Ray デーモンは同じマシンで動作していますか？

そうでない場合は、X-Ray デーモンドキュメントを参照して設定します。

アプリケーションログに「X-Ray エージェントレコーダーの初期化」というメッセージが表示されますか？

アプリケーションにエージェントを正しく追加した場合、このメッセージはアプリケーションの起動時に、リクエストを受け取り始める前に INFO レベルでログに記録されます。このメッセージが表示されない場合、Java エージェントは Java プロセスで実行されていません。入力ミスがない状態で、すべてのセットアップ手順を正しく実行していることを確認してください。

アプリケーションログに、 AWS X-Ray 「コンテキストの欠落例外の抑制」のようなエラーメッセージがいくつか表示されていますか？

これらのエラーは、エージェントが AWS SDK リクエストや SQL クエリなどのダウンストリームリクエストを計測しようとしたが、エージェントがセグメントを自動的に作成できなかったために発生します。これらのエラーが多く見られる場合、エージェントはユースケースに最適なツールではない可能性がありますので、代わりに X-Ray SDK を使用した手動計測を検討することをお勧めします。また、X-Ray SDK の デバッグログ を有効にすると、コンテキスト欠落例外が発生している場所のスタックトレースを確認できます。コードのこれらの部分をカスタムセグメントでラップできます。これにより、これらのエラーを解決できます。ダウンストリームリクエストをカスタムセグメントでラップする例については、スタートアップコードの計測のサンプルコードを参照してください。

問題: 期待していたセグメントの一部が、X-Ray コンソールに表示されない

アプリケーションでマルチスレッドを使用していますか？

作成される予定のセグメントがコンソールに表示されない場合は、アプリケーションのバックグラウンドスレッドが原因である可能性があります。アプリケーションが AWS SDK を使用して Lambda 関数を 1 回だけ呼び出す、HTTP エンドポイントを定期的にポーリングするなど、「発砲と忘れ」のバックグラウンドスレッドを使用してタスクを実行する場合、スレッド間でコンテキストを伝達している間にエージェントを混乱させる可能性があります。この問題が発生しているかどうかを確認するには、X-Ray SDK のデバッグログを有効にして、次のようなメッセージがないかどうかを確認してください。「進行中のサブセグメントの親となる、<NAME> という名前のセグメントを出力していません」 この問題を回避するには、サーバーが戻る前にバックグラウンドスレッドに参加して、そのスレッドで行われたすべての作業が記録されるようにします。または、エージェントの `contextPropagation` の設定を `false` にすると、バックグラウンドスレッドでのコンテキスト伝

播を無効にすることができます。この場合、カスタムセグメントをもつスレッドを手動で計測するか、それらのスレッドが生成するコンテキスト欠落例外を無視する必要があります。

サンプリングルールを設定しましたか？

X-Ray コンソールに一見ランダムな、または予期しないセグメントが表示される場合、あるいはコンソールに表示されるはずのセグメントが表示されない場合は、サンプリングの問題が発生している可能性があります。X-Ray エージェントは、X-Ray コンソールのルールを使用して、作成したすべてのセグメントに集中サンプリングを適用します。デフォルトのルールは、1 秒あたり 1 セグメントが、それ以降はセグメントの 5% がサンプリングされます。つまり、エージェントで迅速に作成されたセグメントはサンプリングされない可能性があります。これを解決するには、目的のセグメントを適切にサンプリングするカスタムサンプリングルールを X-Ray コンソールで作成する必要があります。詳しくは、サンプリングを参照してください。

X-Ray SDK for Java の設定

X-Ray SDK for Java には、グローバルレコーダーを提供する `AWSXRay` というクラスが含まれます。これは、コードの計測に使用できる `TracingHandler` です。グローバルレコーダーを設定して、受信 HTTP 呼び出しのセグメントを作成する `AWSXRayServletFilter` をカスタマイズできます。

セクション

- サービスプラグイン
- サンプリングルール
- ロギング
- セグメントリスナー
- 環境変数
- システムプロパティ

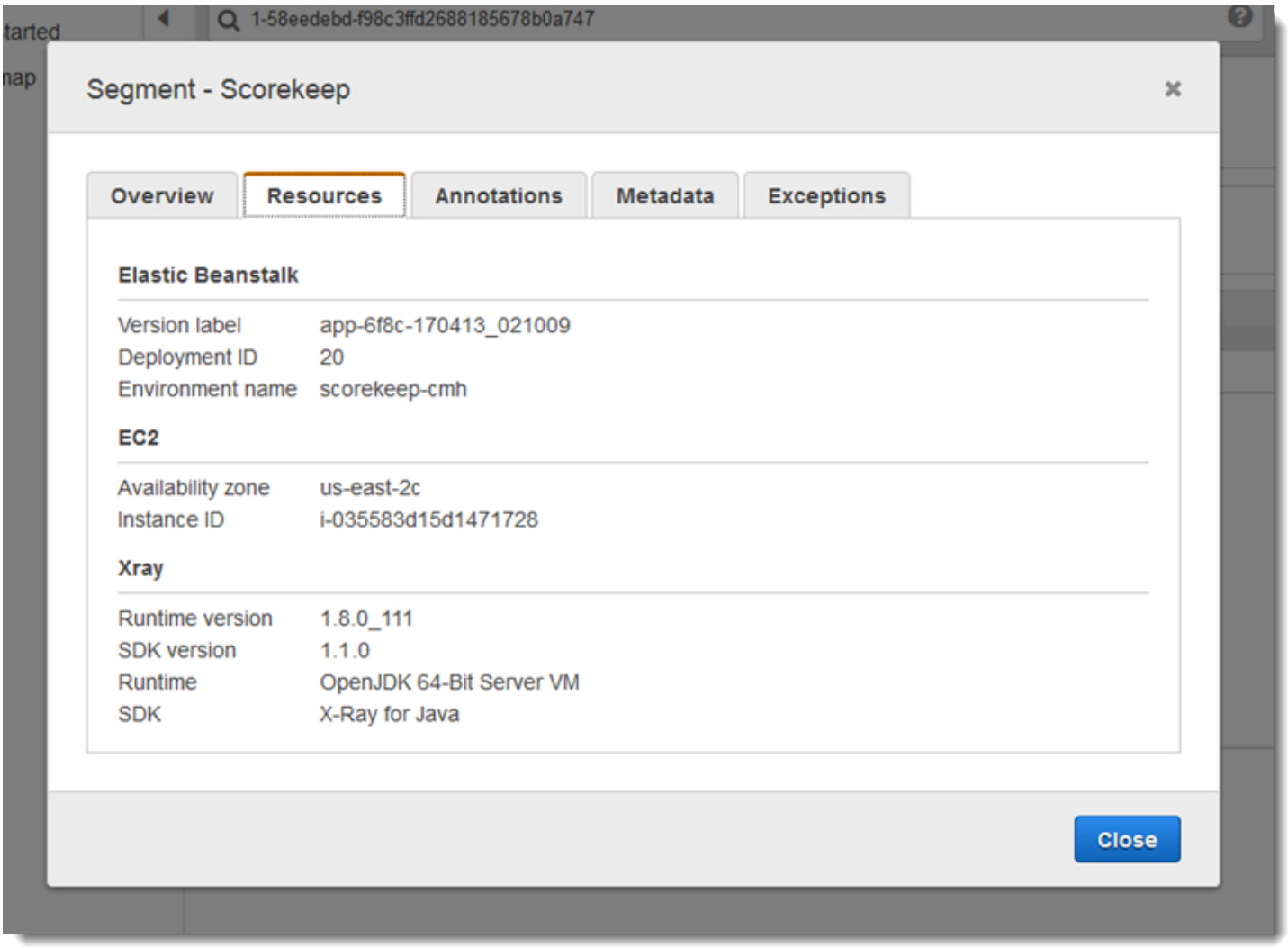
サービスプラグイン

`plugins`を使用して、アプリケーションをホストしているサービスに関する情報を記録します。

プラグイン

- Amazon EC2 —`EC2Plugin`は、インスタンス ID、アベイラビリティーゾーン、および CloudWatch Logs グループを追加します。
- ElasticBeanstalk– `ElasticBeanstalkPlugin`は、環境名、バージョンラベル、およびデプロイ ID を追加します。

- Amazon ECS — `ECSPlugin`は、コンテナ ID を追加します。
- Amazon EKS – `EKSPlugin` は、コンテナ ID、クラスター名、ポッド ID、および CloudWatch Logs グループを追加します。



プラグインを使用するには、`AWSXRayRecorderBuilder` で `withPlugin` を呼び出します。

Example src/main/java/scorekeep/WebConfig.java - レコーダー

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.AWSXRayRecorderBuilder;
import com.amazonaws.xray.plugins.EC2Plugin;
import com.amazonaws.xray.plugins.ElasticBeanstalkPlugin;
import com.amazonaws.xray.strategy.sampling.LocalizedSamplingStrategy;
```

```
@Configuration
public class WebConfig {
...
  static {
    AWSXRayRecorderBuilder builder = AWSXRayRecorderBuilder.standard().withPlugin(new
 EC2Plugin()).withPlugin(new ElasticBeanstalkPlugin());

    URL ruleFile = WebConfig.class.getResource("/sampling-rules.json");
    builder.withSamplingStrategy(new LocalizedSamplingStrategy(ruleFile));

    AWSXRay.setGlobalRecorder(builder.build());
  }
}
```

また、SDK はプラグインの設定を利用して、セグメントの origin フィールドを設定します。これは、アプリケーションを実行する AWS リソースのタイプを示します。複数のプラグインを使用する場合、SDK は次の解決順序を使用して起点を決定します。ElasticBeanStalk > EKS > ECS > EC2。

サンプリングルール

SDK は X-Ray コンソールで定義したサンプリングルールを使用し、記録するリクエストを決定します。デフォルトルールでは、最初のリクエストを毎秒トレースし、X-Ray にトレースを送信するすべてのサービスで追加のリクエストの 5% をトレースします。X-Ray コンソールに追加のルールを作成するをクリックして、各アプリケーションで記録されるデータ量をカスタマイズします。

SDK は、定義された順序でカスタムルールを適用します。リクエストが複数のカスタムルールと一致する場合、SDK は最初のルールのみを適用します。

> **Note**
>
> SDK が X-Ray に到達してサンプリングルールを取得できない場合、1 秒ごとに最初のリクエストのデフォルトのローカルルールに戻り、ホストあたりの追加リクエストの 5% に戻ります。これは、ホストがサンプリング API を呼び出す権限を持っていない場合や、SDK によって行われる API 呼び出しの TCP プロキシとして機能する X-Ray デーモンに接続できない場合に発生します。

JSON ドキュメントからサンプリングルールをロードするように SDK を設定することもできます。SDK は、X-Ray サンプリングが利用できない場合のバックアップとしてローカルルールを使用することも、ローカルルールを排他的に使用することもできます。

Example sampling-rules.json

```
{
  "version": 2,
  "rules": [
    {
      "description": "Player moves.",
      "host": "*",
      "http_method": "*",
      "url_path": "/api/move/*",
      "fixed_target": 0,
      "rate": 0.05
    }
  ],
  "default": {
    "fixed_target": 1,
    "rate": 0.1
  }
}
```

この例では、1 つのカスタムルールとデフォルトルールを定義します。カスタムルールでは、5 パーセントのサンプリングレートが適用され、/api/move/以下のパスに対してトレースするリクエストの最小数はありません。デフォルトのルールでは、1秒ごとの最初のリクエストおよび追加リクエストの 10 パーセントをトレースします。

ルールをローカルで定義することの欠点は、固定ターゲットが X-Ray サービスによって管理されるのではなく、レコーダーの各インスタンスによって個別に適用されることです。より多くのホストをデプロイすると、固定レートが乗算され、記録されるデータ量の制御が難しくなります。

オンの場合 AWS Lambda、サンプリングレートを変更することはできません。関数がインストルメント化されたサービスによって呼び出された場合、そのサービスによってサンプリングされたリクエストを生成した呼び出しは Lambda によって記録されます。アクティブなトレースが有効で、トレースヘッダーが存在しない場合、Lambda はサンプリングを決定します。

Spring でバックアップルールを提供するには、設定クラスの `CentralizedSamplingStrategy` にグローバルレコーダーを設定します。

Example src/main/java/myapp/WebConfig.java – レコーダーの設定

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.AWSXRayRecorderBuilder;
```

```
import com.amazonaws.xray.javax.servlet.AWSXRayServletFilter;
import com.amazonaws.xray.plugins.EC2Plugin;
import com.amazonaws.xray.strategy.sampling.LocalizedSamplingStrategy;

@Configuration
public class WebConfig {

  static {
  AWSXRayRecorderBuilder builder = AWSXRayRecorderBuilder.standard().withPlugin(new
 EC2Plugin());

  URL ruleFile = WebConfig.class.getResource("/sampling-rules.json");
  builder.withSamplingStrategy(new CentralizedSamplingStrategy(ruleFile));

  AWSXRay.setGlobalRecorder(builder.build());
}
```

Tomcat の場合、ServletContextListener を拡張するリスナーを追加し、デプロイ記述子にリスナーを登録します。

Example src/com/myapp/web/Startup.java

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.AWSXRayRecorderBuilder;
import com.amazonaws.xray.plugins.EC2Plugin;
import com.amazonaws.xray.strategy.sampling.LocalizedSamplingStrategy;

import java.net.URL;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

public class Startup implements ServletContextListener {

    @Override
    public void contextInitialized(ServletContextEvent event) {
        AWSXRayRecorderBuilder builder =
 AWSXRayRecorderBuilder.standard().withPlugin(new EC2Plugin());

        URL ruleFile = Startup.class.getResource("/sampling-rules.json");
        builder.withSamplingStrategy(new CentralizedSamplingStrategy(ruleFile));

        AWSXRay.setGlobalRecorder(builder.build());
    }
```

```
    @Override
    public void contextDestroyed(ServletContextEvent event) { }
}
```

Example WEB-INF/web.xml

```
...
  <listener>
    <listener-class>com.myapp.web.Startup</listener-class>
  </listener>
```

ローカルルールのみを使用するには、CentralizedSamplingStrategy を LocalizedSamplingStrategy に置き換えます。

```
builder.withSamplingStrategy(new LocalizedSamplingStrategy(ruleFile));
```

ロギング

デフォルトでは、SDK はアプリケーションログに ERROR-レベルのメッセージを出力します。SDK でデバッグレベルのログを有効にすると、より詳細なログをアプリケーションログファイルに出力できます。有効なログレベルは、DEBUG、INFO、WARN、ERROR、FATAL です。FATAL ログレベルは、SDK が致命的なレベルでログを記録しないため、すべてのログメッセージを消去します。

Example application.properties

logging.level.com.amazonaws.xray プロパティを使用してログレベルを設定します。

```
logging.level.com.amazonaws.xray = DEBUG
```

デバッグログを使用して問題を識別します。たとえば、「サブセグメントを手動で生成する」場合にサブセグメントが閉じない問題などです。

ログへのトレース ID の挿入

ログステートメントに現在の完全修飾トレース ID を公開するには、マップされた診断コンテキスト (MDC) に ID を挿入できます。SegmentListener インターフェイスを使用して、セグメントライフサイクルイベント中に X-Ray レコーダーからメソッドが呼び出されます。セグメントまたはサブセグメントが開始されると、修飾トレース ID がキー AWS-XRAY-TRACE-ID と共に MDC に挿入さ

れます。そのセグメントが終了すると、キーは MDC から削除されます。これにより、トレース ID が使用中のログ記録ライブラリに公開されます。サブセグメントが終了すると、その親 ID が MDC に挿入されます。

Example 完全修飾トレース ID

完全修飾 ID は `TraceID@EntityID` として表されます

```
1-5df42873-011e96598b447dfca814c156@541b3365be3dafc3
```

この機能は、 AWS X-Ray SDK for Java で計測された Java アプリケーションで動作し、次のログ記録設定をサポートします。

- Logback バックエンドを使用する SLF4J フロントエンド API
- Log4J2 バックエンドを使用する SLF4J フロントエンド API
- Log4J2 バックエンドを使用する Log4J2 フロントエンド API

各フロントエンドと各バックエンドのニーズについては、以下のタブを参照してください。

SLF4J Frontend

1. 以下の Maven 依存関係をプロジェクトに追加します。

   ```
   <dependency>
       <groupId>com.amazonaws</groupId>
       <artifactId>aws-xray-recorder-sdk-slf4j</artifactId>
       <version>2.11.0</version>
   </dependency>
   ```

2. `AWSXRayRecorder` を構築するときに `withSegmentListener` メソッドを含めます。これにより、`SegmentListener` クラスが追加されて、SLF4J MDC に新しいトレース ID が自動的に挿入されるようになります。

   `SegmentListener` は、ログステートメントのプレフィクスを設定するためのパラメータとしてオプションの文字列を取ります。プレフィクスは、次の方法で設定できます。

   - なし – デフォルトの `AWS-XRAY-TRACE-ID` プレフィックスを使用します。
   - 空 – 空の文字列を使用します (例:"")。
   - カスタム – 文字列で定義されているカスタムプレフィックスを使用します。

Example **AWSXRayRecorderBuilder** ステートメント

```
AWSXRayRecorderBuilder builder = AWSXRayRecorderBuilder
        .standard().withSegmentListener(new SLF4JSegmentListener("CUSTOM-
PREFIX"));
```

Log4J2 front end

1. 以下の Maven 依存関係をプロジェクトに追加します。

```
<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-xray-recorder-sdk-log4j</artifactId>
    <version>2.11.0</version>
</dependency>
```

2. AWSXRayRecorder を構築するときに withSegmentListener メソッドを含めます。これにより、SegmentListener クラスが追加されて、SLF4J MDC に新しい完全修飾トレース ID が自動的に挿入されるようになります。

   SegmentListener は、ログステートメントのプレフィクスを設定するためのパラメータとしてオプションの文字列を取ります。プレフィクスは、次の方法で設定できます。

   - なし – デフォルトの AWS-XRAY-TRACE-ID プレフィックスを使用します。
   - 空 – 空の文字列 (例: "") を使用して、プレフィックスを削除します。
   - カスタム – 文字列で定義されているカスタムプレフィックスを使用します。

   Example **AWSXRayRecorderBuilder** ステートメント

```
AWSXRayRecorderBuilder builder = AWSXRayRecorderBuilder
        .standard().withSegmentListener(new Log4JSegmentListener("CUSTOM-
PREFIX"));
```

Logback backend

トレース ID をログイベントに挿入するには、各ログ記録ステートメントを書式設定するロガーの `PatternLayout` を変更する必要があります。

1. `patternLayout` が設定されている場所を見つけます。これは、プログラムで行うことも、XML 設定ファイルを使用して行うこともできます。詳細については、「Logback の設定」を参照してください。
2. 以降のログ記録ステートメントにトレース ID を挿入するには、`patternLayout` の任意の場所に `%X{AWS-XRAY-TRACE-ID}` を挿入します。`%X{}` は、MDC から提供されたキーを使用して値を取得することを示します。Logback の PatternLayouts の詳細については、「PatternLayout」を参照してください。

Log4J2 backend

1. `patternLayout` が設定されている場所を見つけます。これは、プログラムで行うことも、XML、JSON、YAML、またはプロパティ形式で記述された設定ファイルを使用して行うこともできます。

   設定ファイルを使用した Log4J2 の設定の詳細については、「設定」を参照してください。

   プログラムによる Log4J2 の設定の詳細については、「プログラムによる設定」を参照してください。
2. 以降のログ記録ステートメントにトレース ID を挿入するには、`PatternLayout` の任意の場所に `%X{AWS-XRAY-TRACE-ID}` を挿入します。`%X{}` は、MDC から提供されたキーを使用して値を取得することを示します。Log4J2 の PatternLayouts の詳細については、「パターンレイアウト」を参照してください。

トレース ID の挿入の例

以下に示しているのは、トレース ID を含むように変更された `PatternLayout` 文字列です。トレース ID は、スレッド名 (`%t`) の後、ログレベル (`%-5p`) の前に出力されます。

Example ID を挿入した **`PatternLayout`**

```
%d{HH:mm:ss.SSS} [%t] %X{AWS-XRAY-TRACE-ID} %-5p %m%n
```

AWS X-Ray は、簡単に解析できるように、ログステートメントにキーとトレース ID を自動的に出力します。以下に示しているのは、変更した PatternLayout を使用したログステートメントです。

Example ID を挿入したログステートメント

```
2019-09-10 18:58:30.844 [nio-5000-exec-4]  AWS-XRAY-TRACE-ID:
 1-5d77f256-19f12e4eaa02e3f76c78f46a@1ce7df03252d99e1 WARN 1 - Your logging message
 here
```

ログメッセージ自体はパターン %m に格納され、ロガーを呼び出すときに設定されます。

セグメントリスナー

セグメントリスナーは、AWSXRayRecorder によって生成されたセグメントの開始と終了などのライフサイクルイベントをインターセプトするためのインターフェイスです。セグメントリスナーイベント関数の実装では、onBeginSubsegment で作成された場合にすべてのサブセグメントに同じ注釈を追加したり、afterEndSegment を使用して各セグメントがデーモンに送信された後にメッセージをログに記録したりします。または、beforeEndSubsegment を使用して SQL インターセプタによって送信されたクエリを記録し、サブセグメントが SQL クエリを表しているかどうかを確認して、そうであればメタデータを追加します。

SegmentListener 関数の完全なリストについては、AWS X-Ray Recorder SDK for Java API のドキュメントを参照してください。

次の例は、onBeginSubsegment での作成時にすべてのサブセグメントに一貫性のある注釈を追加し、afterEndSegment を使用して各セグメントの最後にログメッセージを出力する方法を示しています。

Example MySegmentListener.java

```
import com.amazonaws.xray.entities.Segment;
import com.amazonaws.xray.entities.Subsegment;
import com.amazonaws.xray.listeners.SegmentListener;

public class MySegmentListener implements SegmentListener {
    .....

    @Override
    public void onBeginSubsegment(Subsegment subsegment) {
        subsegment.putAnnotation("annotationKey", "annotationValue");
```

```
    }

    @Override
    public void afterEndSegment(Segment segment) {
        // Be mindful not to mutate the segment
        logger.info("Segment with ID " + segment.getId());
    }
}
```

このカスタムセグメントリスナーは、AWSXRayRecorder を構築するときに参照されます。

Example AWSXRayRecorderBuilder statement

```
AWSXRayRecorderBuilder builder = AWSXRayRecorderBuilder
        .standard().withSegmentListener(new MySegmentListener());
```

環境変数

環境変数を使用して、X-Ray SDK for Java を設定できます。SDK は次の変数をサポートしています。

- AWS_XRAY_CONTEXT_MISSING – 計測されたコードが、セグメントが開いていないときにデータを記録しようとした場合に例外をスローするには、RUNTIME_ERROR に設定します。

  有効な値

  - RUNTIME_ERROR— ランタイム例外をスローします。
  - LOG_ERROR – エラーをログ記録して続行します (デフォルト)。
  - IGNORE_ERROR – エラーを無視して続行します。

  オープン状態のリクエストがない場合、または新しいスレッドを発生させるコードで、スタートアップコードに実装されたクライアントを使用しようとした場合に発生する可能性がある、セグメントまたはサブセグメントの欠落に関連するエラー。
- AWS_XRAY_DAEMON_ADDRESS – X-Ray デーモンリスナーのホストとポートを設定します。デフォルトでは、SDK はトレースデータ (UDP) とサンプリング (TCP) の両方に127.0.0.1:2000を使用します。この変数は、デーモンを次のように構成している場合に使用します。別のポートでリッスンするまたは、別のホストで実行されている場合。

  形式

  - 同じポート – *address*:*port*

  - 異なるポート – `tcp:address:port udp:address:port`
- `AWS_LOG_GROUP` - ロググループの名前を、ご使用のアプリケーションに関連付けられたロググループに設定します。ロググループがアプリケーションと同じ AWS アカウントとリージョンを使用している場合、X-Ray はこの指定されたロググループを使用してアプリケーションのセグメントデータを自動的に検索します。ロググループの詳細については、「ロググループとストリームの操作」を参照してください。
- `AWS_XRAY_TRACING_NAME` – SDK がセグメントに使用するサービス名を設定します。サーブレットフィルタのセグメント命名ルールで設定したサービス名を上書きします。

環境変数は、同等の「システムプロパティ」と、コードで設定される値を上書きします。

システムプロパティ

システムプロパティは、「環境変数」に代わる JVM 固有の代替的な方法として使用できます。SDK では、以下のプロパティをサポートしています。

- `com.amazonaws.xray.strategy.tracingName` – `AWS_XRAY_TRACING_NAME` と同等です。
- `com.amazonaws.xray.emitters.daemonAddress` – `AWS_XRAY_DAEMON_ADDRESS` と同等です。
- `com.amazonaws.xray.strategy.contextMissingStrategy` – `AWS_XRAY_CONTEXT_MISSING` と同等です。

環境変数と同等の環境変数のいずれも設定されている場合は、環境変数の値が使用されます。どちらのメソッドでも、コードで設定される値は上書きされます。

X-Ray SDK for Java を使用して受信リクエストをトレースします。

X-Ray SDK を使用して、アプリケーションが Amazon EC2 の EC2 インスタンス AWS Elastic Beanstalk、または Amazon ECS で処理する受信 HTTP リクエストをトレースできます。 Amazon EC2

`Filter` を使用して受信 HTTP リクエストを計測します。X-Ray サーブレットフィルターをアプリケーションに追加すると、X-Ray SDK for Java によってサンプリングされた各リクエストのセグメントが作成されます。このセグメントには、時間、メソッド、HTTP リクエストの処理などが含まれます。追加の計測により、このセグメントでサブセグメントが作成されます。

> **Note**
>
> AWS Lambda 関数の場合、Lambda はサンプリングされたリクエストごとにセグメントを作成します。詳細については「AWS Lambda and AWS X-Ray」を参照してください。

各セグメントには、サービスマップ内のアプリケーションを識別する名前があります。セグメントの名前は静的に指定することも、受信リクエストのホストヘッダーに基づいて動的に名前を付けるように SDK を設定することもできます。動的ネーミングでは、リクエスト内のドメイン名に基づいてトレースをグループ化でき、名前が予想されるパターンと一致しない場合 (たとえば、ホストヘッダーが偽造されている場合) 、デフォルト名を適用できます。

> **転送されたリクエスト**
>
> ロードバランサーまたは他の仲介者がアプリケーションにリクエストを転送する場合、X-Ray は、クライアントの IP をIP パケットの送信元 IP からではなく、リクエストのX-Forwarded-Forヘッダーから取得します。転送されたリクエストについて記録されたクライアント IP は偽造される可能性があるため、信頼されるべきではありません。

リクエストが転送されると、それを示す追加フィールドが SDK によってセグメントに設定されます。セグメントのフィールド `x_forwarded_for` が `true` に設定されている場合、クライアント IP が HTTP リクエストの `X-Forwarded-For` ヘッダーから取得されます。

メッセージハンドラーは、次の情報が含まれる `http` ブロックを使用して、各受信リクエスト用にセグメントを作成します。

- HTTP メソッド – GET、POST、PUT、DELETE、その他。
- クライアントアドレス – リクエストを送信するクライアントの IP アドレス。
- レスポンスコード – 完了したリクエストの HTTP レスポンスコード。
- タイミング – 開始時間 (リクエストが受信された時間) および終了時間 (レスポンスが送信された時間)。
- ユーザーエージェント — リクエストからの`user-agent`
- コンテンツの長さ — レスポンスからの`content-length`

セクション

- トレースフィルタをアプリケーション (Tomcat) に追加する
- トレースフィルタをアプリケーション (Spring) に追加する
- セグメント命名ルールの設定

トレースフィルタをアプリケーション (Tomcat) に追加する

Tomcat の場合は、プロジェクトの `<filter>` ファイルに `web.xml` を追加します。`fixedName` パラメーターを使用して、サービス名を指定し、着信リクエスト用に作成されたセグメントに適用します。

Example WEB-INF/web.xml - Tomcat

```
<filter>
  <filter-name>AWSXRayServletFilter</filter-name>
  <filter-class>com.amazonaws.xray.javax.servlet.AWSXRayServletFilter</filter-class>
  <init-param>
    <param-name>fixedName</param-name>
    <param-value>MyApp</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>AWSXRayServletFilter</filter-name>
  <url-pattern>*</url-pattern>
</filter-mapping>
```

トレースフィルタをアプリケーション (Spring) に追加する

Spring の場合は、`WebConfig` クラスに `Filter` を追加します。セグメント名を文字列として `AWSXRayServletFilter` コンストラクタに渡します。

Example src/main/java/myapp/WebConfig.java - Spring

```
package myapp;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Bean;
import javax.servlet.Filter;
import com.amazonaws.xray.javax.servlet.AWSXRayServletFilter;

@Configuration
public class WebConfig {
```

```
  @Bean
  public Filter TracingFilter() {
    return new AWSXRayServletFilter("Scorekeep");
  }
}
```

セグメント命名ルールの設定

AWS X-Ray は、サービス名を使用してアプリケーションを識別し、アプリケーションが使用する他のアプリケーション、データベース、外部 APIs、 AWS リソースと区別します。X-Ray SDK が受信リクエストのセグメントを生成すると、アプリケーションのサービス名がセグメントの名前フィールドに記録されます。

X-Ray SDK では、HTTP リクエストヘッダーのホスト名の後にセグメントの名前を指定できます。ただし、このヘッダーは偽造され、サービスマップに予期しないノードが発生する可能性があります。偽造されたホストヘッダーを持つリクエストによって SDK がセグメントの名前を間違えないようにするには、受信リクエストのデフォルト名を指定する必要があります。

アプリケーションが複数のドメインのリクエストを処理する場合、動的ネーミングストラテジーを使用してセグメント名にこれを反映するように SDK を設定できます。動的ネーミングストラテジーにより、SDK は予想されるパターンに一致するリクエストにホスト名を使用し、そうでないリクエストにデフォルト名を適用できます。

たとえば、3 つのサブドメイン (`www.example.com`,`api.example.com`,および`static.example.com`) に対してリクエストを処理する単一のアプリケーションがあるとします。動的ネーミングストラテジーをパターン `*.example.com` で使用して、異なる名前を持つ各サブドメインのセグメントを識別することができます。結果的にはサービスマップ上に 3 つのサービスノードを作成することになります。アプリケーションがパターンと一致しないホスト名のリクエストを受信すると、指定したフォールバック名を持つ 4 番目のノードがサービスマップに表示されます。

すべてのリクエストセグメントに対して同じ名前を使用するには、前のセクションで示すとおり、サーブレットフィルタを初期化するときに、アプリケーションの名前を指定します。これは、`SegmentNamingStrategy.fixed()` を呼び出して固定 SegmentNamingStrategy を作成し、それを `AWSXRayServletFilter` コンストラクタに渡すのと同じ効果があります。

> **Note**
>
> コードで定義したデフォルトのサービス名は、AWS_XRAY_TRACING_NAME 環境変数で上書きできます。

動的な命名戦略は、ホスト名と一致するようパターンを定義し、HTTP リクエストのホスト名がパターンと一致しない場合はデフォルトの名前を使用します。Tomcat で動的にセグメントに命名するには、dynamicNamingRecognizedHosts および dynamicNamingFallbackName を使用して、パターンとデフォルト名をそれぞれ定義します。

Example WEB-INF/web.xml - 動的名前付けのサーブレットフィルタ

```
<filter>
  <filter-name>AWSXRayServletFilter</filter-name>
  <filter-class>com.amazonaws.xray.javax.servlet.AWSXRayServletFilter</filter-class>
  <init-param>
    <param-name>dynamicNamingRecognizedHosts</param-name>
    <param-value>*.example.com</param-value>
  </init-param>
  <init-param>
    <param-name>dynamicNamingFallbackName</param-name>
    <param-value>MyApp</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>AWSXRayServletFilter</filter-name>
  <url-pattern>*</url-pattern>
</filter-mapping>
```

Spring の場合、SegmentNamingStrategy.dynamic()を呼び出して 動的 SegmentNamingStrategy を作成し、AWSXRayServletFilter コンストラクタに渡します。

Example src/main/java/myapp/WebConfig.java - servlet filter with dynamic naming

```
package myapp;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Bean;
import javax.servlet.Filter;
import com.amazonaws.xray.javax.servlet.AWSXRayServletFilter;
import com.amazonaws.xray.strategy.SegmentNamingStrategy;
```

```
@Configuration
public class WebConfig {

  @Bean
  public Filter TracingFilter() {
    return new AWSXRayServletFilter(SegmentNamingStrategy.dynamic("MyApp",
 "*.example.com"));
  }
}
```

X-Ray AWS SDK for Java を使用した SDK 呼び出しのトレース

アプリケーションが AWS のサービス を呼び出してデータの保存、キューへの書き込み、または通知の送信を行うと、X-Ray SDK for Java はサブセグメントのダウンストリームの呼び出しを追跡します。これらのサービス (例えば、Amazon S3 バケットまたは Amazon SQS キュー) 内でアクセスするトレースされた AWS のサービス およびリソースは、X-Ray コンソールのトレースマップではダウンストリームノードとして表示されます。

aws-sdk および aws-sdk-instrumentor サブモジュールをビルドに含めると、X-Ray SDK for Java では自動的にすべての AWS SDK クライアントを計測します。Instrumentor サブモジュールを含めない場合は、一部のクライアントを計測して他を除外できます。

個々のクライアントを計測するには、ビルドからaws-sdk-instrumentorサブモジュールを削除し、サービスのクライアントビルダーを使用して AWS SDK クライアントTracingHandlerに XRayClientとして を追加します。

たとえば、AmazonDynamoDB を計測するには、トレースハンドラーを AmazonDynamoDBClientBuilder に渡します。

Example MyModel.java - DynamoDB クライアント

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.handlers.TracingHandler;

...
public class MyModel {
  private AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
        .withRegion(Regions.fromName(System.getenv("AWS_REGION")))
        .withRequestHandlers(new TracingHandler(AWSXRay.getGlobalRecorder()))
        .build();
...
```

すべてのサービスで、X-Ray コンソールで呼び出された API の名前を確認できます。サービスのサブセットの場合、X-Ray SDK はセグメントに情報を追加して、サービスマップでより細かく指定します。

たとえば、実装された DynamoDB クライアントでコールすると、SDK はテーブルをターゲットとするコールのセグメントにテーブル名を追加します。コンソールで、各テーブルはサービスマップ内に個別のノードとして表示され、テーブルをターゲットにしないコール用の汎用の DynamoDB ノードが表示されます。

Example 項目を保存するための DynamoDB に対するコールのサブセグメント

```
{
  "id": "24756640c0d0978a",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "DynamoDB",
  "namespace": "aws",
  "http": {
    "response": {
      "content_length": 60,
      "status": 200
    }
  },
  "aws": {
    "table_name": "scorekeep-user",
    "operation": "UpdateItem",
    "request_id": "UBQNSO5AEM8T4FDA4RQDEB94OVTDRVV4K4HIRGVJF66Q9ASUAAJG",
  }
}
```

名前付きリソースにアクセスしたとき、次のサービスをコールすると、サービスマップに追加のノードが作成されます。特定のリソースをターゲットとしないコールでは、サービスの汎用ノードが作成されます。

- Amazon DynamoDB – テーブル名
- Amazon Simple Storage Service – バケットとキー名
- Amazon Simple Queue Service – キュー名

AWS SDK for Java 2.2 以降 AWS のサービス で へのダウンストリーム呼び出しを計測するには、ビルド設定から`aws-xray-recorder-sdk-aws-sdk-v2-instrumentor`モジュー

ルを省略できます。その代わりに、aws-xray-recorder-sdk-aws-sdk-v2 module を含め、TracingInterceptor で設定して個々のクライアントを実装します。

Example AWS SDK for Java 2.2 以降 - インターセプターのトレース

```
import com.amazonaws.xray.interceptors.TracingInterceptor;
import software.amazon.awssdk.core.client.config.ClientOverrideConfiguration
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
//...
public class MyModel {
private DynamoDbClient client = DynamoDbClient.builder()
.region(Region.US_WEST_2)
.overrideConfiguration(ClientOverrideConfiguration.builder()
.addExecutionInterceptor(new TracingInterceptor())
.build()
)
.build();
//...
```

X-Ray SDK for Java を使用してダウンストリーム HTTP ウェブサービスの呼び出しをトレースする

アプリケーションがマイクロサービスまたはパブリック HTTP API に呼び出しを実行する場合に、X-Ray SDK for Java の HttpClient バージョンを使用してこれらの呼び出しを計測し、API をダウンストリームサービスとしてサービスグラフに追加できます。

X-Ray SDK for Java には、送信 HTTP 呼び出しの計測と同等の Apache HttpComponents の代わりに使用できる DefaultHttpClient および HttpClientBuilder クラスが含まれています。

- com.amazonaws.xray.proxies.apache.http.DefaultHttpClient - org.apache.http.impl.client.DefaultHttpClient
- com.amazonaws.xray.proxies.apache.http.HttpClientBuilder - org.apache.http.impl.client.HttpClientBuilder

これらのライブラリは、aws-xray-recorder-sdk-apache-httpサブモジュールにあります。

既存のインポートステートメントを X-Ray の該当部分に置き換えてすべてのクライアントを計測するか、クライアントを初期化する際に完全修飾名を使用して特定のクライアントを計測できます。

Example HttpClientBuilder

```
import com.fasterxml.jackson.databind.ObjectMapper;
import org.apache.http.HttpEntity;
import org.apache.http.client.methods.CloseableHttpResponse;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.util.EntityUtils;
import com.amazonaws.xray.proxies.apache.http.HttpClientBuilder;
...
  public String randomName() throws IOException {
    CloseableHttpClient httpclient = HttpClientBuilder.create().build();
    HttpGet httpGet = new HttpGet("http://names.example.com/api/");
    CloseableHttpResponse response = httpclient.execute(httpGet);
    try {
      HttpEntity entity = response.getEntity();
      InputStream inputStream = entity.getContent();
      ObjectMapper mapper = new ObjectMapper();
      Map<String, String> jsonMap = mapper.readValue(inputStream, Map.class);
      String name = jsonMap.get("name");
      EntityUtils.consume(entity);
      return name;
    } finally {
      response.close();
    }
  }
```

ダウンストリームウェブ API に対する呼び出しを計測すると、X-Ray SDK for Java は HTTP リクエストおよびレスポンスに関する情報を含むセグメントを記録します。X-Ray はサブセグメントを使用してリモート API の推測セグメントを生成します。

Example ダウンストリーム HTTP 呼び出しのサブセグメント

```
{
  "id": "004f72be19cddc2a",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "name": "names.example.com",
  "namespace": "remote",
  "http": {
    "request": {
      "method": "GET",
      "url": "https://names.example.com/"
```

```
    },
    "response": {
      "content_length": -1,
      "status": 200
    }
  }
}
```

Example ダウンストリーム HTTP 呼び出しの推定セグメント

```
{
  "id": "168416dc2ea97781",
  "name": "names.example.com",
  "trace_id": "1-62be1272-1b71c4274f39f122afa64eab",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "parent_id": "004f72be19cddc2a",
  "http": {
    "request": {
      "method": "GET",
      "url": "https://names.example.com/"
    },
    "response": {
      "content_length": -1,
      "status": 200
    }
  },
  "inferred": true
}
```

X-Ray SDK for Java での SQL クエリのトレース

SQL インターセプター

X-Ray SDK for Java JDBC インターセプターをデータソース設定に追加して、SQL データベースのクエリを計測します。

- PostgreSQL – `com.amazonaws.xray.sql.postgres.TracingInterceptor`
- MySQL – `com.amazonaws.xray.sql.mysql.TracingInterceptor`

これらのインターセプタは、それぞれaws-xray-recorder-sql-postgres と aws-xray-recorder-sql-mysql サブモジュールにあります。これらは、Tomcat の接続プールと互換性がある `org.apache.tomcat.jdbc.pool.JdbcInterceptor` を実装します。

> **Note**
>
> SQL インターセプタは、セキュリティ上の目的で SQL クエリ自体をサブセグメント内に記録しません。

Spring の場合は、プロパティファイルのインターセプターを追加し、スプリングブートの `DataSourceBuilder` を使用して、データソースを構築します。

Example **`src/main/java/resources/application.properties`** - PostgreSQL JDBC Interceptor

```
spring.datasource.continue-on-error=true
spring.jpa.show-sql=false
spring.jpa.hibernate.ddl-auto=create-drop
spring.datasource.jdbc-interceptors=com.amazonaws.xray.sql.postgres.TracingInterceptor
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQL94Dialect
```

Example **`src/main/java/myapp/WebConfig.java`** - データソース

```
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.jdbc.DataSourceBuilder;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

import javax.servlet.Filter;
import javax.sql.DataSource;
import java.net.URL;

@Configuration
@EnableAutoConfiguration
@EnableJpaRepositories("myapp")
public class RdsWebConfig {

  @Bean
  @ConfigurationProperties(prefix = "spring.datasource")
```

```
  public DataSource dataSource() {
      logger.info("Initializing PostgreSQL datasource");
      return DataSourceBuilder.create()
              .driverClassName("org.postgresql.Driver")
              .url("jdbc:postgresql://" + System.getenv("RDS_HOSTNAME") + ":" +
 System.getenv("RDS_PORT") + "/ebdb")
              .username(System.getenv("RDS_USERNAME"))
              .password(System.getenv("RDS_PASSWORD"))
              .build();
  }
...
}
```

Tomcat の場合、X-Ray SDK for Java クラスを参照する JDBC データソースの `setJdbcInterceptors` を呼び出します。

Example **`src/main/myapp/model.java`** - データソース

```
import org.apache.tomcat.jdbc.pool.DataSource;
...
DataSource source = new DataSource();
source.setUrl(url);
source.setUsername(user);
source.setPassword(password);
source.setDriverClassName("com.mysql.jdbc.Driver");
source.setJdbcInterceptors("com.amazonaws.xray.sql.mysql.TracingInterceptor;");
```

Tomcat JDBC データソースライブラリは、X-Ray SDK for Java に含まれていますが、使用するドキュメントへの指定された依存関係として宣言できます。

Example **`pom.xml`** - JDBC Data Source

```
<dependency>
  <groupId>org.apache.tomcat</groupId>
  <artifactId>tomcat-jdbc</artifactId>
  <version>8.0.36</version>
  <scope>provided</scope>
</dependency>
```

ネイティブ SQL トレースデコレータ

- `aws-xray-recorder-sdk-sql` を依存関係に追加します。

- データベースのデータソース、接続、またはステートメントを修飾します。

```
dataSource = TracingDataSource.decorate(dataSource)
connection = TracingConnection.decorate(connection)
statement = TracingStatement.decorateStatement(statement)
preparedStatement = TracingStatement.decoratePreparedStatement(preparedStatement,
 sql)
callableStatement = TracingStatement.decorateCallableStatement(callableStatement,
 sql)
```

X-Ray SDK for Java を使用したカスタムサブセグメントの生成

サブセグメントは、トレースの セグメント をリクエストを処理するために行われた作業の詳細で拡張します。計測済みクライアント内で呼び出しを行うたびに、X-Ray SDK によってサブセグメントに生成された情報が記録されます。追加のサブセグメントを作成して、他のサブセグメントをグループ化したり、コードセクションのパフォーマンスを測定したり、注釈とメタデータを記録したりできます。

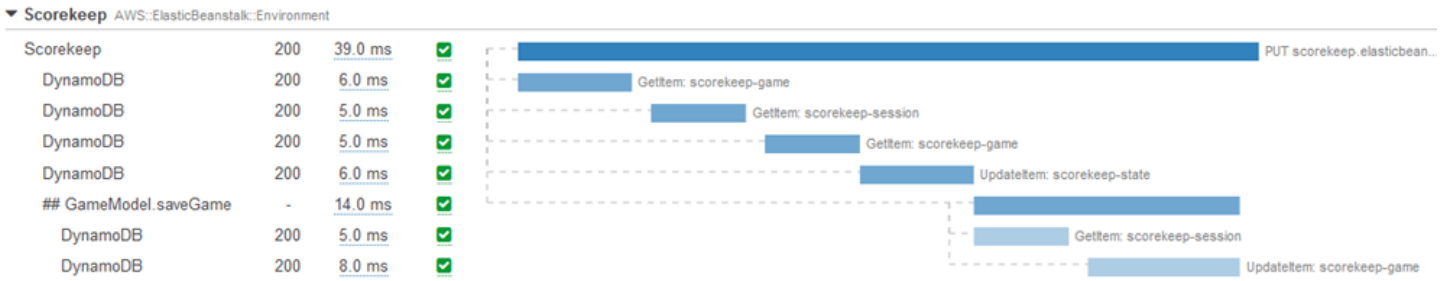
サブセグメントを管理するには、beginSubsegment および endSubsegment メソッドを使用します。

Example GameModel.java - カスタムサブセグメント

```
import com.amazonaws.xray.AWSXRay;
...
  public void saveGame(Game game) throws SessionNotFoundException {
    // wrap in subsegment
    Subsegment subsegment = AWSXRay.beginSubsegment("Save Game");
    try {
      // check session
      String sessionId = game.getSession();
      if (sessionModel.loadSession(sessionId) == null ) {
        throw new SessionNotFoundException(sessionId);
      }
      mapper.save(game);
    } catch (Exception e) {
      subsegment.addException(e);
      throw e;
    } finally {
      AWSXRay.endSubsegment();
    }
```

```
  }
```

この例では、サブセグメント内のコードは、セッションモデルの メソッドを使用して DynamoDB からゲームのセッションをロードし、 AWS SDK for Javaの DynamoDB マッパーを使用してゲームを保存します。このコードをサブセグメントにラップすることで、呼び出しが `Save Game` サブセグメントの DynamoDB の子としてコンソールのトレースビューに表示されます。



サブセグメントのコードがチェック例外をスローした場合は、`try` ブロックにコードをラップして、`finally` ブロックで `AWSXRay.endSubsegment()` を呼び出し、常にサブセグメントが閉じられるようにします。サブセグメントが閉じていない場合は、親セグメントが完了できず、X-Ray に送信されません。

チェック例外をスローしないコードの場合は、コードを Lambda 関数として `AWSXRay.CreateSubsegment` に渡すことができます。

Example Lambda 関数のサブセグメント

```
import com.amazonaws.xray.AWSXRay;

AWSXRay.createSubsegment("getMovies", (subsegment) -> {
    // function code
});
```

セグメントまたは別のサブセグメント内にサブセグメントを作成する場合、X-Ray SDK for Java によってその ID が生成され、開始時刻と終了時刻が記録されます。

Example サブセグメントとメタデータ

```
"subsegments": [{
  "id": "6f1605cd8a07cb70",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "Custom subsegment for UserModel.saveUser function",
```

```
  "metadata": {
    "debug": {
      "test": "Metadata string from UserModel.saveUser"
    }
  },
```

非同期やマルチスレッドのプログラミングでは、非同期実行中に X-Ray コンテキストが変更されることがあるため、サブセグメントを endSubsegment() メソッドに手動で渡して正しく閉じるようにする必要があります。親セグメントが閉じられた後に非同期サブセグメントが閉じられた場合、このメソッドはセグメント全体を X-Ray デーモンに自動的にストリームします。

Example 非同期サブセグメント

```
@GetMapping("/api")
public ResponseEntity<?> api() {
  CompletableFuture.runAsync(() -> {
      Subsegment subsegment = AWSXRay.beginSubsegment("Async Work");
      try {
          Thread.sleep(3000);
      } catch (InterruptedException e) {
          subsegment.addException(e);
          throw e;
      } finally {
          AWSXRay.endSubsegment(subsegment);
      }
  });
  return ResponseEntity.ok().build();
}
```

X-Ray SDK for Java を使用してセグメントに注釈とメタデータを追加する

注釈やメタデータにより、リクエストや環境、アプリケーションに関する追加情報を記録できます。X-Ray SDK が作成するセグメントまたは作成するカスタムサブセグメントに、注釈およびメタデータを追加できます。

注釈は文字列、数値、またはブール値を使用したキーと値のペアです。注釈は、フィルタ式用にインデックス付けされます。注釈を使用して、コンソールでトレースをグループ化するため、またはGetTraceSummaries API を呼び出すときに使用するデータを記録します。

メタデータは、オブジェクトとリストを含む、任意のタイプの値を持つことができるキーバリューのペアですが、フィルタ式に使用するためにインデックスは作成されません。メタデータを使用してトレースに保存する追加のデータを記録しますが、検索で使用する必要はありません。

注釈とメタデータに加えて、セグメントにユーザー ID 文字列を記録することもできます。ユーザー ID はセグメントの個別のフィールドに記録され、検索用にインデックスが作成されます。

セクション

- X-Ray SDK for Java での注釈の記録
- X-Ray SDK for Java を使用したメタデータの記録
- X-Ray SDK for Javaを使用したユーザー ID の記録

X-Ray SDK for Java での注釈の記録

注釈を使用して、検索用にインデックスを作成するセグメントまたはサブセグメントに情報を記録します。

注釈の要件

- キー – X-Ray 注釈のキーには最大 500 文字の英数字を使用できます。スペースまたはドットやピリオド (.) 以外の記号は使用できません。
- 値 – X-Ray 注釈の値には最大 1,000 の Unicode 文字を使用できます。
- 注釈の数 – トレースごとに最大 50 の注釈を使用できます。

注釈を記録するには

1. AWSXRay から現在のセグメントまたはサブセグメントの参照を取得します。

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.entities.Segment;
...
Segment document = AWSXRay.getCurrentSegment();
```

or

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.entities.Subsegment;
```

```
...
Subsegment document = AWSXRay.getCurrentSubsegment();
```

2. 文字列キー、および、ブール値、数値、文字列値を使用して putAnnotation を呼び出します。

```
document.putAnnotation("mykey", "my value");
```

次の例は、ドット、ブール値、数値、または文字列値を含む文字列キーを使用して putAnnotation を呼び出す方法を示しています。

```
document.putAnnotation("testkey.test", "my value");
```

SDK は、セグメントドキュメントの annotations オブジェクトにキーと値のペアとして、注釈を記録します。同じキーで putAnnotation を 2 回呼び出すと、同じセグメントまたはサブセグメントに以前記録された値が上書きされます。

特定の値を持つ注釈のあるトレースを見つけるには、annotation[*key*]フィルタ式の キーワードを使用します。

Example **src/main/java/scorekeep/GameModel.java** – 注釈とメタデータ

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.entities.Segment;
import com.amazonaws.xray.entities.Subsegment;
...
  public void saveGame(Game game) throws SessionNotFoundException {
    // wrap in subsegment
    Subsegment subsegment = AWSXRay.beginSubsegment("## GameModel.saveGame");
    try {
      // check session
      String sessionId = game.getSession();
      if (sessionModel.loadSession(sessionId) == null ) {
        throw new SessionNotFoundException(sessionId);
      }
      Segment segment = AWSXRay.getCurrentSegment();
      subsegment.putMetadata("resources", "game", game);
      segment.putAnnotation("gameid", game.getId());
      mapper.save(game);
    } catch (Exception e) {
      subsegment.addException(e);
```

```
      throw e;
    } finally {
      AWSXRay.endSubsegment();
    }
  }
```

X-Ray SDK for Java を使用したメタデータの記録

メタデータを使用して、検索用にインデックスを作成する必要のないセグメントまたはサブセグメントに情報を記録します。メタデータ値は、文字列、数値、ブール値、または JSON オブジェクトや JSON 配列にシリアル化できる任意のオブジェクトになります。

メタデータを記録するには

1. AWSXRay から現在のセグメントまたはサブセグメントの参照を取得します。

   ```
   import com.amazonaws.xray.AWSXRay;
   import com.amazonaws.xray.entities.Segment;
   ...
   Segment document = AWSXRay.getCurrentSegment();
   ```

   or

   ```
   import com.amazonaws.xray.AWSXRay;
   import com.amazonaws.xray.entities.Subsegment;
   ...
   Subsegment document = AWSXRay.getCurrentSubsegment();
   ```

2. 文字列名前空間、文字列キー、ブール値、数値、文字列値、オブジェクト値を使用して putMetadata を呼び出します。

   ```
   document.putMetadata("my namespace", "my key", "my value");
   ```

   or

   キーと値だけを使用して putMetadata を呼び出します。

   ```
   document.putMetadata("my key", "my value");
   ```

名前空間を指定しない場合、SDK は default を使用します。同じキーで putMetadata を 2 回呼び出すと、同じセグメントまたはサブセグメントに以前記録された値が上書きされます。

Example **src/main/java/scorekeep/GameModel.java** – 注釈とメタデータ

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.entities.Segment;
import com.amazonaws.xray.entities.Subsegment;
...
  public void saveGame(Game game) throws SessionNotFoundException {
    // wrap in subsegment
    Subsegment subsegment = AWSXRay.beginSubsegment("## GameModel.saveGame");
    try {
      // check session
      String sessionId = game.getSession();
      if (sessionModel.loadSession(sessionId) == null ) {
        throw new SessionNotFoundException(sessionId);
      }
      Segment segment = AWSXRay.getCurrentSegment();
      subsegment.putMetadata("resources", "game", game);
      segment.putAnnotation("gameid", game.getId());
      mapper.save(game);
    } catch (Exception e) {
      subsegment.addException(e);
      throw e;
    } finally {
      AWSXRay.endSubsegment();
    }
  }
```

X-Ray SDK for Javaを使用したユーザー ID の記録

リクエストセグメントにユーザー ID を記録して、リクエストを送信したユーザーを識別します。

ユーザー ID を記録するには

1. AWSXRay から現在のセグメントへの参照を取得します。

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.entities.Segment;
...
Segment document = AWSXRay.getCurrentSegment();
```

2. リクエストを送信したユーザーの文字列 ID を使用して setUser を呼び出します。

```
document.setUser("U12345");
```

コントローラーで setUser を呼び出し、アプリケーションがリクエストの処理を開始するとすぐに、ユーザー ID を記録できます。ユーザー ID を設定するためだけにセグメントを使用する場合、呼び出しを 1 行で連鎖させることができます。

Example src/main/java/scorekeep/MoveController.java – ユーザー ID

```
import com.amazonaws.xray.AWSXRay;
...
  @RequestMapping(value="/{userId}", method=RequestMethod.POST)
  public Move newMove(@PathVariable String sessionId, @PathVariable String
 gameId, @PathVariable String userId, @RequestBody String move) throws
 SessionNotFoundException, GameNotFoundException, StateNotFoundException,
 RulesException {
    AWSXRay.getCurrentSegment().setUser(userId);
    return moveFactory.newMove(sessionId, gameId, userId, move);
  }
```

ユーザー ID のトレースを見つけるには、userフィルタ式で、 キーワードを使用します。

AWS X-Ray X-Ray SDK for Java の メトリクス

このトピックでは、 AWS X-Ray 名前空間、メトリクス、ディメンションについて説明します。X-Ray SDK for Java を使用して、収集した X-Ray セグメントから、サンプリングされていない Amazon CloudWatch メトリクスを公開できます。これらのメトリクスは、セグメントの開始時間と終了時間、さらにエラー、障害、スロットリングのステータスフラグから取得されます。これらの追跡メトリクスを使用して、サブセグメント内の再試行と依存関係の問題を公開します。

CloudWatch はメトリクスリポジトリです。メトリクスは CloudWatch の基本的概念で、時系列に並んだデータポイントのセットを表しています。ユーザー (または AWS のサービス) はメトリクスデータポイントを CloudWatch に発行し、それらのデータポイントに関する統計を時系列データの順序付けられたセットとして取得します。

メトリクスは名前、名前空間、1 つ以上のディメンションで一意に定義されます。各データポイントには、タイムスタンプと、オプションとして測定単位があります。統計を要求した場合、返されるデータストリームは、名前空間、メトリクス名、ディメンションによって識別されます。

CloudWatch の詳細については、「Amazon CloudWatch ユーザーガイド」を参照してください。

X-Ray CloudWatch メトリクス

`ServiceMetrics/SDK` 名前空間には、次のメトリクスが含まれます。

メトリクス	利用可能な統計情報	説明	単位
`Latency`	Average、Minimum、Maximum、Count	開始時間と終了時間の差。Average、Minimum、Maximum はすべて、オペレーション関連のレイテンシーを表します。Count は、呼び出し数を表します。	ミリ秒
`ErrorRate`	Average、Sum	`4xx Client Error` ステータスコードで失敗し、エラーになったリクエストの割合。	割合 (%)
`FaultRate`	Average、Sum	`5xx Server Error` ステータスコードで失敗し、障害になったトレースの割合。	割合 (%)
`ThrottleRate`	Average、Sum	429 ステータスコードを返すスロットリングトレースの割合。これは `ErrorRate` メトリクスのサブセットです。	割合 (%)
`OkRate`	Average、Sum	トレースされた後、OK ステータスコード	割合 (%)

メトリクス	利用可能な統計情報	説明	単位
		になったリクエストの割合。	

X-Ray CloudWatch ディメンション

以下の表のディメンションを使用して、X-Ray によって計測された Java アプリケーションに対して返されるメトリクスを絞り込みます。

ディメンション	説明
`ServiceType`	不明な場合、サービスのタイプ (`AWS::EC2::Instance` 、`NONE` など)。
`ServiceName`	サービスの正規名。

X-Ray CloudWatch メトリクスを有効にする

以下の手順を使用して、計測された Java アプリケーションでトレースメトリクスを有効にします。

トレースメトリクスを設定するには

1. `aws-xray-recorder-sdk-metrics` パッケージを Apache Maven 依存関係として追加します。詳細は、X-Ray SDK for Java サブモジュールを参照してください。
2. グローバルレコーダービルドの一部として新しい `MetricsSegmentListener()` を有効にします。

   Example src/com/myapp/web/Startup.java

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.AWSXRayRecorderBuilder;
import com.amazonaws.xray.plugins.EC2Plugin;
import com.amazonaws.xray.plugins.ElasticBeanstalkPlugin;
import com.amazonaws.xray.strategy.sampling.LocalizedSamplingStrategy;

@Configuration
public class WebConfig {
```

```
...
  static {
    AWSXRayRecorderBuilder builder = AWSXRayRecorderBuilder
                                        .standard()
                                        .withPlugin(new EC2Plugin())
                                        .withPlugin(new ElasticBeanstalkPlugin())
                                        .withSegmentListener(new
 MetricsSegmentListener());

    URL ruleFile = WebConfig.class.getResource("/sampling-rules.json");
    builder.withSamplingStrategy(new LocalizedSamplingStrategy(ruleFile));

    AWSXRay.setGlobalRecorder(builder.build());
  }
}
```

3. CloudWatch エージェントをデプロイして、Amazon Elastic Compute Cloud (Amazon EC2)、Amazon Elastic Container Service (Amazon ECS)、または Amazon Elastic Kubernetes Service (Amazon EKS) を使用してメトリクスを収集します。

   - Amazon EC2 を設定するには、「CloudWatch エージェントのインストール」を参照してください。
   - Amazon ECS を設定するには、「Container Insights を使用して Amazon ECS コンテナをモニタリングする」を参照してください。
   - Amazon ESK を設定するには、「Amazon CloudWatch Observability EKS アドオンを使用して CloudWatch エージェントをインストールする」を参照してください。

4. CloudWatch エージェントと通信するように SDK を設定します。デフォルトでは、SDK はアドレス `127.0.0.1` の CloudWatch エージェントと通信します。環境変数または Java プロパティを `address:port` に設定することで、代替アドレスを設定できます。

   Example 環境変数

   ```
   AWS_XRAY_METRICS_DAEMON_ADDRESS=address:port
   ```

   Example Java のプロパティ

   ```
   com.amazonaws.xray.metrics.daemonAddress=address:port
   ```

設定を検証するには

1. にサインイン AWS Management Console し、https://console.aws.amazon.com/cloudwatch/:// www.com」で CloudWatch コンソールを開きます。
2. [Metrics (メトリクス)] タブを開いて、メトリクスの到着を確認します。
3. (オプション) CloudWatch コンソールのLogs (ログ) タブで、`ServiceMetricsSDK`ロググループを開きます。ホストメトリクスに一致するログストリームを探し、ログメッセージを確認します。

マルチスレッドアプリケーションでのスレッド間のセグメントコンテキストの受け渡し

アプリケーションで新しいスレッドを作成すると、`AWSXRayRecorder` は現在のセグメントまたはサブセグメント Entity への参照を保持しません。実装されたクライアントを新しいスレッドで使用すると、SDK は存在しないセグメントに書き込みを試み、SegmentNotFoundException が発生します。

開発中に例外をスローしないようにするには、代わりにエラーを記録するよう指示する ContextMissingStrategy でレコーダーを設定します。SetContextMissingStrategy を使用してコード内に戦略を設定するか、環境変数 または システムプロパティ を使用して同等のオプションを設定することができます。

エラーに対処する 1 つの方法として、スレッドを開始するときに beginSegment を呼び出して新しいセグメントを使用する方法と、スレッドを終了するときに endSegment を使用する方法があります。これは、アプリケーションの起動時に実行されるコードのように、HTTP リクエストに応答して実行されないコードを実装する場合に機能します。

複数のスレッドを使用して着信リクエストを処理する場合は、現在のセグメントまたはサブセグメントを新しいスレッドに渡してグローバルレコーダーに渡すことができます。これにより、新しいスレッド内に記録された情報が、そのリクエストに関して記録された残りの情報と同じセグメントに関連付けられることが保証されます。新しいスレッドでセグメントが使用可能になると、そのセグメントのコンテキストにアクセスできる任意の実行可能なファイルを `segment.run(() -> { ... })` メソッドで実行できるようになります。

例については、「実装されたクライアントをワーカースレッドで使用する」を参照してください。

非同期プログラミングでの X-Ray の使用

X-Ray SDK for Java は、SegmentContextExecutors を使用した非同期 Java プログラムで使用できます。SegmentContextExecutor は Executor インターフェイスを実装しており、CompletableFuture のすべての非同期操作に渡せます。これにより、非同期オペレーションがそのコンテキスト内で正しいセグメントで実行されることが保証されます。

Example App.java 例: SegmentContextExecutor を CompletableFuture に渡す

```
DynamoDbAsyncClient client = DynamoDbAsyncClient.create();

AWSXRay.beginSegment();

// ...

client.getItem(request).thenComposeAsync(response -> {
    // If we did not provide the segment context executor, this request would not be
 traced correctly.
    return client.getItem(request2);
}, SegmentContextExecutors.newSegmentContextExecutor());
```

Spring による AOP と X-Ray SDK for Java

このトピックでは、X-Ray SDK および Spring Framework を使用して、コアロジックを変更せずにアプリケーションを計測する方法について説明します。つまり、リモートで実行されているアプリケーションをインストルメントする非侵入的な方法があるということです AWS。

Spring で AOP を有効にするには

1. Spring を設定します
2. トレースフィルターをアプリケーションに追加する
3. コードに注釈を付けるか、インターフェイスを実装します
4. アプリケーションで X-Ray を有効化します

Spring の設定

Maven または Gradle を使用して、AOP でアプリケーションを計測するように Spring を設定できます。

Maven を使用してアプリケーションを構築する場合は、次の依存関係を pom.xml ファイルに追加します。

```
<dependency>
     <groupId>com.amazonaws</groupId>
     <artifactId>aws-xray-recorder-sdk-spring</artifactId>
     <version>2.11.0</version>
</dependency>
```

Gradle の場合は、次の依存関係を build.gradle ファイルに追加します。

```
compile 'com.amazonaws:aws-xray-recorder-sdk-spring:2.11.0'
```

Spring Boot の設定

前のセクションで説明した Spring の依存関係に加えて、Spring Boot を使用している場合で、それがまだクラスパス上にない場合は、次の依存関係を追加します。

Maven:

```
<dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-aop</artifactId>
     <version>2.5.2</version>
</dependency>
```

Gradle:

```
compile 'org.springframework.boot:spring-boot-starter-aop:2.5.2'
```

トレースフィルターをアプリケーションに追加する

WebConfig のクラスに Filter を追加します。セグメント名を文字列として AWSXRayServletFilter コンストラクタに渡します。フィルターのトレースおよび受信リクエストの計測の詳細については、「X-Ray SDK for Java を使用して受信リクエストをトレースします。」を参照してください。

Example src/main/java/myapp/WebConfig.java - Spring

```
package myapp;
import org.springframework.context.annotation.Configuration;
```

```
import org.springframework.context.annotation.Bean;
import javax.servlet.Filter;
import com.amazonaws.xray.javax.servlet.AWSXRayServletFilter;

@Configuration
public class WebConfig {

  @Bean
  public Filter TracingFilter() {
    return new AWSXRayServletFilter("Scorekeep");
  }
}
```

Jakarta のサポート

Spring 6 の Enterprise Edition では、Javax ではなく Jakarta を使用しています。この新しい名前空間をサポートするために、X-Ray では独自の Jakarta 名前空間に存在するクラスの並列セットを作成しました。

フィルタークラスの場合は、`javax` を `jakarta` に置き換えてください。セグメント命名ルールを設定するときは、次の例のように、命名ルールクラス名の前に `jakarta` を追加してください。

```
package myapp;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Bean;
import jakarta.servlet.Filter;
import com.amazonaws.xray.jakarta.servlet.AWSXRayServletFilter;
import com.amazonaws.xray.strategy.jakarta.SegmentNamingStrategy;

@Configuration
public class WebConfig {
    @Bean
    public Filter TracingFilter() {
        return new AWSXRayServletFilter(SegmentNamingStrategy.dynamic("Scorekeep"));
    }
}
```

コードに注釈を付ける、またはインターフェイスを実装する

クラスには `@XRayEnabled` の注釈を付けるか、`XRayTraced` インターフェイスを実装する必要があります。これにより、X-Ray の計測のため、影響を受けた関数をラップするように AOP システムに伝えます。

アプリケーションでの X-Ray のアクティベーション

アプリケーションで X-Ray トレースを有効化するには、コードで次のメソッドをオーバーライドして、抽象クラス BaseAbstractXRayInterceptor を拡張する必要があります。

- generateMetadata—この関数では、現在の関数のトレースにアタッチされたメタデータをカスタマイズできます。デフォルトでは、実行中の関数のクラス名がメタデータに記録されます。追加情報が必要な場合は、さらにデータを追加できます。
- xrayEnabledClasses—この関数は空であり、空のままにしておく必要があります。これは、ラップするメソッドをインターセプターに指示するポイントカットのホストとして機能します。@XRayEnabled の注釈が付けられたどのクラスをトレースするか指定して、ポイントカットを定義します。次のステートメントは、@XRayEnabled という注釈が付けられたすべてのコントローラービーンをラップするようにインターセプターに伝えます。

  ```
  @Pointcut(“@within(com.amazonaws.xray.spring.aop.XRayEnabled) && bean(*Controller)”)
  ```

プロジェクトで Spring Data JPA を使用している場合は、BaseAbstractXRayInterceptor ではなく AbstractXRayInterceptor から拡張することを検討してください。

例

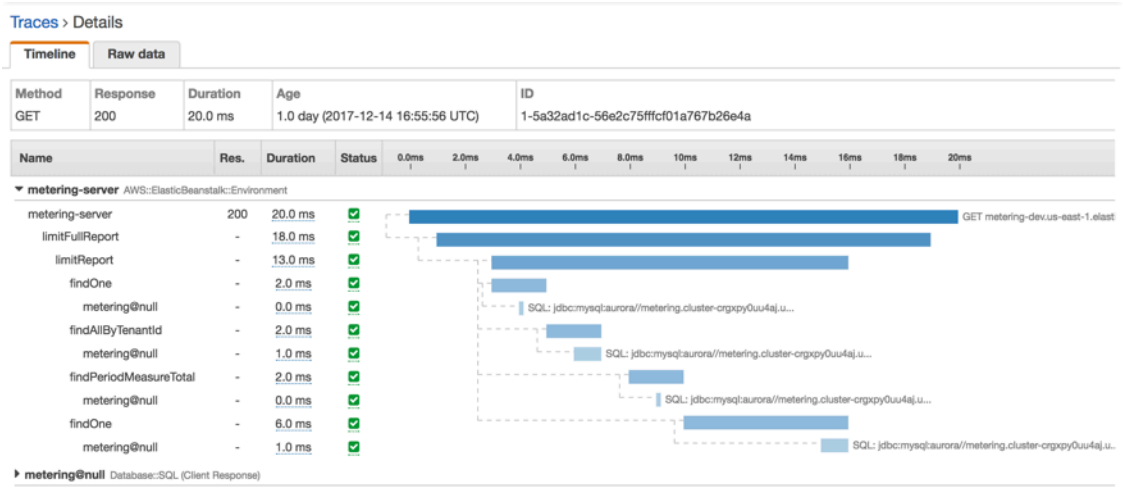
次のコードは、抽象クラス BaseAbstractXRayInterceptorを示しています。

```
@Aspect
@Component
public class XRayInspector extends BaseAbstractXRayInterceptor {
    @Override
    protected Map<String, Map<String, Object>> generateMetadata(ProceedingJoinPoint
 proceedingJoinPoint, Subsegment subsegment) throws Exception {
        return super.generateMetadata(proceedingJoinPoint, subsegment);
    }

  @Override
  @Pointcut("@within(com.amazonaws.xray.spring.aop.XRayEnabled) && bean(*Controller)")

  public void xrayEnabledClasses() {}

}
```

次のコードは、X-Ray によって計測されるクラスです。

```
@Service
@XRayEnabled
public class MyServiceImpl implements MyService {
    private final MyEntityRepository myEntityRepository;

    @Autowired
    public MyServiceImpl(MyEntityRepository myEntityRepository) {
        this.myEntityRepository = myEntityRepository;
    }

    @Transactional(readOnly = true)
    public List<MyEntity> getMyEntities(){
        try(Stream<MyEntity> entityStream = this.myEntityRepository.streamAll()){

            return entityStream.sorted().collect(Collectors.toList());
        }
    }
}
```

アプリケーションを正しく設定した場合は、コンソールの次のスクリーンショットに示すように、コントローラーからサービス呼び出しまで、アプリケーションの完全なコールスタックが表示されます。



Node.js の使用

X-Ray にトレースを送信する Node.js アプリケーションを計測するには、次の 2 つの方法があります。

- AWS Distro for OpenTelemetry JavaScript – AWS Distro for OpenTelemetry Collector を介して、相関メトリクスとトレースを Amazon CloudWatch、 AWS X-Ray Amazon OpenSearch Service などの複数の AWS モニタリングソリューションに送信するための一連のオープンソースライブラリを提供する AWS ディストリビューション。
- AWS X-Ray SDK for Node.js – X-Ray デーモンを介してトレースを生成して X-Ray に送信するためのライブラリのセット。

詳細については、「Distro for OpenTelemetry AWS と X-Ray SDKs の選択」を参照してください。

AWS Distro for OpenTelemetry JavaScript

AWS Distro for OpenTelemetry (ADOT) JavaScript を使用すると、アプリケーションを一度計測し、相関メトリクスとトレースを Amazon CloudWatch AWS X-Ray、Amazon OpenSearch Service などの複数の AWS モニタリングソリューションに送信できます。OpenTelemetry 用 AWS Distro で X-Ray を使用するには、2 つのコンポーネントが必要です。X-Ray での使用が有効になっている OpenTelemetry SDK と、X-Ray での使用が有効になっている AWS OpenTelemetry Collector 用 Distro。

開始するには、AWS Distro for OpenTelemetry JavaScript ドキュメントを参照してください。

> **Note**
>
> ADOT JavaScript は、すべてのサーバーサイドの Node.js アプリケーションでサポートされています。ADOT JavaScript はブラウザクライアントから X-Ray にデータをエクスポートできません。

AWS Distro for OpenTelemetry を AWS X-Ray およびその他の で使用する方法の詳細については AWS のサービス、AWS 「 Distro for OpenTelemetry」またはAWS 「 Distro for OpenTelemetry ドキュメント」を参照してください。

言語サポートと使用方法の詳細については、「AWS Observability on Github」を参照してください。

AWS Node.js 用 X-Ray SDK

X-Ray SDK for Node.js は、Express ウェブアプリケーションと Node.js Lambda 関数用のライブラリです。トレースデータを作成して X-Ray デーモンに送信するためのクラスとメソッドを提供します。トレースデータには、アプリケーションによって提供される受信 HTTP リクエストに関する情報と、アプリケーションが AWS SDK または HTTP クライアントを使用してダウンストリームサービスに対して行う呼び出しが含まれます。

> **Note**
>
> X-Ray SDK for Node.js は、Node.js バージョン 14.x 以降でサポートされるオープンソースプロジェクトです。プロジェクトに従って、GitHub github.com/aws/aws-xray-sdk-node で問題とプルリクエストを送信できます。

Express を使用する場合は、アプリケーションサーバーで SDK をミドルウェアとして追加し、受信リクエストをトレースします。ミドルウェアでは、トレース対象リクエストごとに「セグメント」を作成し、レスポンスが送信されるとセグメントを完了します。セグメントが開いている間、SDK クライアントのメソッドを使用してセグメントに情報を追加し、サブセグメントを作成してダウンストリーム呼び出しをトレースできます。また、SDK では、セグメントが開いている間にアプリケーションがスローする例外を自動的に記録します。

インストルメント済みアプリケーションまたはサービスによって呼び出される Lambda 関数の場合、Lambda は トレースヘッダー を読み取り、サンプリングされたリクエストを自動的にトレースします。その他の関数については、Lambda の設定 から受信リクエストのサンプリングとトレースを行うことができます。いずれの場合も、Lambda はセグメントを作成し、X-Ray SDK に提供します。

> **Note**
>
> Lambda では、X-Ray SDK はオプションです。関数でこれを使用しない場合、サービスマップには Lambda サービスのノードと Lambda 関数ごとに 1 つのノードが含まれます。SDK を追加することで、関数コードをインストルメントして、Lambda で記録された関数セグメントにサブセグメントを追加することができます。詳細については「AWS Lambda and AWS X-Ray」を参照してください。

次に、X-Ray SDK for Node.js を使用してAWS 、SDK for JavaScript を Node.js クライアントに計測します。計測されたクライアントを使用してダウンストリーム AWS のサービス またはリソースを呼び出すたびに、SDK は呼び出しに関する情報をサブセグメントに記録します。 AWS のサービス また、サービス内でアクセスするリソースは、トレースマップにダウンストリームノードとして表示され、個々の接続のエラーやスロットリングの問題を特定するのに役立ちます。

また、X-Ray SDK for Node.js では、HTTP ウェブ API と SQL クエリに対するダウンストリーム呼び出しの計測もできます。HTTP クライアントを SDK のキャプチャメソッドでラップして、送信 HTTP 呼び出しについての情報を記録します。SQL クライアントでは、データベースタイプのキャプチャメソッドを使用します。

ミドルウェアでは、受信リクエストにサンプリングルールを適用して、トレースするリクエストを決定します。X-Ray SDK for Node.js を設定して、サンプリング動作を調整したり、アプリケーションが実行されている AWS コンピューティングリソースに関する情報を記録したりできます。

アプリケーションが注釈やメタデータで行うリクエストや作業に関する追加情報を記録します。注釈は、フィルタ式で使用するためにインデックス化されたシンプルなキーと値のペアで、特定のデータが含まれているトレースを検索できます。メタデータのエントリは制約が緩やかで、JSON にシリアル化できるオブジェクトと配列全体を記録できます。

> **注釈とメタデータ**
>
> 注釈およびメタデータとは、X-Ray SDK を使用してセグメントに追加する任意のテキストです。注釈は、フィルタ式用にインデックス付けされます。メタデータはインデックス化されませんが、X-Ray コンソールまたは API を使用して raw セグメントで表示できます。X-Ray への読み取りアクセスを許可した人は誰でも、このデータを表示できます。

コードに多数の計測されたクライアントがある場合、単一のリクエストセグメントには計測されたクライアントで行われた呼び出しごとに 1 個の多数のサブセグメントを含めることができます。カスタムサブセグメントで、クライアント呼び出しをラップすることで、サブセグメントを整理してグループできます。関数全体またはコードの任意のセクションのサブセグメントを作成し、親セグメントにすべてのレコードを記述する代わりにサブセグメントにメタデータと注釈を記録できます。

SDK のクラスとメソッドに関するリファレンスドキュメントについては、AWS X-Ray SDK for Node.js API リファレンスを参照してください。

要件

X-Ray SDK for Node.js には Node.js および次のライブラリが必要です。

- `atomic-batcher` – 1.0.2
- `cls-hooked` – 4.2.2
- `pkginfo` – 0.4.0
- `semver` – 5.3.0

NPM を使用して SDK をインストールするときに、SDK ではこれらのライブラリを引き出します。

AWS SDK クライアントをトレースするには、X-Ray SDK for Node.js には、Node.js の AWS SDK for JavaScript の最小バージョンが必要です。

- `aws-sdk` - 2.7.15

依存関係管理

The X-Ray SDK for Node.js は NPM から入手できます。

- パッケージ – `aws-xray-sdk`

ローカル開発の場合は、npm を使用してプロジェクトディレクトリに SDK をインストールします。

```
~/nodejs-xray$ npm install aws-xray-sdk
aws-xray-sdk@3.3.3
  ### aws-xray-sdk-core@3.3.3
  # ### @aws-sdk/service-error-classification@3.15.0
  # ### @aws-sdk/types@3.15.0
  # ### @types/cls-hooked@4.3.3
  # # ### @types/node@15.3.0
  # ### atomic-batcher@1.0.2
  # ### cls-hooked@4.2.2
  # # ### async-hook-jl@1.7.6
  # # # ### stack-chain@1.3.7
  # # ### emitter-listener@1.1.2
  # #   ### shimmer@1.2.1
  # ### semver@5.7.1
  ### aws-xray-sdk-express@3.3.3
  ### aws-xray-sdk-mysql@3.3.3
  ### aws-xray-sdk-postgres@3.3.3
```

`--save` オプションを使用して、SDK を依存関係としてアプリケーションの `package.json` に保存します。

```
~/nodejs-xray$ npm install aws-xray-sdk --save
aws-xray-sdk@3.3.3
```

アプリケーションに X-Ray SDK の依存関係と競合するバージョンの依存関係がある場合、互換性を確保するために両方のバージョンがインストールされます。詳細については、依存関係の解決に関する公式 NPM ドキュメンテーションを参照してください。

Node.js サンプル

AWS X-Ray SDK for Node.js を使用して、Node.js アプリケーションを通過するリクエストのend-to-endビューを取得します。

- GitHub 上の Node.js サンプルアプリケーション。

X-Ray SDK for Node.js の設定

X-Ray SDK for Node.js にプラグインを設定して、アプリケーションが実行されているサービスに関する情報が含めたり、デフォルトのサンプリング動作を変更したり、特定のパスに対するリクエストに適用されるサンプリングルールを追加したりできます。

セクション
- サービスプラグイン
- サンプリングルール
- ロギング
- X-Ray デーモンのアドレス
- 環境変数

サービスプラグイン

`plugins`を使用して、アプリケーションをホストしているサービスに関する情報を記録します。

プラグイン

- Amazon EC2 —`EC2Plugin`は、インスタンス ID、アベイラビリティーゾーン、および CloudWatch Logs グループを追加します。

- ElasticBeanstalk– `ElasticBeanstalkPlugin`は、環境名、バージョンラベル、およびデプロイ ID を追加します。
- Amazon ECS —`ECSPlugin`は、コンテナ ID を追加します。

プラグインを使用するには、`config` メソッドを使用して X-Ray SDK for Node.js クライアントを設定します。

Example app.js - プラグイン

```
var AWSXRay = require('aws-xray-sdk');
AWSXRay.config([AWSXRay.plugins.EC2Plugin,AWSXRay.plugins.ElasticBeanstalkPlugin]);
```

SDK はプラグイン設定を使用して、セグメントの`origin`フィールドを設定します。これは、アプリケーションを実行する AWS リソースのタイプを示します。複数のプラグインを使用する場合、SDK は次の解決順序を使用して起点を決定します。ElasticBeanStalk > EKS > ECS > EC2。

サンプリングルール

SDK は X-Ray コンソールで定義したサンプリングルールを使用し、記録するリクエストを決定します。デフォルトルールでは、最初のリクエストを毎秒トレースし、X-Ray にトレースを送信するすべてのサービスで追加のリクエストの 5% をトレースします。X-Ray コンソールに追加のルールを作成するをクリックして、各アプリケーションで記録されるデータ量をカスタマイズします。

SDK は、定義された順序でカスタムルールを適用します。リクエストが複数のカスタムルールと一致する場合、SDK は最初のルールのみを適用します。

> **Note**
>
> SDK が X-Ray に到達してサンプリングルールを取得できない場合、1 秒ごとに最初のリクエストのデフォルトのローカルルールに戻り、ホストあたりの追加リクエストの 5% に戻ります。これは、ホストがサンプリング API を呼び出す権限を持っていない場合や、SDK によって行われる API 呼び出しの TCP プロキシとして機能する X-Ray デーモンに接続できない場合に発生します。

JSON ドキュメントからサンプリングルールをロードするように SDK を設定することもできます。SDK は、X-Ray サンプリングが利用できない場合のバックアップとしてローカルルールを使用することも、ローカルルールを排他的に使用することもできます。

Example sampling-rules.json

```
{
  "version": 2,
  "rules": [
    {
      "description": "Player moves.",
      "host": "*",
      "http_method": "*",
      "url_path": "/api/move/*",
      "fixed_target": 0,
      "rate": 0.05
    }
  ],
  "default": {
    "fixed_target": 1,
    "rate": 0.1
  }
}
```

この例では、1 つのカスタムルールとデフォルトルールを定義します。カスタムルールでは、5 パーセントのサンプリングレートが適用され、/api/move/以下のパスに対してトレースするリクエストの最小数はありません。デフォルトのルールでは、1秒ごとの最初のリクエストおよび追加リクエストの 10 パーセントをトレースします。

ルールをローカルで定義することの欠点は、固定ターゲットが X-Ray サービスによって管理されるのではなく、レコーダーの各インスタンスによって個別に適用されることです。より多くのホストをデプロイすると、固定レートが乗算され、記録されるデータ量の制御が難しくなります。

オンの場合 AWS Lambda、サンプリングレートを変更することはできません。関数がインストルメント化されたサービスによって呼び出された場合、そのサービスによってサンプリングされたリクエストを生成した呼び出しは Lambda によって記録されます。アクティブなトレースが有効で、トレースヘッダーが存在しない場合、Lambda はサンプリングを決定します。

バックアップルールを設定するには、setSamplingRulesの ファイルからサンプリングルールをロードするようX-Ray SDK for Node.js に指示します。

Example app.js - ファイルのサンプリングルール

```
var AWSXRay = require('aws-xray-sdk');
AWSXRay.middleware.setSamplingRules('sampling-rules.json');
```

コードのルールを定義し、オブジェクトとして setSamplingRules に渡すこともできます。

Example app.js - オブジェクトのサンプリングルール

```
var AWSXRay = require('aws-xray-sdk');
var rules = {
  "rules": [ { "description": "Player moves.", "service_name": "*", "http_method": "*",
 "url_path": "/api/move/*", "fixed_target": 0, "rate": 0.05 } ],
  "default": { "fixed_target": 1, "rate": 0.1 },
  "version": 1
  }

AWSXRay.middleware.setSamplingRules(rules);
```

ローカルルールのみを使用するには、disableCentralizedSampling を呼び出します。

```
AWSXRay.middleware.disableCentralizedSampling()
```

ロギング

SDK からログ出力するには、AWSXRay.setLogger(logger) を呼び出します。logger は、標準のログ記録メソッド (warn、info、など) を提供するオブジェクトです。

デフォルトでは、SDK はコンソールオブジェクトの標準メソッドを使用してコンソールにエラーメッセージを記録します。組み込みロガーのログレベルは、AWS_XRAY_DEBUG_MODEまたはAWS_XRAY_LOG_LEVEL環境変数を使って設定できます。有効なログレベル値の一覧については、「環境変数」を参照してください。

ログに別の形式または宛先を指定したい場合は、次に示すように、SDK に独自のロガーインターフェイスの実装を提供できます。このインタフェースを実装するあらゆるオブジェクトを使用できます。つまり、Winston など、多くのロギングライブラリを使用して SDK に直接渡すことができるということです。

Example app.js - ログ記録

```
var AWSXRay = require('aws-xray-sdk');

// Create your own logger, or instantiate one using a library.
var logger = {
  error: (message, meta) => { /* logging code */ },
```

```
  warn: (message, meta) => { /* logging code */ },
  info: (message, meta) => { /* logging code */ },
  debug: (message, meta) => { /* logging code */ }
}

AWSXRay.setLogger(logger);
AWSXRay.config([AWSXRay.plugins.EC2Plugin]);
```

他の設定方法を実行する前に、setLogger を呼び出して、これらの操作から出力をキャプチャすることを確認します。

X-Ray デーモンのアドレス

X-Ray デーモンが、127.0.0.1:2000 以外のポートまたはホスト上でリッスンする場合は、X-Ray SDK for Node.js を使用して、トレースデータを別のアドレスに送信することができます。

```
AWSXRay.setDaemonAddress('host:port');
```

ホストは、名前または IPv4 アドレス で指定できます。

Example app.js - デーモンのアドレス

```
var AWSXRay = require('aws-xray-sdk');
AWSXRay.setDaemonAddress('daemonhost:8082');
```

TCP および UDP の別のポートでリッスンするようデーモンを設定する場合、両方ともデーモンアドレスの設定で指定できます。

Example app.js - 別々のポートのデーモンのアドレス

```
var AWSXRay = require('aws-xray-sdk');
AWSXRay.setDaemonAddress('tcp:daemonhost:8082 udp:daemonhost:8083');
```

または、AWS_XRAY_DAEMON_ADDRESS 環境変数を使用して、デーモンアドレスを設定することもできます。

環境変数

環境変数を使用して、X-Ray SDK for Node.js を設定できます。SDK は次の変数をサポートしています。

- AWS_XRAY_CONTEXT_MISSING – 計測されたコードが、セグメントが開いていないときにデータを記録しようとした場合に例外をスローするには、RUNTIME_ERROR に設定します。

  有効な値

  - RUNTIME_ERROR— ランタイム例外をスローします。
  - LOG_ERROR – エラーをログ記録して続行します (デフォルト)。
  - IGNORE_ERROR – エラーを無視して続行します。

  オープン状態のリクエストがない場合、または新しいスレッドを発生させるコードで、スタートアップコードに実装されたクライアントを使用しようとした場合に発生する可能性がある、セグメントまたはサブセグメントの欠落に関連するエラー。
- AWS_XRAY_DAEMON_ADDRESS – X-Ray デーモンリスナーのホストとポートを設定します。デフォルトでは、SDK はトレースデータ (UDP) とサンプリング (TCP) の両方に127.0.0.1:2000を使用します。この変数は、デーモンを次のように構成している場合に使用します。別のポートでリッスンするまたは、別のホストで実行されている場合。

  形式

  - 同じポート – *address*:*port*
  - 異なるポート – tcp:*address*:*port* udp:*address*:*port*
- AWS_XRAY_DEBUG_MODE— debugレベルでコンソールにログを出力するように SDK を設定するには、TRUEに設定します。
- AWS_XRAY_LOG_LEVEL – デフォルトロガーのログレベルを設定します。有効な値は、debug、info、warn、error、silent です。この値は、AWS_XRAY_DEBUG_MODE が TRUE に設定されている場合、無視されます。
- AWS_XRAY_TRACING_NAME – SDK がセグメントに使用するサービス名を設定します。Express ミドルウェアを設定したセグメント名を上書きします。

X-Ray SDK for Node.js を使用して受信リクエストをトレースします。

X-Ray SDK for Node.js を使用して、Express および Restify アプリケーションが Amazon EC2 の EC2 インスタンス AWS Elastic Beanstalk、または Amazon ECS で処理する受信 HTTP リクエストをトレースできます。 Amazon EC2

X-Ray SDK for Node.js は Express フレームワークおよび Restify フレームワークを使用するアプリケーションのミドルウェアを提供します。X-Ray ミドルウェアをアプリケーションに追加する

と、X-Ray SDK for Node.js によってサンプリングされた各リクエストのセグメントが作成されます。このセグメントには、時間、メソッド、HTTP リクエストの処理などが含まれます。追加の計測により、このセグメントでサブセグメントが作成されます。

> **Note**
>
> AWS Lambda 関数の場合、Lambda はサンプリングされたリクエストごとにセグメントを作成します。詳細については「AWS Lambda and AWS X-Ray」を参照してください。

各セグメントには、サービスマップ内のアプリケーションを識別する名前があります。セグメントの名前は静的に指定することも、受信リクエストのホストヘッダーに基づいて動的に名前を付けるように SDK を設定することもできます。動的ネーミングでは、リクエスト内のドメイン名に基づいてトレースをグループ化でき、名前が予想されるパターンと一致しない場合 (たとえば、ホストヘッダーが偽造されている場合) 、デフォルト名を適用できます。

> **転送されたリクエスト**
>
> ロードバランサーまたは他の仲介者がアプリケーションにリクエストを転送する場合、X-Ray は、クライアントの IP をIP パケットの送信元 IP からではなく、リクエストのX-Forwarded-Forヘッダーから取得します。転送されたリクエストについて記録されたクライアント IP は偽造される可能性があるため、信頼されるべきではありません。

リクエストが転送されると、それを示す追加フィールドが SDK によってセグメントに設定されます。セグメントのフィールド `x_forwarded_for` が `true` に設定されている場合、クライアント IP が HTTP リクエストの `X-Forwarded-For` ヘッダーから取得されます。

メッセージハンドラーは、次の情報が含まれる `http` ブロックを使用して、各受信リクエスト用にセグメントを作成します。

- HTTP メソッド – GET、POST、PUT、DELETE、その他。
- クライアントアドレス – リクエストを送信するクライアントの IP アドレス。
- レスポンスコード – 完了したリクエストの HTTP レスポンスコード。
- タイミング – 開始時間 (リクエストが受信された時間) および終了時間 (レスポンスが送信された時間)。
- ユーザーエージェント — リクエストからのuser-agent

- コンテンツの長さ — レスポンスからのcontent-length

セクション

Express を使用した受信リクエストのトレース

Express ミドルウェアを使用するには、SDK クライアントを初期化して、ルートを定義する前に express.openSegment 関数によって返されたミドルウェアを使用します。

Example app.js - Express

```
var app = express();

var AWSXRay = require('aws-xray-sdk');
app.use(AWSXRay.express.openSegment('MyApp'));

app.get('/', function (req, res) {
  res.render('index');
});

app.use(AWSXRay.express.closeSegment());
```

ルートを定義した後、express.closeSegment の出力を図のように使用して X-Ray SDK for Node.js によって返されたエラーを処理します。

Restify を使用した受信リクエストのトレース

Restify ミドルウェアを使用するには、SDK クライアントを初期化し、enable を実行します。Restify サーバーおよびセグメント名を渡します。

Example app.js - Restify

```
var AWSXRay = require('aws-xray-sdk');
var AWSXRayRestify = require('aws-xray-sdk-restify');

var restify = require('restify');
```

```
var server = restify.createServer();
AWSXRayRestify.enable(server, 'MyApp'));

server.get('/', function (req, res) {
  res.render('index');
});
```

セグメント命名ルールの設定

AWS X-Ray は、サービス名を使用してアプリケーションを識別し、アプリケーションが使用する他のアプリケーション、データベース、外部 APIs、 AWS リソースと区別します。X-Ray SDK が受信リクエストのセグメントを生成すると、アプリケーションのサービス名がセグメントの名前フィールドに記録されます。

X-Ray SDK では、HTTP リクエストヘッダーのホスト名の後にセグメントの名前を指定できます。ただし、このヘッダーは偽造され、サービスマップに予期しないノードが発生する可能性があります。偽造されたホストヘッダーを持つリクエストによって SDK がセグメントの名前を間違えないようにするには、受信リクエストのデフォルト名を指定する必要があります。

アプリケーションが複数のドメインのリクエストを処理する場合、動的ネーミングストラテジーを使用してセグメント名にこれを反映するように SDK を設定できます。動的ネーミングストラテジーにより、SDK は予想されるパターンに一致するリクエストにホスト名を使用し、そうでないリクエストにデフォルト名を適用できます。

たとえば、3 つのサブドメイン (`www.example.com`,`api.example.com`,および`static.example.com`) に対してリクエストを処理する単一のアプリケーションがあるとします。動的ネーミングストラテジーをパターン `*.example.com` で使用して、異なる名前を持つ各サブドメインのセグメントを識別することができます。結果的にはサービスマップ上に 3 つのサービスノードを作成することになります。アプリケーションがパターンと一致しないホスト名のリクエストを受信すると、指定したフォールバック名を持つ 4 番目のノードがサービスマップに表示されます。

すべてのリクエストセグメントに対して同じ名前を使用するには、前のセクションで示すとおり、ミドルウェアを初期化するときに、アプリケーションの名前を指定します。

> **Note**
>
> コードで定義したデフォルトのサービス名は、`AWS_XRAY_TRACING_NAME` 環境変数で上書きできます。

動的な命名戦略は、ホスト名と一致するようパターンを定義し、HTTP リクエストのホスト名がパターンと一致しない場合はデフォルトの名前を使用します。動的にセグメントに命名するには、AWSXRay.middleware.enableDynamicNaming を使用します。

Example app.js - 動的セグメントの名前

リクエストのホスト名がパターン *.example.com と一致する場合は、そのホスト名を使用します。それ以外の場合は、MyApp を使用します。

```
var app = express();

var AWSXRay = require('aws-xray-sdk');
app.use(AWSXRay.express.openSegment('MyApp'));
AWSXRay.middleware.enableDynamicNaming('*.example.com');

app.get('/', function (req, res) {
  res.render('index');
});

app.use(AWSXRay.express.closeSegment());
```

X-Ray AWS SDK for Node.js を使用した SDK 呼び出しのトレース

アプリケーションが AWS のサービス を呼び出してデータの保存、キューへの書き込み、または通知の送信を行うと、X-Ray SDK for Node.js はサブセグメントのダウンストリームの呼び出しを追跡します。トレースされた およびこれらのサービス内でアクセスするリソース (Amazon S3 バケットや Amazon SQS キューなど) は AWS のサービス、X-Ray コンソールのトレースマップにダウンストリームノードとして表示されます。

VAWS SDK for JavaScript V2AWS SDK for JavaScript V3 を介して作成する AWS SDK クライアントを計測します。各 AWS SDK バージョンには、 AWS SDK クライアントを計測するためのさまざまな方法が用意されています。

> **Note**
>
> 現在、 AWS X-Ray SDK for Node.js は、 AWS SDK for JavaScript V3 V2クライアントの計測時に返されるセグメント情報が少なくなります。例えば、DynamoDB への呼び出しを表すサブセグメントはテーブル名を返しません。トレースでこのセグメント情報が必要な場合は、 AWS SDK for JavaScript V2 の使用を検討してください。

AWS SDK for JavaScript V2

への呼び出しで aws-sdk require ステートメントをラップすることで、すべての AWS SDK V2 クライアントを計測できますAWSXRay.captureAWS。

Example app.js - AWS SDK 計測

```
const AWS = AWSXRay.captureAWS(require('aws-sdk'));
```

個々のクライアントを計測するには、 AWS SDK クライアントを への呼び出しでラップしますAWSXRay.captureAWSClient。たとえば、AmazonDynamoDB クライアントを計測するには:

Example app.js - DynamoDB クライアント計測

```
    const AWSXRay = require('aws-xray-sdk');
...
    const ddb = AWSXRay.captureAWSClient(new AWS.DynamoDB());
```

> **Warning**
>
> captureAWS と captureAWSClient との両方を使用しないでください。これにより、サブセグメントが重複します。

ECMAScript モジュール (ESM) で TypeScript を使用して JavaScript コードをロードする場合は、次の例を使用してライブラリをインポートします。

Example app.js - AWS SDK 計測

```
import * as AWS from 'aws-sdk';
import * as AWSXRay from 'aws-xray-sdk';
```

ESM を使用してすべての AWS クライアントを計測するには、次のコードを使用します。

Example app.js - AWS SDK 計測

```
import * as AWS from 'aws-sdk';
import * as AWSXRay from 'aws-xray-sdk';
```

```
const XRAY_AWS = AWSXRay.captureAWS(AWS);
const ddb = new XRAY_AWS.DynamoDB();
```

すべてのサービスにおいて、X-Ray コンソールでコールされた API の名前を確認できます。サービスのサブセットの場合、X-Ray SDK はセグメントに情報を追加して、サービスマップでより細かく指定します。

たとえば、実装された DynamoDB クライアントでコールすると、SDK はテーブルをターゲットとするコールのセグメントにテーブル名を追加します。コンソールで、各テーブルはサービスマップ内に個別のノードとして表示され、テーブルをターゲットにしないコール用の汎用の DynamoDB ノードが表示されます。

Example 項目を保存するための DynamoDB に対するコールのサブセグメント

```
{
  "id": "24756640c0d0978a",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "DynamoDB",
  "namespace": "aws",
  "http": {
    "response": {
      "content_length": 60,
      "status": 200
    }
  },
  "aws": {
    "table_name": "scorekeep-user",
    "operation": "UpdateItem",
    "request_id": "UBQNSO5AEM8T4FDA4RQDEB94OVTDRVV4K4HIRGVJF66Q9ASUAAJG",
  }
}
```

名前付きリソースにアクセスしたとき、次のサービスをコールすると、サービスマップに追加のノードが作成されます。特定のリソースをターゲットとしないコールでは、サービスの汎用ノードが作成されます。

- Amazon DynamoDB – テーブル名
- Amazon Simple Storage Service – バケットとキー名
- Amazon Simple Queue Service – キュー名

AWS SDK for JavaScript V3

AWS SDK for JavaScript V3 はモジュール式であるため、コードは必要なモジュールのみをロードします。このため、V3 は `captureAWS`メソッドをサポートしていないため、すべての AWS SDK クライアントを計測することはできません。

ECMAScript Modules (ESM) で TypeScript を使用して JavaScript コードをロードする場合は、次の例を使用してライブラリをインポートできます。

```
import * as AWS from 'aws-sdk';
import * as AWSXRay from 'aws-xray-sdk';
```

`AWSXRay.captureAWSv3Client` メソッドを使用して各 AWS SDK クライアントを計測します。たとえば、`AmazonDynamoDB` クライアントを計測するには:

Example app.js - JavaScript V3 用 SDK を使用した DynamoDB クライアント計測

```
    const AWSXRay = require('aws-xray-sdk');
    const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
...
    const ddb = AWSXRay.captureAWSv3Client(new DynamoDBClient({ region:
 "region" }));
```

AWS SDK for JavaScript V3 を使用する場合、テーブル名、バケット名、キー名、キュー名などのメタデータは現在返されないため、トレースマップには AWS SDK for JavaScript V2 を使用して AWS SDK クライアントを計測する場合と同様に、名前付きリソースごとに個別のノードは含まれません。

Example AWS SDK for JavaScript V3 を使用する場合に項目を保存するための DynamoDB への呼び出しのサブセグメント

```
{
  "id": "24756640c0d0978a",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "DynamoDB",
  "namespace": "aws",
  "http": {
    "response": {
      "content_length": 60,
```

```
      "status": 200
    }
  },
  "aws": {
    "operation": "UpdateItem",
    "request_id": "UBQNSO5AEM8T4FDA4RQDEB94OVTDRVV4K4HIRGVJF66Q9ASUAAJG",
  }
}
```

X-Ray SDK for Node.js を使用してダウンストリーム HTTP ウェブサービスの呼び出しをトレースする

アプリケーションがマイクロサービスまたはパブリック HTTP API に呼び出しを実行する場合に、X-Ray SDK for Node.js クライアントを使用してこれらの呼び出しを計測し、API をダウンストリームサービスとしてサービスグラフに追加できます。

`http` または `https` クライアントを X-Ray SDK for Node.js の `captureHTTPs` メソッドに渡して、送信呼び出しをトレースします。

> **Note**
>
> Axios や Superagent などのサードパーティー製の HTTP リクエストライブラリを使用する呼び出しは `captureHTTPsGlobal()` API を通じてサポートされ、ネイティブ `http` モジュールを使用する場合でもトレースされます。

Example app.js - HTTP クライアント

```
var AWSXRay = require('aws-xray-sdk');
var http = AWSXRay.captureHTTPs(require('http'));
```

すべての HTTP クライアントのトレースを有効にするには、`http` をロードする前に `captureHTTPsGlobal` を呼び出します。

Example app.js - HTTP クライアント (グローバル)

```
var AWSXRay = require('aws-xray-sdk');
AWSXRay.captureHTTPsGlobal(require('http'));
```

```
var http = require('http');
```

ダウンストリームウェブ API に対する呼び出しを計測すると、X-Ray SDK for Node.js は HTTP リクエストおよびレスポンスに関する情報を含むセグメントを記録します。X-Ray はサブセグメントを使用してリモート API の推測セグメントを生成します。

Example ダウンストリーム HTTP 呼び出しのサブセグメント

```
{
  "id": "004f72be19cddc2a",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "name": "names.example.com",
  "namespace": "remote",
  "http": {
    "request": {
      "method": "GET",
      "url": "https://names.example.com/"
    },
    "response": {
      "content_length": -1,
      "status": 200
    }
  }
}
```

Example ダウンストリーム HTTP 呼び出しの推定セグメント

```
{
  "id": "168416dc2ea97781",
  "name": "names.example.com",
  "trace_id": "1-62be1272-1b71c4274f39f122afa64eab",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "parent_id": "004f72be19cddc2a",
  "http": {
    "request": {
      "method": "GET",
      "url": "https://names.example.com/"
    },
    "response": {
      "content_length": -1,
```

```
      "status": 200
    }
  },
  "inferred": true
}
```

X-Ray SDK for Node.js を使用して SQL クエリをトレースします。

SQL クライアントを対応する X-Ray SDK for Node.js クライアントメソッドでラップすることにより、SQL データベースクエリを計測します。

- PostgreSQL – `AWSXRay.capturePostgres()`

```
var AWSXRay = require('aws-xray-sdk');
var pg = AWSXRay.capturePostgres(require('pg'));
var client = new pg.Client();
```

- MySQL – `AWSXRay.captureMySQL()`

```
var AWSXRay = require('aws-xray-sdk');
var mysql = AWSXRay.captureMySQL(require('mysql'));
...
var connection = mysql.createConnection(config);
```

計測済みクライアントを使用して SQL クエリを作成すると、&X-Ray-nodejssdk; は、サブセグメントに接続およびクエリに関する情報を記録します。

SQL サブセグメントに追加データを含める

許可リストに登録された SQL フィールドにマップされている限り、SQL クエリ用に生成されたサブセグメントに情報を追加できます。たとえば、サニタイズされた SQL クエリ文字列をサブセグメントに記録するには、サブセグメントの SQL オブジェクトに直接追加できます。

Example サブセグメントへの SQL の割り当て

```
    const queryString = 'SELECT * FROM MyTable';
connection.query(queryString, ...);

// Retrieve the most recently created subsegment
const subs = AWSXRay.getSegment().subsegments;
```

```
if (subs & & subs.length > 0) {
  var sqlSub = subs[subs.length - 1];
  sqlSub.sql.sanitized_query = queryString;
}
```

許可リストに登録されている SQL フィールドの完全な一覧については、AWS X-Ray デベロッパーガイドにある「SQL クエリ」を参照してください。

X-Ray SDK for Node.js を使用したカスタムサブセグメントの生成

サブセグメントはリクエストを処理するために行われた作業の詳細を含んだトレースのセグメントを拡張します。計測済みクライアント内で呼び出しを行うたびに、X-Ray SDK によってサブセグメントに生成された情報が記録されます。追加のサブセグメントを作成して、他のサブセグメントをグループ化したり、コードセクションのパフォーマンスを測定したり、注釈とメタデータを記録したりできます。

カスタム Express サブセグメント

ダウンストリームサービス呼び出しを行う関数用のカスタムセグメントを作成するには、`captureAsyncFunc` 関数を使用します。

Example app.js - カスタムサブセグメント Express

```
var AWSXRay = require('aws-xray-sdk');

app.use(AWSXRay.express.openSegment('MyApp'));

app.get('/', function (req, res) {
  var host = 'api.example.com';

  AWSXRay.captureAsyncFunc('send', function(subsegment) {
    sendRequest(host, function() {
      console.log('rendering!');
      res.render('index');
      subsegment.close();
    });
  });
});

app.use(AWSXRay.express.closeSegment());
```

```
function sendRequest(host, cb) {
  var options = {
    host: host,
    path: '/',
  };

  var callback = function(response) {
    var str = '';

    response.on('data', function (chunk) {
      str += chunk;
    });

    response.on('end', function () {
      cb();
    });
  }

  http.request(options, callback).end();
};
```

この例では、アプリケーションにより、sendRequest 関数への呼び出すために send という名前のカスタムサブセグメントが作成されます。captureAsyncFunc は、非同期呼び出しが完了したときにコールバック関数内で閉じる必要があるサブセグメントを渡します。

同期関数の場合は、captureFunc 関数を使用できます。これにより、関数ブロックの実行が終了するとサブセグメントが自動的に閉じられます。

セグメントまたは別のサブセグメント内にサブセグメントを作成する場合、X-Ray SDK for Node.js によってその ID が生成され、開始時刻と終了時刻が記録されます。

Example サブセグメントとメタデータ

```
"subsegments": [{
  "id": "6f1605cd8a07cb70",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "Custom subsegment for UserModel.saveUser function",
  "metadata": {
    "debug": {
      "test": "Metadata string from UserModel.saveUser"
    }
```

```
  },
```

カスタム Lambda サブセグメント

SDK は、Lambda で実行中であることを検出したときに、プレースホルダファサードセグメントを自動的に作成するように設定されています。X-Ray トレースマップ上に単一の `AWS::Lambda::Function` ノードを作成する基本的なサブセグメントを作成するには、呼び出してファサードセグメントを再利用します。新しい ID を使用して新しいセグメントを手動で作成すると (トレース ID、親 ID、サンプリングデシジョンを共有しているときに)、新しいセグメントを送信できるようになります。

Example app.js - 手動カスタムサブセグメント

```
const segment = AWSXRay.getSegment(); //returns the facade segment
const subsegment = segment.addNewSubsegment('subseg');
...
subsegment.close();
//the segment is closed by the SDK automatically
```

X-Ray SDK for Node.js を使用してセグメントに注釈とメタデータを追加する

リクエスト、環境、または注釈やメタデータを使用するアプリケーションに関する追加情報を記録できます。X-Ray SDK が作成するセグメントまたは作成するカスタムサブセグメントに、注釈およびメタデータを追加できます。

注釈は文字列、数値、またはブール値を使用したキーと値のペアです。注釈は、フィルタ式用にインデックス付けされます。注釈を使用して、コンソールでトレースをグループ化するため、または`GetTraceSummaries` API を呼び出すときに使用するデータを記録します。

メタデータは、オブジェクトとリストを含む、任意のタイプの値を持つことができるキーバリューのペアですが、フィルタ式に使用するためにインデックスは作成されません。メタデータを使用してトレースに保存する追加のデータを記録しますが、検索で使用する必要はありません。

注釈とメタデータに加えて、セグメントにユーザー ID 文字列を記録することもできます。ユーザー ID はセグメントの個別のフィールドに記録され、検索用にインデックスが作成されます。

セクション

- X-Ray SDK for Node.js を使用して注釈を記録

- X-Ray SDK for Node.js を使用したメタデータの記録
- X-Ray SDK for Node.js を使用したユーザー ID の記録

X-Ray SDK for Node.js を使用して注釈を記録

注釈を使用して、検索用にインデックスを作成するセグメントまたはサブセグメントに情報を記録します。

注釈の要件

- キー – X-Ray 注釈のキーには最大 500 文字の英数字を使用できます。スペースまたはドットやピリオド (.) 以外の記号は使用できません。
- 値 – X-Ray 注釈の値には最大 1,000 の Unicode 文字を使用できます。
- 注釈の数 – トレースごとに最大 50 の注釈を使用できます。

注釈を記録するには

1. 現在のセグメントまたはサブセグメントの参照を取得します。

```
var AWSXRay = require('aws-xray-sdk');
...
var document = AWSXRay.getSegment();
```

2. 文字列キー、および、ブール値、数値、文字列値を使用して addAnnotation を呼び出します。

```
document.addAnnotation("mykey", "my value");
```

次の例は、ドット、ブール値、数値、または文字列値を含む文字列キーを使用して putAnnotation を呼び出す方法を示しています。

```
document.putAnnotation("testkey.test", "my value");
```

SDK は、セグメントドキュメントの annotations オブジェクトにキーと値のペアとして、注釈を記録します。同じキーで addAnnotation を 2 回呼び出すと、同じセグメントまたはサブセグメントに以前記録された値が上書きされます。

特定の値を持つ注釈のあるトレースを見つけるには、annotation[*key*]フィルタ式の キーワードを使用します。

Example app.js - 注釈

```
var AWS = require('aws-sdk');
var AWSXRay = require('aws-xray-sdk');
var ddb = AWSXRay.captureAWSClient(new AWS.DynamoDB());
...
app.post('/signup', function(req, res) {
    var item = {
        'email': {'S': req.body.email},
        'name': {'S': req.body.name},
        'preview': {'S': req.body.previewAccess},
        'theme': {'S': req.body.theme}
    };

    var seg = AWSXRay.getSegment();
    seg.addAnnotation('theme', req.body.theme);

    ddb.putItem({
      'TableName': ddbTable,
      'Item': item,
      'Expected': { email: { Exists: false } }
  }, function(err, data) {
...
```

X-Ray SDK for Node.js を使用したメタデータの記録

メタデータを使用して、検索用にインデックスを作成する必要のないセグメントまたはサブセグメントに情報を記録します。メタデータ値は、文字列、数値、ブール値、または JSON オブジェクトや JSON 配列にシリアル化できるその他の任意のオブジェクトになります。

メタデータを記録するには

1. 現在のセグメントまたはサブセグメントの参照を取得します。

   ```
   var AWSXRay = require('aws-xray-sdk');
   ...
   var document = AWSXRay.getSegment();
   ```

2. 文字列キー、ブール値、数値、文字列値、オブジェクト値、文字列名前空間を使用して addMetadata を呼び出します。

```
document.addMetadata("my key", "my value", "my namespace");
```

or

キーと値だけを使用して addMetadata を呼び出します。

```
document.addMetadata("my key", "my value");
```

名前空間を指定しない場合、SDK は default を使用します。同じキーで addMetadata を 2 回呼び出すと、同じセグメントまたはサブセグメントに以前記録された値が上書きされます。

X-Ray SDK for Node.js を使用したユーザー ID の記録

リクエストセグメントにユーザー ID を記録して、リクエストを送信したユーザーを識別します。Lambda 環境のセグメントはイミュータブルであるため、このオペレーションは AWS Lambda 関数と互換性がありません。setUser の呼び出しはセグメントにのみ適用でき、サブセグメントには適用できません。

ユーザー ID を記録するには

1. 現在のセグメントまたはサブセグメントの参照を取得します。

```
var AWSXRay = require('aws-xray-sdk');
...
var document = AWSXRay.getSegment();
```

2. リクエストを送信したユーザーの文字列 ID を使用して setUser() を呼び出します。

```
var user = 'john123';

AWSXRay.getSegment().setUser(user);
```

setUser を呼び出し、Express アプリケーションがリクエストの処理を開始するとすぐに、ユーザー ID を記録できます。ユーザー ID を設定するためだけにセグメントを使用する場合、呼び出しを 1 行で連鎖させることができます。

Example app.js - ユーザー ID

```
var AWS = require('aws-sdk');
var AWSXRay = require('aws-xray-sdk');
var uuidv4 = require('uuid/v4');
var ddb = AWSXRay.captureAWSClient(new AWS.DynamoDB());
...
    app.post('/signup', function(req, res) {
    var userId = uuidv4();
    var item = {
        'userId': {'S': userId},
        'email': {'S': req.body.email},
        'name': {'S': req.body.name}
    };

    var seg = AWSXRay.getSegment().setUser(userId);

    ddb.putItem({
      'TableName': ddbTable,
      'Item': item,
      'Expected': { email: { Exists: false } }
  }, function(err, data) {
...
```

ユーザー ID のトレースを見つけるには、userフィルタ式で、 キーワードを使用します。

Python の使用

X-Ray にトレースを送信する Python アプリケーションを計測するには、次の 2 つの方法があります。

- AWS Distro for OpenTelemetry Python – AWS Distro for OpenTelemetry Collector を介して、相関メトリクスとトレースを Amazon CloudWatch、 AWS X-Ray Amazon OpenSearch Service などの複数の AWS モニタリングソリューションに送信するための一連のオープンソースライブラリを提供する AWS ディストリビューション。
- AWS X-Ray SDK for Python – X-Ray デーモンを介してトレースを生成して X-Ray に送信するためのライブラリのセット。

詳細については、「Distro for OpenTelemetry AWS と X-Ray SDKs の選択」を参照してください。

AWS Distro for OpenTelemetry Python

AWS Distro for OpenTelemetry (ADOT) Python を使用すると、アプリケーションを一度計測し、相関メトリクスとトレースを Amazon CloudWatch AWS X-Ray、Amazon OpenSearch Service などの複数の AWS モニタリングソリューションに送信できます。X-Ray を ADOT で使用するには、X-Ray で使用できる OpenTelemetry SDK と、X-Ray で使用できる AWS Distro for OpenTelemetry Collector の 2 つのコンポーネントが必要です。ADOT Python には自動計測のサポートが含まれており、アプリケーションでコードを変更せずにトレースを送信できます。

開始するには、AWS Distro for OpenTelemetry Java ドキュメントを参照してください。

AWS Distro for OpenTelemetry を AWS X-Ray およびその他の で使用する方法の詳細については AWS のサービス、AWS 「 Distro for OpenTelemetry」またはAWS 「 Distro for OpenTelemetry ドキュメント」を参照してください。

言語サポートと使用方法の詳細については、「AWS Observability on Github」を参照してください。

AWS X-Ray SDK for Python

X-Ray SDK for Python は、Python ウェブアプリケーション用のライブラリです。トレースデータを作成し X-Ray デーモンに送信するためのクラスおよびメソッドを提供します。トレースデータには、アプリケーションによって提供される受信 HTTP リクエストに関する情報、および AWS SDK、HTTP クライアント、または SQL データベースコネクタを使用してアプリケーションがダウ

ンストリームサービスに対して行う呼び出しが含まれます。セグメントを手動で作成し、注釈およびメタデータにデバッグ情報を追加することもできます。

SDK は `pip` を使用してダウンロードできます。

```
$ pip install aws-xray-sdk
```

> **Note**
>
> X-Ray SDK for Python は、オープンソースプロジェクトです。プロジェクトに従って、GitHub github.com/aws/aws-xray-sdk-python で問題とプルリクエストを送信できます。

Django または Flask を使用する場合は、アプリケーションに SDK ミドルウェアを追加し、受信リクエストをトレースします。ミドルウェアでは、トレース対象リクエストごとに「セグメント」を作成し、レスポンスが送信されるとセグメントを完了します。セグメントが開いている間、SDK クライアントのメソッドを使用してセグメントに情報を追加し、サブセグメントを作成してダウンストリーム呼び出しをトレースできます。また、SDK では、セグメントが開いている間にアプリケーションがスローする例外を自動的に記録します。他のアプリケーションの場合、手動でセグメントを作成することができます。

測定されたアプリケーションまたはサービスから呼び出された Lambda 関数に対して、Lambda はトレースヘッダー を読み込み、サンプリングされたリクエストを自動的にトレースします。その他の関数については、Lambda の設定 から受信リクエストのサンプリングとトレースを行うことができます。いずれの場合も、Lambda はセグメントを作成し、X-Ray SDK に提供します。

> **Note**
>
> Lambda では、X-Ray SDK はオプションです。関数でこれを使用しない場合、サービスマップには Lambda サービスのノードと Lambda 関数ごとに 1 つのノードが含まれます。SDK を追加することで、関数コードをインストルメントして、Lambda で記録された関数セグメントにサブセグメントを追加することができます。詳細については「AWS Lambda and AWS X-Ray」を参照してください。

Lambda で実行されているサンプル Python 関数については、「ワーカー」を参照してください。

次に、X-Ray SDK for Python を使用してダウンストリーム呼び出しを実装するには、アプリケーションが使用するライブラリにパッチを適用します。SDK は次のライブラリをサポートしています。

サポートされているライブラリ

- botocore、 boto3 – 計測 AWS SDK for Python (Boto) クライアント。
- pynamodb – 測定された Amazon DynamoDB クライアントの PynamoDB のバージョン。
- aiobotocore、aioboto3 - 測定された asyncio統合バージョンの SDK for Python クライアント。
- requests、aiohttp - 測定された高レベルの HTTP クライアント。
- httplib、 http.client - 測定された低レベルの HTTP クライアントおよびそれらを使用する高レベルのライブラリ。
- sqlite3 - SQLite クライアントを測定します。
- mysql-connector-python - MySQL クライアントを測定します。
- pg8000 - Pure-Python PostgreSQL インターフェイスを測定します。
- psycopg2 - PostgreSQL データベースアダプターを測定します。
- pymongo - MongoDB クライアントを測定します。
- pymysql - MySQL と MariaDB の測定された PyMySQL ベースのクライアント。

アプリケーションが AWS、SQL データベース、またはその他の HTTP サービスを呼び出すたびに、SDK は呼び出しに関する情報をサブセグメントに記録します。 AWS のサービス また、サービス内でアクセスするリソースは、トレースマップにダウンストリームノードとして表示され、個々の接続のエラーやスロットリングの問題を識別しやすくなります。

SDK を入手したら、レコーダーとミドルウェアを設定して、その動作をカスタマイズします。プラグインを追加して、アプリケーションを実行しているコンピューティングリソースに関するデータを記録したり、サンプリングルールを定義することでサンプリングの動作をカスタマイズしたり、アプリケーションログに SDK からの情報をより多くあるいは少なく表示するように口グレベルを設定できます。

アプリケーションが注釈やメタデータで行うリクエストや作業に関する追加情報を記録します。注釈は、フィルタ式で使用するためにインデックス化されたシンプルなキーと値のペアで、特定のデータが含まれているトレースを検索できます。メタデータのエントリは制約が緩やかで、JSON にシリアル化できるオブジェクトと配列全体を記録できます。

> **注釈とメタデータ**
>
> 注釈およびメタデータとは、X-Ray SDK を使用してセグメントに追加する任意のテキストです。注釈は、フィルタ式用にインデックス付けされます。メタデータはインデックス化され

ませんが、X-Ray コンソールまたは API を使用して raw セグメントで表示できます。X-Ray への読み取りアクセスを許可した人は誰でも、このデータを表示できます。

コードに多数の計測されたクライアントがある場合、単一のリクエストセグメントには計測されたクライアントで行われた呼び出しごとに 1 個の多数のサブセグメントを含めることができます。カスタムサブセグメントで、クライアント呼び出しをラップすることで、サブセグメントを整理してグループできます。関数全体またはコードの任意のセクションに対して、カスタムサブセグメントを作成できます。親セグメントのすべてを書き込むのではなく、サブセグメントにメタデータと注釈を記録することができます。

SDK のクラスとメソッドのリファレンスドキュメントについては、AWS X-Ray SDK for Python API リファレンスを参照してください。

要件

X-Ray SDK for Python では、次の言語とライブラリのバージョンがサポートされています。

- Python – 2.7、3.4 以降
- Django – 1.10 以降
- Flask – 0.10 以降
- aiohttp – 2.3.0 以降
- AWS SDK for Python (Boto) - 1.4.0 以降
- botocore – 1.5.0 以降
- enum – 0.4.7 以降 (Python バージョン 3.4.0 以前)
- jsonpickle – 1.0.0 以降
- setuptools – 40.6.3 以降
- wrapt – 1.11.0 以降

依存関係管理

X-Ray SDK for Python は、`pip`から入手できます。

- パッケージ – `aws-xray-sdk`

1. SDK を依存関係として `requirements.txt` ファイルに追加します。

Example requirements.txt

```
aws-xray-sdk==2.4.2
boto3==1.4.4
botocore==1.5.55
Django==1.11.3
```

Elastic Beanstalk を使用してアプリケーションをデプロイする場合、Elastic Beanstalk は `requirements.txt` のパッケージをすべて自動的にインストールします。

X-Ray SDK for Python の設定

X-Ray SDK for Python には、グローバルレコーダーを提供する `xray_recorder` というクラスがあります。グローバルレコーダーを設定して、受信 HTTP コールのセグメントを作成するミドルウェアをカスタマイズできます。

セクション

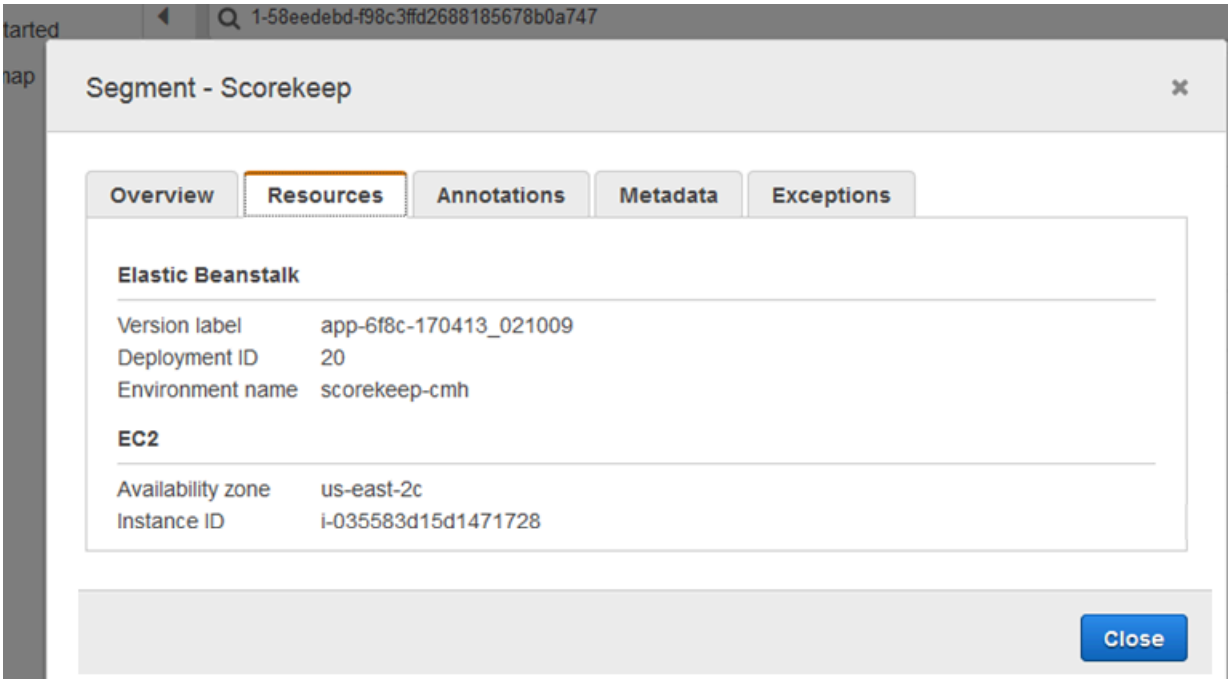
- サービスプラグイン
- サンプリングルール
- ロギング
- コード内のレコーダー設定
- Django でのレコーダー設定
- 環境変数

サービスプラグイン

`plugins`を使用して、アプリケーションをホストしているサービスに関する情報を記録します。

プラグイン

- Amazon EC2 —`EC2Plugin`は、インスタンス ID、アベイラビリティーゾーン、および CloudWatch Logs グループを追加します。
- ElasticBeanstalk– `ElasticBeanstalkPlugin`は、環境名、バージョンラベル、およびデプロイ ID を追加します。
- Amazon ECS —`ECSPlugin`は、コンテナ ID を追加します。



プラグインを使用するには、configure で xray_recorder を呼び出します。1 はタプルとして渡されるため、単一のプラグインを指定する場合は必ず末尾に 2 を付けてください。

```
from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch_all

xray_recorder.configure(service='My app')
plugins = ('ElasticBeanstalkPlugin', 'EC2Plugin')
xray_recorder.configure(plugins=plugins)
patch_all()
```

> **Note**
>
> plugins はタプルとして渡されるため、単一のプラグインを指定する場合は必ず末尾に , を付けてください。例えば、plugins = ('EC2Plugin',)

また、コードで設定した値よりも優先される環境変数を使用して、レコーダーを設定することもできます。

ダウンストリームコールを記録するには、パッチライブラリの前にプラグインを設定します。

また、SDK はプラグインの設定を利用して、セグメントの origin フィールドを設定します。これは、アプリケーションを実行する AWS リソースのタイプを示します。複数のプラグインを使用する場合、SDK は次の解決順序を使用して起点を決定します。ElasticBeanStalk > EKS > ECS > EC2。

サンプリングルール

SDK は X-Ray コンソールで定義したサンプリングルールを使用し、記録するリクエストを決定します。デフォルトルールでは、最初のリクエストを毎秒トレースし、X-Ray にトレースを送信するすべてのサービスで追加のリクエストの 5% をトレースします。X-Ray コンソールに追加のルールを作成するをクリックして、各アプリケーションで記録されるデータ量をカスタマイズします。

SDK は、定義された順序でカスタムルールを適用します。リクエストが複数のカスタムルールと一致する場合、SDK は最初のルールのみを適用します。

> **Note**
>
> SDK が X-Ray に到達してサンプリングルールを取得できない場合、1 秒ごとに最初のリクエストのデフォルトのローカルルールに戻り、ホストあたりの追加リクエストの 5% に戻ります。これは、ホストがサンプリング API を呼び出す権限を持っていない場合や、SDK によって行われる API 呼び出しの TCP プロキシとして機能する X-Ray デーモンに接続できない場合に発生します。

JSON ドキュメントからサンプリングルールをロードするように SDK を設定することもできます。SDK は、X-Ray サンプリングが利用できない場合のバックアップとしてローカルルールを使用することも、ローカルルールを排他的に使用することもできます。

Example sampling-rules.json

```
{
  "version": 2,
  "rules": [
    {
      "description": "Player moves.",
      "host": "*",
      "http_method": "*",
      "url_path": "/api/move/*",
      "fixed_target": 0,
      "rate": 0.05
    }
```

```
  ],
  "default": {
    "fixed_target": 1,
    "rate": 0.1
  }
}
```

この例では、1 つのカスタムルールとデフォルトルールを定義します。カスタムルールでは、5 パーセントのサンプリングレートが適用され、/api/move/以下のパスに対してトレースするリクエストの最小数はありません。デフォルトのルールでは、1秒ごとの最初のリクエストおよび追加リクエストの 10 パーセントをトレースします。

ルールをローカルで定義することの欠点は、固定ターゲットが X-Ray サービスによって管理されるのではなく、レコーダーの各インスタンスによって個別に適用されることです。より多くのホストをデプロイすると、固定レートが乗算され、記録されるデータ量の制御が難しくなります。

オンの場合 AWS Lambda、サンプリングレートを変更することはできません。関数がインストルメント化されたサービスによって呼び出された場合、そのサービスによってサンプリングされたリクエストを生成した呼び出しは Lambda によって記録されます。アクティブなトレースが有効で、トレースヘッダーが存在しない場合、Lambda はサンプリングを決定します。

バックアップサンプリングルールを設定するには、次の例に示すように、`xray_recorder.configure` を呼び出します。*`rules`* は、ルールの辞書または JSON ファイルの絶対パスサンプリングルールです。

```
xray_recorder.configure(sampling_rules=rules)
```

ローカルルールのみを使用するには、`LocalSampler` でレコーダーを設定します。

```
from aws_xray_sdk.core.sampling.local.sampler import LocalSampler
xray_recorder.configure(sampler=LocalSampler())
```

サンプリングを無効にしてすべての着信リクエストを実装するように、グローバルレコーダーを設定することもできます。

Example main.py - サンプリングを無効にする

```
xray_recorder.configure(sampling=False)
```

ロギング

SDK は、Python の組み込み `logging` モジュールを使用し、`WARNING`デフォルトのログ記録レベルを使用します。`aws_xray_sdk` クラスのロガーへの参照を取得し、その上で `setLevel` を呼び出して、ライブラリとアプリケーションに別のログレベルを設定します。

Example app.py - ログ記録

```
logging.basicConfig(level='WARNING')
logging.getLogger('aws_xray_sdk').setLevel(logging.ERROR)
```

デバッグログを使用して問題を識別します。たとえば、「サブセグメントを手動で生成する」場合にサブセグメントが閉じない問題などです。

コード内のレコーダー設定

追加の設定は、`xray_recorder` の `configure` メソッドから利用できます。

- `context_missing` - 測定されたコードが、セグメントが開いていないときにデータを記録しようとした場合に例外のスローを回避するには、`LOG_ERROR` に設定します。
- `daemon_address` – X-Ray デーモンリスナーのホストとポートを設定します。
- `service` - SDK がセグメントに使用するサービス名を設定します。
- `plugins` – アプリケーションの AWS リソースに関する情報を記録します。
- `sampling` – `False` に設定してサンプリングを無効にします。
- `sampling_rules` – サンプリングルールを含む JSON ファイルのパスを設定します。

Example main.py - コンテキスト欠落例外を無効にする

```
from aws_xray_sdk.core import xray_recorder

xray_recorder.configure(context_missing='LOG_ERROR')
```

Django でのレコーダー設定

Django フレームワークを使用している場合は、Django `settings.py` ファイルを使用してグローバルレコーダーのオプションを設定できます。

- AUTO_INSTRUMENT (Django のみ) - 組み込みデータベースおよびテンプレートレンダリング操作のサブセグメントを記録します。
- AWS_XRAY_CONTEXT_MISSING - LOG_ERROR に設定すると、セグメントを開いていないときに測定されたコードがデータを記録しようとしたときに例外が発生するのを防ぐことができます。
- AWS_XRAY_DAEMON_ADDRESS – X-Ray デーモンリスナーのホストとポートを設定します。
- AWS_XRAY_TRACING_NAME - SDK がセグメントに使用するサービス名を設定します。
- PLUGINS – アプリケーションの AWS リソースに関する情報を記録します。
- SAMPLING – False に設定してサンプリングを無効にします。
- SAMPLING_RULES – サンプリングルールを含む JSON ファイルのパスを設定します。

settings.py でレコーダ設定を有効にするには、インストールされているアプリのリストに Django ミドルウェアを追加します。

Example settings.py - インストールされたアプリ

```
INSTALLED_APPS = [
    ...
    'django.contrib.sessions',
    'aws_xray_sdk.ext.django',
]
```

XRAY_RECORDER という名前の dict で利用可能な設定を行います。

Example settings.py - インストールされたアプリ

```
XRAY_RECORDER = {
    'AUTO_INSTRUMENT': True,
    'AWS_XRAY_CONTEXT_MISSING': 'LOG_ERROR',
    'AWS_XRAY_DAEMON_ADDRESS': '127.0.0.1:5000',
    'AWS_XRAY_TRACING_NAME': 'My application',
    'PLUGINS': ('ElasticBeanstalkPlugin', 'EC2Plugin', 'ECSPlugin'),
    'SAMPLING': False,
}
```

環境変数

環境変数を使用して、X-Ray SDK for Python を設定できます。SDK は次の変数をサポートしています。

- AWS_XRAY_TRACING_NAME - SDK がセグメントに使用するサービス名を設定します。プログラムによって設定したサービス名を上書きします。
- AWS_XRAY_SDK_ENABLED – falseに設定されている場合、SDK は無効になります。デフォルトでは、環境変数が false に設定されている場合を除き、SDK は有効です。
  - 無効にすると、グローバルレコーダーによって、デーモンに送信されないダミーセグメントとサブセグメントが自動的に生成され、自動パッチ適用は無効になります。ミドルウェアはグローバルレコーダーのラッパーとして記述されています。ミドルウェアによるセグメントやサブセグメントの生成もすべて、ダミーセグメントおよびダミーサブセグメントになります。
  - AWS_XRAY_SDK_ENABLED の値を設定するには、環境変数を使用するか、aws_xray_sdk ライブラリの global_sdk_config オブジェクトと直接やり取りします。環境変数を設定すると、これらの通信は上書きされます。
- AWS_XRAY_DAEMON_ADDRESS – X-Ray デーモンリスナーのホストとポートを設定します。デフォルトでは、SDK はトレースデータ (UDP) とサンプリング (TCP) の両方に127.0.0.1:2000を使用します。この変数は、デーモンを次のように構成している場合に使用します。別のポートでリッスンするまたは、別のホストで実行されている場合。

  形式

  - 同じポート – *address*:*port*
  - 異なるポート – tcp:*address*:*port* udp:*address*:*port*
- AWS_XRAY_CONTEXT_MISSING – 計測されたコードが、セグメントが開いていないときにデータを記録しようとした場合に例外をスローするには、RUNTIME_ERROR に設定します。

  有効な値

  - RUNTIME_ERROR— ランタイム例外をスローします。
  - LOG_ERROR – エラーをログ記録して続行します (デフォルト)。
  - IGNORE_ERROR – エラーを無視して続行します。

  リクエストが開かれていないときに実行されるスタートアップコード、または新しいスレッドを生成するコードで測定されたクライアントを使用しようとしたときに発生する可能性があるセグメントまたはサブセグメントの欠落に関連するエラー。

環境変数は、コードで設定される値を上書きします。

X-Ray SDK for Python ミドルウェアを使用して受信リクエストをトレースします。

ミドルウェアをアプリケーションに追加してセグメント名を設定すると、X-Ray SDK for Python はサンプリングされた要求ごとにセグメントを作成します。このセグメントには、時間、メソッド、HTTP リクエストの処理などが含まれます。追加の計測により、このセグメントでサブセグメントが作成されます。

X-Ray SDK for Python は、受信 HTTP リクエストを測定する次のミドルウェアをサポートしています。

- Django
- Flask
- Bottle

> **Note**
>
> AWS Lambda 関数の場合、Lambda はサンプリングされたリクエストごとにセグメントを作成します。詳細については「AWS Lambda and AWS X-Ray」を参照してください。

Lambda で測定されているサンプル Python 関数については、「ワーカー」を参照してください。

他のフレームワークのスクリプトまたは Python アプリケーションの場合、セグメントを手動で作成することができます。

各セグメントには、サービスマップ内のアプリケーションを識別する名前があります。セグメントの名前は静的に指定することも、受信リクエストのホストヘッダーに基づいて動的に名前を付けるように SDK を設定することもできます。動的ネーミングでは、リクエスト内のドメイン名に基づいてトレースをグループ化でき、名前が予想されるパターンと一致しない場合 (たとえば、ホストヘッダーが偽造されている場合) 、デフォルト名を適用できます。

> **転送されたリクエスト**
>
> ロードバランサーまたは他の仲介者がアプリケーションにリクエストを転送する場合、X-Ray は、クライアントの IP をIP パケットの送信元 IP からではなく、リクエストのX-

Forwarded-Forヘッダーから取得します。転送されたリクエストについて記録されたクライアント IP は偽造される可能性があるため、信頼されるべきではありません。

リクエストが転送されると、それを示す追加フィールドが SDK によってセグメントに設定されます。セグメントのフィールド x_forwarded_for が true に設定されている場合、クライアント IP が HTTP リクエストの X-Forwarded-For ヘッダーから取得されます。

ミドルウェアは、次の情報が含まれる http ブロックを使用して、各受信リクエスト用にセグメントを作成します。

- HTTP メソッド – GET、POST、PUT、DELETE、その他。
- クライアントアドレス – リクエストを送信するクライアントの IP アドレス。
- レスポンスコード – 完了したリクエストの HTTP レスポンスコード。
- タイミング – 開始時間 (リクエストが受信された時間) および終了時間 (レスポンスが送信された時間)。
- ユーザーエージェント — リクエストからのuser-agent
- コンテンツの長さ — レスポンスからのcontent-length

セクション

アプリケーションにミドルウェアを追加する (Django)

ミドルウェアを MIDDLEWARE ファイルの settings.py リストに追加します。X-Ray ミドルウェアは、他のミドルウェアで失敗した要求が確実に記録されるように、settings.py ファイルの最初の行にする必要があります。

Example settings.py - X-Ray SDK for Python ミドルウェア

```
MIDDLEWARE = [
    'aws_xray_sdk.ext.django.middleware.XRayMiddleware',
```

```
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware'
]
```

settings.py ファイルの INSTALLED_APPS リストに X-Ray SDK Django アプリを追加します。これにより、アプリの起動時にX-Ray レコーダーを設定できるようになります。

Example settings.py - X-Ray SDK for Python Django アプリ

```
INSTALLED_APPS = [
    'aws_xray_sdk.ext.django',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

使用する settings.py ファイルにセグメント名を設定します。

Example settings.py - セグメント名

```
XRAY_RECORDER = {
    'AWS_XRAY_TRACING_NAME': 'My application',
    'PLUGINS': ('EC2Plugin',),
}
```

これは、X-Ray レコーダーに、デフォルトのサンプリングレートで Django アプリケーションが提供する要求をトレースするように指示します。使用する Django 設定ファイルにレコーダーを設定して、カスタムサンプリングルールを適用するか、その他の設定を変更することができます。

> **Note**
>
> plugins はタプルとして渡されるため、単一のプラグインを指定する場合は必ず末尾に , を付けてください。例えば、plugins = ('EC2Plugin',)

アプリケーションにミドルウェアを追加する (Flask)

Flask アプリケーションを計測するには、最初に `xray_recorder` にセグメント名を設定します。次に、`XRayMiddleware` 関数を使用して Flask アプリケーションをコードにパッチします。

Example app.py

```
from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.ext.flask.middleware import XRayMiddleware

app = Flask(__name__)

xray_recorder.configure(service='My application')
XRayMiddleware(app, xray_recorder)
```

これは、X-Ray レコーダーに、デフォルトのサンプリングレートで Flask アプリケーションが提供する要求をトレースするように指示します。レコーダーをコードに設定して、カスタムサンプリングルールを適用するか、その他の設定を変更することができます。

アプリケーションにミドルウェアを追加する (Bottle)

Bottle アプリケーションを計測するには、最初に `xray_recorder` にセグメント名を設定します。次に、`XRayMiddleware` 関数を使用して Bottle アプリケーションをコードにパッチします。

Example app.py

```
from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.ext.bottle.middleware import XRayMiddleware

app = Bottle()

xray_recorder.configure(service='fallback_name', dynamic_naming='My application')
app.install(XRayMiddleware(xray_recorder))
```

これは、X-Ray レコーダーに、デフォルトのサンプリングレートで Bottle アプリケーションが提供する要求をトレースするように指示します。レコーダーをコードに設定して、カスタムサンプリングルールを適用するか、その他の設定を変更することができます。

Python コードを手動で実装する

Django または Flask を使用していない場合、手動でセグメントを作成することができます。受信リクエストごとにセグメントを作成するか、パッチが適用された HTTP または AWS SDK クライアン

トの周囲にセグメントを作成して、レコーダーがサブセグメントを追加するためのコンテキストを提供できます。

Example main.py - 手動測定

```
from aws_xray_sdk.core import xray_recorder

# Start a segment
segment = xray_recorder.begin_segment('segment_name')
# Start a subsegment
subsegment = xray_recorder.begin_subsegment('subsegment_name')

# Add metadata and annotations
segment.put_metadata('key', dict, 'namespace')
subsegment.put_annotation('key', 'value')

# Close the subsegment and segment
xray_recorder.end_subsegment()
xray_recorder.end_segment()
```

セグメント命名ルールの設定

AWS X-Ray は、サービス名を使用してアプリケーションを識別し、アプリケーションが使用する他のアプリケーション、データベース、外部 APIs、 AWS リソースと区別します。X-Ray SDK が受信リクエストのセグメントを生成すると、アプリケーションのサービス名がセグメントの名前フィールドに記録されます。

X-Ray SDK では、HTTP リクエストヘッダーのホスト名の後にセグメントの名前を指定できます。ただし、このヘッダーは偽造され、サービスマップに予期しないノードが発生する可能性があります。偽造されたホストヘッダーを持つリクエストによって SDK がセグメントの名前を間違えないようにするには、受信リクエストのデフォルト名を指定する必要があります。

アプリケーションが複数のドメインのリクエストを処理する場合、動的ネーミングストラテジーを使用してセグメント名にこれを反映するように SDK を設定できます。動的ネーミングストラテジーにより、SDK は予想されるパターンに一致するリクエストにホスト名を使用し、そうでないリクエストにデフォルト名を適用できます。

たとえば、3 つのサブドメイン (`www.example.com`,`api.example.com`,および`static.example.com`) に対してリクエストを処理する単一のアプリケーションがあるとします。動的ネーミングストラテジーをパターン `*.example.com` で使用して、異なる名前を持つ各サブドメインのセグメントを識別することができます。結果的にはサービスマップ上に 3 つのサービ

スノードを作成することになります。アプリケーションがパターンと一致しないホスト名のリクエストを受信すると、指定したフォールバック名を持つ 4 番目のノードがサービスマップに表示されます。

すべてのリクエストセグメントに同じ名前を使用するには、前のセクションで示されたように、レコーダーを設定するときにアプリケーションの名前を指定します。

動的命名ルールは、ホスト名と一致するようパターンを定義し、HTTP リクエストのホスト名がパターンと一致しない場合はデフォルトの名前を使用します。Django でセグメントに動的な名前を付けるには、DYNAMIC_NAMING 設定を使用する settings.py ファイルに追加します。

Example settings.py - 動的ネーミング

```
XRAY_RECORDER = {
    'AUTO_INSTRUMENT': True,
    'AWS_XRAY_TRACING_NAME': 'My application',
    'DYNAMIC_NAMING': '*.example.com',
    'PLUGINS': ('ElasticBeanstalkPlugin', 'EC2Plugin')
}
```

パターン内で任意の文字列に一致させるには「*」を、また、任意の 1 文字に一致させるには「?」を使用することができます。Flask の場合は、コードでレコーダーを設定します。

Example main.py - セグメント名

```
from aws_xray_sdk.core import xray_recorder
xray_recorder.configure(service='My application')
xray_recorder.configure(dynamic_naming='*.example.com')
```

> **Note**
>
> コードで定義したデフォルトのサービス名は、AWS_XRAY_TRACING_NAME 環境変数で上書きできます。

ダウンストリームコールを実装するためのライブラリへのパッチ適用

ダウンストリーム呼び出しを測定するには、X-Ray SDK for Python を使用して、アプリケーションが使用するライブラリにパッチを適用します。X-Ray SDK for Python では、以下のライブラリにパッチを適用できます。

サポートされているライブラリ

- botocore、 boto3 – 計測 AWS SDK for Python (Boto) クライアント。
- pynamodb – 測定された Amazon DynamoDB クライアントの PynamoDB のバージョン。
- aiobotocore、aioboto3 - 測定された asyncio統合バージョンの SDK for Python クライアント。
- requests、aiohttp - 測定された高レベルの HTTP クライアント。
- httplib、 http.client - 測定された低レベルの HTTP クライアントおよびそれらを使用する高レベルのライブラリ。
- sqlite3 - SQLite クライアントを測定します。
- mysql-connector-python - MySQL クライアントを測定します。
- pg8000 - Pure-Python PostgreSQL インターフェイスを測定します。
- psycopg2 - PostgreSQL データベースアダプターを測定します。
- pymongo - MongoDB クライアントを測定します。
- pymysql - MySQL と MariaDB 用 PyMySQL ベースクライアントを測定します。

パッチ適用されたライブラリを使用すると、X-Ray SDK for Python は呼び出しのサブセグメントが作成され、リクエストとレスポンスの情報を記録します。SDK ミドルウェアまたは AWS Lambdaのいずれかから、サブセグメントを作成するためにSDK でセグメントを使用できる必要があります。

> **Note**
>
> SQLAlchemy ORM を使用する場合は、SQLAlchemy のセッションクラスとクエリクラスの SDK のバージョンをインポートして、SQL クエリを追加できます。手順については、Use SQLAlchemy ORM を参照してください。

使用可能なすべてのライブラリにパッチを適用するには、`aws_xray_sdk.core` の `patch_all` 関数を使用します。`httplib` や `urllib` などの一部のライブラリでは、`patch_all(double_patch=True)` を呼び出して二重パッチ適用を有効にすることが必要な場合があります。

Example main.py - サポートされているすべてのライブラリにパッチを適用

```
import boto3
import botocore
```

```
import requests
import sqlite3

from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch_all

patch_all()
```

単一のライブラリにパッチを適用するには、ライブラリ名のタプルを使用して patch を呼び出します。これを行うには、単一の要素リストを用意する必要があります。

Example main.py - 特定のライブラリにパッチを適用

```
import boto3
import botocore
import requests
import mysql-connector-python

from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch

libraries = ['botocore']
patch(libraries)
```

> **Note**
>
> 場合によっては、ライブラリにパッチを適用するために使用するキーがライブラリ名と一致しない場合があります。一部のキーは、1 つまたは複数のライブラリのエイリアスとして機能します。
>
> ライブラリエイリアス
>
> - `httplib` – `httplib` および `http.client`
> - `mysql` – `mysql-connector-python`

非同期作業のコンテキストのトレース

`asyncio` によって統合されたライブラリの場合や、非同期関数用のサブセグメントを作成する場合は、非同期コンテキストで X-Ray SDK for Python も設定する必要があります。`AsyncContext` クラスをインポートし、そのインスタンスを X-Ray レコーダーに渡します。

> **Note**
>
> ウェブフレームワークサポートライブラリ (例: AIOHTTP) は、`aws_xray_sdk.core.patcher` モジュールで処理することはできません。これらのライブラリは、サポートされているライブラリの `patcher` カタログに表示されません。

Example main.py - aioboto3 にパッチを適用

```
import asyncio
import aioboto3
import requests

from aws_xray_sdk.core.async_context import AsyncContext
from aws_xray_sdk.core import xray_recorder
xray_recorder.configure(service='my_service', context=AsyncContext())
from aws_xray_sdk.core import patch

libraries = ['aioboto3']
patch(libraries)
```

X-Ray AWS SDK for Python を使用した SDK 呼び出しのトレース

アプリケーションが AWS のサービス を呼び出してデータの保存、キューへの書き込み、または通知の送信を行うと、X-Ray SDK for Python はサブセグメントのダウンストリームの呼び出しを追跡します。これらのサービス (Amazon S3 バケットや Amazon SQS キューなど) 内でアクセスするトレースされた AWS のサービス および リソースは、X-Ray コンソールのトレースマップにダウンストリームノードとして表示されます。

X-Ray SDK for Python は、ライブラリにパッチを適用すると、すべての AWS SDK クライアントを自動的に計測します。 `botocore`個々のクライアントを実装することはできません。

すべてのサービスにおいて、X-Ray コンソールでコールされた API の名前を確認できます。サービスのサブセットの場合、X-Ray SDK はセグメントに情報を追加して、サービスマップでより細かく指定します。

たとえば、実装された DynamoDB クライアントでコールすると、SDK はテーブルをターゲットとするコールのセグメントにテーブル名を追加します。コンソールで、各テーブルはサービスマップ内に個別のノードとして表示され、テーブルをターゲットにしないコール用の汎用の DynamoDB ノードが表示されます。

Example 項目を保存するための DynamoDB に対するコールのサブセグメント

```
{
  "id": "24756640c0d0978a",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "DynamoDB",
  "namespace": "aws",
  "http": {
    "response": {
      "content_length": 60,
      "status": 200
    }
  },
  "aws": {
    "table_name": "scorekeep-user",
    "operation": "UpdateItem",
    "request_id": "UBQNSO5AEM8T4FDA4RQDEB94OVTDRVV4K4HIRGVJF66Q9ASUAAJG",
  }
}
```

名前付きリソースにアクセスしたとき、次のサービスをコールすると、サービスマップに追加のノードが作成されます。特定のリソースをターゲットとしないコールでは、サービスの汎用ノードが作成されます。

- Amazon DynamoDB – テーブル名
- Amazon Simple Storage Service – バケットとキー名
- Amazon Simple Queue Service – キュー名

X-Ray SDK for Python を使用してダウンストリーム HTTP ウェブサービスの呼び出しをトレースする

アプリケーションがマイクロサービスまたはパブリック HTTP API に呼び出しを実行する場合に、X-Ray SDK for Python を使用してこれらの呼び出しを計測し、API をダウンストリームサービスとしてサービスグラフに追加できます。

HTTP クライアントを設定するには、送信呼び出しに使用するライブラリにパッチを適用します。`requests` や Python で実装されている HTTP クライアントを使っている場合は、必要な作業はこれだけです。`aiohttp` の場合は、非同期コンテキストを使用してレコーダーも設定します。

aiohttp 3 のクライアント API を使用する場合は、SDK で提供されるトレース設定のインスタンスを使用して、ClientSession を設定する必要もあります。

Example **aiohttp** 3 クライアント API

```
from aws_xray_sdk.ext.aiohttp.client import aws_xray_trace_config

async def foo():
    trace_config = aws_xray_trace_config()
    async with ClientSession(loop=loop, trace_configs=[trace_config]) as session:
        async with session.get(url) as resp
            await resp.read()
```

ダウンストリームウェブ API に対する呼び出しを計測すると、X-Ray SDK for Python は HTTP リクエストおよびレスポンスに関する情報を含むセグメントを記録します。X-Ray はサブセグメントを使用してリモート API の推測セグメントを生成します。

Example ダウンストリーム HTTP 呼び出しのサブセグメント

```
{
  "id": "004f72be19cddc2a",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "name": "names.example.com",
  "namespace": "remote",
  "http": {
    "request": {
      "method": "GET",
      "url": "https://names.example.com/"
    },
    "response": {
      "content_length": -1,
      "status": 200
    }
  }
}
```

Example ダウンストリーム HTTP 呼び出しの推定セグメント

```
{
  "id": "168416dc2ea97781",
  "name": "names.example.com",
```

```
  "trace_id": "1-62be1272-1b71c4274f39f122afa64eab",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "parent_id": "004f72be19cddc2a",
  "http": {
    "request": {
      "method": "GET",
      "url": "https://names.example.com/"
    },
    "response": {
      "content_length": -1,
      "status": 200
    }
  },
  "inferred": true
}
```

X-Ray SDK for Python を使用したカスタムサブセグメントの生成

サブセグメントは、トレースの セグメント をリクストに対応するために行われた作業の詳細で拡張します。計測済みクライアント内で呼び出しを行うたびに、X-Ray SDK によってサブセグメントに生成された情報が記録されます。追加のサブセグメントを作成して、他のサブセグメントをグループ化したり、コードセクションのパフォーマンスを測定したり、注釈とメタデータを記録したりできます。

サブセグメントを管理するには、begin_subsegment および end_subsegment メソッドを使用します。

Example main.py - カスタムサブセグメント

```
from aws_xray_sdk.core import xray_recorder

subsegment = xray_recorder.begin_subsegment('annotations')
subsegment.put_annotation('id', 12345)
xray_recorder.end_subsegment()
```

同期関数のサブセグメントを作成するには、@xray_recorder.capture デコレータを使用します。サブセグメントの名前をキャプチャ関数に渡すことも、関数名を使用することもできます。

Example main.py - 関数サブセグメント

```
from aws_xray_sdk.core import xray_recorder
```

```
@xray_recorder.capture('## create_user')
def create_user():
...
```

非同期関数の場合、@xray_recorder.capture_async デコレータを使用し、非同期コンテキストをレコーダーに渡します。

Example main.py - 非同期関数サブセグメント

```
from aws_xray_sdk.core.async_context import AsyncContext
from aws_xray_sdk.core import xray_recorder
xray_recorder.configure(service='my_service', context=AsyncContext())

@xray_recorder.capture_async('## create_user')
async def create_user():
    ...

async def main():
    await myfunc()
```

セグメントまたは別のサブセグメント内にサブセグメントを作成する場合、X-Ray SDK for Python によってその ID が生成され、開始時刻と終了時刻が記録されます。

Example サブセグメントとメタデータ

```
"subsegments": [{
  "id": "6f1605cd8a07cb70",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "Custom subsegment for UserModel.saveUser function",
  "metadata": {
    "debug": {
      "test": "Metadata string from UserModel.saveUser"
    }
  },
```

X-Ray SDK for Python を使用してセグメントに注釈とメタデータを追加する

注釈やメタデータにより、リクエスト、環境、またはアプリケーションに関する追加情報を記録することができます。X-Ray SDK が作成するセグメントまたは作成するカスタムサブセグメントに、注釈およびメタデータを追加できます。

注釈は文字列、数値、またはブール値を使用したキーと値のペアです。注釈は、フィルタ式用にインデックス付けされます。注釈を使用して、コンソールでトレースをグループ化するため、または`GetTraceSummaries` API を呼び出すときに使用するデータを記録します。

メタデータは、オブジェクトとリストを含む、任意のタイプの値を持つことができるキーバリューのペアですが、フィルタ式に使用するためにインデックスは作成されません。メタデータを使用してトレースに保存する追加のデータを記録しますが、検索で使用する必要はありません。

注釈とメタデータに加えて、セグメントにユーザー ID 文字列を記録することもできます。ユーザー ID はセグメントの個別のフィールドに記録され、検索用にインデックスが作成されます。

セクション

- X-Ray SDK for Python で注釈を記録する
- X-Ray SDK for Python でメタデータを記録する
- X-Ray SDK for Python でユーザー ID を記録する

X-Ray SDK for Python で注釈を記録する

注釈を使用して、検索用にインデックスを作成するセグメントまたはサブセグメントに情報を記録します。

注釈の要件

- キー – X-Ray 注釈のキーには最大 500 文字の英数字を使用できます。スペースまたはドットやピリオド (.) 以外の記号は使用できません。
- 値 – X-Ray 注釈の値には最大 1,000 の Unicode 文字を使用できます。
- 注釈の数 – トレースごとに最大 50 の注釈を使用できます。

注釈を記録するには

1. `xray_recorder` から現在のセグメントまたはサブセグメントの参照を取得します。

```
from aws_xray_sdk.core import xray_recorder
...
document = xray_recorder.current_segment()
```

or

```
from aws_xray_sdk.core import xray_recorder
...
document = xray_recorder.current_subsegment()
```

2. 文字列キー、および、ブール値、数値、文字列値を使用して put_annotation を呼び出します。

```
document.put_annotation("mykey", "my value");
```

次の例は、ドット、ブール値、数値、または文字列値を含む文字列キーを使用して putAnnotation を呼び出す方法を示しています。

```
document.putAnnotation("testkey.test", "my value");
```

または、put_annotation メソッドを xray_recorder で使用できます。このメソッドは、現在のサブセグメントの注釈を記録します。サブセグメントが開いていない場合は、セグメントの注釈を記録します。

```
xray_recorder.put_annotation("mykey", "my value");
```

SDK は、セグメントドキュメントの annotations オブジェクトにキーと値のペアとして、注釈を記録します。同じキーで put_annotation を 2 回呼び出すと、同じセグメントまたはサブセグメントに以前記録された値が上書きされます。

特定の値を持つ注釈のあるトレースを見つけるには、annotation[*key*]フィルタ式の キーワードを使用します。

X-Ray SDK for Python でメタデータを記録する

> **Warning**
>
> X-Ray SDK for Python では、循環参照を持つオブジェクトをメタデータ値として追加しないでください。これらのオブジェクトは JSON にシリアル化できず、SDK で無限ループを作成する可能性があります。また、パフォーマンスの問題を防ぐために、大規模で複雑なオブジェクトをメタデータとして追加することは避けてください。

メタデータを使用して、検索用にインデックスを作成する必要のないセグメントまたはサブセグメントに情報を記録します。メタデータ値は、文字列、数値、ブール値、または JSON オブジェクトや JSON 配列にシリアル化できる任意のオブジェクトになります。

メタデータを記録するには

1. `xray_recorder` から現在のセグメントまたはサブセグメントの参照を取得します。

   ```
   from aws_xray_sdk.core import xray_recorder
   ...
   document = xray_recorder.current_segment()
   ```

   or

   ```
   from aws_xray_sdk.core import xray_recorder
   ...
   document = xray_recorder.current_subsegment()
   ```

2. 文字列キー、ブール値、数値、文字列値、オブジェクト値、文字列名前空間を使用して `put_metadata` を呼び出します。

   ```
   document.put_metadata("my key", "my value", "my namespace");
   ```

   or

   キーと値だけを使用して `put_metadata` を呼び出します。

   ```
   document.put_metadata("my key", "my value");
   ```

または、put_metadata メソッドを xray_recorder で使用できます。このメソッドは、現在のサブセグメントのメタデータを記録します。サブセグメントが開いていない場合は、セグメントのメタデータを記録します。

```
xray_recorder.put_metadata("my key", "my value");
```

名前空間を指定しない場合、SDK は default を使用します。同じキーで put_metadata を 2 回呼び出すと、同じセグメントまたはサブセグメントに以前記録された値が上書きされます。

X-Ray SDK for Python でユーザー ID を記録する

リクエストセグメントにユーザー ID を記録して、リクエストを送信したユーザーを識別します。

ユーザー ID を記録するには

1. xray_recorder から現在のセグメントへの参照を取得します。

   ```
   from aws_xray_sdk.core import xray_recorder
   ...
   document = xray_recorder.current_segment()
   ```

2. リクエストを送信したユーザーの文字列 ID を使用して setUser を呼び出します。

   ```
   document.set_user("U12345");
   ```

コントローラーで set_user を呼び出し、アプリケーションがリクエストの処理を開始するとすぐに、ユーザー ID を記録できます。

ユーザー ID のトレースを見つけるには、userフィルタ式で、 キーワードを使用します。

サーバーレス環境にデプロイされたウェブフレームワークの計測

AWS X-Ray SDK for Python は、サーバーレスアプリケーションにデプロイされたウェブフレームワークの計測をサポートしています。サーバーレスはクラウドのネイティブアーキテクチャで、運用上の多くの責任を AWSにシフトさせることができるため、俊敏性とイノベーションを強化できます。

サーバーレスアーキテクチャは、サーバーを意識せずにアプリケーションやサービスを構築および実行できるソフトウェアアプリケーションモデルです。サーバーまたはクラスターのプロビジョニング、パッチ適用、オペレーティングシステムのメンテナンス、キャパシティのプロビジョニング

といったインフラストラクチャ管理のタスクが不要になります。サーバーレスアプリケーションは、ほぼすべてのタイプのアプリケーションやバックエンドサービス向けに構築でき、高可用性を実現しながら、アプリケーションの実行やスケーリングに必要な作業のすべてをユーザーに代わって行います。

このチュートリアルでは、サーバーレス環境にデプロイされた Flask や Django などのウェブフレームワーク AWS X-Ray で を自動的に計測する方法について説明します。アプリケーションの X-Ray 計測により、 AWS Lambda Amazon API Gateway から関数を介して行われたすべてのダウンストリーム呼び出しと、アプリケーションが行う発信呼び出しを表示できます。

X-Ray SDK for Python では、次の Python アプリケーションフレームワークをサポートしています。

- Flask バージョン 0.8 以降
- Django バージョン 1.0 以降

このチュートリアルでは、Lambda にデプロイされ、API Gateway から呼び出されるサーバーレスアプリケーションのサンプルを作成します。このチュートリアルでは、Zappa を使用して、アプリケーションを Lambda に自動的にデプロイし、API Gateway のエンドポイントを設定します。

前提条件

- Zappa
- Python – バージョン 2.7 または 3.6。
- AWS CLI – AWS CLI が、アプリケーションをデプロイする アカウントと AWS リージョン で設定されていることを確認します。
- Pip
- Virtualenv

ステップ 1: 環境を作成する

このステップでは、`virtualenv` を使用して仮想環境を作成し、アプリケーションをホスティングします。

1. を使用して AWS CLI、アプリケーションのディレクトリを作成します。その新しいディレクトリに変更します。

```
mkdir serverless_application
```

```
cd serverless_application
```

2. 次に、新しいディレクトリ内に仮想環境を作成します。アクティベートするには以下のコマンドを使用します。

```
# Create our virtual environment
virtualenv serverless_env

# Activate it
source serverless_env/bin/activate
```

3. X-Ray、Flask、Zappa およびリクエストライブラリをその環境にインストールします。

```
# Install X-Ray, Flask, Zappa, and Requests into your environment
pip install aws-xray-sdk flask zappa requests
```

4. アプリケーションコードを serverless_application ディレクトリに追加します。この例では、Flasks の Hello World の例を基にします。

   serverless_application ディレクトリに my_app.py という名前のファイルを作成します。テキストエディタで、次のコマンドを追加します。このアプリケーションでは、リクエストライブラリを計測し、Flask アプリケーションのミドルウェアにパッチを適用して、エンドポイント '/' を開きます。

```
# Import the X-Ray modules
from aws_xray_sdk.ext.flask.middleware import XRayMiddleware
from aws_xray_sdk.core import patcher, xray_recorder
from flask import Flask
import requests

# Patch the requests module to enable automatic instrumentation
patcher.patch(('requests',))

app = Flask(__name__)

# Configure the X-Ray recorder to generate segments with our service name
xray_recorder.configure(service='My First Serverless App')

# Instrument the Flask application
XRayMiddleware(app, xray_recorder)

@app.route('/')
```

```
def hello_world():
    resp = requests.get("https://aws.amazon.com")
    return 'Hello, World: %s' % resp.url
```

ステップ 2: Zappa 環境の作成とデプロイ

このステップでは、Zappa を使用して API Gateway のエンドポイントを自動的に設定し、Lambda にデプロイします。

1. serverless_application ディレクトリ内から Zappa を初期化します。この例では、デフォルト設定を使用しましたが、カスタマイズ設定がある場合は Zappa に設定手順が表示されます。

```
zappa init
```

```
What do you want to call this environment (default 'dev'): dev
...
What do you want to call your bucket? (default 'zappa-*******'): zappa-*******
...
...
It looks like this is a Flask application.
What's the modular path to your app's function?
This will likely be something like 'your_module.app'.
We discovered: my_app.app
Where is your app's function? (default 'my_app.app'): my_app.app
...
Would you like to deploy this application globally? (default 'n') [y/n/
(p)rimary]: n
```

2. X-Ray を有効にします。zappa_settings.json ファイルを開き、例のように表示されていることを確認します。

```
{
    "dev": {
        "app_function": "my_app.app",
        "aws_region": "us-west-2",
        "profile_name": "default",
        "project_name": "serverless-exam",
        "runtime": "python2.7",
        "s3_bucket": "zappa-*********"
```

```
    }
}
```

3. 設定ファイルのエントリとして "xray_tracing": true を追加します。

```
{
    "dev": {
        "app_function": "my_app.app",
        "aws_region": "us-west-2",
        "profile_name": "default",
        "project_name": "serverless-exam",
        "runtime": "python2.7",
        "s3_bucket": "zappa-*********",
        "xray_tracing": true
    }
}
```

4. アプリケーションをデプロイします。これにより、API Gateway エンドポイントが設定され、コードは Lambda に更新されます。

```
zappa deploy
```

```
...
Deploying API Gateway..
Deployment complete!: https://**********.execute-api.us-west-2.amazonaws.com/dev
```

ステップ 3: API Gateway 用 X-Ray トレースを有効にする

このステップでは、API Gateway コンソールを使用して、X-Ray トレースを有効にします。

1. にサインイン AWS Management Console し、https://console.aws.amazon.com/apigateway/:// www.com」で API Gateway コンソールを開きます。
2. 新しく生成された API を探します。serverless-exam-dev のようになります。
3. [Stages] (ステージ) を選択します。
4. API のデプロイステージの名前を選択します。デフォルト: dev。
5. [Logs/Tracing] タブで、[X-Ray トレースを有効にする] チェックボックスをオンにします。
6. [Save Changes] を選択します。

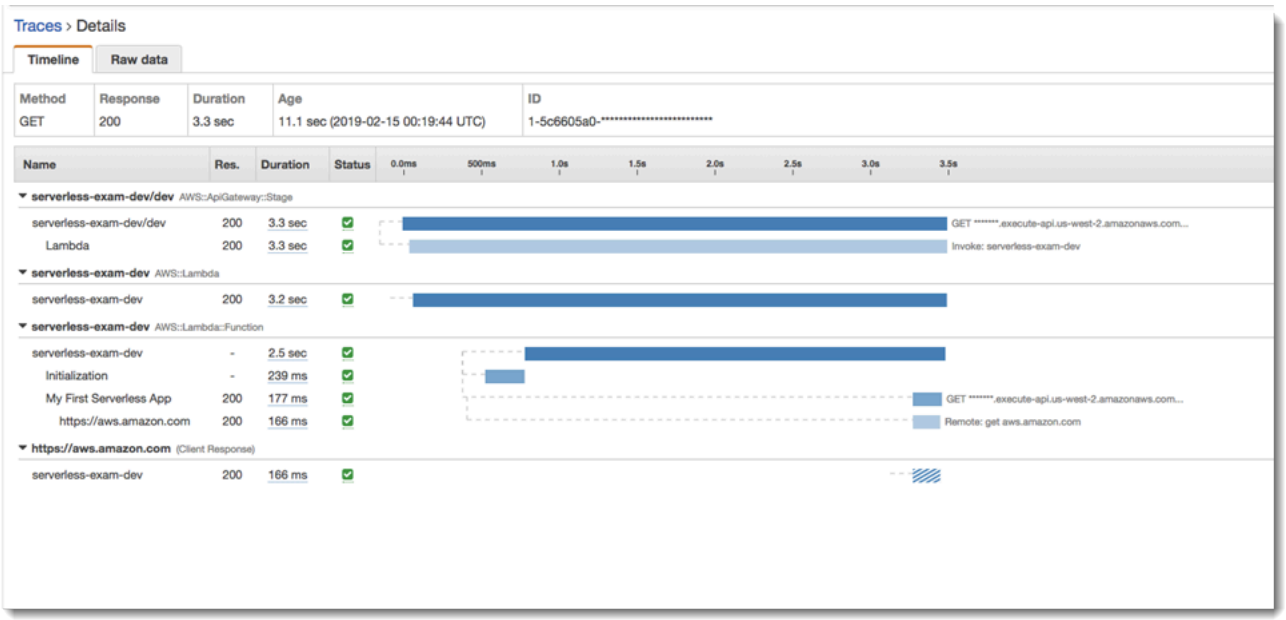
7. ブラウザでエンドポイントにアクセスします。サンプルの `Hello World` アプリケーションを使用した場合は、次のように表示されます。

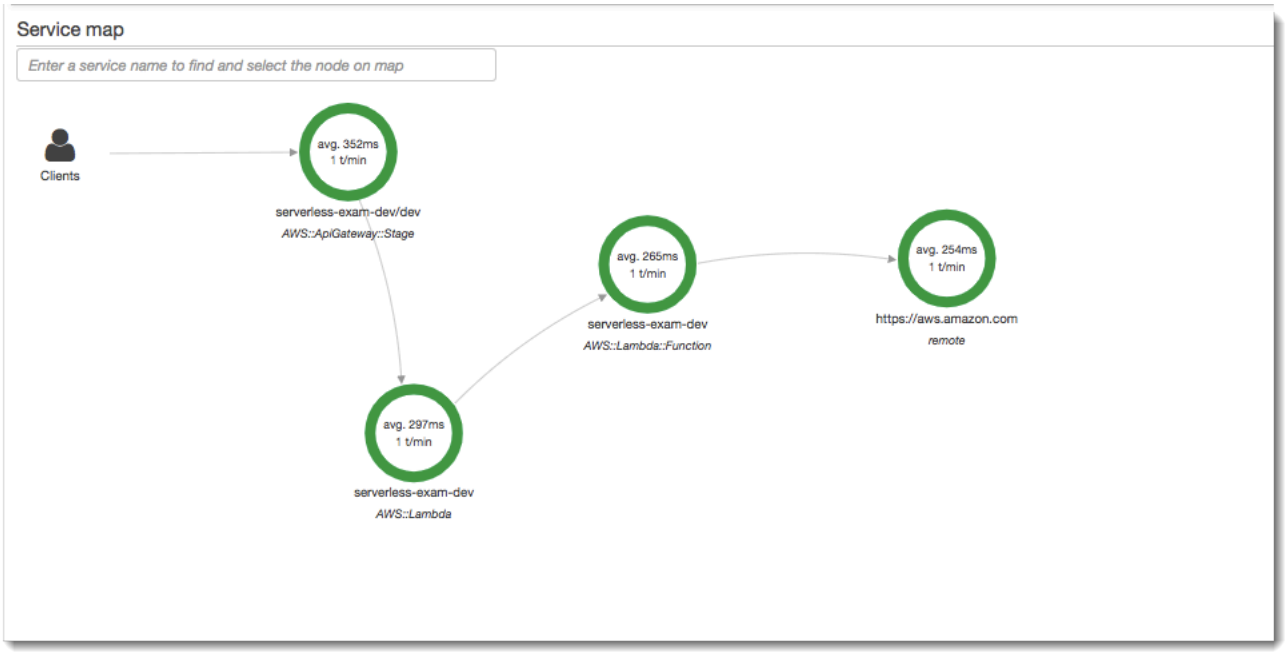
```
"Hello, World: https://aws.amazon.com/"
```

ステップ 4: 作成したトレースを表示する

このステップでは、X-Ray コンソールを使用して、サンプルアプリケーションで作成されたトレースを表示します。トレース解析の詳細なチュートリアルについては、「サービスマップの表示」を参照してください。

1. にサインイン AWS Management Console し、https://console.aws.amazon.com/xray/home:// www.com」で X-Ray コンソールを開きます。
2. API Gateway、Lambda 関数、Lambda コンテナによって生成されたセグメントを表示します。
3. Lambda 関数のセグメントで、`My First Serverless App` という名前のサブセグメントを表示します。次に、`https://aws.amazon.com` という名前の 2 番目のサグメントが続きます。
4. Lambda では、初期化中に `initialization` という名前の 3 番目のサブセグメントが生成されることがあります。





ステップ 5：クリーンアップ

不要なコストがかからないように、使用しなくなったリソースは必ず終了してください。このチュートリアルで示されているように、Zappa のようなツールを使用することで、サーバーレスの再デプロイを効率化することができます。

Lambda、API Gateway、Amazon S3 からアプリケーションを削除するには、 AWS CLIを使用して、プロジェクトディレクトリで次のコマンドを実行します。

```
zappa undeploy dev
```

次のステップ

AWS クライアントを追加し、X-Ray で計測することで、アプリケーションに機能を追加します。サーバーレスコンピューティングのオプションについては、AWSのサーバーレスを参照してください。

.NET の使用

X-Ray にトレースを送信する .NET アプリケーションを計測するには、次の 2 つの方法があります。

- AWS Distro for OpenTelemetry .NET – AWS Distro for OpenTelemetry Collector を介して、相関メトリクスとトレースを Amazon CloudWatch AWS X-Ray、Amazon OpenSearch Service などの複数の AWS モニタリングソリューションに送信するための一連のオープンソースライブラリを提供する AWS ディストリビューション。
- AWS X-Ray SDK for .NET – X-Ray デーモンを介してトレースを生成して X-Ray に送信するためのライブラリのセット。

詳細については、「Distro for OpenTelemetry AWS と X-Ray SDKs の選択」を参照してください。

AWS Distro for OpenTelemetry .NET

AWS Distro for OpenTelemetry .NET を使用すると、アプリケーションを一度計測し、相関メトリクスとトレースを Amazon CloudWatch AWS X-Ray、Amazon OpenSearch Service などの複数の AWS モニタリングソリューションに送信できます。OpenTelemetry 用 AWS Distro で X-Ray を使用するには、2 つのコンポーネントが必要です。X-Ray での使用が有効になっている OpenTelemetry SDK と、X-Ray での使用が有効になっている AWS OpenTelemetry Collector 用 Distro。

開始するには、「」を参照してください。AWS Distro for OpenTelemetry .NET のドキュメント。

AWS Distro for OpenTelemetry を AWS X-Ray およびその他の で使用する方法の詳細については AWS のサービス、AWS 「 Distro for OpenTelemetry」またはAWS 「 Distro for OpenTelemetry ドキュメント」を参照してください。

言語サポートと使用方法の詳細については、「AWS Observability on Github」を参照してください。

AWS X-Ray SDK for .NET

X-Ray SDK for .NET は、C# .NET ウェブアプリケーション、.NET Core ウェブアプリケーション、.NET Core 関数を計測するためのライブラリです AWS Lambda。トレースデータを生成して X-Ray デーモンに送信するためのクラスとメソッドを提供します。これには、アプリケーションによって提供される受信リクエスト、およびアプリケーションがダウンストリーム AWS のサービス、HTTP ウェブ APIs、SQL データベースに対して行う呼び出しに関する情報が含まれます。

> **Note**
>
> X-Ray SDK for .NET は、オープンソースプロジェクトです。プロジェクトに従って、GitHub: github.com/aws/aws-xray-sdk-dotnet で問題とプルリクエストを送信できます。

ウェブアプリケーションの場合は、最初にメッセージハンドラーをウェブ設定に追加して、受信リクエストをトレースします。メッセージハンドラーでは、トレース対象リクエストごとに「セグメント」を作成し、レスポンスが送信されるとセグメントを完了します。セグメントが開いている間、SDK クライアントのメソッドを使用してセグメントに情報を追加し、サブセグメントを作成してダウンストリーム呼び出しをトレースできます。また、SDK では、セグメントが開いている間にアプリケーションがスローする例外を自動的に記録します。

インストルメント済みアプリケーションまたはサービスによって呼び出される Lambda 関数の場合、Lambda は トレースヘッダー を読み取り、サンプリングされたリクエストを自動的にトレースします。その他の関数については、Lambda の設定 から受信リクエストのサンプリングとトレースを行うことができます。いずれの場合も、Lambda はセグメントを作成し、X-Ray SDK に提供します。

> **Note**
>
> Lambda では、X-Ray SDK はオプションです。関数でこれを使用しない場合、サービスマップには Lambda サービスのノードと Lambda 関数ごとに 1 つのノードが含まれます。SDK を追加することで、関数コードをインストルメントして、Lambda で記録された関数セグメントにサブセグメントを追加することができます。詳細については「AWS Lambda and AWS X-Ray」を参照してください。

次に、X-Ray SDK for .NET を使用して次の操作を行います。を計測します AWS SDK for .NET クライアント。計測されたクライアントを使用してダウンストリーム AWS のサービス またはリソースを呼び出すたびに、SDK はサブセグメントの呼び出しに関する情報を記録します。 AWS サービスおよびサービス内でアクセスするリソースは、トレースマップにダウンストリームノードとして表示され、個々の接続のエラーとスロットリングの問題を識別しやすくなります。

X-Ray SDK for .NET は、また、HTTP ウェブ API および SQL データベースに対するダウンストリーム呼び出しの計測もできます。`System.Net.HttpWebRequest` の `GetResponseTraced` 拡張メソッドは送信 HTTP 呼び出しをトレースします。X-Ray SDK for .NET の `SqlCommand` バージョンを使用して SQL クエリを計測します。

SDK の活用を開始したら、レコーダーとメッセージハンドラーを設定して、その動作をカスタマイズします。プラグインを追加して、アプリケーションを実行しているコンピューティングリソースに関するデータを記録したり、サンプリングルールを定義することでサンプリングの動作をカスタマイズしたり、アプリケーションログに SDK からの情報をより多くあるいは少なく表示するようにログレベルを設定できます。

アプリケーションが注釈やメタデータで行うリクエストや作業に関する追加情報を記録します。注釈は、フィルタ式で使用するためにインデックス化されたシンプルなキーと値のペアで、特定のデータが含まれているトレースを検索できます。メタデータのエントリは制約が緩やかで、JSON にシリアル化できるオブジェクトと配列全体を記録できます。

> **注釈とメタデータ**
>
> 注釈およびメタデータとは、X-Ray SDK を使用してセグメントに追加する任意のテキストです。注釈は、フィルタ式用にインデックス付けされます。メタデータはインデックス化されませんが、X-Ray コンソールまたは API を使用して raw セグメントで表示できます。X-Ray への読み取りアクセスを許可した人は誰でも、このデータを表示できます。

コードに多数の計測されたクライアントがある場合、単一のリクエストセグメントには計測されたクライアントで行われた呼び出しごとに 1 個の多数のサブセグメントを含めることができます。カスタムサブセグメントで、クライアント呼び出しをラップすることで、サブセグメントを整理してグループできます。関数全体またはコードの任意のセクションのサブセグメントを作成し、親セグメントにすべてのレコードを記述する代わりにサブセグメントにメタデータと注釈を記録できます。

SDK のクラスとメソッドに関するリファレンスドキュメントについては、以下を参照してください。

- AWS X-Ray SDK for .NET API リファレンス
- AWS X-Ray SDK for .NET Core API リファレンス

同じパッケージが .NET および .NET Core の両方をサポートしますが、使用されるクラスは異なります。この章の例は、そのクラスが .NET Core に固有でない限り、.NET API リファレンスにリンクされています。

要件

X-Ray SDK for .NET には、.NET Framework 4.5 以降および が必要です AWS SDK for .NET。

.NET Core アプリケーションと関数の場合、SDK では .NET Core 2.0 以降が必要になります。

.NET X-Ray SDK をアプリケーションに追加する

NuGet を使用して、X-Ray SDK for .NET をアプリケーションに追加します。

NuGet パッケージマネージャーで X-Ray SDK for .NET を Visual Studio にインストールするには

1. [ツール]、[NuGet パッケージマネージャー]、[Manage NuGet Packages for Solution (ソリューションに対する NuGet パッケージの管理)] を選択します。
2. の検索AWSXrayRecorder。
3. パッケージを選択し、[Install] を選択します。

依存関係管理

X-Ray SDK for .NETNuGet。SDK の パッケージマネージャーを使ったインストール

```
Install-Package AWSXRayRecorder -Version 2.10.1
```

-AWSXRayRecorder v2.10.1NuGet パッケージには次の依存関係があります。

.NET フレームワーク 4.5

```
AWSXRayRecorder (2.10.1)
|
|-- AWSXRayRecorder.Core (>= 2.10.1)
|   |-- AWSSDK.Core (>= 3.3.25.1)
|
|-- AWSXRayRecorder.Handlers.AspNet (>= 2.7.3)
|   |-- AWSXRayRecorder.Core (>= 2.10.1)
|
|-- AWSXRayRecorder.Handlers.AwsSdk (>= 2.8.3)
|   |-- AWSXRayRecorder.Core (>= 2.10.1)
|
|-- AWSXRayRecorder.Handlers.EntityFramework (>= 1.1.1)
|   |-- AWSXRayRecorder.Core (>= 2.10.1)
|   |-- EntityFramework (>= 6.2.0)
|
```

```
|-- AWSXRayRecorder.Handlers.SqlServer (>= 2.7.3)
|   |-- AWSXRayRecorder.Core (>= 2.10.1)
|
|-- AWSXRayRecorder.Handlers.System.Net (>= 2.7.3)
    |-- AWSXRayRecorder.Core (>= 2.10.1)
```

.NET フレームワーク 2.0

```
AWSXRayRecorder (2.10.1)
|
|-- AWSXRayRecorder.Core (>= 2.10.1)
|   |-- AWSSDK.Core (>= 3.3.25.1)
|   |-- Microsoft.AspNetCore.Http (>= 2.0.0)
|   |-- Microsoft.Extensions.Configuration (>= 2.0.0)
|   |-- System.Net.Http (>= 4.3.4)
|
|-- AWSXRayRecorder.Handlers.AspNetCore (>= 2.7.3)
|   |-- AWSXRayRecorder.Core (>= 2.10.1)
|   |-- Microsoft.AspNetCore.Http.Extensions (>= 2.0.0)
|   |-- Microsoft.AspNetCore.Mvc.Abstractions (>= 2.0.0)
|
|-- AWSXRayRecorder.Handlers.AwsSdk (>= 2.8.3)
|   |-- AWSXRayRecorder.Core (>= 2.10.1)
|
|-- AWSXRayRecorder.Handlers.EntityFramework (>= 1.1.1)
|   |-- AWSXRayRecorder.Core (>= 2.10.1)
|   |-- Microsoft.EntityFrameworkCore.Relational (>= 3.1.0)
|
|-- AWSXRayRecorder.Handlers.SqlServer (>= 2.7.3)
|   |-- AWSXRayRecorder.Core (>= 2.10.1)
|   |-- System.Data.SqlClient (>= 4.4.0)
|
|-- AWSXRayRecorder.Handlers.System.Net (>= 2.7.3)
    |-- AWSXRayRecorder.Core (>= 2.10.1)
```

依存関係管理の詳細については、マイクロソフトのドキュメントを参照してください。NuGet の依存関係そしてNuGet 依存関係の解決。

NET 用 X-Ray SDK の設定

X-Ray SDK for .NET にプラグインを設定して、アプリケーションが実行されているサービスに関する情報が含めたり、デフォルトのサンプリング動作を変更したり、特定のパスに対するリクエストに適用されるサンプリングルールを追加したりできます。

.NET ウェブアプリケーションの場合は、appSettings ファイルの Web.config セクションにキーを追加します。

Example Web.config

```
<configuration>
  <appSettings>
    <add key="AWSXRayPlugins" value="EC2Plugin"/>
    <add key="SamplingRuleManifest" value="sampling-rules.json"/>
  </appSettings>
</configuration>
```

.NET Core の場合は、appsettings.json という最上位のキーを持つ XRay という名前のファイルを作成します。

Example .NET appsettings.json

```
{
  "XRay": {
    "AWSXRayPlugins": "EC2Plugin",
    "SamplingRuleManifest": "sampling-rules.json"
  }
}
```

次に、アプリケーションコードで、設定オブジェクトを構築し、それを使用して X-Ray レコーダーを初期化します。この操作は、レコーダーを初期化する前に実行します。

Example .NET Core Program.cs – Recorder 設定

```
using Amazon.XRay.Recorder.Core;
...
AWSXRayRecorder.InitializeInstance(configuration);
```

.NET Core ウェブアプリケーションを計測する場合は、UseXRayメッセージ ハンドラーを設定するときに、設定オブジェクトを メソッドに渡すこともできます。Lambda 関数の場合は、上記のように InitializeInstance メソッドを使用します。

.NET Core 設定 API の詳細については、docs.microsoft.com の「ASP.NET Core アプリを構成する」を参照してください。

セクション

- プラグイン
- サンプリングルール
- ログ記録 (.NET)
- ログ記録 (.NET Core)
- 環境変数

プラグイン

プラグインを使用して、アプリケーションをホストしているサービスに関するデータを追加します。

プラグイン

- Amazon EC2 —EC2Pluginは、インスタンス ID、アベイラビリティーゾーン、および CloudWatch Logs グループを追加します。
- ElasticBeanstalk– ElasticBeanstalkPluginは、環境名、バージョンラベル、およびデプロイ ID を追加します。
- Amazon ECS —ECSPluginは、コンテナ ID を追加します。

プラグインを使用するには、AWSXRayPlugins 設定を追加して X-Ray SDK for .NET クライアントを設定します。複数のプラグインがアプリケーションに適用される場合は、そのすべてをカンマで区切って同じ設定で指定します。

Example Web.config - プラグイン

```
<configuration>
  <appSettings>
    <add key="AWSXRayPlugins" value="EC2Plugin,ElasticBeanstalkPlugin"/>
  </appSettings>
```

```
</configuration>
```

Example .NET Core appsettings.json – プラグイン

```
{
  "XRay": {
    "AWSXRayPlugins": "EC2Plugin,ElasticBeanstalkPlugin"
  }
}
```

サンプリングルール

SDK は X-Ray コンソールで定義したサンプリングルールを使用し、記録するリクエストを決定します。デフォルトルールでは、最初のリクエストを毎秒トレースし、X-Ray にトレースを送信するすべてのサービスで追加のリクエストの 5% をトレースします。X-Ray コンソールに追加のルールを作成するをクリックして、各アプリケーションで記録されるデータ量をカスタマイズします。

SDK は、定義された順序でカスタムルールを適用します。リクエストが複数のカスタムルールと一致する場合、SDK は最初のルールのみを適用します。

> **Note**
>
> SDK が X-Ray に到達してサンプリングルールを取得できない場合、1 秒ごとに最初のリクエストのデフォルトのローカルルールに戻り、ホストあたりの追加リクエストの 5% に戻ります。これは、ホストがサンプリング API を呼び出す権限を持っていない場合や、SDK によって行われる API 呼び出しの TCP プロキシとして機能する X-Ray デーモンに接続できない場合に発生します。

JSON ドキュメントからサンプリングルールをロードするように SDK を設定することもできます。SDK は、X-Ray サンプリングが利用できない場合のバックアップとしてローカルルールを使用することも、ローカルルールを排他的に使用することもできます。

Example sampling-rules.json

```
{
  "version": 2,
  "rules": [
    {
      "description": "Player moves.",
```

```
      "host": "*",
      "http_method": "*",
      "url_path": "/api/move/*",
      "fixed_target": 0,
      "rate": 0.05
    }
  ],
  "default": {
    "fixed_target": 1,
    "rate": 0.1
  }
}
```

この例では、1 つのカスタムルールとデフォルトルールを定義します。カスタムルールでは、5 パーセントのサンプリングレートが適用され、/api/move/以下のパスに対してトレースするリクエストの最小数はありません。デフォルトのルールでは、1秒ごとの最初のリクエストおよび追加リクエストの 10 パーセントをトレースします。

ルールをローカルで定義することの欠点は、固定ターゲットが X-Ray サービスによって管理されるのではなく、レコーダーの各インスタンスによって個別に適用されることです。より多くのホストをデプロイすると、固定レートが乗算され、記録されるデータ量の制御が難しくなります。

オンの場合 AWS Lambda、サンプリングレートを変更することはできません。関数がインストルメント化されたサービスによって呼び出された場合、そのサービスによってサンプリングされたリクエストを生成した呼び出しは Lambda によって記録されます。アクティブなトレースが有効で、トレースヘッダーが存在しない場合、Lambda はサンプリングを決定します。

バックアップルールを設定するには、SamplingRuleManifest 設定を使用して X-Ray SDK for .NET にファイルからサンプリングルールをロードするように指示します。

Example .NET Web.config - サンプリングルール

```
<configuration>
  <appSettings>
    <add key="SamplingRuleManifest" value="sampling-rules.json"/>
  </appSettings>
</configuration>
```

Example .NET Core appsettings.json – サンプリングルール

```
{
```

```
  "XRay": {
    "SamplingRuleManifest": "sampling-rules.json"
  }
}
```

ローカルルールのみを使用するには、LocalizedSamplingStrategy でレコーダーをビルドします。バックアップルールが設定されている場合、その設定を削除します。

Example .NET global.asax – ローカルサンプリングルール

```
var recorder = new AWSXRayRecorderBuilder().WithSamplingStrategy(new
 LocalizedSamplingStrategy("samplingrules.json")).Build();
AWSXRayRecorder.InitializeInstance(recorder: recorder);
```

Example .NET Core Program.cs – Local サンプリングルール

```
var recorder = new AWSXRayRecorderBuilder().WithSamplingStrategy(new
 LocalizedSamplingStrategy("sampling-rules.json")).Build();
AWSXRayRecorder.InitializeInstance(configuration,recorder);
```

ログ記録 (.NET)

X-Ray SDK for .NET では、AWS SDK for .NET。 AWS SDK for .NET 出力をログに記録するようにアプリケーションを設定済みである場合、X-Ray SDK for .NET からの出力にも同じ設定が適用されます。

ログ記録を設定するには、aws ファイルまたは App.config ファイルに、Web.config という名前の設定セクションを追加します。

Example Web.config - ログ記録

```
...
<configuration>
  <configSections>
    <section name="aws" type="Amazon.AWSSection, AWSSDK.Core"/>
  </configSections>
  <aws>
    <logging logTo="Log4Net"/>
  </aws>
</configuration>
```

詳細については、『AWS SDK for .NET 開発者ガイド』の「AWS SDK for .NET アプリケーションの設定」を参照してください。

ログ記録 (.NET Core)

X-Ray SDK for .NET では、AWS SDK for .NET。.NET Core アプリケーションのロギングを構成するには、logging オプションを`AWSXRayRecorder.RegisterLogger`方法。

たとえば、log4net を使用するには、ロガー、出力形式、およびファイルの場所を定義する設定ファイルを作成します。

Example .NET Core log4net.config

```
<?xml version="1.0" encoding="utf-8" ?>
<log4net>
  <appender name="FileAppender" type="log4net.Appender.FileAppender,log4net">
    <file value="c:\logs\sdk-log.txt" />
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%date [%thread] %level %logger - %message%newline" />
    </layout>
  </appender>
  <logger name="Amazon">
    <level value="DEBUG" />
    <appender-ref ref="FileAppender" />
  </logger>
</log4net>
```

次に、ロガーを作成し、プログラムコードで設定を適用します。

Example .NET Core Program.cs – ロギング

```
using log4net;
using Amazon.XRay.Recorder.Core;

class Program
{
  private static ILog log;
  static Program()
  {
    var logRepository = LogManager.GetRepository(Assembly.GetEntryAssembly());
    XmlConfigurator.Configure(logRepository, new FileInfo("log4net.config"));
    log = LogManager.GetLogger(typeof(Program));
```

```
    AWSXRayRecorder.RegisterLogger(LoggingOptions.Log4Net);
  }
  static void Main(string[] args)
  {
  ...
  }
}
```

log4net の設定の詳細については、logging.apache.org で設定を参照してください。

環境変数

環境変数を使用して、X-Ray SDK を .NET 用に設定できます。SDK は次の変数をサポートしています。

- AWS_XRAY_TRACING_NAME – SDK がセグメントに使用するサービス名を設定します。サーブレットフィルタのセグメント命名ルールで設定したサービス名を上書きします。
- AWS_XRAY_DAEMON_ADDRESS – X-Ray デーモン リスナーのホストとポートを設定します。デフォルトでは、SDK はトレースデータ (UDP) とサンプリング (TCP) の両方に127.0.0.1:2000を使用します。この変数は、デーモンを次のように構成している場合に使用します。別のポートでリッスンするまたは、別のホストで実行されている場合。

  形式

  - 同じポート – *address*:*port*
  - 異なるポート – tcp:*address*:*port* udp:*address*:*port*
- AWS_XRAY_CONTEXT_MISSING – 計測されたコードが、セグメントが開いていないときにデータを記録しようとした場合に例外をスローするには、RUNTIME_ERROR に設定します。

  有効な値

  - RUNTIME_ERROR— ランタイム例外をスローします。
  - LOG_ERROR – エラーをログ記録して続行します (デフォルト)。
  - IGNORE_ERROR – エラーを無視して続行します。

  オープン状態のリクエストがない場合、または新しいスレッドを発生させるコードで、スタートアップコードに実装されたクライアントを使用しようとした場合に発生する可能性がある、セグメントまたはサブセグメントの欠落に関連するエラー。

X-Ray SDK for .NET

X-Ray SDK を使用して、アプリケーションが Amazon EC2 の EC2 インスタンス AWS Elastic Beanstalk、または Amazon ECS で処理する受信 HTTP リクエストをトレースできます。 Amazon EC2

メッセージハンドラーを使用して受信 HTTP リクエストを計測します。X-Ray メッセージハンドラーをアプリケーションに追加すると、サンプリングされた各リクエストに X-Ray SDK for .NET によってセグメントが作成されます。このセグメントには、時間、メソッド、HTTP リクエストの処理などが含まれます。追加の計測により、このセグメントでサブセグメントが作成されます。

> **Note**
>
> AWS Lambda 関数の場合、Lambda はサンプリングされたリクエストごとにセグメントを作成します。詳細については「AWS Lambda and AWS X-Ray」を参照してください。

各セグメントには、サービスマップ内のアプリケーションを識別する名前があります。セグメントの名前は静的に指定することも、受信リクエストのホストヘッダーに基づいて動的に名前を付けるように SDK を設定することもできます。動的ネーミングでは、リクエスト内のドメイン名に基づいてトレースをグループ化でき、名前が予想されるパターンと一致しない場合 (たとえば、ホストヘッダーが偽造されている場合) 、デフォルト名を適用できます。

> **転送されたリクエスト**
>
> ロードバランサーまたは他の仲介者がアプリケーションにリクエストを転送する場合、X-Ray は、クライアントの IP をIP パケットの送信元 IP からではなく、リクエストのX-Forwarded-Forヘッダーから取得します。転送された要求に対して記録されたクライアント IP は偽造される可能性があるため、信頼されるべきではありません。

メッセージハンドラーは、次の情報が含まれる `http` ブロックを使用して、各受信リクエスト用にセグメントを作成します。

- HTTP メソッド – GET、POST、PUT、DELETE、その他。
- クライアントアドレス – リクエストを送信するクライアントの IP アドレス。
- レスポンスコード – 完了したリクエストの HTTP レスポンスコード。

- タイミング – 開始時間 (リクエストが受信された時間) および終了時間 (レスポンスが送信された時間)。
- ユーザーエージェント — リクエストからのuser-agent
- コンテンツの長さ — レスポンスからのcontent-length

セクション

受信リクエストの計測 (.NET)

アプリケーションによって処理されるリクエストを計測するには、RegisterXRay ファイルの Init メソッドで global.asax を呼び出します。

Example global.asax - メッセージハンドラー

```
using System.Web.Http;
using Amazon.XRay.Recorder.Handlers.AspNet;

namespace SampleEBWebApplication
{
  public class MvcApplication : System.Web.HttpApplication
  {
    public override void Init()
    {
      base.Init();
      AWSXRayASPNET.RegisterXRay(this, "MyApp");
    }
  }
}
```

受信リクエストの計測 (.NET Core)

アプリケーションによって処理されるリクエストを計測するには、UseXRay他のミドルウェアより前のメソッドConfigureStartup クラスのメソッドは、理想的にはX-Rayミドルウェアがリクエストを処理する最初のミドルウェアであり、パイプラインでレスポンスを処理する最後のミドルウェアにする必要があります。

> **Note**
>
> .NET Core 2.0 の場合、UseExceptionHandlerアプリケーションのメソッドで、必ず呼び出してくださいUseXRay後UseExceptionHandlerメソッドを使用して、例外が記録されるようにします。

Example Startup.cs

<caption>.NET Core 2.1 and above</caption>

```
using Microsoft.AspNetCore.Builder;

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
  {
    app.UseXRay("MyApp");
    // additional middleware
    ...
  }
```

<caption>.NET Core 2.0</caption>

```
using Microsoft.AspNetCore.Builder;

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
  {
    app.UseExceptionHandler("/Error");
    app.UseXRay("MyApp");
    // additional middleware
    ...
  }
```

この UseXRay メソッドは、2 番目の引数として設定オブジェクトを受け取ることもできます。

```
app.UseXRay("MyApp", configuration);
```

セグメント命名ルールの設定

AWS X-Ray は、サービス名を使用してアプリケーションを識別し、アプリケーションが使用する他のアプリケーション、データベース、外部 APIs、 AWS リソースと区別します。X-Ray SDK が受信

リクエストのセグメントを生成すると、アプリケーションのサービス名がセグメントの名前フィールドに記録されます。

X-Ray SDK では、HTTP リクエストヘッダーのホスト名の後にセグメントの名前を指定できます。ただし、このヘッダーは偽造され、サービスマップに予期しないノードが発生する可能性があります。偽造されたホストヘッダーを持つリクエストによって SDK がセグメントの名前を間違えないようにするには、受信リクエストのデフォルト名を指定する必要があります。

アプリケーションが複数のドメインのリクエストを処理する場合、動的ネーミングストラテジーを使用してセグメント名にこれを反映するように SDK を設定できます。動的ネーミングストラテジーにより、SDK は予想されるパターンに一致するリクエストにホスト名を使用し、そうでないリクエストにデフォルト名を適用できます。

たとえば、3 つのサブドメイン (`www.example.com`,`api.example.com`,および`static.example.com`) に対してリクエストを処理する単一のアプリケーションがあるとします。動的ネーミングストラテジーをパターン `*.example.com` で使用して、異なる名前を持つ各サブドメインのセグメントを識別することができます。結果的にはサービスマップ上に 3 つのサービスノードを作成することになります。アプリケーションがパターンと一致しないホスト名のリクエストを受信すると、指定したフォールバック名を持つ 4 番目のノードがサービスマップに表示されます。

すべてのリクエストセグメントに対して同じ名前を使用するには、前のセクションで示すとおり、メッセージハンドラを初期化するときに、アプリケーションの名前を指定します。これは、`FixedSegmentNamingStrategy` を作成して、`RegisterXRay` メソッドに渡すのと同じ効果があります。

```
AWSXRayASPNET.RegisterXRay(this, new FixedSegmentNamingStrategy("MyApp"));
```

> **Note**
>
> コードで定義したデフォルトのサービス名は、`AWS_XRAY_TRACING_NAME` 環境変数で上書きできます。

動的な命名戦略は、ホスト名と一致するようパターンを定義し、HTTP リクエストのホスト名がパターンと一致しない場合はデフォルトの名前を使用します。動的にセグメントに命名するには、`DynamicSegmentNamingStrategy` を作成して、`RegisterXRay` メソッドに渡します。

```
AWSXRayASPNET.RegisterXRay(this, new DynamicSegmentNamingStrategy("MyApp",
 "*.example.com"));
```

X-Ray AWS SDK for .NET を使用した SDK 呼び出しのトレース

アプリケーションが AWS のサービス を呼び出してデータの保存、キューへの書き込み、または通知の送信を行うと、X-Ray SDK for .NET はサブセグメントのダウンストリームの呼び出しを追跡します。これらのサービス (Amazon S3 バケットや Amazon SQS キューなど) 内でアクセスするトレースされた AWS のサービス および リソースは、X-Ray コンソールのトレースマップにダウンストリームノードとして表示されます。

クライアントを作成するRegisterXRayForAllServices前に を呼び出すことで、すべての AWS SDK for .NET クライアントを計測できます。

Example SampleController.cs - DynamoDB クライアント計測

```
using Amazon;
using Amazon.Util;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DocumentModel;
using Amazon.XRay.Recorder.Core;
using Amazon.XRay.Recorder.Handlers.AwsSdk;

namespace SampleEBWebApplication.Controllers
{
  public class SampleController : ApiController
  {
    AWSSDKHandler.RegisterXRayForAllServices();
    private static readonly Lazy<AmazonDynamoDBClient> LazyDdbClient = new
 Lazy<AmazonDynamoDBClient>(() =>
    {
      var client = new AmazonDynamoDBClient(EC2InstanceMetadata.Region ??
 RegionEndpoint.USEast1);
      return client;
    });
```

一部のサービスのクライアントのみを計測するには、RegisterXRayForAllServices ではなく RegisterXRay を呼び出します。強調表示されたテキストを、サービスのクライアントインターフェイスの名前で置き換えます。

```
AWSSDKHandler.RegisterXRay<IAmazonDynamoDB>()
```

すべてのサービスで、X-Ray コンソールで呼び出される API の名前を確認できます。サービスのサブセットの場合、X-Ray SDK はセグメントに情報を追加して、サービスマップでより細かく指定します。

たとえば、実装された DynamoDB クライアントでコールすると、SDK はテーブルをターゲットとするコールのセグメントにテーブル名を追加します。コンソールで、各テーブルはサービスマップ内に個別のノードとして表示され、テーブルをターゲットにしないコール用の汎用の DynamoDB ノードが表示されます。

Example 項目を保存するための DynamoDB に対するコールのサブセグメント

```
{
  "id": "24756640c0d0978a",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "DynamoDB",
  "namespace": "aws",
  "http": {
    "response": {
      "content_length": 60,
      "status": 200
    }
  },
  "aws": {
    "table_name": "scorekeep-user",
    "operation": "UpdateItem",
    "request_id": "UBQNSO5AEM8T4FDA4RQDEB94OVTDRVV4K4HIRGVJF66Q9ASUAAJG",
  }
}
```

名前付きリソースにアクセスしたとき、次のサービスをコールすると、サービスマップに追加のノードが作成されます。特定のリソースをターゲットとしないコールでは、サービスの汎用ノードが作成されます。

- Amazon DynamoDB – テーブル名
- Amazon Simple Storage Service – バケットとキー名
- Amazon Simple Queue Service – キュー名

X-Ray SDK for .NET を使用してダウンストリーム HTTP ウェブサービスの呼び出しをトレースする

アプリケーションがマイクロサービスまたはパブリック HTTP API に呼び出しを実行する場合に、GetResponseTraced で X-Ray SDK for .NET の System.Net.HttpWebRequest 拡張メソッドを使用してこれらの呼び出しを計測し、API をダウンストリームサービスとしてサービスグラフに追加できます。

Example HttpWebRequest

```
using System.Net;
using Amazon.XRay.Recorder.Core;
using Amazon.XRay.Recorder.Handlers.System.Net;

private void MakeHttpRequest()
{
  HttpWebRequest request = (HttpWebRequest)WebRequest.Create("http://names.example.com/
api");
  request.GetResponseTraced();
}
```

非同期呼び出しには、GetAsyncResponseTraced を使用します。

```
request.GetAsyncResponseTraced();
```

system.net.http.httpclient を使用する場合は、HttpClientXRayTracingHandler 委任ハンドラを使用して呼び出しを記録します。

Example HttpClient

```
using System.Net.Http;
using Amazon.XRay.Recorder.Core;
using Amazon.XRay.Recorder.Handlers.System.Net;

private void MakeHttpRequest()
{
  var httpClient = new HttpClient(new HttpClientXRayTracingHandler(new
 HttpClientHandler()));
  httpClient.GetAsync(URL);
}
```

ダウンストリームウェブ API に対する呼び出しを計測すると、X-Ray SDK for .NET は HTTP リクエストおよびレスポンスに関する情報を含むセグメントを記録します。X-Ray はサブセグメントを使用して API の推測セグメントを生成します。

Example ダウンストリーム HTTP 呼び出しのサブセグメント

```
{
  "id": "004f72be19cddc2a",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "name": "names.example.com",
  "namespace": "remote",
  "http": {
    "request": {
      "method": "GET",
      "url": "https://names.example.com/"
    },
    "response": {
      "content_length": -1,
      "status": 200
    }
  }
}
```

Example ダウンストリーム HTTP 呼び出しの推定セグメント

```
{
  "id": "168416dc2ea97781",
  "name": "names.example.com",
  "trace_id": "1-62be1272-1b71c4274f39f122afa64eab",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "parent_id": "004f72be19cddc2a",
  "http": {
    "request": {
      "method": "GET",
      "url": "https://names.example.com/"
    },
    "response": {
      "content_length": -1,
      "status": 200
    }
  },
```

```
  "inferred": true
}
```

X-Ray SDK for .NET

X-Ray SDK for .NET は、SqlCommand の代わりに使用できる TraceableSqlCommand という名前の System.Data.SqlClient.SqlCommand のラッパークラスを提供します。TraceableSqlCommand クラスを使用して、SQL コマンドを初期化できます。

同期メソッドと非同期メソッドを使用した SQL クエリのトレース

以下の例では、TraceableSqlCommand を使用して自動的に SQL Server クエリを同期的および非同期的にトレースする方法を示しています。

Example **Controller.cs** - SQL クライアント計測 (同期)

```
using Amazon;
using Amazon.Util;
using Amazon.XRay.Recorder.Core;
using Amazon.XRay.Recorder.Handlers.SqlServer;

private void QuerySql(int id)
{
  var connectionString = ConfigurationManager.AppSettings["RDS_CONNECTION_STRING"];
  using (var sqlConnection = new SqlConnection(connectionString))
  using (var sqlCommand = new TraceableSqlCommand("SELECT " + id, sqlConnection))
  {
    sqlCommand.Connection.Open();
    sqlCommand.ExecuteNonQuery();
  }
}
```

ExecuteReaderAsync メソッドを使用して、クエリを非同期的に実行できます。

Example **Controller.cs** - SQL クライアント計測 (非同期)

```
using Amazon;
using Amazon.Util;
using Amazon.XRay.Recorder.Core;
using Amazon.XRay.Recorder.Handlers.SqlServer;
private void QuerySql(int id)
```

```
{
  var connectionString = ConfigurationManager.AppSettings["RDS_CONNECTION_STRING"];
  using (var sqlConnection = new SqlConnection(connectionString))
  using (var sqlCommand = new TraceableSqlCommand("SELECT " + id, sqlConnection))
  {
    await sqlCommand.ExecuteReaderAsync();
  }
}
```

SQL Server に対して実行された SQL クエリの収集

SQL クエリによって作成されたサブセグメントの一部として SqlCommand.CommandText のキャプチャを有効にできます。SqlCommand.CommandText は、サブセグメント JSON のフィールド sanitized_query として表示されます。デフォルトでは、この機能はセキュリティのために無効になっています。

> **Note**
>
> SQL クエリに機密情報をクリアテキストとして含める場合は、収集機能を有効にしないでください。

SQL クエリの収集を有効にするには、以下の 2 つの方法があります。

- アプリケーションのグローバル設定で CollectSqlQueries プロパティを true に設定する。
- インスタンス内の呼び出しを収集するには、TraceableSqlCommand インスタンスの collectSqlQueries パラメータを true に設定する。

Global CollectSqlQueries プロパティを有効にする

以下の例では、.NET および .NET Core の CollectSqlQueries プロパティを有効にする方法を示しています。

.NET

NET のアプリケーションのグローバル設定で CollectSqlQueries プロパティを true に指定するには、ここで示しているように App.config または Web.config ファイルの appsettings を変更します。

Example **App.config** または **Web.config** – SQL クエリの収集をグローバルに有効にする

```
<configuration>
<appSettings>
    <add key="CollectSqlQueries" value="true">
</appSettings>
</configuration>
```

.NET Core

.NET Core のアプリケーションのグローバル設定で CollectSqlQueries プロパティを true に指定するには、ここで示しているように appsettings.jsonX-Ray キーの ファイルを変更します。

Example **appsettings.json** – SQL クエリの収集をグローバルに有効にする

```
{
  "XRay": {
    "CollectSqlQueries":"true"
  }
}
```

collectSqlQueries パラメータを有効にする

TraceableSqlCommand インスタンスの collectSqlQueries パラメータを true に設定することで、そのインスタンスを使用して実行された SQL Server クエリの SQL クエリテキストを収集できます。このパラメータを false に設定すると、TraceableSqlCommand インスタンスの CollectSqlQuery 機能が無効になります。

> **Note**
>
> TraceableSqlCommand インスタンスの collectSqlQueries の値は、CollectSqlQueries プロパティのグローバル設定で指定された値を上書きします。

Example サンプル **Controller.cs** – インスタンスの SQL クエリの収集を有効にする

```
using Amazon;
using Amazon.Util;
using Amazon.XRay.Recorder.Core;
```

```
using Amazon.XRay.Recorder.Handlers.SqlServer;

private void QuerySql(int id)
{
  var connectionString = ConfigurationManager.AppSettings["RDS_CONNECTION_STRING"];
  using (var sqlConnection = new SqlConnection(connectionString))
  using (var command = new TraceableSqlCommand("SELECT " + id, sqlConnection,
 collectSqlQueries: true))
  {
    command.ExecuteNonQuery();
  }
}
```

追加のサブセグメントを作成する

サブセグメントはトレースを拡張しますセグメントリクエストを処理するために行われた作業の詳細を含む。計測済みクライアント内で呼び出しを行うたびに、X-Ray SDK によってサブセグメントに生成された情報が記録されます。追加のサブセグメントを作成して、他のサブセグメントをグループ化したり、コードセクションのパフォーマンスを測定したり、注釈とメタデータを記録したりできます。

サブセグメントを管理するには、BeginSubsegment および EndSubsegment メソッドを使用します。try ブロックでサブセグメントの任意の作業を実行し、AddException を使用して例外をトレースします。ブロックで EndSubsegment を呼び出し、サブセグメントが閉じられたことを確認します。finally

Example Controller.cs – カスタムサブセグメント

```
AWSXRayRecorder.Instance.BeginSubsegment("custom method");
try
{
  DoWork();
}
catch (Exception e)
{
  AWSXRayRecorder.Instance.AddException(e);
}
finally
{
  AWSXRayRecorder.Instance.EndSubsegment();
}
```

セグメントまたは別のサブセグメント内にサブセグメントを作成する場合、X-Ray SDK for .NET によってその ID が生成され、開始時刻と終了時刻が記録されます。

Example サブセグメントとメタデータ

```
"subsegments": [{
  "id": "6f1605cd8a07cb70",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "Custom subsegment for UserModel.saveUser function",
  "metadata": {
    "debug": {
      "test": "Metadata string from UserModel.saveUser"
    }
  },
```

X-Ray SDK for .NET を使用してセグメントに注釈とメタデータを追加する

アプリケーションが注釈やメタデータで行うリクエストや環境や、作業に関する追加情報を記録します。X-Ray SDK が作成するセグメントまたは作成するカスタムサブセグメントに、注釈およびメタデータを追加できます。

注釈は文字列、数値、またはブール値を使用したキーと値のペアです。注釈は、フィルタ式用にインデックス付けされます。注釈を使用して、コンソールでトレースをグループ化するため、またはGetTraceSummaries API を呼び出すときに使用するデータを記録します。

メタデータは、オブジェクトとリストを含む、任意のタイプの値を持つことができるキーバリューのペアですが、フィルタ式に使用するためにインデックスは作成されません。メタデータを使用してトレースに保存する追加のデータを記録しますが、トレースの検索用に使用する必要はありません。

セクション

- X-Ray SDK を使用して.NET の注釈を記録する
- X-Ray SDK for .NET

X-Ray SDK を使用して.NET の注釈を記録する

注釈を使用して、検索用にインデックスを作成するセグメントまたはサブセグメントに情報を記録します。

X-Ray のすべての注釈には以下が必要です。

注釈の要件

- キー – X-Ray 注釈のキーには最大 500 文字の英数字を使用できます。スペースまたはドットやピリオド (.) 以外の記号は使用できません。
- 値 – X-Ray 注釈の値には最大 1,000 の Unicode 文字を使用できます。
- 注釈の数 – トレースごとに最大 50 の注釈を使用できます。

AWS Lambda 関数の外部に注釈を記録するには

1. `AWSXRayRecorder` のインスタンスを取得します。

   ```
   using Amazon.XRay.Recorder.Core;
   ...
   AWSXRayRecorder recorder = AWSXRayRecorder.Instance;
   ```

2. 文字列キー、およびブール値、Int32、Int64、Double、文字列値を使用して `addAnnotation` を呼び出します。

   ```
   recorder.AddAnnotation("mykey", "my value");
   ```

   次の例は、ドット、ブール値、数値、または文字列値を含む文字列キーを使用して `putAnnotation` を呼び出す方法を示しています。

   ```
   document.putAnnotation("testkey.test", "my value");
   ```

AWS Lambda 関数内の注釈を記録するには

Lambda ランタイム環境では Lambda 関数内のセグメントとサブセグメントの両方が管理されます。Lambda 関数内のセグメントまたはサブセグメントに注釈を追加する場合は、以下の内容を実行する必要があります。

1. Lambda 関数内にセグメントまたはサブセグメントを作成します。
2. セグメントまたはサブセグメントに注釈を追加します。
3. セグメントまたはサブセグメントを終了します。

次のコード例は、Lambda 関数内のサブセグメントに注釈を追加する方法を示しています。

```
#Create the subsegment
AWSXRayRecorder.Instance.BeginSubsegment("custom method");
#Add an annotation
AWSXRayRecorder.Instance.AddAnnotation("My", "Annotation");
try
{
  YourProcess(); #Your function
}
catch (Exception e)
{
  AWSXRayRecorder.Instance.AddException(e);
}
finally #End the subsegment
{
  AWSXRayRecorder.Instance.EndSubsegment();
}
```

X-Ray SDK は、セグメントドキュメントの annotations オブジェクトにキーと値のペアとして、注釈を記録します。同じキーで addAnnotation を 2 回呼び出すと、同じセグメントまたはサブセグメントに以前記録された値は上書きされます。

特定の値を持つ注釈のあるトレースを見つけるには、annotation[*key*]フィルタ式の キーワードを使用します。

X-Ray SDK for .NET

メタデータを使用して、検索内でインデックスを使用する必要がないセグメントまたはサブセグメントに情報を記録します。メタデータ値は、文字列、数値、ブール値、または JSON オブジェクトや JSON 配列にシリアル化できるその他の任意のオブジェクトになります。

メタデータを記録するには

1. 次のコード例に示すように、AWSXRayRecorder のインスタンスを取得します。

   ```
   using Amazon.XRay.Recorder.Core;
   ...
   AWSXRayRecorder recorder = AWSXRayRecorder.Instance;
   ```

2. 次のコード例に示すように、文字列名前空間、文字列キー、およびオブジェクト値を使用して AddMetadata を呼び出します。

```
recorder.AddMetadata("my namespace", "my key", "my value");
```

次のコード例に示すように、キーと値のペアのみを使用して AddMetadata オペレーションを呼び出すこともできます。

```
recorder.AddMetadata("my key", "my value");
```

名前空間の値を指定しない場合、X-Ray SDK は default を使用します。同じキーで AddMetadata を 2 回呼び出すと、同じセグメントまたはサブセグメントに以前記録された値は上書きされます。

Ruby の使用

X-Ray にトレースを送信する Ruby アプリケーションを計測するには、次の 2 つの方法があります。

- AWS Distro for OpenTelemetry Ruby – AWS Distro for OpenTelemetry Collector を介して、相関メトリクスとトレースを Amazon CloudWatch、 AWS X-Ray Amazon OpenSearch Service などの複数の AWS モニタリングソリューションに送信するための一連のオープンソースライブラリを提供する AWS ディストリビューション。
- AWS X-Ray SDK for Ruby – X-Ray デーモンを介してトレースを生成して X-Ray に送信するためのライブラリのセット。

詳細については、「Distro for OpenTelemetry AWS と X-Ray SDKs の選択」を参照してください。

AWS Distro for OpenTelemetry Ruby

AWS Distro for OpenTelemetry (ADOT) Ruby を使用すると、アプリケーションを一度計測し、相関メトリクスとトレースを Amazon CloudWatch AWS X-Ray、Amazon OpenSearch Service などの複数の AWS モニタリングソリューションに送信できます。X-Ray を ADOT で使用するには、X-Ray で使用できる OpenTelemetry SDK と、X-Ray で使用できる AWS Distro for OpenTelemetry Collector の 2 つのコンポーネントが必要です。

開始するには、「AWS Distro for OpenTelemetry Ruby documentation」を参照してください。

AWS Distro for OpenTelemetry を AWS X-Ray およびその他の で使用する方法の詳細については AWS のサービス、AWS 「 Distro for OpenTelemetry」またはAWS 「 Distro for OpenTelemetry ドキュメント」を参照してください。

言語サポートと使用方法の詳細については、「AWS Observability on Github」を参照してください。

AWS X-Ray SDK for Ruby

X-Ray SDKは、Ruby ウェブアプリケーション用のライブラリです。トレースデータを作成して X-Ray デーモンに送信するためのクラスとメソッドを提供します。トレースデータには、アプリケーションによって提供される受信 HTTP リクエストに関する情報、および AWS SDK、HTTP クライアント、またはアクティブなレコードクライアントを使用してアプリケーションがダウンストリーム

サービスに対して行う呼び出しが含まれます。セグメントを手動で作成し、注釈およびメタデータにデバッグ情報を追加することもできます。

gemfile に追加し、`bundle install` を実行することで、SDK をダウンロードできます。

Example Gemfile

```
gem 'aws-sdk'
```

Rails を使用する場合は、最初に X-Ray SDK ミドルウェアを追加して受信リクエストをトレースします。リクエストフィルタにより、セグメントが作成されます。セグメントが開いている間、SDK クライアントのメソッドを使用してセグメントに情報を追加し、サブセグメントを作成してダウンストリーム呼び出しをトレースできます。また、SDK では、セグメントが開いている間にアプリケーションがスローする例外を自動的に記録します。Rails 以外のアプリケーションの場合、手動でセグメントを作成することができます。

次に、X-Ray SDK を使用して AWS SDK for Ruby、関連付けられたライブラリにパッチを適用するようにレコーダーを設定して、、HTTP、SQL クライアントを計測します。計測されたクライアントを使用してダウンストリーム AWS のサービス またはリソースを呼び出すたびに、SDK は呼び出しに関する情報をサブセグメントに記録します。 AWS のサービス また、サービス内でアクセスするリソースは、トレースマップにダウンストリームノードとして表示され、個々の接続のエラーやスロットリングの問題を特定するのに役立ちます。

SDK を入手したら、レコーダーを設定して動作をカスタマイズします。プラグインを追加して、アプリケーションを実行しているコンピューティングリソースに関するデータを記録したり、サンプリングルールを定義することでサンプリングの動作をカスタマイズしたり、ロガーを提供してアプリケーションログに SDK からの情報をより多くあるいは少なく表示することができます。

アプリケーションが注釈やメタデータで行うリクエストや作業に関する追加情報を記録します。注釈は、フィルタ式で使用するためにインデックス化されたシンプルなキーと値のペアで、特定のデータが含まれているトレースを検索できます。メタデータのエントリは制約が緩やかで、JSON にシリアル化できるオブジェクトと配列全体を記録できます。

> **注釈とメタデータ**
>
> 注釈およびメタデータとは、X-Ray SDK を使用してセグメントに追加する任意のテキストです。注釈は、フィルタ式用にインデックス付けされます。メタデータはインデックス化されませんが、X-Ray コンソールまたは API を使用して raw セグメントで表示できます。X-Ray への読み取りアクセスを許可した人は誰でも、このデータを表示できます。

コードに多数の計測されたクライアントがある場合、単一のリクエストセグメントには計測されたクライアントで行われた呼び出しごとに 1 個の多数のサブセグメントを含めることができます。カスタムサブセグメントで、クライアント呼び出しをラップすることで、サブセグメントを整理してグループできます。関数全体またはコードの任意のセクションのサブセグメントを作成し、親セグメントにすべてのレコードを記述する代わりにサブセグメントにメタデータと注釈を記録できます。

SDK のクラスとメソッドのリファレンス ドキュメントについては、AWS X-Ray SDK for Ruby API リファレンスを参照してください。

要件

X-Ray SDK では Ruby 2.3 以降が必要です。また、次のライブラリと互換性があります。

- AWS SDK for Ruby バージョン 3.0 以降
- Rails バージョン 5.1 以降

X-Ray SDK for Ruby の設定

X-Ray SDK for Ruby には、グローバルレコーダーを提供する `XRay.recorder` というクラスがあります。グローバルレコーダーを設定して、受信 HTTP コールのセグメントを作成するミドルウェアをカスタマイズできます。

セクション

- サービスプラグイン
- サンプリングルール
- ロギング
- コード内のレコーダー設定
- Rails でのレコーダー設定
- 環境変数

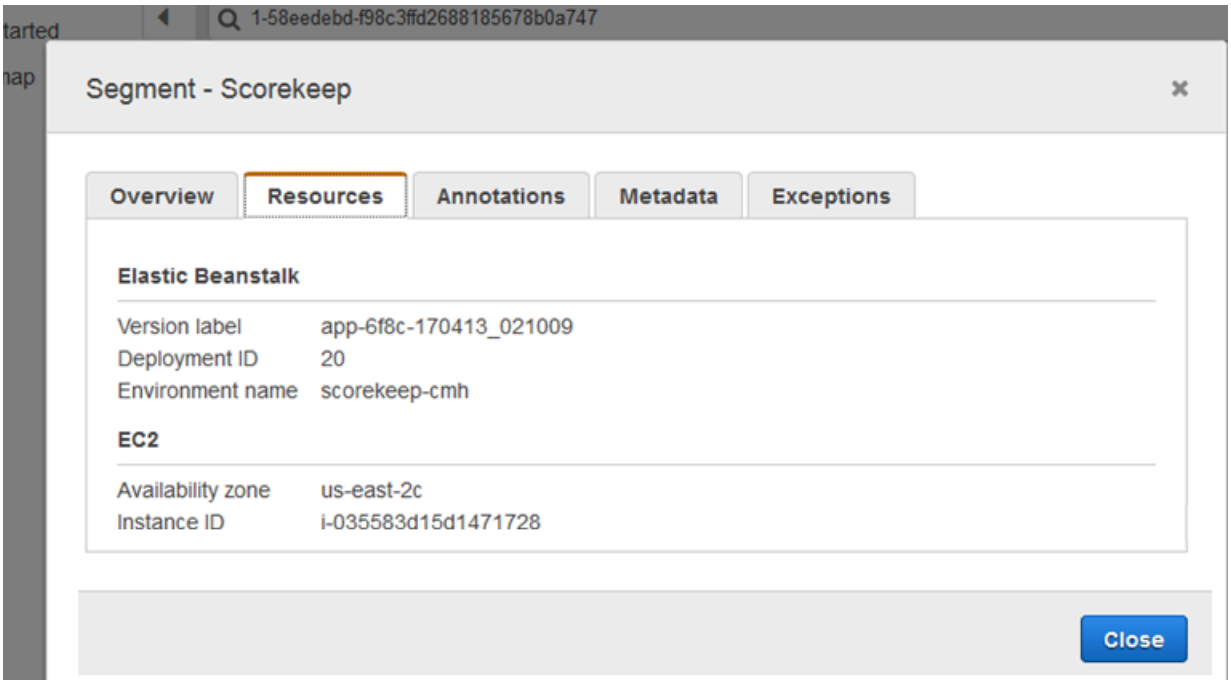
サービスプラグイン

`plugins`を使用して、アプリケーションをホストしているサービスに関する情報を記録します。

プラグイン

- Amazon EC2 – `ec2` はインスタンス ID とアベイラビリティーゾーンを追加します。

- Elastic Beanstalk – `elastic_beanstalk` は環境名、バージョンラベル、およびデプロイ ID を追加します。
- Amazon ECS —`ecs`は、コンテナ ID を追加します。



プラグインを使用するには、レコーダーに渡す設定オブジェクトでそのプラグインを指定します。

Example main.rb – プラグインの設定

```
my_plugins = %I[ec2 elastic_beanstalk]

config = {
  plugins: my_plugins,
  name: 'my app',
}

XRay.recorder.configure(config)
```

また、コードで設定した値よりも優先される環境変数を使用して、レコーダーを設定することもできます。

また、SDK はプラグイン設定を使用して、セグメントの `origin` フィールドを設定します。これは、アプリケーションを実行する AWS リソースのタイプを示します。複数のプラグインを使用する場合、SDK は次の解決順序を使用して起点を決定します。ElasticBeanStalk > EKS > ECS > EC2。

サンプリングルール

SDK は X-Ray コンソールで定義したサンプリングルールを使用し、記録するリクエストを決定します。デフォルトルールでは、最初のリクエストを毎秒トレースし、X-Ray にトレースを送信するすべてのサービスで追加のリクエストの 5% をトレースします。X-Ray コンソールに追加のルールを作成するをクリックして、各アプリケーションで記録されるデータ量をカスタマイズします。

SDK は、定義された順序でカスタムルールを適用します。リクエストが複数のカスタムルールと一致する場合、SDK は最初のルールのみを適用します。

> **Note**
>
> SDK が X-Ray に到達してサンプリングルールを取得できない場合、1 秒ごとに最初のリクエストのデフォルトのローカルルールに戻り、ホストあたりの追加リクエストの 5% に戻ります。これは、ホストがサンプリング API を呼び出す権限を持っていない場合や、SDK によって行われる API 呼び出しの TCP プロキシとして機能する X-Ray デーモンに接続できない場合に発生します。

JSON ドキュメントからサンプリングルールをロードするように SDK を設定することもできます。SDK は、X-Ray サンプリングが利用できない場合のバックアップとしてローカルルールを使用することも、ローカルルールを排他的に使用することもできます。

Example sampling-rules.json

```
{
  "version": 2,
  "rules": [
    {
      "description": "Player moves.",
      "host": "*",
      "http_method": "*",
      "url_path": "/api/move/*",
      "fixed_target": 0,
      "rate": 0.05
    }
  ],
  "default": {
    "fixed_target": 1,
    "rate": 0.1
  }
```

```
}
```

この例では、1 つのカスタムルールとデフォルトルールを定義します。カスタムルールでは、5 パーセントのサンプリングレートが適用され、/api/move/以下のパスに対してトレースするリクエストの最小数はありません。デフォルトのルールでは、1秒ごとの最初のリクエストおよび追加リクエストの 10 パーセントをトレースします。

ルールをローカルで定義することの欠点は、固定ターゲットが X-Ray サービスによって管理されるのではなく、レコーダーの各インスタンスによって個別に適用されることです。より多くのホストをデプロイすると、固定レートが重複し、記録されるデータ量の制御が難しくなります。

バックアップルールを設定するには、レコーダーに渡す設定オブジェクトで、ドキュメントのハッシュを定義します。

Example main.rb – Backupルールの設定

```
require 'aws-xray-sdk'
my_sampling_rules =  {
  version: 1,
  default: {
    fixed_target: 1,
    rate: 0.1
  }
}
config = {
  sampling_rules: my_sampling_rules,
  name: 'my app',
}
XRay.recorder.configure(config)
```

サンプリングルールを個別に保存するには、別個のファイルにハッシュを定義し、アプリケーションにそのハッシュをプルするようにファイルに要求します。

Example config/sampling-rules.rb

```
my_sampling_rules =  {
  version: 1,
  default: {
    fixed_target: 1,
    rate: 0.1
  }
}
```

Example main.rb – ファイルからのサンプリングルール

```
require 'aws-xray-sdk'
require 'config/sampling-rules.rb'

config = {
  sampling_rules: my_sampling_rules,
  name: 'my app',
}
XRay.recorder.configure(config)
```

ローカルルールのみを使用するには、サンプリングルールと `LocalSampler` を設定する必要があります。

Example main.rb – ローカルルールサンプリング

```
require 'aws-xray-sdk'
require 'aws-xray-sdk/sampling/local/sampler'

config = {
  sampler: LocalSampler.new,
  name: 'my app',
}
XRay.recorder.configure(config)
```

サンプリングを無効にしてすべての着信リクエストを実装するように、グローバルレコーダーを設定することもできます。

Example main.rb – サンプリングを無効にする

```
require 'aws-xray-sdk'
config = {
  sampling: false,
  name: 'my app',
}
XRay.recorder.configure(config)
```

ロギング

デフォルトでは、レコーダーは情報レベルのイベントを `$stdout` に出力します。レコーダーに渡す設定オブジェクトで、ロガーを定義してログ記録をカスタマイズできます。

Example main.rb – ログ記録

```
require 'aws-xray-sdk'
config = {
  logger: my_logger,
  name: 'my app',
}
XRay.recorder.configure(config)
```

デバッグログを使用して問題を識別します。たとえば、「サブセグメントを手動で生成する」場合にサブセグメントが閉じない問題などです。

コード内のレコーダー設定

追加の設定は、XRay.recorder の configure メソッドから利用できます。

- context_missing - 測定されたコードが、セグメントが開いていないときにデータを記録しようとした場合に例外のスローを回避するには、LOG_ERROR に設定します。
- daemon_address – X-Ray デーモンリスナーのホストとポートを設定します。
- name – SDK がセグメントに使用するサービス名を設定します。
- naming_pattern – 動的な命名を使用するようにドメイン名を設定します。
- plugins – プラグインを使用して、アプリケーションの AWS リソースに関する情報を記録します。
- sampling – false に設定してサンプリングを無効にします。
- sampling_rules – サンプリングルールを含むハッシュを設定します。

Example main.py – コンテキスト欠落例外を無効にする

```
require 'aws-xray-sdk'
config = {
  context_missing: 'LOG_ERROR'
}

XRay.recorder.configure(config)
```

Rails でのレコーダー設定

Rails フレームワークを使用している場合は、Ruby ファイルの app_root/initializers 以下で、グローバルレコーダーのオプションを設定できます。X-Ray SDK は、Rails で使用する追加の設定キーをサポートしています。

- active_record – trueに設定して、Active Record データベーストランザクションのサブセグメントを記録します。

Rails.application.config.xray という名前の設定オブジェクトで利用可能な設定を行います。

Example config/initializers/aws_xray.rb

```
Rails.application.config.xray = {
  name: 'my app',
  patch: %I[net_http aws_sdk],
  active_record: true
}
```

環境変数

環境変数を使用して、X-Ray SDK for Ruby を設定できます。SDK は次の変数をサポートしています。

- AWS_XRAY_TRACING_NAME – SDK がセグメントに使用するサービス名を設定します。サーブレットフィルタのセグメント命名ルールで設定したサービス名を上書きします。
- AWS_XRAY_DAEMON_ADDRESS –X-Ray デーモンリスナーのホストとポートを設定します。デフォルトでは、SDK は、トレースデータをに送信します127.0.0.1:2000。この変数は、デーモンを次のように構成している場合に使用します。別のポートでリッスンするまたは、別のホストで実行されている場合。
- AWS_XRAY_CONTEXT_MISSING – 計測されたコードが、セグメントが開いていないときにデータを記録しようとした場合に例外をスローするには、RUNTIME_ERROR に設定します。

  有効な値

  - RUNTIME_ERROR— ランタイム例外をスローします。
  - LOG_ERROR – エラーをログ記録して続行します (デフォルト)。
  - IGNORE_ERROR – エラーを無視して続行します。

リクエストが開かれていないときに実行されるスタートアップコード、または新しいスレッドを生成するコードで測定されたクライアントを使用しようとしたときに発生する可能性があるセグメントまたはサブセグメントの欠落に関連するエラー。

環境変数は、コードで設定される値を上書きします。

X-Ray SDK for Ruby ミドルウェアでの受信リクエストのトレーシング

X-Ray SDK を使用して、アプリケーションが Amazon EC2 の EC2 インスタンス AWS Elastic Beanstalk、または Amazon ECS で処理する受信 HTTP リクエストをトレースできます。 Amazon EC2

Rails を使用する場合は、Rails ミドルウェアを使用して、受信 HTTP リクエストを計測します。ミドルウェアをアプリケーションに追加してセグメント名を設定すると、X-Ray SDK for Ruby はサンプリングされたリクエストごとにセグメントを作成します。追加実装で作成されたセグメントは、HTTP リクエストおよびレスポンスに関する情報を提供するリクエストレベルのセグメントのサブセグメントになります。この情報には、時間、メソッド、リクエストの処理などがあります。

各セグメントには、サービスマップ内のアプリケーションを識別する名前があります。セグメントの名前は静的に指定することも、受信リクエストのホストヘッダーに基づいて動的に名前を付けるように SDK を設定することもできます。動的ネーミングでは、リクエスト内のドメイン名に基づいてトレースをグループ化でき、名前が予想されるパターンと一致しない場合 (たとえば、ホストヘッダーが偽造されている場合) 、デフォルト名を適用できます。

> **転送されたリクエスト**
>
> ロードバランサーまたは他の仲介者がアプリケーションにリクエストを転送する場合、X-Ray は、クライアントの IP をIP パケットの送信元 IP からではなく、リクエストのX-Forwarded-Forヘッダーから取得します。転送されたリクエストについて記録されたクライアント IP は偽造される可能性があるため、信頼されるべきではありません。

リクエストが転送されると、それを示す追加フィールドが SDK によってセグメントに設定されます。セグメントのフィールド `x_forwarded_for` が `true` に設定されている場合、クライアント IP が HTTP リクエストの `X-Forwarded-For` ヘッダーから取得されます。

ミドルウェアは、次の情報が含まれる `http` ブロックを使用して、各受信リクエスト用にセグメントを作成します。

- HTTP メソッド – GET、POST、PUT、DELETE、その他。
- クライアントアドレス – リクエストを送信するクライアントの IP アドレス。
- レスポンスコード – 完了したリクエストの HTTP レスポンスコード。
- タイミング – 開始時間 (リクエストが受信された時間) および終了時間 (レスポンスが送信された時間)。
- ユーザーエージェント — リクエストからのuser-agent
- コンテンツの長さ – レスポンスからの content-length。

Rails ミドルウェアの使用

ミドルウェアを使用するには、gemfile を更新して必要な railtie を含めます。

Example Gemfile - rails

```
gem 'aws-xray-sdk', require: ['aws-xray-sdk/facets/rails/railtie']
```

ミドルウェアを使用するには、トレースマップのアプリケーションを表す名前でレコーダーを設定する必要もあります。

Example config/initializers/aws_xray.rb

```
Rails.application.config.xray = {
  name: 'my app'
}
```

手動によるコードの実装

Rails を使用しない場合は、手動でセグメントを作成します。受信リクエストごとにセグメントを作成するか、パッチが適用された HTTP または AWS SDK クライアントの周囲にセグメントを作成して、レコーダーがサブセグメントを追加するためのコンテキストを提供できます。

```
# Start a segment
segment = XRay.recorder.begin_segment 'my_service'
# Start a subsegment
subsegment = XRay.recorder.begin_subsegment 'outbound_call', namespace: 'remote'

# Add metadata or annotation here if necessary
```

```
my_annotations = {
  k1: 'v1',
  k2: 1024
}
segment.annotations.update my_annotations

# Add metadata to default namespace
subsegment.metadata[:k1] = 'v1'

# Set user for the segment (subsegment is not supported)
segment.user = 'my_name'

# End segment/subsegment
XRay.recorder.end_subsegment
XRay.recorder.end_segment
```

セグメント命名ルールの設定

AWS X-Ray は、サービス名を使用してアプリケーションを識別し、アプリケーションが使用する他のアプリケーション、データベース、外部 APIs、 AWS リソースと区別します。X-Ray SDK が受信リクエストのセグメントを生成すると、アプリケーションのサービス名がセグメントの名前フィールドに記録されます。

X-Ray SDK では、HTTP リクエストヘッダーのホスト名の後にセグメントの名前を指定できます。ただし、このヘッダーは偽造され、サービスマップに予期しないノードが発生する可能性があります。偽造されたホストヘッダーを持つリクエストによって SDK がセグメントの名前を間違えないようにするには、受信リクエストのデフォルト名を指定する必要があります。

アプリケーションが複数のドメインのリクエストを処理する場合、動的ネーミングストラテジーを使用してセグメント名にこれを反映するように SDK を設定できます。動的ネーミングストラテジーにより、SDK は予想されるパターンに一致するリクエストにホスト名を使用し、そうでないリクエストにデフォルト名を適用できます。

たとえば、3 つのサブドメイン (`www.example.com`,`api.example.com`,および`static.example.com`) に対してリクエストを処理する単一のアプリケーションがあるとします。動的ネーミングストラテジーをパターン `*.example.com` で使用して、異なる名前を持つ各サブドメインのセグメントを識別することができます。結果的にはサービスマップ上に 3 つのサービスノードを作成することになります。アプリケーションがパターンと一致しないホスト名のリクエストを受信すると、指定したフォールバック名を持つ 4 番目のノードがサービスマップに表示されます。

すべてのリクエストセグメントに同じ名前を使用するには、前のセクションで示されたように、レコーダーを設定するときにアプリケーションの名前を指定します。

動的命名ルールは、ホスト名と一致するようパターンを定義し、HTTP リクエストのホスト名がパターンと一致しない場合はデフォルトの名前を使用します。セグメントに動的に名前を付けるには、config ハッシュで命名パターンを指定します。

Example main.rb – 動的命名

```
config = {
  naming_pattern: '*mydomain*',
  name: 'my app',
}

XRay.recorder.configure(config)
```

パターン内で任意の文字列に一致させるには「*」を、また、任意の 1 文字に一致させるには「?」を使用することができます。

> **Note**
>
> コードで定義したデフォルトのサービス名は、AWS_XRAY_TRACING_NAME 環境変数で上書きできます。

ダウンストリームコールを実装するためのライブラリへのパッチ適用

ダウンストリームコールを実装するには、X-Ray SDK for Ruby を使用して、アプリケーションが使用するライブラリにパッチを適用します。X-Ray SDK for Ruby では、次のライブラリにパッチを適用できます。

サポートされているライブラリ

- net/http – 実装 HTTP クライアント。
- aws-sdk – クライアントを計測 AWS SDK for Ruby します。

パッチ適用されたライブラリを使用すると、X-Ray SDK for Ruby はコールのサブセグメントを作成し、リクエストとレスポンスの情報を記録します。SDK ミドルウェアまたは

`XRay.recorder.begin_segment` のコールのいずれかから、サブセグメントを作成するために SDK でセグメントを使用できる必要があります。

ライブラリにパッチを適用するには、X-Ray レコーダーに渡す設定オブジェクトでそのパッチを指定します。

Example main.rb – パッチライブラリ

```
require 'aws-xray-sdk'

config = {
  name: 'my app',
  patch: %I[net_http aws_sdk]
}

XRay.recorder.configure(config)
```

X-Ray AWS SDK for Ruby を使用した SDK 呼び出しのトレース

アプリケーションが AWS のサービス を呼び出してデータの保存、キューへの書き込み、または通知の送信を行うと、X-Ray SDK for Ruby はサブセグメントのダウンストリームの呼び出しを追跡します。これらのサービス (Amazon S3 バケットや Amazon SQS キューなど) 内でアクセスするトレースされた AWS のサービス および リソースは、X-Ray コンソールのトレースマップにダウンストリームノードとして表示されます。

X-Ray SDK for Ruby は、ライブラリにパッチを適用すると、すべての AWS SDK クライアントを自動的に計測します。 `aws-sdk`個々のクライアントを実装することはできません。

すべてのサービスにおいて、X-Ray コンソールでコールされた API の名前を確認できます。サービスのサブセットの場合、X-Ray SDK はセグメントに情報を追加して、サービスマップでより細かく指定します。

たとえば、実装された DynamoDB クライアントでコールすると、SDK はテーブルをターゲットとするコールのセグメントにテーブル名を追加します。コンソールで、各テーブルはサービスマップ内に個別のノードとして表示され、テーブルをターゲットにしないコール用の汎用の DynamoDB ノードが表示されます。

Example 項目を保存するための DynamoDB に対するコールのサブセグメント

```
{
  "id": "24756640c0d0978a",
```

```
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "DynamoDB",
  "namespace": "aws",
  "http": {
    "response": {
      "content_length": 60,
      "status": 200
    }
  },
  "aws": {
    "table_name": "scorekeep-user",
    "operation": "UpdateItem",
    "request_id": "UBQNSO5AEM8T4FDA4RQDEB94OVTDRVV4K4HIRGVJF66Q9ASUAAJG",
  }
}
```

名前付きリソースにアクセスしたとき、次のサービスをコールすると、サービスマップに追加のノードが作成されます。特定のリソースをターゲットとしないコールでは、サービスの汎用ノードが作成されます。

- Amazon DynamoDB – テーブル名
- Amazon Simple Storage Service – バケットとキー名
- Amazon Simple Queue Service – キュー名

X-Ray SDK でのカスタムサブセグメントの生成

サブセグメントはリクエストを提供するために行われた作業の詳細を記載したトレースのセグメントを拡張します。計測済みクライアント内で呼び出しを行うたびに、X-Ray SDK によってサブセグメントに生成された情報が記録されます。追加のサブセグメントを作成して、他のサブセグメントをグループ化したり、コードセクションのパフォーマンスを測定したり、注釈とメタデータを記録したりできます。

サブセグメントを管理するには、begin_subsegment および end_subsegment メソッドを使用します。

```
subsegment = XRay.recorder.begin_subsegment name: 'annotations', namespace: 'remote'
my_annotations = { id: 12345 }
subsegment.annotations.update my_annotations
XRay.recorder.end_subsegment
```

関数のサブセグメントを作成するには、XRay.recorder.capture へのコールでラップします。

```
XRay.recorder.capture('name_for_subsegment') do |subsegment|
  resp = myfunc() # myfunc is your function
  subsegment.annotations.update k1: 'v1'
  resp
end
```

セグメントまたは別のサブセグメント内にサブセグメントを作成する場合、X-Ray SDK によってその ID が生成され、開始時刻と終了時刻が記録されます。

Example サブセグメントとメタデータ

```
"subsegments": [{
  "id": "6f1605cd8a07cb70",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "Custom subsegment for UserModel.saveUser function",
  "metadata": {
    "debug": {
      "test": "Metadata string from UserModel.saveUser"
    }
  },
```

X-Ray SDK for Ruby を使用してセグメントに注釈とメタデータを追加する

注釈やメタデータにより、リクエスト、環境、またはアプリケーションに関する追加情報を記録することができます。X-Ray SDK が作成するセグメントまたは作成するカスタムサブセグメントに、注釈およびメタデータを追加できます。

注釈は文字列、数値、またはブール値を使用したキーと値のペアです。注釈は、フィルタ式用にインデックス付けされます。注釈を使用して、コンソールでトレースをグループ化するため、または`GetTraceSummaries` API を呼び出すときに使用するデータを記録します。

メタデータは、オブジェクトとリストを含む、任意のタイプの値を持つことができるキーバリューのペアですが、フィルタ式に使用するためにインデックスは作成されません。メタデータを使用してトレースに保存する追加のデータを記録しますが、検索で使用する必要はありません。

注釈とメタデータに加えて、セグメントにユーザー ID 文字列を記録することもできます。ユーザー ID はセグメントの個別のフィールドに記録され、検索用にインデックスが作成されます。

セクション

X-Ray SDK for Ruby で注釈を記録する

注釈を使用して、検索用にインデックスを作成するセグメントまたはサブセグメントに情報を記録します。

注釈の要件

- キー – X-Ray 注釈のキーには最大 500 文字の英数字を使用できます。スペースまたはドットやピリオド (.) 以外の記号は使用できません。
- 値 – X-Ray 注釈の値には最大 1,000 の Unicode 文字を使用できます。
- 注釈の数 – トレースごとに最大 50 の注釈を使用できます。

注釈を記録するには

1. `xray_recorder` から現在のセグメントまたはサブセグメントの参照を取得します。

```
require 'aws-xray-sdk'
...
document = XRay.recorder.current_segment
```

or

```
require 'aws-xray-sdk'
...
document = XRay.recorder.current_subsegment
```

2. ハッシュ値を使って `update` を呼び出します。

```
my_annotations = { id: 12345 }
document.annotations.update my_annotations
```

以下は、ドットを含む注釈キーを使用して `update` を呼び出す方法を示す例です。

```
my_annotations = { testkey.test: 12345 }
document.annotations.update my_annotations
```

SDK は、セグメントドキュメントの annotations オブジェクトにキーと値のペアとして、注釈を記録します。同じキーで add_annotations を 2 回呼び出すと、同じセグメントまたはサブセグメントに以前記録された値が上書きされます。

特定の値を持つ注釈のあるトレースを見つけるには、annotation[*key*]フィルタ式の キーワードを使用します。

X-Ray SDK for Ruby でメタデータを記録する

メタデータを使用して、検索用にインデックスを作成する必要のないセグメントまたはサブセグメントに情報を記録します。メタデータ値は、文字列、数値、ブール値、または JSON オブジェクトや JSON 配列にシリアル化できる任意のオブジェクトになります。

メタデータを記録するには

1. xray_recorder から現在のセグメントまたはサブセグメントの参照を取得します。

   ```
   require 'aws-xray-sdk'
   ...
   document = XRay.recorder.current_segment
   ```

   or

   ```
   require 'aws-xray-sdk'
   ...
   document = XRay.recorder.current_subsegment
   ```

2. 文字列キー、ブール値、数値、文字列値、オブジェクト値、文字列名前空間を使用して metadata を呼び出します。

   ```
   my_metadata = {
     my_namespace: {
       key: 'value'
     }
   }
   subsegment.metadata my_metadata
   ```

同じキーで metadata を 2 回呼び出すと、同じセグメントまたはサブセグメントに以前記録された値が上書きされます。

X-Ray SDK for Ruby でユーザー ID を記録する

リクエストセグメントにユーザー ID を記録して、リクエストを送信したユーザーを識別します。

ユーザー ID を記録するには

1. xray_recorder から現在のセグメントへの参照を取得します。

```
require 'aws-xray-sdk'
...
document = XRay.recorder.current_segment
```

2. セグメントのユーザーフィールドを、リクエストを送信したユーザーの文字列 ID に設定します。

```
segment.user = 'U12345'
```

コントローラーでユーザーを設定し、アプリケーションがリクエストの処理を開始するとすぐに、ユーザー ID を記録できます。

ユーザー ID のトレースを見つけるには、userフィルタ式で、 キーワードを使用します。

X-Ray 計測から OpenTelemetry 計測への移行

X-Ray は、アプリケーショントレースとオブザーバビリティの主要な計測標準として OpenTelemetry (OTel) に移行しています。この戦略的シフトは、業界のベストプラクティス AWS に沿ったものであり、オブザーバビリティのニーズに応じて、より包括的で柔軟、かつ将来を見据えたソリューションを提供します。OpenTelemetry は業界で広く採用 AWS されているため、X-Ray と直接統合されない可能性のある外部のシステムなど、さまざまなシステム間でリクエストをトレースできます。

この章では、スムーズな移行に関する推奨事項を提供し、OpenTelemetry ベースのソリューションへの移行の重要性を強調して、アプリケーションの計測とオブザーバビリティにおける最新の機能を継続的にサポートし、アクセスできるようにします。

アプリケーションを計測するためのオブザーバビリティソリューションとして OpenTelemetry を採用することをお勧めします。

トピック

OpenTelemetry について

OpenTelemetry は、テレメトリデータを収集するための標準化されたプロトコルとツールを提供する業界標準のオブザーバビリティフレームワークです。メトリクス、ログ、トレースなどのテレメトリデータを計測、生成、収集、エクスポートするための統一されたアプローチを提供します。

X-Ray SDKs から OpenTelemetry に移行すると、次の利点があります。

- フレームワークとライブラリの計測サポートの強化
- 追加のプログラミング言語のサポート
- 自動計測機能
- 柔軟なサンプリング設定オプション
- メトリクス、ログ、トレースの統合コレクション

OpenTelemetry コレクターには、X-Ray デーモンよりも多くのデータ収集形式とエクスポート先オプションが用意されています。

での OpenTelemetry サポート AWS

AWS には、OpenTelemetry を操作するための複数のソリューションが用意されています。

- AWS Distro for OpenTelemetry

  OpenTelemetry トレースをセグメントとして X-Ray にエクスポートします。

  詳細については、AWS 「 Distro for OpenTelemetry」を参照してください。
- CloudWatch Application Signals

  カスタマイズされた OpenTelemetry トレースとメトリクスをエクスポートして、アプリケーションのヘルスをモニタリングします。

  詳細については、「Application Signals の使用」を参照してください。
- CloudWatch OTel エンドポイント

  HTTP OTel エンドポイントとネイティブ OpenTelemetry 計測を使用して、OpenTelemetry トレースを X-Ray にエクスポートします。

  詳細については、OTel エンドポイントの使用」を参照してください。

AWS CloudWatch での OpenTelemetry の使用

AWS CloudWatch は、クライアント側のアプリケーション計測と、Application Signals、Trace、Map、Metrics、Logs などのネイティブ AWS CloudWatch サービスを通じて OpenTelemetry トレースをサポートします。詳細については、「OpenTelemetry」を参照してください。

移行のための OpenTelemetry の概念を理解する

次の表は、X-Ray の概念を OpenTelemetry の同等の概念にマッピングしたものです。これらのマッピングを理解することで、既存の X-Ray 計測を OpenTelemetry に変換できます。

X-Ray の概念	OpenTelemetry の概念
X-Ray レコーダー	トレーサープロバイダーとトレーサー
サービスプラグイン	Resource Detector
Segment	(サーバー) スパン
サブセグメント	(サーバー以外) スパン
X-Ray サンプリングルール	OpenTelemetry サンプリング (カスタマイズ可能)
X-Ray エミッタ	Span Exporter (カスタマイズ可能)
注釈/メタデータ	属性
ライブラリの計測	ライブラリの計測
X-Ray トレースコンテキスト	スパンコンテキスト
X-Ray トレースコンテキストの伝播	W3C トレースコンテキストの伝播
X-Ray トレースサンプリング	OpenTelemetry トレースサンプリング
該当なし	スパン処理
該当なし	パーティクル
X-Ray デーモン	OpenTelemetry Collector

> **Note**
>
> OpenTelemetry の概念の詳細については、OpenTelemetry ドキュメントを参照してください。

機能の比較

次の表は、両方のサービスでサポートされている機能を示しています。この情報を使用して、移行中に対処する必要があるギャップを特定します。

機能	X-Ray 計測	OpenTelemetry 計測
ライブラリの計測	サポート	サポート
X-Ray サンプリング	サポート	OTel Java/ でサポートされています。NET/Go ADOT Java/ でサポートされています。NET/Python/Node.js
X-Ray トレースコンテキストの伝播	サポート	サポート
リソース検出	サポート	サポート
セグメント注釈	サポート	サポート
セグメントメタデータ	サポート	サポート
ゼロコード自動計測	Java でサポート	OTel Java/ でサポートされています。NET/Python/Node.js ADOT Java/ でサポートされています。NET/Python/Node.js
作成を手動でトレースする	サポート	サポート

トレースのセットアップと設定

OpenTelemetry でトレースを作成するには、トレーサーが必要です。アプリケーションで Tracer プロバイダーを初期化することで、トレーサーを取得します。これは、X-Ray レコーダーを使用して X-Ray を設定し、X-Ray トレースでセグメントとサブセグメントを作成する方法と似ています。

> **Note**
>
> OpenTelemetry トレーサープロバイダーは、X-Ray レコーダーよりも多くの設定オプションを提供します。

トレースデータ構造について

基本的な概念と機能マッピングを理解したら、トレースデータ構造やサンプリングなどの特定の実装の詳細について学習できます。

OpenTelemetry は、セグメントやサブセグメントの代わりにスパンを使用してトレースデータを構造化します。各スパンには、次のコンポーネントが含まれます。

- 名前
- 一意の ID
- 開始タイムスタンプと終了タイムスタンプ
- スパンの種類
- スパンコンテキスト
- 属性 (キーと値のメタデータ)
- イベント (タイムスタンプ付きログ)
- 他のスパンへのリンク
- ステータス情報
- 親スパンリファレンス

OpenTelemetry に移行すると、スパンは自動的に X-Ray セグメントまたはサブセグメントに変換されます。これにより、既存の CloudWatch コンソールエクスペリエンスは変更されません。

スパン属性の使用

X-Ray SDK には、セグメントとサブセグメントにデータを追加する 2 つの方法があります。

注釈

フィルタリングと検索のためにインデックス付けされるキーと値のペア

メタデータ

検索用にインデックス化されていない複雑なデータを含むキーと値のペア

デフォルトでは、OpenTelemetry スパン属性は X-Ray raw データ内のメタデータに変換されます。代わりに特定の属性を注釈に変換するには、それらのキーを`aws.xray.annotations`属性リストに追加します。

- OpenTelemetry の概念の詳細については、OpenTelemetry トレース」を参照してください。
- OpenTelemetry データが X-Ray データにどのようにマッピングされるかの詳細については、OpenTelemetry to X-Ray data model translation」を参照してください。

環境内のリソースの検出

OpenTelemetry は Resource Detectors を使用して、テレメトリデータを生成するリソースに関するメタデータを収集します。このメタデータはリソース属性として保存されます。たとえば、テレメトリを生成するエンティティは Amazon ECS クラスターまたは Amazon EC2 インスタンスであり、これらのエンティティから記録できるリソース属性には Amazon ECS クラスター ARN または Amazon EC2 インスタンス ID を含めることができます。

- サポートされているリソースタイプの詳細については、OpenTelemetry Resource Semantic Conventions」を参照してください。
- X-Ray サービスプラグインの詳細については、「X-Ray SDK の設定」を参照してください。

サンプリング戦略の管理

トレースサンプリングは、すべてのリクエストではなく、リクエストの代表的なサブセットからデータを収集することで、コストを管理するのに役立ちます。OpenTelemetry と X-Ray はどちらもサンプリングをサポートしていますが、実装は異なります。

> **Note**
>
> トレースを 100% 未満サンプリングすると、アプリケーションのパフォーマンスに関する有意義なインサイトを維持しながら、オブザーバビリティのコストを削減できます。

OpenTelemetry にはいくつかの組み込みサンプリング戦略があり、カスタム戦略を作成できます。OpenTelemetry で X-Ray サンプリングルールを使用するように、一部の SDK 言語の X-Ray リモートサンプラーを設定することもできます。

OpenTelemetry からの追加のサンプリング戦略は次のとおりです。

- 親ベースのサンプリング – 追加のサンプリング戦略を適用する前に、親スパンのサンプリング決定を尊重します
- トレース ID 比率ベースのサンプリング – >指定されたスパンの割合をランダムにサンプリングする
- テールサンプリング – サンプリングルールを適用して OpenTelemetry Collector のトレースを完了します
- カスタムサンプラー – サンプリングインターフェイスを使用して独自のサンプリングロジックを実装する

X-Ray サンプリングルールの詳細については、「X-Ray コンソールのサンプリングルール」を参照してください。

OpenTelemetry テールサンプリングの詳細については、「テールサンプリングプロセッサ」を参照してください。

トレースコンテキストの管理

X-Ray SDKs はセグメントコンテキストを管理し、トレース内のセグメントとサブセグメント間の親子関係を正しく処理します。OpenTelemetry は同様のメカニズムを使用して、スパンが正しい親スパンを持つようにします。リクエストコンテキスト全体でトレースデータを保存して伝播します。たとえば、アプリケーションがリクエストを処理し、そのリクエストを表すサーバースパンを作成すると、OpenTelemetry はサーバースパンを OpenTelemetry コンテキストに保存し、子スパンが作成されると、その子スパンはコンテキスト内のスパンを親として参照できるようにします。

トレースコンテキストの伝播

X-Ray と OpenTelemetry はどちらも HTTP ヘッダーを使用して、サービス間でトレースコンテキストを伝達します。これにより、さまざまなサービスによって生成されたトレースデータをリンクし、サンプリングの決定を維持できます。

X-Ray SDK は、X-Ray トレースヘッダーを使用してトレースコンテキストを自動的に伝播します。あるサービスが別のサービスを呼び出すと、トレースヘッダーには、トレース間の親子関係を維持するために必要なコンテキストが含まれます。

OpenTelemetry は、コンテキストの伝播用に次のような複数のトレースヘッダー形式をサポートしています。

- W3C トレースコンテキスト (デフォルト)
- X-Ray トレースヘッダー
- その他のカスタム形式

> **Note**
>
> 1 つ以上のヘッダー形式を使用するように OpenTelemetry を設定できます。たとえば、X-Ray Propagator を使用して、X-Ray トレースをサポートする AWS サービスにトレースコンテキストを送信します。

X-Ray Propagator を設定して使用し、 AWS サービス間のトレースを有効にします。これにより、トレースコンテキストを API Gateway エンドポイントおよび X-Ray をサポートする他の サービスに伝達できます。

- X-Ray トレースヘッダーの詳細については、「X-Ray デベロッパーガイド」の「トレースヘッダー」を参照してください。
- OpenTelemetry コンテキストの伝播の詳細については、OpenTelemetry ドキュメントの「コンテキストとコンテキストの伝播」を参照してください。

ライブラリ計測の使用

X-Ray と OpenTelemetry はどちらも、アプリケーションにトレースを追加するために最小限のコード変更を必要とするライブラリ計測を提供します。

X-Ray はライブラリ計測機能を提供します。これにより、アプリケーションコードの変更を最小限に抑えながら、構築済みの X-Ray 計測を追加できます。これらの計測は、 AWS SDK や HTTP クライアントなどの特定のライブラリと、Spring Boot や Express.js などのウェブフレームワークをサポートしています。

OpenTelemetry の計測ライブラリは、ライブラリフックまたは自動コード変更を通じてライブラリの詳細なスパンを生成し、コードの変更を最小限に抑えます。

OpenTelemetry のライブラリ計測がライブラリをサポートしているかどうかを確認するには、OpenTelemetry Registry の OpenTelemetry Registry でライブラリを検索します。

トレースのエクスポート

X-Ray と OpenTelemetry は、トレースデータをエクスポートするためにさまざまな方法を使用します。

X-Ray トレースのエクスポート

X-Ray SDKs はエミッタを使用してトレースデータを送信します。

- セグメントとサブセグメントを X-Ray デーモンに送信します
- ノンブロッキング I/O に UDP を使用する
- SDK でデフォルトで設定

OpenTelemetry トレースのエクスポート

OpenTelemetry は、設定可能なスパンエクスポーターを使用してトレースデータを送信します。

- http/protobuf または grpc プロトコルを使用する
- OpenTelemetry Collector または CloudWatch エージェントによってモニタリングされるエンドポイントにスパンをエクスポートする
- カスタムエクスポーター設定の許可

トレースの処理と転送

X-Ray と OpenTelemetry はどちらも、トレースデータを受信、処理、転送するためのコンポーネントを提供します。

X-Ray トレース処理

X-Ray デーモンはトレース処理を処理します。

- X-Ray SDKs

- セグメントとサブセグメントをバッチ処理する
- X-Ray サービスにバッチをアップロードします

OpenTelemetry トレース処理

OpenTelemetry Collector はトレース処理を処理します。

- 計測されたサービスからトレースを受信します
- トレースデータを処理してオプションで変更する
- X-Ray を含むさまざまなバックエンドに処理されたトレースを送信します

> **Note**
>
> AWS CloudWatch エージェントはOpenTelemetry トレースを受信して X-Ray に送信することもできます。詳細については、OpenTelemetry を使用してメトリクスとトレースを収集する」を参照してください。

スパン処理 (OpenTelemetry 固有の概念)

OpenTelemetry はスパンプロセッサを使用して、スパンの作成時にスパンを変更します。

- 作成時または完了時にスパンの読み取りと変更を許可する
- スパン処理のカスタムロジックを有効にする

孤立 (OpenTelemetry-soecific 概念)

OpenTelemetry のパーティショニング機能を使用すると、キーと値のデータを伝達できます。

- トレースコンテキストとともに任意のデータを渡すことを可能にします
- サービス境界を越えてアプリケーション固有の情報を伝達するのに役立ちます

OpenTelemetry Collector の詳細については、OpenTelemetry Collector」を参照してください。

X-Ray の概念の詳細については、「X-Ray デベロッパーガイド」の「X-Ray の概念」を参照してください。

移行の概要

このセクションでは、移行に必要なコード変更の概要を説明します。以下のリストは、言語固有のガイダンスと X-Ray デーモンの移行手順です。

> **Important**
>
> X-Ray 計測から OpenTelemetry 計測に完全に移行するには、以下が必要です。
>
> 1. X-Ray SDK の使用を OpenTelemetry ソリューションに置き換える
> 2. X-Ray デーモンを CloudWatch エージェントまたは OpenTelemetry Collector (X-Ray Exporter) に置き換えます。

- OpenTelemetry Java への移行
- OpenTelemetry Go への移行
- OpenTelemetry Node.js への移行
- OpenTelemetry .NET への移行
- OpenTelemetry Python への移行
- OpenTelemetry Ruby への移行

新規および既存のアプリケーションの推奨事項

新規および既存のアプリケーションでは、次のソリューションを使用してアプリケーションでトレースを有効にすることをお勧めします。

インストルメンテーション

- OpenTelemetry SDKs
- AWS Distro for OpenTelemetry Instrumentation

データ収集

- OpenTelemetry Collector
- CloudWatch エージェント

OpenTelemetry ベースのソリューションに移行した後も、CloudWatch エクスペリエンスは変わりません。CloudWatch コンソールのトレースページとトレースマップページでトレースを同じ形式で表示したり、X-Ray APIs を使用してトレースデータを取得したりできます。

セットアップ変更のトレース

X-Ray セットアップを OpenTelemetry セットアップに置き換える必要があります。

X-Ray と OpenTelemetry の設定の比較

機能	X-Ray SDK	OpenTelemetry
デフォルト設定	• X-Ray 集中サンプリング • X-Ray トレースコンテキストの伝播 • X-Ray デーモンへのトレースエクスポート	• OpenTelemetry Collector または CloudWatch エージェントへのトレースのエクスポート (HTTP/gRPC) • W3C トレースコンテキストの伝播
手動設定	• ローカルサンプリングルール • リソース検出プラグイン	• X-Ray サンプリング (一部の言語では使用できない場合があります) • リソース検出 • X-Ray トレースコンテキストの伝播

ライブラリ計測の変更

AWS SDK、HTTP クライアント、ウェブフレームワーク、およびその他のライブラリの X-Ray ライブラリ計測の代わりに OpenTelemetry Library 計測を使用するようにコードを更新します。これにより、X-Ray トレースの代わりに OpenTelemetry トレースが生成されます。

> **Note**
>
> コードの変更は言語とライブラリによって異なります。詳細な手順については、言語固有の移行ガイドを参照してください。

Lambda 環境計測の変更

Lambda 関数で OpenTelemetry を使用するには、次のいずれかの設定オプションを選択します。

1. 自動計測 Lambda レイヤーを使用します。
   - （推奨) CloudWatch Application Signals Lambda レイヤー

     > **Note**
     >
     > トレースのみを使用するには、Lambda 環境変数 を設定しますOTEL_AWS_APPLICATION_SIGNALS_ENABLED=false。

   - AWS ADOT 用 マネージド Lambda Layer
2. Lambda 関数の OpenTelemetry を手動でセットアップします。
   - X-Ray UDP Span Exporter を使用してシンプルなスパンプロセッサを設定する
   - X-Ray Lambda プロパゲーターをセットアップする

トレースデータを手動で作成する

X-Ray セグメントとサブセグメントを OpenTelemetry スパンに置き換えます。

- OpenTelemetry トレーサーを使用してスパンを作成する
- スパンに属性を追加する (X-Ray メタデータと注釈に相当)

> **Important**
>
> X-Ray に送信した場合：
>
> - サーバースパンを X-Ray セグメントに変換する
> - その他のスパンは X-Ray サブセグメントに変換されます
> - 属性はデフォルトでメタデータに変換されます

属性を注釈に変換するには、そのキーをaws.xray.annotations属性リストに追加します。詳細については、「カスタマイズされた X-Ray 注釈を有効にする」を参照してください。

X-Ray デーモンから AWS CloudWatch エージェントまたは OpenTelemetry コレクターへの移行

CloudWatch エージェントまたは OpenTelemetry コレクターを使用して、計測されたアプリケーションからトレースを受信し、X-Ray に送信できます。

> **Note**
>
> CloudWatch エージェントバージョン 1.300025.0 以降では、OpenTelemetry トレースを収集できます。X-Ray デーモンの代わりに CloudWatch エージェントを使用すると、管理する必要があるエージェントの数が減ります。詳細については、CloudWatch エージェントを使用したメトリクス、ログ、トレースの収集」を参照してください。

セクション

Amazon EC2 またはオンプレミスサーバーでの移行

> **Important**
>
> CloudWatch エージェントまたは OpenTelemetry コレクターを使用してポートの競合を防ぐ前に、X-Ray デーモンプロセスを停止します。

既存の X-Ray デーモンのセットアップ

デーモンのインストール

既存の X-Ray デーモンの使用法は、次のいずれかの方法を使用してインストールされました。

手動インストール

X-Ray デーモン Amazon S3 バケットから実行可能ファイルをダウンロードして実行します。

自動インストール

このスクリプトを使用して、インスタンスの起動時に デーモンをインストールします。

```
#!/bin/bash
curl https://s3.us-east-2.amazonaws.com/aws-xray-assets.us-east-2/xray-daemon/aws-xray-daemon-3.x.rpm \
    -o /home/ec2-user/xray.rpm
yum install -y /home/ec2-user/xray.rpm
```

デーモンを設定する

既存の X-Ray デーモンの使用法は、次のいずれかを使用して設定されました。

- コマンドライン引数
- 設定ファイル (`xray-daemon.yaml`)

Example 設定ファイルを使用する

```
./xray -c ~/xray-daemon.yaml
```

デーモンを実行する

既存の X-Ray デーモンの使用が次のコマンドで開始されました。

```
~/xray-daemon$ ./xray -o -n us-east-1
```

デーモンの削除

Amazon EC2 インスタンスから X-Ray デーモンを削除するには :

1. デーモンサービスを停止します。

   ```
   systemctl stop xray
   ```

2. 設定ファイルを削除します。

   ```
   rm ~/path/to/xray-daemon.yaml
   ```

3. 設定されている場合は、ログファイルを削除します。

> **Note**
>
> ログファイルの場所は、設定によって異なります。
>
> - コマンドライン設定: `/var/log/xray-daemon.log`
> - 設定ファイル: `LogPath`設定を確認する

CloudWatch エージェントのセットアップ

エージェントのインストール

インストール手順については、「オンプレミスサーバーへの CloudWatch エージェントのインストール」を参照してください。

エージェントの設定

1. トレース収集を有効にする設定ファイルを作成します。詳細については、CloudWatch エージェント設定ファイルの作成」を参照してください。
2. IAM アクセス許可を設定します。
   - IAM ロールをアタッチするか、エージェントの認証情報を指定します。詳細については、「IAM ロールのセットアップ」を参照してください。
   - ロールまたは認証情報に アクセス`xray:PutTraceSegments`許可が含まれていることを確認します。

エージェントを開始する

エージェントを起動する手順については、「 コマンドラインを使用した CloudWatch エージェントの開始」を参照してください。

OpenTelemetry コレクターのセットアップ

コレクターのインストール

オペレーティングシステムの OpenTelemetry コレクターをダウンロードしてインストールします。手順については、「コレクターのインストール」を参照してください。

コレクターの設定

コレクターで次のコンポーネントを設定します。

- awsproxy 拡張機能

  X-Ray サンプリングに必要です

- OTel レシーバー

  アプリケーションからトレースを収集します

- xray エクスポーター

  X-Ray にトレースを送信する

Example サンプルコレクター設定 — otel-collector-config.yaml

```
extensions:
  awsproxy:
    endpoint: 127.0.0.1:2000
  health_check:

receivers:
  otlp:
    protocols:
      grpc:
        endpoint: 127.0.0.1:4317
      http:
        endpoint: 127.0.0.1:4318

processors:
  batch:

exporters:
  awsxray:
    region: 'us-east-1'

service:
  pipelines:
    traces:
      receivers: [otlp]
      exporters: [awsxray]
  extensions: [awsproxy, health_check]
```

> **Important**
>
> アクセス`xray:PutTraceSegments`許可を使用して AWS 認証情報を設定します。詳細については、「認証情報の指定」を参照してください。

コレクターの起動

設定ファイルを使用してコレクターを実行します。

```
otelcol --config=otel-collector-config.yaml
```

Amazon ECS での移行

> **Important**
>
> タスクロールには、使用するコレクターに対する アクセス`xray:PutTraceSegments`許可が必要です。
> ポートの競合を防ぐために、同じホストで CloudWatch エージェントまたは OpenTelemetry コレクターコンテナを実行する前に、既存の X-Ray デーモンコンテナを停止します。

CloudWatch エージェントの使用

1. Amazon ECR Public Gallery から Docker イメージを取得します。
2. という名前の設定ファイルを作成します`cw-agent-otel.json`。

```
{
  "traces": {
    "traces_collected": {
      "xray": {
        "tcp_proxy": {
          "bind_address": "0.0.0.0:2000"
        }
      },
      "otlp": {
        "grpc_endpoint": "0.0.0.0:4317",
        "http_endpoint": "0.0.0.0:4318"
      }
```

```
    }
  }
}
```

3. 設定を Systems Manager パラメータストアに保存します。

   1. https://console.aws.amazon.com/systems-manager/ を開きます。
   2. Create パラメータを選択する
   3. 次の値を入力します。
      - 名前: `/ecs/cwagent/otel-config`
      - 階層: 標準
      - タイプ: 文字列
      - データ型: テキスト
      - 値: [ここに cw-agent-otel.json 設定を貼り付けます〕

4. ブリッジネットワークモードを使用してタスク定義を作成します。

   タスク定義では、使用するネットワーキングモードによって設定が異なります。デフォルトはブリッジネットワーキングで、デフォルトの VPC で使用できます。ブリッジネットワークで、`OTEL_EXPORTER_OTLP_TRACES_ENDPOINT`環境変数を設定して、CloudWatch エージェント用のエンドポイントとポートを OpenTelemetry SDK に指示します。また、アプリケーションの OpenTelemetry SDK から Collector コンテナにトレースを送信するには、アプリケーションコンテナから Collector コンテナへのリンクを作成する必要があります。

   Example CloudWatch エージェントタスク定義

```
{
    "containerDefinitions": [
        {
            "name": "cwagent",
            "image": "public.ecr.aws/cloudwatch-agent/cloudwatch-agent:latest",
            "portMappings": [
                {
                    "containerPort": 4318,
                    "hostPort": 4318,
                    "protocol": "tcp"
                },
                {
                    "containerPort": 4317,
                    "hostPort": 4317,
```

```
                    "protocol": "tcp"
                },
                {
                    "containerPort": 2000,
                    "hostPort": 2000,
                    "protocol": "tcp"
                }
            ],
            "secrets": [
                {
                    "name": "CW_CONFIG_CONTENT",
                    "valueFrom": "/ecs/cwagent/otel-config"
                }
            ]
        },
        {
            "name": "application",
            "image": "APPLICATION_IMAGE",
            "links": ["cwagent"],
            "environment": [
                {
                    "name": "OTEL_EXPORTER_OTLP_TRACES_ENDPOINT",
                    "value": "http://cwagent:4318/v1/traces"
                }
            ]
        }
    ]
}
```

詳細については、CloudWatch エージェントをデプロイして Amazon ECS で Amazon EC2 インスタンスレベルのメトリクスを収集する」を参照してください。

OpenTelemetry コレクターの使用

1. Docker Hub `otel/opentelemetry-collector-contrib`から Docker イメージを取得します。
2. Amazon EC2 コレクターを設定するセクションで示しているのと同じコンテンツ`otel-collector-config.yaml`を使用して という設定ファイルを作成しますが、 `0.0.0.0`の代わりに を使用するようにエンドポイントを更新します`127.0.0.1`。

3. Amazon ECS でこの設定を使用するには、Systems Manager パラメータストアに設定を保存できます。まず、Systems Manager パラメータストアコンソールに移動し、新しいパラメータの作成を選択します。次の情報を使用して新しいパラメータを作成します。
   - 名前: /ecs/otel/config (この名前はコレクターのタスク定義で参照されます)
   - 階層: 標準
   - タイプ: 文字列
   - データ型: テキスト
   - 値: [otel-collector-config.yaml 設定をここに貼り付ける〕
4. 例としてブリッジネットワークモードを使用して OpenTelemetry コレクターをデプロイするタスク定義を作成します。

   タスク定義では、設定は使用するネットワークモードによって異なります。デフォルトはブリッジネットワーキングで、デフォルトの VPC で使用できます。ブリッジネットワークで、OpenTelemetry Collector のエンドポイントとポートを OpenTelemetry SDK に指示するようにOTEL_EXPORTER_OTLP_TRACES_ENDPOINT環境変数を設定します。また、アプリケーションの OpenTelemetry SDK から Collector コンテナにトレースを送信するには、アプリケーションコンテナから Collector コンテナへのリンクを作成する必要があります。

   Example OpenTelemetry コレクタータスク定義

```
{
    "containerDefinitions": [
        {
            "name": "otel-collector",
            "image": "otel/opentelemetry-collector-contrib",
            "portMappings": [
                {
                    "containerPort": 2000,
                    "hostPort": 2000
                },
                {
                    "containerPort": 4317,
                    "hostPort": 4317
                },
                {
                    "containerPort": 4318,
                    "hostPort": 4318
                }
            ],
```

```
            "command": [
                "--config",
                "env:SSM_CONFIG"
            ],
            "secrets": [
                {
                    "name": "SSM_CONFIG",
                    "valueFrom": "/ecs/otel/config"
                }
            ]
        },
        {
            "name": "application",
            "image": "APPLICATION_IMAGE",
            "links": ["otel-collector"],
            "environment": [
                {
                    "name": "OTEL_EXPORTER_OTLP_TRACES_ENDPOINT",
                    "value": "http://otel-collector:4318/v1/traces"
                }
            ]
        }
    ]
}
```

Elastic Beanstalk での移行

> **Important**
>
> CloudWatch エージェントを使用してポートの競合を防ぐ前に、X-Ray デーモンプロセスを停止します。

既存の X-Ray デーモン統合は、Elastic Beanstalk コンソールを使用するか、設定ファイルを使用してアプリケーションソースコードで X-Ray デーモンを設定することで有効になりました。

CloudWatch エージェントの使用

Amazon Linux 2 プラットフォームで、設定ファイルを使用して CloudWatch エージェント.ebextensionsを設定します。

1. プロジェクトルート.ebextensionsに という名前のディレクトリを作成する
2. 次の内容で、 .ebextensions ディレクトリcloudwatch.config内に という名前のファイルを作成します。

```
files:
  "/opt/aws/amazon-cloudwatch-agent/etc/config.json":
    mode: "0644"
    owner: root
    group: root
    content: |
      {
        "traces": {
          "traces_collected": {
            "otlp": {
              "grpc_endpoint": "12.0.0.1:4317",
              "http_endpoint": "12.0.0.1:4318"
            }
          }
        }
      }
container_commands:
  start_agent:
    command: /opt/aws/amazon-cloudwatch-agent/bin/amazon-cloudwatch-agent-ctl -a
 append-config -c file:/opt/aws/amazon-cloudwatch-agent/etc/config.json -s
```

3. デプロイ時にアプリケーションソースバンドルに .ebextensions ディレクトリを含める

Elastic Beanstalk 設定ファイルの詳細については、「設定ファイルを使用した高度な環境のカスタマイズ」を参照してください。

OpenTelemetry Java への移行

このセクションでは、X-Ray SDK から OpenTelemetry SDK for Java アプリケーションへの移行に関するガイダンスを提供します。

セクション

- ゼロコード自動計測ソリューション
- SDK を使用した手動計測ソリューション
- 受信リクエストのトレース (スプリングフレームワークの計測)

- AWS SDK v2 計測
- 送信 HTTP 呼び出しの計測
- 他のライブラリの計測サポート
- トレースデータを手動で作成する
- Lambda 計測

ゼロコード自動計測ソリューション

With X-Ray Java agent

X-Ray Java エージェントを有効にするには、アプリケーションの JVM 引数を変更する必要があります。

```
-javaagent:/path-to-disco/disco-java-agent.jar=pluginPath=/path-to-disco/disco-plugins
```

With OpenTelemetry-based Java agent

OpenTelemetry ベースの Java エージェントを使用するには。

- AWS Distro for OpenTelemetry (ADOT) Auto-Instrumentation Java エージェントを使用して、ADOT Java エージェントによる自動計測を行います。詳細については、「Java エージェントを使用したトレースとメトリクスの自動計測」を参照してください。トレースのみが必要な場合は、 `OTEL_METRICS_EXPORTER=none` 環境変数を無効にします。 は Java エージェントからメトリクスをエクスポートします。

  (オプション) ADOT Java 自動計測 AWS を使用して でアプリケーションを自動的に計測するときに CloudWatch Application Signals を有効にして、現在のアプリケーションの状態をモニタリングし、長期的なアプリケーションパフォーマンスを追跡することもできます。Application Signals は、アプリケーション、サービス、依存関係の統合されたアプリケーション中心のビューを提供し、アプリケーションのヘルスをモニタリングおよびトリアージするのに役立ちます。詳細については、「Application Signals」を参照してください。
- 自動計測には OpenTelemetry Java エージェントを使用します。詳細については、「Java エージェントによるゼロコード計測」を参照してください。

SDK を使用した手動計測ソリューション

Tracing setup with X-Ray SDK

X-Ray SDK for Java を使用してコードを計測するには、まず AWSXRay クラスをサービスプラグインとローカルサンプリングルールで設定する必要があり、次に提供されたレコーダーが使用されました。

```
static { AWS XRayRecorderBuilder builder = AWS
 XRayRecorderBuilder.standard().withPlugin(new EC2Plugin()).withPlugin(new
 ECSPlugin()); AWS XRay.setGlobalRecorder(builder.build());
}
```

Tracing setup with OpenTelemetry SDK

次の依存関係が必要です。

```
<dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>io.opentelemetry</groupId>
                <artifactId>opentelemetry-bom</artifactId>
                <version>1.49.0</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
            <dependency>
                <groupId>io.opentelemetry.instrumentation</groupId>
                <artifactId>opentelemetry-instrumentation-bom</artifactId>
                <version>2.15.0</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>io.opentelemetry</groupId>
            <artifactId>opentelemetry-sdk</artifactId>
        </dependency>
        <dependency>
            <groupId>io.opentelemetry</groupId>
            <artifactId>opentelemetry-api</artifactId>
```

```
        </dependency>
        <dependency>
            <groupId>io.opentelemetry.semconv</groupId>
            <artifactId>opentelemetry-semconv</artifactId>
        </dependency>
        <dependency>
            <groupId>io.opentelemetry</groupId>
            <artifactId>opentelemetry-exporter-otlp</artifactId>
        </dependency>
        <dependency>
            <groupId>io.opentelemetry.contrib</groupId>
            <artifactId>opentelemetry-aws-xray</artifactId>
            <version>1.46.0</version>
        </dependency>
        <dependency>
            <groupId>io.opentelemetry.contrib</groupId>
            <artifactId>opentelemetry-aws-xray-propagator</artifactId>
            <version>1.46.0-alpha</version>
        </dependency>
        <dependency>
            <groupId>io.opentelemetry.contrib</groupId>
            <artifactId>opentelemetry-aws-resources</artifactId>
            <version>1.46.0-alpha</version>
        </dependency>
    </dependencies>
```

をインスタンス化し、OpenTelemetrySdkオブジェクトをTracerProviderグローバルに登録して、OpenTelemetry SDK を設定します。これらのコンポーネントを設定します。

- OTLP Span Exporter (OtlpGrpcSpanExporter など) - CloudWatch エージェントまたは OpenTelemetry Collector にトレースをエクスポートするために必要です
- AWS X-Ray プロパゲーター – X-Ray と統合された AWS サービスにトレースコンテキストを伝達するために必要です
- AWS X-Ray リモートサンプラー – X-Ray サンプリングルールを使用してリクエストをサンプリングする必要がある場合に必要です
- Resource Detectors (EcsResource や Ec2Resource など) – アプリケーションを実行しているホストのメタデータを検出する

```
import io.opentelemetry.api.common.Attributes;
import io.opentelemetry.context.propagation.ContextPropagators;
import io.opentelemetry.contrib.aws.resource.Ec2Resource;
```

```
import io.opentelemetry.contrib.aws.resource.EcsResource;
import io.opentelemetry.contrib.awsxray.AwsXrayRemoteSampler;
import io.opentelemetry.contrib.awsxray.propagator.AwsXrayPropagator;
import io.opentelemetry.exporter.otlp.trace.OtlpGrpcSpanExporter;
import io.opentelemetry.sdk.OpenTelemetrySdk;
import io.opentelemetry.sdk.resources.Resource;
import io.opentelemetry.sdk.trace.SdkTracerProvider;
import io.opentelemetry.sdk.trace.export.BatchSpanProcessor;
import io.opentelemetry.sdk.trace.samplers.Sampler;
import static io.opentelemetry.semconv.ServiceAttributes.SERVICE_NAME;

// ...

    private static final Resource otelResource =
        Resource.create(Attributes.of(SERVICE_NAME, "YOUR_SERVICE_NAME"))
            .merge(EcsResource.get())
            .merge(Ec2Resource.get());
    private static final SdkTracerProvider sdkTracerProvider =
        SdkTracerProvider.builder()
            .addSpanProcessor(BatchSpanProcessor.create(
                OtlpGrpcSpanExporter.getDefault()
            ))
            .addResource(otelResource)
            .setSampler(Sampler.parentBased(
                AwsXrayRemoteSampler.newBuilder(otelResource).build()
            ))
            .build();
    // Globally registering a TracerProvider makes it available throughout the
 application to create as many Tracers as needed.
    private static final OpenTelemetrySdk openTelemetry =
        OpenTelemetrySdk.builder()
            .setTracerProvider(sdkTracerProvider)

 .setPropagators(ContextPropagators.create(AwsXrayPropagator.getInstance()))
            .buildAndRegisterGlobal();
```

受信リクエストのトレース (スプリングフレームワークの計測)

With X-Ray SDK

Spring フレームワークで X-Ray SDK を使用してアプリケーションを計測する方法については、「Spring での AOP」と「X-Ray SDK for Java」を参照してください。Spring で AOP を有効にするには、以下の手順を実行します。

1. Spring を設定します
2. トレースフィルターをアプリケーションに追加する
3. コードに注釈を付けるか、 インターフェイスを実装する
4. アプリケーションで X-Ray を有効化します

With OpenTelemetry SDK

OpenTelemetry は、Spring Boot アプリケーションの受信リクエストのトレースを収集するための計測ライブラリを提供します。最小限の設定で Spring Boot 計測を有効にするには、次の依存関係を含めます。

```
<dependency>
            <groupId>io.opentelemetry.instrumentation</groupId>
             <artifactId>opentelemetry-spring-boot-starter</artifactId>
        </dependency>
```

OpenTelemetry のセットアップで Spring Boot 計測を有効にして設定する方法の詳細については、OpenTelemetry の開始方法」を参照してください。

Using OpenTelemetry-based Java agents

Spring Boot アプリケーションを計測するためのデフォルトの推奨方法は、OpenTelemetry Java エージェントとバイトコード計測を使用することです。これにより、SDK を直接使用するよりもout-of-the-box使用できる計測と設定も増えます。開始方法については、「」を参照してくださいゼロコード自動計測ソリューション。

AWS SDK v2 計測

With X-Ray SDK

X-Ray SDK for Java は、ビルドにaws-xray-recorder-sdk-aws-sdk-v2-instrumentorサブモジュールを追加したときに、すべての AWS SDK v2 クライアントを自動的に計測できます。

AWS SDK for Java 2.2 以降 AWS で個々のクライアントのダウンストリームクライアント呼び出しを計測するために、ビルド設定の aws-xray-recorder-sdk-aws-sdk-v2-instrumentor モジュールが除外され、 aws-xray-recorder-sdk-aws-sdk-v2モジュールが含まれていました。個々のクライアントは、 で設定することで計測されましたTracingInterceptor。

```
import com.amazonaws.xray.interceptors.TracingInterceptor;
import software.amazon.awssdk.core.client.config.ClientOverrideConfiguration
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
//...

public class MyModel {
  private DynamoDbClient client = DynamoDbClient.builder()
    .region(Region.US_WEST_2)
    .overrideConfiguration(
      ClientOverrideConfiguration.builder()
        .addExecutionInterceptor(new TracingInterceptor())
        .build()
      )
    .build();
//...
```

With OpenTelemetry SDK

すべての AWS SDK クライアントを自動的に計測するには、 opentelemetry-aws-sdk-2.2-autoconfigureサブモジュールを追加します。

```
<dependency>
            <groupId>io.opentelemetry.instrumentation</groupId>
            <artifactId>opentelemetry-aws-sdk-2.2-autoconfigure</artifactId>
            <version>2.15.0-alpha</version>
            <scope>runtime</scope>
        </dependency>
```

個々の AWS SDK クライアントを計測するには、 opentelemetry-aws-sdk-2.2サブモジュールを追加します。

```
<dependency>
            <groupId>io.opentelemetry.instrumentation</groupId>
            <artifactId>opentelemetry-aws-sdk-2.2</artifactId>
            <version>2.15.0-alpha</version>
            <scope>compile</scope>
        </dependency>
```

次に、 AWS SDK クライアントを作成するときにインターセプターを登録します。

```
import io.opentelemetry.instrumentation.awssdk.v2_2.AwsSdkTelemetry;

// ...

    AwsSdkTelemetry telemetry = AwsSdkTelemetry.create(openTelemetry);
    private final S3Client S3_CLIENT = S3Client.builder()
      .overrideConfiguration(ClientOverrideConfiguration.builder()
        .addExecutionInterceptor(telemetry.newExecutionInterceptor())
        .build())
      .build();
```

送信 HTTP 呼び出しの計測

With X-Ray SDK

X-Ray を使用して送信 HTTP リクエストを計測するには、X-Ray SDK for Java のバージョンの Apache HttpClient が必要です。

```
import com.amazonaws.xray.proxies.apache.http.HttpClientBuilder;
...
  public String randomName() throws IOException {
    CloseableHttpClient httpclient = HttpClientBuilder.create().build();
```

With OpenTelemetry SDK

X-Ray Java SDK と同様に、OpenTelemetry は、Apache HttpClient に OpenTelemetry ベースのスパンとコンテキスト伝達を提供する `HttpClientBuilder` のインスタンスの作成を許可するビルダーメソッドを持つ `ApacheHttpClientTelemetry` クラスを提供します。

```
<dependency>
```

```
            <groupId>io.opentelemetry.instrumentation</groupId>
            <artifactId>opentelemetry-apache-httpclient-5.2</artifactId>
            <version>2.15.0-alpha</version>
            <scope>compile</scope>
        </dependency>
```

opentelemetry-java-instrumentation のコード例を次に示します。newHttpClient() によって提供される HTTP クライアントは、実行されたリクエストのトレースを生成します。

```
import io.opentelemetry.api.OpenTelemetry;
import
 io.opentelemetry.instrumentation.apachehttpclient.v5_2.ApacheHttpClientTelemetry;
import org.apache.hc.client5.http.classic.HttpClient;
import org.apache.hc.client5.http.impl.classic.HttpClientBuilder;

public class ApacheHttpClientConfiguration {

  private OpenTelemetry openTelemetry;

  public ApacheHttpClientConfiguration(OpenTelemetry openTelemetry) {
    this.openTelemetry = openTelemetry;
  }

  // creates a new http client builder for constructing http clients with open
 telemetry instrumentation
  public HttpClientBuilder createBuilder() {
    return
 ApacheHttpClientTelemetry.builder(openTelemetry).build().newHttpClientBuilder();
  }

  // creates a new http client with open telemetry instrumentation
  public HttpClient newHttpClient() {
    return ApacheHttpClientTelemetry.builder(openTelemetry).build().newHttpClient();
  }
}
```

他のライブラリの計測サポート

OpenTelemetry Java でサポートされているライブラリ計測の完全なリストは、サポートされているライブラリ、フレームワーク、アプリケーションサーバー、および JVMs の下にある、それぞれの計測 GitHub リポジトリ にあります。

または、OpenTelemetry Registry を検索して、OpenTelemetry が計測をサポートしているかどうかを調べることもできます。検索を開始するには、「レジストリ」を参照してください。

トレースデータを手動で作成する

With X-Ray SDK

X-Ray SDK では、X-Ray セグメントbeginSegmentとサブセグメントを手動で作成するには、メソッドと beginSubsegmentメソッドが必要です。

```
 Segment segment = xrayRecorder.beginSegment("ManualSegment");
       segment.putAnnotation("annotationKey", "annotationValue");
       segment.putMetadata("metadataKey", "metadataValue");

       try {
           Subsegment subsegment =
xrayRecorder.beginSubsegment("ManualSubsegment");
           subsegment.putAnnotation("key", "value");

           // Do something here

       } catch (Exception e) {
           subsegment.addException(e);
       } finally {
           xrayRecorder.endSegment();
       }
```

With OpenTelemetry SDK

カスタムスパンを使用して、計測ライブラリでキャプチャされない内部アクティビティのパフォーマンスをモニタリングできます。スパンの種類サーバーのみが X-Ray セグメントに変換され、他のすべてのスパンは X-Ray サブセグメントに変換されることに注意してください。

まず、 openTelemetry.getTracerメソッドで取得できるスパンを生成するには、トレーサーを作成する必要があります。これにより、このSDK を使用した手動計測ソリューション例でグローバルに登録TracerProviderされた の Tracer インスタンスが提供されます。必要に応じていくつでも Tracer インスタンスを作成できますが、アプリケーション全体に 1 つの Tracer を持つのが一般的です。

```
Tracer tracer = openTelemetry.getTracer("my-app");
```

Tracer を使用してスパンを作成できます。

```
import io.opentelemetry.api.common.AttributeKey;
import io.opentelemetry.api.trace.Span;
import io.opentelemetry.api.trace.SpanKind;
import io.opentelemetry.api.trace.Tracer;
import io.opentelemetry.context.Scope;

...

// SERVER span will become an X-Ray segment
Span span = tracer.spanBuilder("get-token")
  .setKind(SpanKind.SERVER)
  .setAttribute("key", "value")
  .startSpan();
try (Scope ignored = span.makeCurrent()) {

  span.setAttribute("metadataKey", "metadataValue");
  span.setAttribute("annotationKey", "annotationValue");

  // The following ensures that "annotationKey: annotationValue" is an annotation in
 X-Ray raw data.
  span.setAttribute(AttributeKey.stringArrayKey("aws.xray.annotations"),
 List.of("annotationKey"));

  // Do something here
}

span.end();
```

スパンのデフォルトタイプは INTERNAL です。

```
// Default span of type INTERNAL will become an X-Ray subsegment
Span span = tracer.spanBuilder("process-header")
  .startSpan();
try (Scope ignored = span.makeCurrent()) {
  doProcessHeader();
}
```

OpenTelemetry SDK を使用してトレースに注釈とメタデータを追加する

上記の例では、 `setAttribute`メソッドを使用して各スパンに属性を追加します。デフォルトでは、すべてのスパン属性は X-Ray raw データ内のメタデータに変換されま

す。属性がメタデータではなく注釈に変換されるように、上記の例では、その属性のキーをaws.xray.annotations属性のリストに追加します。詳細については、「カスタマイズされた X-Ray 注釈と注釈とメタデータを有効にする」を参照してください。 https://docs.aws.amazon.com/xray/latest/devguide/xray-concepts.html#xray-concepts-annotations

OpenTelemetry ベースの Java エージェントを使用

Java エージェントを使用してアプリケーションを自動的に計測する場合は、アプリケーションで手動計測を実行する必要があります。たとえば、自動計測ライブラリでカバーされていないセクションのコードをアプリケーション内で計測するには。

エージェントで手動計測を実行するには、 opentelemetry-api アーティファクトを使用する必要があります。アーティファクトバージョンをエージェントバージョンより新しいものにすることはできません。

```
import io.opentelemetry.api.GlobalOpenTelemetry;
import io.opentelemetry.api.trace.Span;

// ...

        Span parentSpan = Span.current();
        Tracer tracer = GlobalOpenTelemetry.getTracer("my-app");
        Span span = tracer.spanBuilder("my-span-name")
            .setParent(io.opentelemetry.context.Context.current().with(parentSpan))
            .startSpan();
        span.end();
```

Lambda 計測

With X-Ray SDK

X-Ray SDK を使用すると、Lambda でアクティブトレースを有効にした後、X-Ray SDK を使用するための追加の設定は必要ありません。Lambda は Lambda ハンドラー呼び出しを表すセグメントを作成し、追加の設定なしで X-Ray SDK を使用してサブセグメントまたは計測ライブラリを作成できます。

With OpenTelemetry-based solutions

自動計測 Lambda レイヤー – 次のソリューションを使用して AWS 、販売された Lambda レイヤーを使用して Lambda を自動的に計測できます。

- CloudWatch Application Signals Lambda レイヤー (推奨)

  > **Note**
  >
  > この Lambda レイヤーでは、デフォルトで CloudWatch Application Signals が有効になっているため、メトリクスとトレースの両方を収集することで、Lambda アプリケーションのパフォーマンスとヘルスのモニタリングが可能になります。トレースのみの場合は、Lambda 環境変数 を設定します `OTEL_AWS_APPLICATION_SIGNALS_ENABLED=false`。

  - Lambda アプリケーションのパフォーマンスとヘルスのモニタリングを有効にします
  - デフォルトでは、メトリクスとトレースの両方を収集します
- AWS ADOT Java 用の マネージド Lambda レイヤー。詳細については、AWS 「 Distro for OpenTelemetry Lambda Support for Java」を参照してください。

自動計測レイヤーとともに手動計測を使用するには、「」を参照してくださいSDK を使用した手動計測ソリューション。コールドスタートを減らすには、OpenTelemetry 手動計測を使用して Lambda 関数の OpenTelemetry トレースを生成することを検討してください。

AWS Lambda の OpenTelemetry 手動計測

Amazon S3 ListBuckets 呼び出しを行う次の Lambda 関数コードを検討してください。

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.s3.model.ListBucketsRequest;
import software.amazon.awssdk.services.s3.model.ListBucketsResponse;
import software.amazon.awssdk.services.s3.model.S3Exception;

public class ListBucketsLambda implements RequestHandler<String, String> {

    private final S3Client S3_CLIENT = S3Client.builder()
        .build();
```

```
    @Override
    public String handleRequest(String input, Context context) {
        try {
            ListBucketsResponse response = makeListBucketsCall();
            context.getLogger().log("response: " + response.toString());
            return "Success";
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    private ListBucketsResponse makeListBucketsCall() {
        try {
            ListBucketsRequest listBucketsRequest = ListBucketsRequest.builder()
                .build();
            ListBucketsResponse response = S3_CLIENT.listBuckets(listBucketsRequest);
            return response;
        } catch (S3Exception e) {
            throw new RuntimeException("Failed to call S3 listBuckets" +
 e.awsErrorDetails().errorMessage(), e);
        }
    }
}
```

依存関係は次のとおりです。

```
dependencies {
    implementation('com.amazonaws:aws-lambda-java-core:1.2.3')
    implementation('software.amazon.awssdk:s3:2.28.29')
    implementation('org.slf4j:slf4j-nop:2.0.16')
}
```

Lambda ハンドラーと Amazon S3 クライアントを手動で計測するには、次の手順を実行します。

1. （`RequestHandler`または RequestStreamHandler) を実装する関数クラスを、拡張する関数クラス `TracingRequestHandler` (または TracingRequestStreamHandler) に置き換えます。
2. TracerProvider をインスタンス化し、OpenTelemetrySdk オブジェクトをグローバルに登録します。TracerProvider は、以下を使用して設定することをお勧めします。

   a. Lambda の UDP X-Ray エンドポイントにトレースを送信するための X-Ray UDP スパンエクスポーターを備えたシンプルなスパンプロセッサ

   b. ParentBased always on sampler (設定されていない場合はデフォルト)
   c. service.name が Lambda 関数名に設定されているリソース
   d. X-Ray Lambda プロパゲータ
3. handleRequest メソッドを に変更doHandleRequestし、OpenTelemetrySdkオブジェクトをベースクラスに渡します。
4. クライアントの構築時にインターセプターを登録して、OpenTemetry AWS SDK 計測を使用して Amazon S3 クライアントを計測します。

以下の OpenTelemetry 関連の依存関係が必要です。

```
dependencies {
    ...

    implementation("software.amazon.distro.opentelemetry:aws-distro-opentelemetry-xray-
udp-span-exporter:0.1.0")

    implementation(platform('io.opentelemetry.instrumentation:opentelemetry-
instrumentation-bom:2.14.0'))
    implementation(platform('io.opentelemetry:opentelemetry-bom:1.48.0'))

    implementation('io.opentelemetry:opentelemetry-sdk')
    implementation('io.opentelemetry:opentelemetry-api')
    implementation('io.opentelemetry.contrib:opentelemetry-aws-xray-propagator:1.45.0-
alpha')
    implementation('io.opentelemetry.contrib:opentelemetry-aws-resources:1.45.0-alpha')
    implementation('io.opentelemetry.instrumentation:opentelemetry-aws-lambda-
core-1.0:2.14.0-alpha')
    implementation('io.opentelemetry.instrumentation:opentelemetry-aws-sdk-2.2:2.14.0-
alpha')
}
```

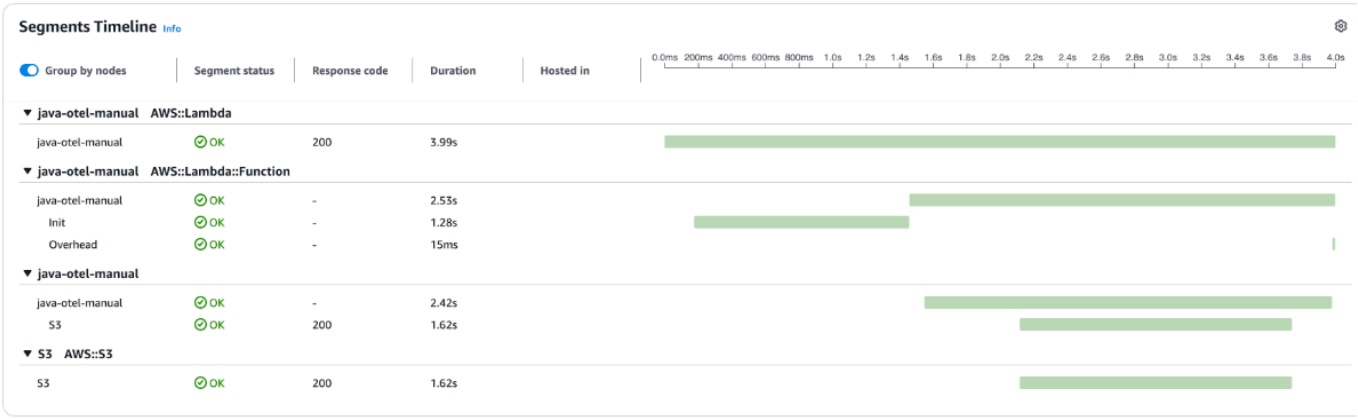
次のコードは、必要な変更後の Lambda 関数を示しています。追加のカスタムスパンを作成して、自動的に提供されるスパンを補完できます。

```
package example;

import java.time.Duration;

import com.amazonaws.services.lambda.runtime.Context;
```

```
import io.opentelemetry.api.common.Attributes;
import io.opentelemetry.context.propagation.ContextPropagators;
import io.opentelemetry.contrib.aws.resource.LambdaResource;
import io.opentelemetry.contrib.awsxray.propagator.AwsXrayLambdaPropagator;
import io.opentelemetry.instrumentation.awslambdacore.v1_0.TracingRequestHandler;
import io.opentelemetry.instrumentation.awssdk.v2_2.AwsSdkTelemetry;
import io.opentelemetry.sdk.OpenTelemetrySdk;
import io.opentelemetry.sdk.resources.Resource;
import io.opentelemetry.sdk.trace.SdkTracerProvider;
import io.opentelemetry.sdk.trace.export.SimpleSpanProcessor;
import io.opentelemetry.sdk.trace.samplers.Sampler;
import static io.opentelemetry.semconv.ServiceAttributes.SERVICE_NAME;
import software.amazon.awssdk.core.client.config.ClientOverrideConfiguration;
import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.s3.model.ListBucketsRequest;
import software.amazon.awssdk.services.s3.model.ListBucketsResponse;
import software.amazon.awssdk.services.s3.model.S3Exception;
import
 software.amazon.distro.opentelemetry.exporter.xray.udp.trace.AwsXrayUdpSpanExporterBuilder;

public class ListBucketsLambda extends TracingRequestHandler<String, String> {
    private static final Resource lambdaResource = LambdaResource.get();
    private static final SdkTracerProvider sdkTracerProvider =
        SdkTracerProvider.builder()
            .addSpanProcessor(SimpleSpanProcessor.create(
                new AwsXrayUdpSpanExporterBuilder().build()
            ))
            .addResource(
                lambdaResource
                .merge(Resource.create(Attributes.of(SERVICE_NAME,
 System.getenv("AWS_LAMBDA_FUNCTION_NAME"))))
            )
            .setSampler(Sampler.parentBased(Sampler.alwaysOn()))
            .build();
    private static final OpenTelemetrySdk openTelemetry =
        OpenTelemetrySdk.builder()
            .setTracerProvider(sdkTracerProvider)

 .setPropagators(ContextPropagators.create(AwsXrayLambdaPropagator.getInstance()))
            .buildAndRegisterGlobal();
    private static final AwsSdkTelemetry telemetry =
 AwsSdkTelemetry.create(openTelemetry);
    private final S3Client S3_CLIENT = S3Client.builder()
        .overrideConfiguration(ClientOverrideConfiguration.builder()
```

```
            .addExecutionInterceptor(telemetry.newExecutionInterceptor())
            .build())
        .build();

    public ListBucketsLambda() {
        super(openTelemetry, Duration.ofMillis(0));
    }

    @Override
    public String doHandleRequest(String input, Context context) {
        try {
            ListBucketsResponse response = makeListBucketsCall();
            context.getLogger().log("response: " + response.toString());
            return "Success";
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    private ListBucketsResponse makeListBucketsCall() {
        try {
            ListBucketsRequest listBucketsRequest = ListBucketsRequest.builder()
                .build();
            ListBucketsResponse response = S3_CLIENT.listBuckets(listBucketsRequest);
            return response;
        } catch (S3Exception e) {
            throw new RuntimeException("Failed to call S3 listBuckets" +
 e.awsErrorDetails().errorMessage(), e);
        }
    }
}
```

Lambda 関数を呼び出すと、CloudWatch コンソールのトレースマップの下に次のトレースが表示されます。



OpenTelemetry Go への移行

次のコード例を使用して、X-Ray から移行するときに OpenTelemetry SDK を使用して Go アプリケーションを手動で計測します。

SDK を使用した手動計測

Tracing setup with X-Ray SDK

X-Ray SDK for Go を使用する場合、コードを実装する前にサービスプラグインまたはローカルサンプリングルールを設定する必要があります。

```
func init() {
    if os.Getenv("ENVIRONMENT") == "production" {
        ec2.Init()
    }

    xray.Configure(xray.Config{
        DaemonAddr:       "127.0.0.1:2000",
        ServiceVersion:   "1.2.3",
    })
}
```

Set up tracing with OpenTelemetry SDK

TracerProvider をインスタンス化し、グローバルトレーサープロバイダーとして登録して、OpenTelemetry SDK を設定します。以下のコンポーネントを設定することをお勧めします。

- OTLP トレースエクスポーター – CloudWatch エージェントまたは OpenTelemetry Collector へのトレースのエクスポートに必要です
- X-Ray Propagator – X-Ray と統合された AWS サービスにトレースコンテキストを伝達するために必要です
- X-Ray リモートサンプラー – X-Ray サンプリングルールを使用したリクエストのサンプリングに必要です
- リソースディテクター – アプリケーションを実行しているホストのメタデータを検出するには

```
import (
    "go.opentelemetry.io/contrib/detectors/aws/ec2"
    "go.opentelemetry.io/contrib/propagators/aws/xray"
    "go.opentelemetry.io/contrib/samplers/aws/xray"
    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/exporters/otlp/otlptrace/otlptracegrpc"
    "go.opentelemetry.io/otel/sdk/trace"
)

func setupTracing() error {
    ctx := context.Background()

    exporterEndpoint := os.Getenv("OTEL_EXPORTER_OTLP_ENDPOINT")
    if exporterEndpoint == "" {
        exporterEndpoint = "localhost:4317"
    }

    traceExporter, err := otlptracegrpc.New(ctx,
        otlptracegrpc.WithInsecure(),
        otlptracegrpc.WithEndpoint(exporterEndpoint))
    if err != nil {
        return fmt.Errorf("failed to create OTLP trace exporter: %v", err)
    }
```

```
    remoteSampler, err := xray.NewRemoteSampler(ctx, "my-service-name", "ec2")
    if err != nil {
        return fmt.Errorf("failed to create X-Ray Remote Sampler: %v", err)
    }

    ec2Resource, err := ec2.NewResourceDetector().Detect(ctx)
    if err != nil {
        return fmt.Errorf("failed to detect EC2 resource: %v", err)
    }

    tp := trace.NewTracerProvider(
        trace.WithSampler(remoteSampler),
        trace.WithBatcher(traceExporter),
        trace.WithResource(ec2Resource),
    )

    otel.SetTracerProvider(tp)
    otel.SetTextMapPropagator(xray.Propagator{})

    return nil
}
```

受信リクエストのトレース (HTTP ハンドラー計測)

With X-Ray SDK

X-Ray を使用して HTTP ハンドラーを計測するために、X-Ray ハンドラーメソッドを使用して NewFixedSegmentNamer を使用してセグメントを生成しました。

```
func main() {
    http.Handle("/", xray.Handler(xray.NewFixedSegmentNamer("myApp"),
 http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("Hello!"))
    })))
    http.ListenAndServe(":8000", nil)
}
```

With OpenTelemetry SDK

OpenTelemetry を使用して HTTP ハンドラーを計測するには、OpenTelemetry の newHandler メソッドを使用して元のハンドラーコードをラップします。

```
import (
    "go.opentelemetry.io/contrib/instrumentation/net/http/otelhttp"
)

helloHandler := func(w http.ResponseWriter, req *http.Request) {
    ctx := req.Context()
    span := trace.SpanFromContext(ctx)
    span.SetAttributes(attribute.Bool("isHelloHandlerSpan", true),
 attribute.String("attrKey", "attrValue"))

    _, _ = io.WriteString(w, "Hello World!\n")
}

otelHandler := otelhttp.NewHandler(http.HandlerFunc(helloHandler), "Hello")

http.Handle("/hello", otelHandler)
err = http.ListenAndServe(":8080", nil)
if err != nil {
    log.Fatal(err)
}
```

AWS SDK for Go v2 の計測

With X-Ray SDK

AWS SDK からの送信 AWS リクエストを計測するために、クライアントは次のように計測されました。

```
// Create a segment
ctx, root := xray.BeginSegment(context.TODO(), "AWSSDKV2_Dynamodb")
defer root.Close(nil)

cfg, err := config.LoadDefaultConfig(ctx, config.WithRegion("us-west-2"))
if err != nil {
```

```
    log.Fatalf("unable to load SDK config, %v", err)
}
// Instrumenting AWS SDK v2
awsv2.AWSV2Instrumentor(&cfg.APIOptions)
// Using the Config value, create the DynamoDB client
svc := dynamodb.NewFromConfig(cfg)
// Build the request with its input parameters
_, err = svc.ListTables(ctx, &dynamodb.ListTablesInput{
    Limit: aws.Int32(5),
})
if err != nil {
    log.Fatalf("failed to list tables, %v", err)
}
```

With OpenTelemetry SDK

ダウンストリーム AWS SDK 呼び出しのトレースサポートは、OpenTelemetry の AWS SDK for Go v2 Instrumentation によって提供されます。S3 クライアント呼び出しをトレースする例を次に示します。

```
import (
    ...

    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/s3"

    "go.opentelemetry.io/otel"
    oteltrace "go.opentelemetry.io/otel/trace"
    awsConfig "github.com/aws/aws-sdk-go-v2/config"
    "go.opentelemetry.io/contrib/instrumentation/github.com/aws/aws-sdk-go-v2/
otelaws"
)

...


    // init aws config
    cfg, err := awsConfig.LoadDefaultConfig(ctx)
    if err != nil {
        panic("configuration error, " + err.Error())
    }
```

```
    // instrument all aws clients
    otelaws.AppendMiddlewares(&.APIOptions)


    // Call to S3
    s3Client := s3.NewFromConfig(cfg)
    input := &s3.ListBucketsInput{}
    result, err := s3Client.ListBuckets(ctx, input)
    if err != nil {
        fmt.Printf("Got an error retrieving buckets, %v", err)
        return
    }
```

送信 HTTP 呼び出しの計測

With X-Ray SDK

X-Ray を使用して送信 HTTP 呼び出しを計測するために、xray.Client を使用して提供された HTTP クライアントのコピーを作成しました。

```
myClient := xray.Client(http-client)

resp, err := ctxhttp.Get(ctx, xray.Client(nil), url)
```

With OpenTelemetry SDK

OpenTelemetry を使用して HTTP クライアントを計測するには、OpenTelemetry の otelhttp.NewTransport メソッドを使用して http.DefaultTransport をラップします。

```
import (
    "go.opentelemetry.io/contrib/instrumentation/net/http/otelhttp"
)

// Create an instrumented HTTP client.
httpClient := &http.Client{
    Transport: otelhttp.NewTransport(
        http.DefaultTransport,
    ),
}
```

```
req, err := http.NewRequestWithContext(ctx, http.MethodGet, "https://api.github.com/
repos/aws-observability/aws-otel-go/releases/latest", nil)
if err != nil {
    fmt.Printf("failed to create http request, %v\n", err)
}
res, err := httpClient.Do(req)
if err != nil {
    fmt.Printf("failed to make http request, %v\n", err)
}
// Request body must be closed
defer func(Body io.ReadCloser) {
    err := Body.Close()
    if err != nil {
        fmt.Printf("failed to close http response body, %v\n", err)
    }
}(res.Body)
```

他のライブラリの計測サポート

OpenTelemetry Go でサポートされているライブラリインストルメンテーションの完全なリストは、インストルメンテーションパッケージ にあります。

または、OpenTelemetry レジストリを検索して、OpenTelemetry が Registry でライブラリの計測をサポートしているかどうかを調べることもできます。

トレースデータを手動で作成する

With X-Ray SDK

X-Ray SDK では、X-Ray セグメントとサブセグメントを手動で作成するにはBeginSegment メソッドと BeginSubsegment メソッドが必要でした。 BeginSubsegment

```
// Start a segment
ctx, seg := xray.BeginSegment(context.Background(), "service-name")
// Start a subsegment
subCtx, subSeg := xray.BeginSubsegment(ctx, "subsegment-name")

// Add metadata or annotation here if necessary
xray.AddAnnotation(subCtx, "annotationKey", "annotationValue")
xray.AddMetadata(subCtx, "metadataKey", "metadataValue")
```

```
subSeg.Close(nil)
// Close the segment
seg.Close(nil)
```

With OpenTelemetry SDK

カスタムスパンを使用して、計測ライブラリによってキャプチャされない内部アクティビティのパフォーマンスをモニタリングします。この種のスパン Server のみが X-Ray セグメントに変換され、他のすべてのスパンは X-Ray サブセグメントに変換されることに注意してください。

まず、 `otel.Tracer`メソッドで取得できるスパンを生成する Tracer を作成する必要があります。これにより、トレースセットアップの例でグローバルに登録された TracerProvider の Tracer インスタンスが提供されます。必要に応じていくつでも Tracer インスタンスを作成できますが、アプリケーション全体に 1 つの Tracer を持つことが一般的です。

```
    tracer := otel.Tracer("application-tracer")
```

```
import (
    ...

    oteltrace "go.opentelemetry.io/otel/trace"
)

...

    var attributes = []attribute.KeyValue{
        attribute.KeyValue{Key: "metadataKey", Value:
 attribute.StringValue("metadataValue")},
        attribute.KeyValue{Key: "annotationKey", Value:
 attribute.StringValue("annotationValue")},
        attribute.KeyValue{Key: "aws.xray.annotations", Value:
 attribute.StringSliceValue([]string{"annotationKey"})},
    }

    ctx := context.Background()

    parentSpanContext, parentSpan := tracer.Start(ctx,
 "ParentSpan", oteltrace.WithSpanKind(oteltrace.SpanKindServer),
 oteltrace.WithAttributes(attributes...))
    _, childSpan := tracer.Start(parentSpanContext, "ChildSpan",
 oteltrace.WithSpanKind(oteltrace.SpanKindInternal))
```

```
    // ...

    childSpan.End()
    parentSpan.End()
```

OpenTelemetry SDK を使用してトレースに注釈とメタデータを追加する

上記の例では、 `WithAttributes`メソッドを使用して各スパンに属性を追加します。デフォルトでは、すべてのスパン属性は X-Ray raw データ内のメタデータに変換されることに注意してください。属性がメタデータではなく注釈に変換されるようにするには、属性のリストに`aws.xray.annotations`属性のキーを追加します。詳細については、「カスタマイズされた X-Ray 注釈を有効にする」を参照してください。

Lambda 手動計測

With X-Ray SDK

X-Ray SDK では、Lambda でアクティブトレースを有効にした後、X-Ray SDK を使用するために必要な追加の設定はありませんでした。Lambda は Lambda ハンドラー呼び出しを表すセグメントを作成し、追加の設定なしで X-Ray SDK を使用してサブセグメントを作成しました。

With OpenTelemetry SDK

次の Lambda 関数コード (計測なし) は、Amazon S3 ListBuckets 呼び出しと送信 HTTP リクエストを行います。

```
package main

import (
    "context"
    "encoding/json"
    "fmt"
    "io"
    "net/http"
    "os"

    "github.com/aws/aws-lambda-go/events"
    "github.com/aws/aws-lambda-go/lambda"
    awsconfig "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/s3"
```

```
)

func lambdaHandler(ctx context.Context) (interface{}, error) {
    // Initialize AWS config.
    cfg, err := awsconfig.LoadDefaultConfig(ctx)
    if err != nil {
        panic("configuration error, " + err.Error())
    }

    s3Client := s3.NewFromConfig(cfg)

    // Create an HTTP client.
    httpClient := &http.Client{
        Transport: http.DefaultTransport,
    }

    input := &s3.ListBucketsInput{}
    result, err := s3Client.ListBuckets(ctx, input)
    if err != nil {
        fmt.Printf("Got an error retrieving buckets, %v", err)
    }

    fmt.Println("Buckets:")
    for _, bucket := range result.Buckets {
        fmt.Println(*bucket.Name + ": " + bucket.CreationDate.Format("2006-01-02
 15:04:05 Monday"))
    }
    fmt.Println("End Buckets.")

    req, err := http.NewRequestWithContext(ctx, http.MethodGet, "https://
api.github.com/repos/aws-observability/aws-otel-go/releases/latest", nil)
    if err != nil {
        fmt.Printf("failed to create http request, %v\n", err)
    }
    res, err := httpClient.Do(req)
    if err != nil {
        fmt.Printf("failed to make http request, %v\n", err)
    }
    defer func(Body io.ReadCloser) {
        err := Body.Close()
        if err != nil {
            fmt.Printf("failed to close http response body, %v\n", err)
        }
    }(res.Body)
```

```
    var data map[string]interface{}
    err = json.NewDecoder(res.Body).Decode(&data)
    if err != nil {
        fmt.Printf("failed to read http response body, %v\n", err)
    }
    fmt.Printf("Latest ADOT Go Release is '%s'\n", data["name"])

    return events.APIGatewayProxyResponse{
        StatusCode: http.StatusOK,
        Body:       os.Getenv("_X_AMZN_TRACE_ID"),
    }, nil
}

func main() {
    lambda.Start(lambdaHandler)
}
```

Lambda ハンドラーと Amazon S3 クライアントを手動で計測するには、次の手順を実行します。

1. main() で、TracerProvider (tp) をインスタンス化し、グローバルトレーサープロバイダーとして登録します。TracerProvider は、以下を使用して設定することをお勧めします。
   a. Lambda の UDP X-Ray エンドポイントにトレースを送信するための X-Ray UDP スパンエクスポーターを備えたシンプルなスパンプロセッサ
   b. service.name を Lambda 関数名に設定したリソース
2. の使用を `lambda.Start(lambdaHandler)`に変更します`lambda.Start(otellambda.InstrumentHandler(lambdaHandler, xrayconfig.WithRecommendedOptions(tp)...))`。
3. の OpenTelemetry ミドルウェアを Amazon S3 クライアント設定に追加OpenTelemetry SDK 計測を使用して Amazon S3 クライアント`aws-sdk-go-v2`を計測します。 OpenTemetry AWS
4. OpenTelemetry の `otelhttp.NewTransport`メソッドを使用して をラップすることで、http クライアントを計測します`http.DefaultTransport`。

次のコードは、変更後に Lambda 関数がどのように表示されるかの例です。自動的に提供されるスパンに加えて、追加のカスタムスパンを手動で作成できます。

```
package main

import (
    "context"
    "encoding/json"
    "fmt"
    "io"
    "net/http"
    "os"

    "github.com/aws-observability/aws-otel-go/exporters/xrayudp"
    "github.com/aws/aws-lambda-go/events"
    "github.com/aws/aws-lambda-go/lambda"
    awsconfig "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/s3"

    lambdadetector "go.opentelemetry.io/contrib/detectors/aws/lambda"
    "go.opentelemetry.io/contrib/instrumentation/github.com/aws/aws-lambda-go/
otellambda"
    "go.opentelemetry.io/contrib/instrumentation/github.com/aws/aws-lambda-go/
otellambda/xrayconfig"
    "go.opentelemetry.io/contrib/instrumentation/github.com/aws/aws-sdk-go-v2/
otelaws"
    "go.opentelemetry.io/contrib/instrumentation/net/http/otelhttp"
    "go.opentelemetry.io/contrib/propagators/aws/xray"
    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/attribute"
    "go.opentelemetry.io/otel/sdk/resource"
    "go.opentelemetry.io/otel/sdk/trace"
    semconv "go.opentelemetry.io/otel/semconv/v1.26.0"
)

func lambdaHandler(ctx context.Context) (interface{}, error) {
    // Initialize AWS config.
    cfg, err := awsconfig.LoadDefaultConfig(ctx)
    if err != nil {
        panic("configuration error, " + err.Error())
    }

    // Instrument all AWS clients.
    otelaws.AppendMiddlewares(&cfg.APIOptions)
    // Create an instrumented S3 client from the config.
    s3Client := s3.NewFromConfig(cfg)
```

```
    // Create an instrumented HTTP client.
    httpClient := &http.Client{
        Transport: otelhttp.NewTransport(
            http.DefaultTransport,
        ),
    }

    // return func(ctx context.Context) (interface{}, error) {
    input := &s3.ListBucketsInput{}
    result, err := s3Client.ListBuckets(ctx, input)
    if err != nil {
        fmt.Printf("Got an error retrieving buckets, %v", err)
    }

    fmt.Println("Buckets:")
    for _, bucket := range result.Buckets {
        fmt.Println(*bucket.Name + ": " + bucket.CreationDate.Format("2006-01-02
 15:04:05 Monday"))
    }
    fmt.Println("End Buckets.")

    req, err := http.NewRequestWithContext(ctx, http.MethodGet, "https://
api.github.com/repos/aws-observability/aws-otel-go/releases/latest", nil)
    if err != nil {
        fmt.Printf("failed to create http request, %v\n", err)
    }
    res, err := httpClient.Do(req)
    if err != nil {
        fmt.Printf("failed to make http request, %v\n", err)
    }
    defer func(Body io.ReadCloser) {
        err := Body.Close()
        if err != nil {
            fmt.Printf("failed to close http response body, %v\n", err)
        }
    }(res.Body)

    var data map[string]interface{}
    err = json.NewDecoder(res.Body).Decode(&data)
    if err != nil {
        fmt.Printf("failed to read http response body, %v\n", err)
    }
    fmt.Printf("Latest ADOT Go Release is '%s'\n", data["name"])
```

```
    return events.APIGatewayProxyResponse{
        StatusCode: http.StatusOK,
        Body:       os.Getenv("_X_AMZN_TRACE_ID"),
    }, nil
}

func main() {
    ctx := context.Background()
    detector := lambdadetector.NewResourceDetector()
    lambdaResource, err := detector.Detect(context.Background())
    if err != nil {
        fmt.Printf("failed to detect lambda resources: %v\n", err)
    }

    var attributes = []attribute.KeyValue{
        attribute.KeyValue{Key: semconv.ServiceNameKey, Value:
 attribute.StringValue(os.Getenv("AWS_LAMBDA_FUNCTION_NAME"))},
    }
    customResource := resource.NewWithAttributes(semconv.SchemaURL, attributes...)
    mergedResource, _ := resource.Merge(lambdaResource, customResource)

    xrayUdpExporter, _ := xrayudp.NewSpanExporter(ctx)
    tp := trace.NewTracerProvider(
        trace.WithSpanProcessor(trace.NewSimpleSpanProcessor(xrayUdpExporter)),
        trace.WithResource(mergedResource),
    )

    defer func(ctx context.Context) {
        err := tp.Shutdown(ctx)
        if err != nil {
            fmt.Printf("error shutting down tracer provider: %v", err)
        }
    }(ctx)

    otel.SetTracerProvider(tp)
    otel.SetTextMapPropagator(xray.Propagator{})

    lambda.Start(otellambda.InstrumentHandler(lambdaHandler,
 xrayconfig.WithRecommendedOptions(tp)...))
}
```

Lambda を呼び出すと、CloudWatch コンソールTrace Mapの に次のトレースが表示されます。

OpenTelemetry Node.js への移行

このセクションでは、Node.js アプリケーションを X-Ray SDK から OpenTelemetry に移行する方法について説明します。自動計測と手動計測の両方のアプローチについて説明し、一般的なユースケースの具体的な例を示します。

X-Ray Node.js SDK を使用すると、Node.js アプリケーションを手動で計測してトレースできます。このセクションでは、X-Ray から OpenTelemetry 計測に移行するためのコード例を示します。

セクション

- ゼロコード自動計測ソリューション
- 手動計測ソリューション
- 受信リクエストのトレース
- AWS SDK JavaScript V3 計測
- 送信 HTTP 呼び出しの計測
- 他のライブラリの計測サポート
- トレースデータを手動で作成する
- Lambda 計測

ゼロコード自動計測ソリューション

X-Ray SDK for Node.js を使用してリクエストをトレースするには、アプリケーションコードを変更する必要があります。OpenTelemetry を使用すると、ゼロコードの自動計測ソリューションを使用してリクエストをトレースできます。

OpenTelemetry ベースの自動計測を使用したゼロコード自動計測。

1. Node.js の AWS Distro for OpenTelemetry (ADOT) 自動計測の使用 – Node.js アプリケーションの自動計測については、AWS 「 Distro for OpenTelemetry JavaScript 自動計測によるトレースとメトリクス」を参照してください。

   (オプション) ADOT JavaScript 自動計測 AWS を使用して でアプリケーションを自動的に計測するときに CloudWatch Application Signals を有効にして、現在のアプリケーションの状態をモニタリングし、ビジネス目標に照らして長期的なアプリケーションパフォーマンスを追跡することもできます。Application Signals は、アプリケーション、サービス、依存関係をアプリケーション中心の統合ビューで表示し、アプリケーションの状態をモニタリングしてトリアージできるようにします。詳細については、「Application Signals」を参照してください。
2. OpenTelemetry JavaScript ゼロコード自動計測の使用 – OpenTelemetry JavaScript による自動計測については、JavaScript ゼロコード計測」を参照してください。

手動計測ソリューション

Tracing setup with X-Ray SDK

X-Ray SDK for Node.js を使用した場合、SDK を使用してコードを計測する前に、aws-xray-sdkX-Ray SDK をサービスプラグインまたはローカルサンプリングルールで設定するためにパッケージが必要でした。

```
var AWSXRay = require('aws-xray-sdk');

AWSXRay.config([AWSXRay.plugins.EC2Plugin,AWSXRay.plugins.ElasticBeanstalkPlugin]);
AWSXRay.middleware.setSamplingRules(<path to file>);
```

Tracing setup with OpenTelemetry SDK

> **Note**
>
> AWS X-Ray リモートサンプリングは現在、OpenTelemetry JS 用に設定することはできません。ただし、X-Ray リモートサンプリングのサポートは現在、Node.js の ADOT Auto-Instrumentation を通じて利用できます。

以下のコード例では、次の依存関係が必要です。

```
npm install --save \
  @opentelemetry/api \
  @opentelemetry/sdk-node \
  @opentelemetry/exporter-trace-otlp-proto \
  @opentelemetry/propagator-aws-xray \
  @opentelemetry/resource-detector-aws
```

アプリケーションコードを実行する前に、OpenTelemetry SDK をセットアップして設定する必要があります。これは、--require フラグを使用して実行できます。instrumentation.js という名前のファイルを作成します。これには、OpenTelemetry 計測の設定とセットアップが含まれます。

次のコンポーネントを設定することをお勧めします。

- OTLPTraceExporter – CloudWatch エージェント/OpenTelemetry Collector へのトレースのエクスポートに必要です
- AWSXRayPropagator – X-Ray と統合された AWS サービスにトレースコンテキストを伝達するために必要です
- Resource Detectors (Amazon EC2 Resource Detector など) - アプリケーションを実行しているホストのメタデータを検出するには

```
/*instrumentation.js*/
// Require dependencies
const { NodeSDK } = require('@opentelemetry/sdk-node');
const { OTLPTraceExporter } = require('@opentelemetry/exporter-trace-otlp-proto');
const { AWSXRayPropagator } = require("@opentelemetry/propagator-aws-xray");
```

```
const { detectResources } = require('@opentelemetry/resources');
const { awsEc2Detector } = require('@opentelemetry/resource-detector-aws');

const resource = detectResources({
    detectors: [awsEc2Detector],
});

const _traceExporter = new OTLPTraceExporter({
    url: 'http://localhost:4318/v1/traces'
});

const sdk = new NodeSDK({
    resource: resource,
    textMapPropagator: new AWSXRayPropagator(),
    traceExporter: _traceExporter
});

sdk.start();
```

その後、次のような OpenTelemetry 設定でアプリケーションを実行できます。

```
node --require ./instrumentation.js app.js
```

OpenTelemetry SDK ライブラリ計測を使用して、 AWS SDK などのライブラリのスパンを自動的に作成できます。これらを有効にすると、 AWS SDK for JavaScript v3 などのモジュールのスパンが自動的に作成されます。OpenTelemetry には、すべてのライブラリ計測を有効にするか、有効にするライブラリ計測を指定するオプションがあります。

すべての計測を有効にするには、 `@opentelemetry/auto-instrumentations-node`パッケージをインストールします。

```
npm install @opentelemetry/auto-instrumentations-node
```

次に、次のように設定を更新して、すべてのライブラリ計測を有効にします。

```
const { getNodeAutoInstrumentations } = require('@opentelemetry/auto-
instrumentations-node');
```

```
...

const sdk = new NodeSDK({
    resource: resource,
     instrumentations: [getNodeAutoInstrumentations()],
     textMapPropagator: new AWSXRayPropagator(),
    traceExporter: _traceExporter
});
```

Tracing setup with ADOT auto-instrumentation for Node.js

Node.js の ADOT 自動計測を使用して、Node.js アプリケーションの OpenTelemetry を自動的に設定できます。ADOT Auto-Instrumentation を使用すると、手動でコードを変更して受信リクエストをトレースしたり、 AWS SDK や HTTP クライアントなどのライブラリをトレースしたりする必要はありません。詳細については、AWS 「 Distro for OpenTelemetry JavaScript Auto-Instrumentation」の「トレースとメトリクス」を参照してください。

Node.js の ADOT 自動計測は以下をサポートします。

- 環境変数による X-Ray リモートサンプリング – `export OTEL_TRACES_SAMPLER=xray`
- X-Ray トレースコンテキストの伝播 (デフォルトで有効)
- リソース検出 (Amazon EC2、Amazon ECS、Amazon EKS 環境のリソース検出はデフォルトで有効になっています)
- サポートされているすべての OpenTelemetry 計測の自動ライブラリ計測。 `OTEL_NODE_ENABLED_INSTRUMENTATIONS`および `OTEL_NODE_DISABLED_INSTRUMENTATIONS`環境変数を介して選択的に無効化/有効化できます。
- スパンの手動作成

受信リクエストのトレース

With X-Ray SDK

Express.js

Express.js アプリケーションによって受信された HTTP リクエストを追跡する X-Ray SDK では、2 つのミドルウェア`AWSXRay.express.openSegment(<name>)`と は、定義されたルートをすべてラップしてトレースする必要がありAWSXRay.express.closeSegment()ました。

```
app.use(xrayExpress.openSegment('defaultName'));

...

app.use(xrayExpress.closeSegment());
```

復元

Restify アプリケーションによって受信された HTTP リクエストをトレースするために、X-Ray SDK のミドルウェアは、Restify Server の aws-xray-sdk-restify モジュールから enable を実行して使用されました。

```
var AWSXRay = require('aws-xray-sdk');
var AWSXRayRestify = require('aws-xray-sdk-restify');

var restify = require('restify');
var server = restify.createServer();
AWSXRayRestify.enable(server, 'MyApp'));
```

With OpenTelemetry SDK

Express.js

の受信リクエストのトレースサポートExpress.jsは、OpenTelemetry HTTP 計測とOpenTelemetry Express 計測によって提供されます。で次の依存関係をインストールしますnpm。

```
npm install --save @opentelemetry/instrumentation-http @opentelemetry/
instrumentation-express
```

OpenTelemetry SDK 設定を更新して、Express モジュールの計測を有効にします。

```
const { HttpInstrumentation } = require('@opentelemetry/instrumentation-http');
```

```
const { ExpressInstrumentation } = require('@opentelemetry/instrumentation-
express');
...

const sdk = new NodeSDK({
  ...

  instrumentations: [
    ...
    // Express instrumentation requires HTTP instrumentation
    new HttpInstrumentation(),
    new ExpressInstrumentation(),
  ],
});
```

復元

Restify アプリケーションの場合は、OpenTelemetry Restify 計測が必要です。次の依存関係をインストールします。

```
npm install --save @opentelemetry/instrumentation-restify
```

OpenTelemetry SDK 設定を更新して、restify モジュールの計測を有効にします。

```
const { RestifyInstrumentation } = require('@opentelemetry/instrumentation-
restify');
...

const sdk = new NodeSDK({
  ...

  instrumentations: [
    ...
    new RestifyInstrumentation(),
  ],
});
```

AWS SDK JavaScript V3 計測

With X-Ray SDK

AWS SDK からの送信 AWS リクエストを計測するには、次の例のようにクライアントを計測しました。

```
import { S3, PutObjectCommand } from '@aws-sdk/client-s3';

const s3 = AWSXRay.captureAWSv3Client(new S3({}));

await s3.send(new PutObjectCommand({
  Bucket: bucketName,
  Key: keyName,
  Body: 'Hello!',
}));
```

With OpenTelemetry SDK

DynamoDB、Amazon S3 などへのダウンストリーム AWS SDK 呼び出しのトレースサポートは、OpenTelemetry AWS SDK 計測によって提供されます。 Amazon S3 で次の依存関係をインストールしますnpm。

```
npm install --save @opentelemetry/instrumentation-aws-sdk
```

SDK 計測を使用して OpenTelemetry AWS SDK 設定を更新します。

```
import { AwsInstrumentation } from '@opentelemetry/instrumentation-aws-sdk';
...

const sdk = new NodeSDK({
  ...

  instrumentations: [
    ...
    new AwsInstrumentation()
```

```
  ],
});
```

送信 HTTP 呼び出しの計測

With X-Ray SDK

X-Ray を使用して送信 HTTP リクエストを計測するには、クライアントを計測する必要がありました。たとえば、以下の「」を参照してください。

個々の HTTP クライアント

```
var AWSXRay = require('aws-xray-sdk');
var http = AWSXRay.captureHTTPs(require('http'));
```

すべての HTTP クライアント (グローバル)

```
var AWSXRay = require('aws-xray-sdk');
AWSXRay.captureHTTPsGlobal(require('http'));
var http = require('http');
```

With OpenTelemetry SDK

Node.js HTTP クライアントのトレースサポートは、OpenTelemetry HTTP Instrumentation によって提供されます。で次の依存関係をインストールしますnpm。

```
npm install --save @opentelemetry/instrumentation-http
```

OpenTelemetry SDK 設定を次のように更新します。

```
const { HttpInstrumentation } = require('@opentelemetry/instrumentation-http');
...
```

```
const sdk = new NodeSDK({
  ...

  instrumentations: [
    ...
    new HttpInstrumentation(),
  ],
});
```

他のライブラリの計測サポート

OpenTelemetry JavaScript でサポートされているライブラリインストルメンテーションの完全なリストは、サポートされているインストルメンテーション にあります。

または、OpenTelemetry レジストリを検索して、OpenTelemetry が Registry でライブラリの計測をサポートしているかどうかを調べることもできます。

トレースデータを手動で作成する

With X-Ray SDK

X-Ray を使用して、アプリケーションを追跡するセグメントとその子サブセグメントを手動で作成するには、aws-xray-sdkパッケージコードが必要でした。

```
var AWSXRay = require('aws-xray-sdk');

AWSXRay.enableManualMode();

var segment = new AWSXRay.Segment('myApplication');

captureFunc('1', function(subsegment1) {
  captureFunc('2', function(subsegment2) {

  }, subsegment1);
}, segment);

segment.close();
segment.flush();
```

With OpenTelemetry SDK

カスタムスパンを作成して使用することで、計測ライブラリでキャプチャされない内部アクティビティのパフォーマンスをモニタリングできます。この種のスパンのみ サーバーは X-Ray セグメントに変換され、他のすべてのスパンは X-Ray サブセグメントに変換されることに注意してください。詳細については、「セグメント」を参照してください。

トレースセットアップで OpenTelemetry SDK を設定してスパンを作成すると、Tracer インスタンスが必要になります。必要に応じていくつでも Tracer インスタンスを作成できますが、アプリケーション全体に 1 つの Tracer を持つことが一般的です。

```
const { trace, SpanKind } = require('@opentelemetry/api');

// Get a tracer instance
const tracer = trace.getTracer('your-tracer-name');

...

  // This span will appear as a segment in X-Ray
  tracer.startActiveSpan('server', { kind: SpanKind.SERVER }, span => {
    // Do work here

    // This span will appear as a subsegment in X-Ray
    tracer.startActiveSpan('operation2', { kind: SpanKind.INTERNAL }, innerSpan => {
      // Do more work here

      innerSpan.end();
    });
    span.end();
  });
```

OpenTelemetry SDK を使用してトレースに注釈とメタデータを追加する

カスタムキーと値のペアをスパンの属性として追加することもできます。デフォルトでは、これらのスパン属性はすべて X-Ray raw データ内のメタデータに変換されることに注意してください。属性がメタデータではなく注釈に変換されるようにするには、属性のリストに`aws.xray.annotations`属性のキーを追加します。詳細については、「カスタマイズされた X-Ray 注釈を有効にする」を参照してください。

```
 tracer.startActiveSpan('server', { kind: SpanKind.SERVER }, span => {
   span.setAttribute('metadataKey', 'metadataValue');
   span.setAttribute('annotationKey', 'annotationValue');

   // The following ensures that "annotationKey: annotationValue" is an annotation
in X-Ray raw data.
   span.setAttribute('aws.xray.annotations', ['annotationKey']);

   // Do work here

   span.end();
 });
```

Lambda 計測

With X-Ray SDK

Lambda 関数のアクティブトレースを有効にした後、追加の設定なしで X-Ray SDK が必要でした。Lambda は Lambda ハンドラー呼び出しを表すセグメントを作成し、追加の設定なしで X-Ray SDK を使用してサブセグメントまたは計測ライブラリを作成しました。

With OpenTelemetry SDK

AWS 販売された Lambda レイヤーを使用して Lambda を自動的に計測できます。解決策は 2 つあります。

- （推奨) CloudWatch Application Signals Lambda レイヤー

  > **Note**
  >
  > この Lambda レイヤーでは、デフォルトで CloudWatch Application Signals が有効になっているため、メトリクスとトレースの両方を収集することで、Lambda アプリケーションのパフォーマンスとヘルスのモニタリングが可能になります。トレースのみが必要な場合は、Lambda 環境変数 を設定する必要がありますOTEL_AWS_APPLICATION_SIGNALS_ENABLED=false。詳細については、「Lambda でアプリケーションを有効にする」を参照してください。

- AWS ADOT JS 用の マネージド Lambda レイヤー。詳細については、AWS 「 Distro for OpenTelemetry Lambda Support for JavaScript」を参照してください。

Lambda 計測を使用したスパンの手動作成

ADOT JavaScript Lambda Layer は Lambda 関数の自動計測を提供しますが、カスタムデータの提供や、ライブラリ計測の対象ではない Lambda 関数自体内のコードの計測など、Lambda で手動計測を実行する必要がある場合があります。

自動計測とともに手動計測を実行するには、 を依存関係`@opentelemetry/api`として追加する必要があります。この依存関係のバージョンは、ADOT JavaScript SDK で使用されるのと同じ依存関係のバージョンにすることをお勧めします。OpenTelemetry API を使用して、Lambda 関数にスパンを手動で作成できます。

NPM を使用して`@opentelemetry/api`依存関係を追加するには：

```
npm install @opentelemetry/api
```

OpenTelemetry .NET への移行

.NET アプリケーションで X-Ray トレースを使用する場合、手動作業による X-Ray .NET SDK が計測に使用されます。

このセクションでは、X-Ray 手動計測ソリューションから .NET 用 OpenTelemetry 手動計測ソリューションに移行するための SDK を使用した手動計測ソリューションセクションのコード例を示します。または、 ゼロコード自動計測ソリューションセクションでアプリケーションのソースコードを変更することなく、X-Ray 手動計測から OpenTelemetry 自動計測ソリューションに移行して .NET アプリケーションを計測することもできます。

セクション

- ゼロコード自動計測ソリューション
- SDK を使用した手動計測ソリューション
- トレースデータを手動で作成する
- 受信リクエストのトレース (ASP.NET および ASP.NET コア計測)
- AWS SDK 計測
- 送信 HTTP 呼び出しの計測
- 他のライブラリの計測サポート
- Lambda 計測

ゼロコード自動計測ソリューション

OpenTelemetry は、ゼロコードの自動計測ソリューションを提供します。これらのソリューションは、アプリケーションコードを変更することなくリクエストをトレースします。

OpenTelemetry ベースの自動計測オプション

1. AWS Distro for OpenTelemetry (ADOT) auto-Instrumentation for .NET の使用 – .NET アプリケーションを自動的に計測するには、AWS 「 Distro for OpenTelemetry .NET Auto-Instrumentation」を参照してください。

   (オプション) ADOT .NET 自動計測 AWS を使用して でアプリケーションを自動的に計測する場合、CloudWatch Application Signals を有効にして次の操作を行います。

   - 現在のアプリケーションのヘルスをモニタリングする
   - ビジネス目標に対する長期的なアプリケーションパフォーマンスを追跡する
   - アプリケーション、サービス、依存関係の統合されたアプリケーション中心のビューを取得する
   - アプリケーションのヘルスをモニタリングしてトリアージする

   詳細については、「Application Signals」を参照してください。
2. OpenTelemetry .Net ゼロコード自動計測の使用 – OpenTelemetry .Net を使用して自動的に計測するには、AWS 「 Distro for OpenTelemetry .NET Auto-Instrumentation を使用したトレースとメトリクス」を参照してください。

SDK を使用した手動計測ソリューション

Tracing configuration with X-Ray SDK

.NET ウェブアプリケーションの場合、X-Ray SDK は`Web.config`ファイルの appSettings セクションで設定されます。

Web.config の例

```
<configuration>
  <appSettings>
    <add key="AWSXRayPlugins" value="EC2Plugin"/>
  </appSettings>
```

```
</configuration>
```

.NET Core では、 という名前の最上位キーappsettings.jsonを持つ という名前XRayのファイルが使用され、X-Ray レコーダーを初期化するために設定オブジェクトが構築されます。

.NET の例 appsettings.json

```
{
  "XRay": {
    "AWSXRayPlugins": "EC2Plugin"
  }
}
```

.NET Core Program.cs の例 – レコーダー設定

```
using Amazon.XRay.Recorder.Core;
...
AWSXRayRecorder.InitializeInstance(configuration);
```

Tracing configuration with OpenTelemetry SDK

これらの依存関係を追加します。

```
dotnet add package OpenTelemetry
dotnet add package OpenTelemetry.Contrib.Extensions.AWSXRay
dotnet add package OpenTelemetry.Sampler.AWS --prerelease
dotnet add package OpenTelemetry.Resources.AWS
dotnet add package OpenTelemetry.Exporter.OpenTelemetryProtocol
dotnet add package OpenTelemetry.Extensions.Hosting
dotnet add package OpenTelemetry.Instrumentation.AspNetCore
```

.NET アプリケーションの場合は、Global TracerProvider を設定して OpenTelemetry SDK を設定します。次の設定例では、 の計測も有効にしますASP.NET Core。を計測するにはASP.NET、

「」を参照してください受信リクエストのトレース (ASP.NET および ASP.NET コア計測)。他のフレームワークで OpenTelemetry を使用するには、サポートされているフレームワークのその他のライブラリについては「レジストリ」を参照してください。

次のコンポーネントを設定することをお勧めします。

- An `OTLP Exporter` – CloudWatch エージェント/OpenTelemetry Collector へのトレースのエクスポートに必要です
- AWS X-Ray プロパゲーター – AWS X-Ray と統合されたサービスにトレースコンテキストを伝達するために必要です
- AWS X-Ray リモートサンプラー – X-Ray サンプリングルールを使用してリクエストをサンプリングする必要がある場合に必要です
- `Resource Detectors` (Amazon EC2 Resource Detector など) - アプリケーションを実行しているホストのメタデータを検出するには

```
using OpenTelemetry;
using OpenTelemetry.Contrib.Extensions.AWSXRay.Trace;
using OpenTelemetry.Sampler.AWS;
using OpenTelemetry.Trace;
using OpenTelemetry.Resources;

var builder = WebApplication.CreateBuilder(args);

var serviceName = "MyServiceName";
var serviceVersion = "1.0.0";

var resourceBuilder = ResourceBuilder
    .CreateDefault()
    .AddService(serviceName: serviceName)
    .AddAWSEC2Detector();

builder.Services.AddOpenTelemetry()
    .ConfigureResource(resource => resource
        .AddAWSEC2Detector()
        .AddService(
            serviceName: serviceName,
            serviceVersion: serviceVersion))
    .WithTracing(tracing => tracing
        .AddSource(serviceName)
```

```
        .AddAspNetCoreInstrumentation()
        .AddOtlpExporter()
        .SetSampler(AWSXRayRemoteSampler.Builder(resourceBuilder.Build())
            .SetEndpoint("http://localhost:2000")
            .Build()));

Sdk.SetDefaultTextMapPropagator(new AWSXRayPropagator()); // configure  X-Ray
 propagator
```

コンソールアプリに OpenTelemetry を使用するには、プログラムの起動時に次の OpenTelemetry 設定を追加します。

```
using OpenTelemetry;
using OpenTelemetry.Contrib.Extensions.AWSXRay.Trace;
using OpenTelemetry.Trace;
using OpenTelemetry.Resources;

var serviceName = "MyServiceName";

var resourceBuilder = ResourceBuilder
    .CreateDefault()
    .AddService(serviceName: serviceName)
    .AddAWSEC2Detector();

var tracerProvider = Sdk.CreateTracerProviderBuilder()
    .AddSource(serviceName)
    .ConfigureResource(resource =>
        resource
            .AddAWSEC2Detector()
            .AddService(
                serviceName: serviceName,
                serviceVersion: serviceVersion
            )
        )
    .AddOtlpExporter() // default address localhost:4317
    .SetSampler(new TraceIdRatioBasedSampler(1.00))
    .Build();

Sdk.SetDefaultTextMapPropagator(new AWSXRayPropagator()); // configure  X-Ray
 propagator
```

トレースデータを手動で作成する

With X-Ray SDK

X-Ray SDK では、X-Ray セグメントBeginSegmentとサブセグメントを手動で作成するには、メソッドと BeginSubsegmentメソッドが必要でした。

```
using Amazon.XRay.Recorder.Core;

AWSXRayRecorder.Instance.BeginSegment("segment name"); // generates `TraceId` for
 you
try
{
    // Do something here
    // can create custom subsegments
    AWSXRayRecorder.Instance.BeginSubsegment("subsegment name");
    try
    {
        DoSometing();
    }
    catch (Exception e)
    {
        AWSXRayRecorder.Instance.AddException(e);
    }
    finally
    {
        AWSXRayRecorder.Instance.EndSubsegment();
    }
}
catch (Exception e)
{
    AWSXRayRecorder.Instance.AddException(e);
}
finally
{
    AWSXRayRecorder.Instance.EndSegment();
}
```

With OpenTelemetry SDK

.NET では、アクティビティ API を使用してカスタムスパンを作成し、計測ライブラリでキャプチャされない内部アクティビティのパフォーマンスをモニタリングできます。この種のスパン Server のみが X-Ray セグメントに変換され、他のすべてのスパンは X-Ray サブセグメントに変換されることに注意してください。

必要に応じてインスタンスをいくつでも作成できますがActivitySource、アプリケーション/サービス全体でインスタンスを 1 つだけ作成することをお勧めします。

```
using System.Diagnostics;

ActivitySource activitySource = new ActivitySource("ActivitySourceName",
 "ActivitySourceVersion");


...


using (var activity = activitySource.StartActivity("ActivityName",
 ActivityKind.Server)) // this will be translated to a X-Ray Segment
{
    // Do something here

    using (var internalActivity = activitySource.StartActivity("ActivityName",
 ActivityKind.Internal)) // this will be translated to an X-Ray Subsegment
    {
        // Do something here
    }
}
```

OpenTelemetry SDK を使用してトレースに注釈とメタデータを追加する

アクティビティで SetTagメソッドを使用して、カスタムキーと値のペアを属性としてスパンに追加することもできます。デフォルトでは、すべてのスパン属性は X-Ray raw データ内のメタデータに変換されることに注意してください。属性がメタデータではなく注釈に変換されるようにするには、その属性のキーをaws.xray.annotations属性のリストに追加します。

```
using (var activity = activitySource.StartActivity("ActivityName",
 ActivityKind.Server)) // this will be translated to a X-Ray Segment
{
    activity.SetTag("metadataKey", "metadataValue");
    activity.SetTag("annotationKey", "annotationValue");
    string[] annotationKeys = {"annotationKey"};
    activity.SetTag("aws.xray.annotations", annotationKeys);

    // Do something here

    using (var internalActivity = activitySource.StartActivity("ActivityName",
 ActivityKind.Internal)) // this will be translated to an X-Ray Subsegment
    {
        // Do something here
    }
}
```

OpenTelemetry 自動計測を使用する

.NET 用の OpenTelemetry 自動計測ソリューションを使用している場合、アプリケーションで手動計測を実行する必要がある場合。たとえば、自動計測ライブラリでカバーされていないセクションのコードをアプリケーション自体で計測する場合などです。

グローバル は 1 つしかできないためTracerProvider、自動計測と一緒に使用するTracerProviderと、手動計測は独自の をインスタンス化しないでください。を使用する場合、カスタム手動トレースTracerProviderは、OpenTelemetry SDK で自動計測または手動計測を使用する場合と同じように機能します。

受信リクエストのトレース (ASP.NET および ASP.NET コア計測)

With X-Ray SDK

ASP.NET アプリケーションによって処理されるリクエストを計測するには、 `global.asax` ファイルの `Init`メソッドRegisterXRayで https://docs.aws.amazon.com/xray/latest/devguide/xray-sdk-dotnet-messagehandler.html を呼び出す方法の詳細については、「」を参照してください。

```
AWSXRayASPNET.RegisterXRay(this, "MyApp");
```

ASP.NET コアアプリケーションによって処理されるリクエストを計測するには、Startup クラスの UseXRayメソッドの他のミドルウェアの前に Configureメソッドが呼び出されます。

```
app.UseXRay("MyApp");
```

With OpenTelemetry SDK

OpenTelemetry には、ASP.NET および ASP.NET Core の受信ウェブリクエストのトレースを収集するための計測ライブラリも用意されています。次のセクションでは、トレースプロバイダーの作成時に ASP.NET または ASP.NET コアインストルメンテーションを追加する方法など、OpenTelemetry 設定でこれらのライブラリインストルメンテーションを追加および有効にするために必要な手順を示します。

OpenTelemetry.Instrumentation.AspNet,OpenTelemetry.Instrumentation.AspNet を有効にするステップ」を参照してください。また、OpenTelemetry.Instrumentation.AspNetCore,OpenTelemetry.Instrumentation.AspNetCore を有効にするステップ」を参照してください。

AWS SDK 計測

With X-Ray SDK

を呼び出して、すべての AWS SDK クライアントをインストールしますRegisterXRayForAllServices()。

```
using Amazon.XRay.Recorder.Handlers.AwsSdk;
AWSSDKHandler.RegisterXRayForAllServices(); //place this before any instantiation of
 AmazonServiceClient
AmazonDynamoDBClient client = new AmazonDynamoDBClient(RegionEndpoint.USWest2); //
 AmazonDynamoDBClient is automatically registered with X-Ray
```

特定の AWS サービスクライアント計測には、次のいずれかの方法を使用します。

```
AWSSDKHandler.RegisterXRay<IAmazonDynamoDB>(); // Registers specific type of
 AmazonServiceClient : All instances of IAmazonDynamoDB created after this line are
 registered
AWSSDKHandler.RegisterXRayManifest(String path); // To configure custom AWS Service
 Manifest file. This is optional, if you have followed "Configuration" section
```

With OpenTelemetry SDK

次のコード例では、次の依存関係が必要です。

```
dotnet add package OpenTelemetry.Instrumentation.AWS
```

AWS SDK を計測するには、Global TracerProvider がセットアップされている OpenTelemetry SDK 設定を更新します。

```
builder.Services.AddOpenTelemetry()
    ...
    .WithTracing(tracing => tracing
        .AddAWSInstrumentation()
        ...
```

送信 HTTP 呼び出しの計測

With X-Ray SDK

X-Ray .NET SDK は、拡張機能メソッド、 `GetResponseTraced()` の使用`GetAsyncResponseTraced()`時`System.Net.HttpWebRequest`、または の使用時に`HttpClientXRayTracingHandler`ハンドラを使用して、送信 HTTP 呼び出しをトレースします`System.Net.Http.HttpClient`。

With OpenTelemetry SDK

次のコード例では、次の依存関係が必要です。

```
dotnet add package OpenTelemetry.Instrumentation.Http
```

`System.Net.Http.HttpClient` と を計測するには`System.Net.HttpWebRequest`、Global TracerProvider がセットアップされている OpenTelemetry SDK 設定を更新します。

```
builder.Services.AddOpenTelemetry()
    ...
    .WithTracing(tracing => tracing
        .AddHttpClientInstrumentation()
        ...
```

他のライブラリの計測サポート

OpenTelemetry Registry for .NET Instrumentation Libraries を検索してフィルタリングし、OpenTelemetry がライブラリの計測をサポートしているかどうかを確認できます。レジストリを参照して検索を開始します。

Lambda 計測

With X-Ray SDK

Lambda で X-Ray SDK を使用するには、次の手順が必要です。

1. Lambda 関数でアクティブトレースを有効にする
2. Lambda サービスは、ハンドラーの呼び出しを表すセグメントを作成します。
3. X-Ray SDK を使用してサブセグメントまたは計測ライブラリを作成する

With OpenTelemetry-based solutions

AWS 販売された Lambda レイヤーを使用して Lambda を自動的に計測できます。解決策は 2 つあります。

- （推奨) CloudWatch Application Signals Lambda レイヤー
- パフォーマンスを向上させるには、 `OpenTelemetry Manual Instrumentation`を使用して Lambda 関数の OpenTelemetry トレースを生成することを検討してください。

AWS Lambda の OpenTelemetry 手動計測

以下は、Lambda 関数コード (計測なし) の例です。

```
using System;
using System.Text;
using System.Threading.Tasks;
using Amazon.Lambda.Core;
using Amazon.S3;
using Amazon.S3.Model;

// Assembly attribute to enable Lambda function logging
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace ExampleLambda;

public class ListBucketsHandler
{
    private static readonly AmazonS3Client s3Client = new();

    // new Lambda function handler passed in
    public async Task<string> HandleRequest(object input, ILambdaContext context)
    {
        try
        {
            var DoListBucketsAsyncResponse = await DoListBucketsAsync();
            context.Logger.LogInformation($"Results:
 {DoListBucketsAsyncResponse.Buckets}");

            context.Logger.LogInformation($"Successfully called ListBucketsAsync");
            return "Success!";
        }
        catch (Exception ex)
        {
            context.Logger.LogError($"Failed to call ListBucketsAsync: {ex.Message}");
            throw;
        }
    }

    private async Task<ListBucketsResponse> DoListBucketsAsync()
    {
        try
```

```
        {
            var putRequest = new ListBucketsRequest
            {
            };

            var response = await s3Client.ListBucketsAsync(putRequest);
            return response;
        }
        catch (AmazonS3Exception ex)
        {
            throw new Exception($"Failed to call ListBucketsAsync: {ex.Message}", ex);
        }
    }
}
```

Lambda ハンドラーと Amazon S3 クライアントを手動で計測するには、次の手順を実行します。

1. TracerProvider のインスタンス化 – TracerProvider は`XrayUdpSpanExporter`、、ParentBased Always On Sampler、および Lambda 関数名に`service.name`設定された `Resource` で設定することをお勧めします。
2. を呼び出し`AddAWSInstrumentation()`て SDK クライアント計測を に追加することで、OpenTemetry SDK 計測を使用して Amazon S3 クライアントを計測します。 OpenTemetry AWS AWS `TracerProvider`
3. 元の Lambda 関数と同じ署名を持つラッパー関数を作成します。`AWSLambdaWrapper.Trace()` API を呼び出し`TracerProvider`、、元の Lambda 関数、およびその入力をパラメータとして渡します。ラッパー関数を Lambda ハンドラー入力として設定します。

次のコード例では、次の依存関係が必要です。

```
dotnet add package OpenTelemetry.Instrumentation.AWSLambda
dotnet add package OpenTelemetry.Instrumentation.AWS
dotnet add package OpenTelemetry.Resources.AWS
dotnet add package AWS.Distro.OpenTelemetry.Exporter.Xray.Udp
```

次のコードは、必要な変更後の Lambda 関数を示しています。追加のカスタムスパンを作成して、自動的に提供されるスパンを補完できます。

```
using Amazon.Lambda.Core;
using Amazon.S3;
using Amazon.S3.Model;
using OpenTelemetry;
using OpenTelemetry.Instrumentation.AWSLambda;
using OpenTelemetry.Trace;
using AWS.Distro.OpenTelemetry.Exporter.Xray.Udp;
using OpenTelemetry.Resources;

// Assembly attribute to enable Lambda function logging
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace ExampleLambda;

public class ListBucketsHandler
{
    private static readonly AmazonS3Client s3Client = new();

    TracerProvider tracerProvider = Sdk.CreateTracerProviderBuilder()
        .AddAWSLambdaConfigurations()
        .AddProcessor(
            new SimpleActivityExportProcessor(
                // AWS_LAMBDA_FUNCTION_NAME Environment Variable will be defined in AWS
 Lambda Environment
                new
 XrayUdpExporter(ResourceBuilder.CreateDefault().AddService(Environment.GetEnvironmentVariable(
            )
        )
        .AddAWSInstrumentation()
        .SetSampler(new ParentBasedSampler(new AlwaysOnSampler()))
        .Build();

    // new Lambda function handler passed in
    public async Task<string> HandleRequest(object input, ILambdaContext context)
    => await AWSLambdaWrapper.Trace(tracerProvider, OriginalHandleRequest, input,
 context);

    public async Task<string> OriginalHandleRequest(object input, ILambdaContext
 context)
    {
        try
```

```
        {
            var DoListBucketsAsyncResponse = await DoListBucketsAsync();
            context.Logger.LogInformation($"Results:
 {DoListBucketsAsyncResponse.Buckets}");

            context.Logger.LogInformation($"Successfully called ListBucketsAsync");
            return "Success!";
        }
        catch (Exception ex)
        {
            context.Logger.LogError($"Failed to call ListBucketsAsync: {ex.Message}");
            throw;
        }
    }

    private async Task<ListBucketsResponse> DoListBucketsAsync()
    {
        try
        {
            var putRequest = new ListBucketsRequest
            {
            };

            var response = await s3Client.ListBucketsAsync(putRequest);
            return response;
        }
        catch (AmazonS3Exception ex)
        {
            throw new Exception($"Failed to call ListBucketsAsync: {ex.Message}", ex);
        }
    }
}
```

この Lambda を呼び出すと、CloudWatch コンソールのトレースマップに次のトレースが表示されます。

OpenTelemetry Python への移行

このガイドは、Python アプリケーションを X-Ray SDK から OpenTelemetry 計測に移行するのに役立ちます。自動計測アプローチと手動計測アプローチの両方について説明し、一般的なシナリオのコード例を示します。

セクション

- ゼロコード自動計測ソリューション
- アプリケーションを手動で計測する
- トレース設定の初期化
- 受信リクエストのトレース
- AWS SDK 計測
- リクエストによる送信 HTTP コールの計測
- 他のライブラリの計測サポート
- トレースデータを手動で作成する
- Lambda 計測

ゼロコード自動計測ソリューション

X-Ray SDK では、リクエストをトレースするためにアプリケーションコードを変更する必要がありました。OpenTelemetry は、リクエストをトレースするためのゼロコード自動計測ソリューションを提供します。OpenTelemetry では、ゼロコード自動計測ソリューションを使用してリクエストをトレースできます。

OpenTelemetry ベースの自動計測を使用したゼロコード

1. AWS Distro for OpenTelemetry (ADOT) auto-Instrumentation for Python – Python アプリケーションの自動計測については、AWS 「 Distro for OpenTelemetry Python Auto-Instrumentation」を参照してください。

   (オプション) ADOT Python 自動計測 AWS を使用して でアプリケーションを自動的に計測するときに CloudWatch Application Signals を有効にして、現在のアプリケーションの状態をモニタリングし、ビジネス目標に照らして長期的なアプリケーションパフォーマンスを追跡することもできます。Application Signals は、アプリケーション、サービス、依存関係をアプリケーション中心の統合ビューで表示し、アプリケーションの状態をモニタリングしてトリアージできるようにします。

2. OpenTelemetry Python ゼロコード自動計測の使用 – OpenTelemetry Python を使用した自動計測については、「Python ゼロコード計測」を参照してください。

アプリケーションを手動で計測する

pip コマンドを使用して、アプリケーションを手動で計測できます。

With X-Ray SDK

```
pip install aws-xray-sdk
```

With OpenTelemetry SDK

```
pip install opentelemetry-api opentelemetry-sdk opentelemetry-exporter-otlp
 opentelemetry-propagator-aws-xray
```

トレース設定の初期化

With X-Ray SDK

X-Ray では、グローバルxray_recorderが初期化され、セグメントとサブセグメントを生成するために使用されます。

With OpenTelemetry SDK

> **Note**
>
> X-Ray リモートサンプリングは現在、OpenTelemetry Python 用に設定することはできません。ただし、X-Ray リモートサンプリングのサポートは、現在 ADOT Auto-Instrumentation for Python を通じて利用できます。

OpenTelemetry では、グローバル を初期化する必要がありますTracerProvider。この を使用するとTracerProvider、アプリケーション内の任意の場所でスパンを生成するために使用できる Tracer を取得できます。次のコンポーネントを設定することをお勧めします。

- OTLPSpanExporter – CloudWatch エージェント/OpenTelemetry Collector へのトレースのエクスポートに必要です
- AWS X-Ray プロパゲーター – X-Ray と統合された AWS サービスにトレースコンテキストを伝達するために必要です

```
from opentelemetry import (
    trace,
    propagate
)
from opentelemetry.sdk.trace import TracerProvider

from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.exporter.otlp.proto.http.trace_exporter import OTLPSpanExporter
from opentelemetry.sdk.trace.export import BatchSpanProcessor
from opentelemetry.propagators.aws import AwsXRayPropagator

# Sends generated traces in the OTLP format to an OTel Collector running on port
 4318
otlp_exporter = OTLPSpanExporter(endpoint="http://localhost:4318/v1/traces")
# Processes traces in batches as opposed to immediately one after the other
span_processor = BatchSpanProcessor(otlp_exporter)
# More configurations can be done here. We will visit them later.

# Sets the global default tracer provider
provider = TracerProvider(active_span_processor=span_processor)
```

```
trace.set_tracer_provider(provider)

# Configures the global propagator to use the X-Ray Propagator
propagate.set_global_textmap(AwsXRayPropagator())

# Creates a tracer from the global tracer provider
tracer = trace.get_tracer("my.tracer.name")
# Use this tracer to create Spans
```

ADOT auto-Instrumentation for Python を使用

ADOT auto-instrumentation for Python を使用して、Python アプリケーション用に OpenTelemetry を自動的に設定できます。ADOT 自動計測を使用すると、手動でコードを変更して受信リクエストをトレースしたり、 AWS SDK や HTTP クライアントなどのライブラリをトレースしたりする必要はありません。詳細については、AWS 「 Distro for OpenTelemetry Python Auto-Instrumentation を使用したトレースとメトリクス」を参照してください。

ADOT auto-instrumentation for Python は以下をサポートしています。

- 環境変数を介した X-Ray リモートサンプリング `export OTEL_TRACES_SAMPLER=xray`
- X-Ray トレースコンテキストの伝播 (デフォルトで有効)
- リソース検出 (Amazon EC2、Amazon ECS、Amazon EKS 環境のリソース検出はデフォルトで有効になっています)
- サポートされているすべての OpenTelemetry 計測の自動ライブラリ計測は、デフォルトで有効になっています。`OTEL_PYTHON_DISABLED_INSTRUMENTATIONS` 環境変数を使用して選択的にを無効にすることができます (すべてデフォルトで有効になっています)。
- スパンの手動作成

X-Ray サービスプラグインから OpenTelemetry AWS リソースプロバイダーへ

X-Ray SDK には、Amazon EC2、Amazon ECS、Elastic Beanstalk などのホストされたサービスからプラットフォーム固有の情報をキャプチャ`xray_recorder`するために に追加できるプラグインが用意されています。これは、情報をリソース属性としてキャプチャする OpenTelemetry のリソースプロバイダーに似ています。さまざまな AWS プラットフォームで使用できるリソースプロバイダーは複数あります。

- まず、 AWS 拡張機能パッケージをインストールします。 `pip install opentelemetry-sdk-extension-aws`
- 目的のリソースディテクターを設定します。次の例は、OpenTelemetry SDK で Amazon EC2 リソースプロバイダーを設定する方法を示しています。

```
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.extension.aws.resource.ec2 import (
    AwsEc2ResourceDetector,
)
from opentelemetry.sdk.resources import get_aggregated_resources

provider = TracerProvider(
    active_span_processor=span_processor,
    resource=get_aggregated_resources([
        AwsEc2ResourceDetector(),
    ]))

trace.set_tracer_provider(provider)
```

受信リクエストのトレース

With X-Ray SDK

X-Ray Python SDK は、Django、Flask、Bottle などのアプリケーションフレームワークをサポートし、それらで実行されている Python アプリケーションの受信リクエストを追跡します。これは、各フレームワークの `XRayMiddleware`をアプリケーションに追加することによって行われます。

With OpenTelemetry SDK

OpenTelemetry は、特定の計測ライブラリを通じて Django と Flask の計測を提供します。OpenTelemetry で使用できる Bottle の計測はありません。OpenTelemetry WSGI Instrumentation を使用してアプリケーションを追跡できます。

次のコード例では、次の依存関係が必要です。

```
pip install opentelemetry-instrumentation-flask
```

アプリケーションフレームワークの計測を追加する前に、OpenTelemetry SDK を初期化し、グローバル TracerProvider を登録する必要があります。これを行わないと、トレースオペレーションは になりますno-ops。グローバル を設定したらTracerProvider、アプリケーションフレームワークにインストルメンタを使用できます。次の例は、Flask アプリケーションを示しています。

```
from flask import Flask
from opentelemetry import trace
from opentelemetry.instrumentation.flask import FlaskInstrumentor
from opentelemetry.sdk.extension.aws.resource import AwsEc2ResourceDetector
from opentelemetry.sdk.resources import get_aggregated_resources
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor, ConsoleSpanExporter

provider = TracerProvider(resource=get_aggregated_resources(
    [
        AwsEc2ResourceDetector(),
    ]))

processor = BatchSpanProcessor(ConsoleSpanExporter())
provider.add_span_processor(processor)
trace.set_tracer_provider(provider)

# Creates a tracer from the global tracer provider
tracer = trace.get_tracer("my.tracer.name")

app = Flask(__name__)

# Instrument the Flask app
FlaskInstrumentor().instrument_app(app)


@app.route('/')
def hello_world():
    return 'Hello World!'


if __name__ == '__main__':
    app.run()
```

AWS SDK 計測

With X-Ray SDK

X-Ray Python SDK は、botocoreライブラリにパッチを適用して AWS SDK クライアントリクエストをトレースします。詳細については、「X-Ray AWS SDK for Python を使用した SDK 呼び出しのトレース」を参照してください。アプリケーションでは、 patch_all()メソッドを使用して、 または ライブラリを使用してすべてのboto3ライブラリbotocoreまたはパッチを選択的に計測しますpatch((['botocore']))。選択したメソッドのいずれかがアプリケーション内のすべての Boto3 クライアントを計測し、これらのクライアントを使用して行われた呼び出しのサブセグメントを生成します。

With OpenTelemetry SDK

次のコード例では、次の依存関係が必要です。

```
pip install opentelemetry-instrumentation-botocore
```

OpenTelemetry Botocore Instrumentation をプログラムで使用して、アプリケーション内のすべての Boto3 クライアントを計測します。次の例は、 botocoreインストルメンテーションを示しています。

```
import boto3
import opentelemetry.trace as trace
from botocore.exceptions import ClientError
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.resources import get_aggregated_resources
from opentelemetry.sdk.trace.export import (
    BatchSpanProcessor,
    ConsoleSpanExporter,
)
from opentelemetry.instrumentation.botocore import BotocoreInstrumentor

provider = TracerProvider()
processor = BatchSpanProcessor(ConsoleSpanExporter())
provider.add_span_processor(processor)
```

```
trace.set_tracer_provider(provider)

# Creates a tracer from the global tracer provider
tracer = trace.get_tracer("my.tracer.name")

# Instrument BotoCore
BotocoreInstrumentor().instrument()

# Initialize S3 client
s3 = boto3.client("s3", region_name="us-east-1")

# Your bucket name
bucket_name = "my-example-bucket"

# Get bucket location (as an example of describing it)
try:
    response = s3.get_bucket_location(Bucket=bucket_name)
    region = response.get("LocationConstraint") or "us-east-1"
    print(f"Bucket '{bucket_name}' is in region: {region}")

    # Optionally, get bucket's creation date via list_buckets
    buckets = s3.list_buckets()
    for bucket in buckets["Buckets"]:
        if bucket["Name"] == bucket_name:
            print(f"Bucket created on: {bucket['CreationDate']}")
            break
except ClientError as e:
    print(f"Failed to describe bucket: {e}")
```

リクエストによる送信 HTTP コールの計測

With X-Ray SDK

X-Ray Python SDK は、リクエストライブラリにパッチを適用して、リクエストを通じて送信 HTTP 呼び出しをトレースします。詳細については、「X-Ray SDK for Python を使用したダウンストリーム HTTP ウェブサービスの呼び出しのトレース」を参照してください。アプリケーションでは、 `patch_all()`メソッドを使用してすべてのライブラリを計測するか、 を使用してリクエストライブラリに選択的にパッチを適用できます`patch((['requests']))`。オプションのいずれかが`requests`ライブラリを計測し、 を介して行われた呼び出しのサブセグメントを生成します`requests`。

With OpenTelemetry SDK

次のコード例では、次の依存関係が必要です。

```
pip install opentelemetry-instrumentation-requests
```

OpenTelemetry Requests Instrumentation をプログラムで使用してリクエストライブラリを計測し、アプリケーションで OpenTelemetry Requests によって行われた HTTP リクエストのトレースを生成します。詳細については、OpenTelemetry requests Instrumentation」を参照してください。次の例は、`requests`ライブラリの計測を示しています。

```
from opentelemetry.instrumentation.requests import RequestsInstrumentor

# Instrument Requests
RequestsInstrumentor().instrument()

...

    example_session = requests.Session()
    example_session.get(url="https://example.com")
```

または、基盤となる`urllib3`ライブラリを計測して HTTP リクエストをトレースすることもできます。

```
# pip install opentelemetry-instrumentation-urllib3
from opentelemetry.instrumentation.urllib3 import URLLib3Instrumentor

# Instrument urllib3
URLLib3Instrumentor().instrument()

...

    example_session = requests.Session()
    example_session.get(url="https://example.com")
```

他のライブラリの計測サポート

OpenTelemetry Python でサポートされているライブラリ計測の完全なリストは、サポートされているライブラリ、フレームワーク、アプリケーションサーバー、および JVMsにあります。

または、OpenTelemetry Registry を検索して、OpenTelemetry が計測をサポートしているかどうかを調べることもできます。レジストリを参照して検索を開始します。

トレースデータを手動で作成する

Python アプリケーションの `xray_recorder` を使用してセグメントとサブセグメントを作成できます。詳細については、「Python コードの手動計測」を参照してください。トレースデータに注釈とメタデータを手動で追加することもできます。

OpenTelemetry SDK を使用したスパンの作成

`start_as_current_span` API を使用してスパンを開始し、スパンを作成するための設定を行います。スパンの作成例については、「スパンの作成」を参照してください。スパンが開始され、現在のスコープに入ると、属性、イベント、例外、リンクなどを追加して、スパンに詳細情報を追加できます。X-Ray にセグメントとサブセグメントがある場合と同様に、OpenTelemetry にはさまざまな種類のスパンがあります。SERVER 種類スパンのみが X-Ray セグメントに変換され、その他は X-Ray サブセグメントに変換されます。

```
from opentelemetry import trace
from opentelemetry.trace import SpanKind

import time

tracer = trace.get_tracer("my.tracer.name")

# Create a new span to track some work
with tracer.start_as_current_span("parent", kind=SpanKind.SERVER) as parent_span:
    time.sleep(1)

    # Create a nested span to track nested work
    with tracer.start_as_current_span("child", kind=SpanKind.CLIENT) as child_span:
        time.sleep(2)
        # the nested span is closed when it's out of scope

    # Now the parent span is the current span again
    time.sleep(1)
```

```
    # This span is also closed when it goes out of scope
```

OpenTelemetry SDK を使用したトレースへの注釈とメタデータの追加

X-Ray Python SDK には、トレースに注釈とメタデータを追加put_metadataするための個別の APIs put_annotationと が用意されています。OpenTelemetry SDK では、注釈とメタデータは単にスパンの属性であり、set_attributeAPI を介して追加されます。

トレースの注釈にするスパン属性は、値が注釈のキーと値のペアのリストaws.xray.annotationsである予約キーの下に追加されます。他のすべてのスパン属性は、変換されたセグメントまたはサブセグメントのメタデータになります。

さらに、ADOT コレクターを使用している場合は、コレクター設定indexed_attributesで を指定することで、X-Ray 注釈に変換するスパン属性を設定できます。

次の例は、OpenTelemetry SDK を使用してトレースに注釈とメタデータを追加する方法を示しています。

```
with tracer.start_as_current_span("parent", kind=SpanKind.SERVER) as parent_span:
    parent_span.set_attribute("TransactionId", "qwerty12345")
    parent_span.set_attribute("AccountId", "1234567890")

    # This will convert the TransactionId and AccountId to be searchable X-Ray
 annotations
    parent_span.set_attribute("aws.xray.annotations", ["TransactionId", "AccountId"])

    with tracer.start_as_current_span("child", kind=SpanKind.CLIENT) as child_span:

        # The MicroTransactionId will be converted to X-Ray metadata for the child
 subsegment
        child_span.set_attribute("MicroTransactionId", "micro12345")
```

Lambda 計測

X-Ray で Lambda 関数をモニタリングするには、X-Ray を有効にし、関数呼び出しロールに適切なアクセス許可を追加しました。さらに、関数からのダウンストリームリクエストをトレースする場合は、X-Ray Python SDK を使用してコードを計測します。

OpenTelemetry for X-Ray では、Application Signals をオフにして CloudWatch Application Signals Lambda レイヤーを使用することをお勧めします。これにより、関数が自動計測され、関数の呼び出しと関数からのダウンストリームリクエストのスパンが生成されます。トレースに加えて、Application Signals を使用して関数の状態をモニタリングする場合は、「Lambda でアプリケーションを有効にする」を参照してください。

- AWS OpenTelemetry ARN 用の Lambda Layer から関数に必要な Lambda Layer ARNs を見つけて追加します。
- 関数に次の環境変数を設定します。
  - `AWS_LAMBDA_EXEC_WRAPPER=/opt/otel-instrument` – 関数の自動計測をロードします。
  - `OTEL_AWS_APPLICATION_SIGNALS_ENABLED=false` – これにより、Application Signals のモニタリングが無効になります。

Lambda 計測を使用したスパンの手動作成

さらに、 関数内でカスタムスパンを生成して作業を追跡できます。これを行うには、Application Signals Lambda レイヤーの自動計測と組み合わせて `opentelemetry-api`パッケージのみを使用します。

1. 関数の依存関係`opentelemetry-api`として を含める
2. 次のコードスニペットは、カスタムスパンを生成するためのサンプルです。

```
from opentelemetry import trace

# Get the tracer (auto-configured by the Application Signals layer)
tracer = trace.get_tracer(__name__)

def handler(event, context):
    # This span is a child of the layer's root span
    with tracer.start_as_current_span("my-custom-span") as span:
        span.set_attribute("key1", "value1")
        span.add_event("custom-event", {"detail": "something happened"})

        # Any logic you want to trace
        result = some_internal_logic()

    return {
        "statusCode": 200,
```

```
        "body": result
    }
```

OpenTelemetry Ruby への移行

Ruby アプリケーションを X-Ray SDK から OpenTelemetry 計測に移行するには、次のコード例と手動計測のガイダンスを使用します。

セクション

SDK を使用してソリューションを手動で計測する

Tracing setup with X-Ray SDK

X-Ray SDK for Ruby では、サービスプラグインを使用してコードを設定する必要がありました。

```
require 'aws-xray-sdk'

XRay.recorder.configure(plugins: [:ec2, :elastic_beanstalk])
```

Tracing setup with OpenTelemetry SDK

> **Note**
>
> X-Ray リモートサンプリングは現在、OpenTelemetry Ruby 用に設定することはできません。

Ruby on Rails アプリケーションの場合は、Rails イニシャライザに設定コードを配置します。詳細については、「 の使用開始」を参照してください。手動で計測されたすべての Ruby プログラムでは、 `OpenTelemetry::SDK.configure`メソッドを使用して OpenTelemetry Ruby SDK を設定する必要があります。

まず、次のパッケージをインストールします。

```
bundle add opentelemetry-sdk opentelemetry-exporter-otlp opentelemetry-propagator-
xray
```

次に、プログラムが初期化されたときに実行される設定コードを使用して OpenTelemetry SDK を設定します。次のコンポーネントを設定することをお勧めします。

- `OTLP Exporter` – CloudWatch エージェントと OpenTelemetry コレクターへのトレースのエクスポートに必要です
- `An AWS X-Ray Propagator` – X-Ray と統合された AWS サービスにトレースコンテキストを伝達するために必要です

```
require 'opentelemetry-sdk'
require 'opentelemetry-exporter-otlp'

# Import the gem containing the AWS X-Ray for OTel Ruby ID Generator and propagator
require 'opentelemetry-propagator-xray'

OpenTelemetry::SDK.configure do |c|
  c.service_name = 'my-service-name'

  c.add_span_processor(
    # Use the BatchSpanProcessor to send traces in groups instead of one at a time
    OpenTelemetry::SDK::Trace::Export::BatchSpanProcessor.new(
      # Use the default OLTP Exporter to send traces to the ADOT Collector
      OpenTelemetry::Exporter::OTLP::Exporter.new(
        # The OpenTelemetry Collector is running as a sidecar and listening on port
 4318
        endpoint:"http://127.0.0.1:4318/v1/traces"
      )
    )
  )

  # The X-Ray Propagator injects the X-Ray Tracing Header into downstream calls
```

```
  c.propagators = [OpenTelemetry::Propagator::XRay::TextMapPropagator.new]
end
```

OpenTelemetry SDKsには、ライブラリ計測の概念もあります。これらを有効にすると、 AWS SDK などのライブラリのスパンが自動的に作成されます。OpenTelemetry には、すべてのライブラリ計測を有効にするか、有効にするライブラリ計測を指定するオプションがあります。

すべての計測を有効にするには、まず opentelemetry-instrumentation-allパッケージをインストールします。

```
bundle add opentelemetry-instrumentation-all
```

次に、次のように設定を更新して、すべてのライブラリ計測を有効にします。

```
require 'opentelemetry/instrumentation/all'
...

OpenTelemetry::SDK.configure do |c|
   ...

  c.use_all() # Enable all instrumentations
end
```

OpenTelemetry SDKsには、ライブラリ計測の概念もあります。これらを有効にすると、 AWS SDK などのライブラリのスパンが自動的に作成されます。OpenTelemetry には、すべてのライブラリ計測を有効にするか、有効にするライブラリ計測を指定するオプションがあります。

すべての計測を有効にするには、まず opentelemetry-instrumentation-allパッケージをインストールします。

```
bundle add opentelemetry-instrumentation-all
```

次に、次のように設定を更新して、すべてのライブラリ計測を有効にします。

```
require 'opentelemetry/instrumentation/all'
...
```

```
OpenTelemetry::SDK.configure do |c|
   ...

  c.use_all() # Enable all instrumentations
end
```

受信リクエストのトレース (レール計測)

With X-Ray SDK

X-Ray SDK では、X-Ray トレースは初期化時に Rails フレームワーク用に設定されます。

例 – config/initializers/aws_xray.rb

```
Rails.application.config.xray = {
  name: 'my app',
  patch: %I[net_http aws_sdk],
  active_record: true
}
```

With OpenTelemetry SDK

まず、次のパッケージをインストールします。

```
bundle add opentelemetry-instrumentation-rack opentelemetry-instrumentation-
rails opentelemetry-instrumentation-action_pack opentelemetry-instrumentation-
active_record opentelemetry-instrumentation-action_view
```

次に、次のように設定を更新して Rails アプリケーションの計測を有効にします。

```
# During SDK configuration
OpenTelemetry::SDK.configure do |c|

  ...

  c.use 'OpenTelemetry::Instrumentation::Rails'
  c.use 'OpenTelemetry::Instrumentation::Rack'
  c.use 'OpenTelemetry::Instrumentation::ActionPack'
```

```
  c.use 'OpenTelemetry::Instrumentation::ActiveSupport'
  c.use 'OpenTelemetry::Instrumentation::ActionView'

  ...

end
```

AWS SDK 計測

With X-Ray SDK

AWS SDK からの送信 AWS リクエストを計測するために、 AWS SDK クライアントには次の例のように X-Ray のパッチが適用されます。

```
require 'aws-xray-sdk'
require 'aws-sdk-s3'

# Patch AWS SDK clients
XRay.recorder.configure(plugins: [:aws_sdk])

# Use the instrumented client
s3 = Aws::S3::Client.new
s3.list_buckets
```

With OpenTelemetry SDK

AWS SDK for Ruby V3 は、OpenTelemetry トレースの記録と出力をサポートします。サービスクライアントの OpenTelemetry を設定する方法については、 AWS SDK for Ruby の「オブザーバビリティ機能の設定」を参照してください。

送信 HTTP 呼び出しの計測

外部サービスに対して HTTP 呼び出しを行う場合、自動計測が利用できない場合や、十分な詳細が得られない場合は、手動で呼び出しを計測する必要がある場合があります。

With X-Ray SDK

ダウンストリーム呼び出しを計測するために、X-Ray SDK for Ruby を使用して、アプリケーションが使用するnet/httpライブラリにパッチを適用しました。

```
require 'aws-xray-sdk'

config = {
  name: 'my app',
  patch: %I[net_http]
}

XRay.recorder.configure(config)
```

With OpenTelemetry SDK

OpenTelemetry を使用してnet/http計測を有効にするには、まず opentelemetry-instrumentation-net_httpパッケージをインストールします。

```
bundle add opentelemetry-instrumentation-net_http
```

次に、次のように設定を更新してnet/http計測を有効にします。

```
OpenTelemetry::SDK.configure do |c|
   ...

  c.use 'OpenTelemetry::Instrumentation::Net::HTTP'
  ...

end
```

他のライブラリの計測サポート

OpenTelemetry Ruby でサポートされているライブラリ計測の完全なリストは、opentelemetry-ruby-contrib にあります。

または、OpenTelemetry Registry を検索して、OpenTelemetry が計測をサポートしているかどうかを調べることもできます。詳細については、「レジストリ」を参照してください。

トレースデータを手動で作成する

With X-Ray SDK

X-Ray を使用すると、aws-xray-sdkパッケージでは、アプリケーションを追跡するためにセグメントとその子サブセグメントを手動で作成する必要がありました。セグメントまたはサブセグメントに X-Ray 注釈とメタデータを追加している場合もあります。

```
require 'aws-xray-sdk'
...

# Start a segment
segment = XRay.recorder.begin_segment('my-service')

# Add annotations (indexed key-value pairs)
segment.annotations[:user_id] = 'user-123'
segment.annotations[:payment_status] = 'completed'

# Add metadata (non-indexed data)
segment.metadata[:order] = {
  id: 'order-456',
  items: [
    { product_id: 'prod-1', quantity: 2 },
    { product_id: 'prod-2', quantity: 1 }
  ],
  total: 67.99
}

# Add metadata to a specific namespace
segment.metadata(namespace: 'payment') do |metadata|
  metadata[:transaction_id] = 'tx-789'
  metadata[:payment_method] = 'credit_card'
end

# Create a subsegment with annotations and metadata
segment.subsegment('payment-processing') do |subsegment1|
  subsegment1.annotations[:payment_id] = 'pay-123'
  subsegment1.metadata[:details] = { amount: 67.99, currency: 'USD' }

  # Create a nested subsegment
  subsegment1.subsegment('operation-2') do |subsegment2|
    # Do more work...
  end
```

```
end

# Close the segment
segment.close
```

With OpenTelemetry SDK

カスタムスパンを使用して、計測ライブラリによってキャプチャされない内部アクティビティのパフォーマンスをモニタリングできます。X-Ray セグメントに変換されるのは種類のサーバーのみであり、他のすべてのスパンは X-Ray サブセグメントに変換されることに注意してください。デフォルトでは、スパンは ですINTERNAL。

まず、 OpenTelemetry.tracer_provider.tracer('<YOUR_TRACER_NAME>')メソッドで取得できるスパンを生成するために Tracer を作成します。これにより、アプリケーションの OpenTelemetry 設定にグローバルに登録されている Tracer インスタンスが提供されます。アプリケーション全体で 1 つの Tracer を持つことが一般的です。OpenTelemetry トレーサーを作成し、それを使用してスパンを作成します。

```
require 'opentelemetry-sdk'

...

# Get a tracer
tracer = OpenTelemetry.tracer_provider.tracer('my-application')

# Create a server span (equivalent to X-Ray segment)
tracer.in_span('my-application', kind: OpenTelemetry::Trace::SpanKind::SERVER) do |
span|
  # Do work...

  # Create nested spans of default kind INTERNAL will become an X-Ray subsegment
  tracer.in_span('operation-1') do |child_span1|
    # Set attributes (equivalent to X-Ray annotations and metadata)
    child_span1.set_attribute('key', 'value')

    # Do more work...
    tracer.in_span('operation-2') do |child_span2|
      # Do more work...
    end
  end
end
```

OpenTelemetry SDK を使用してトレースに注釈とメタデータを追加する

`set_attribute` メソッドを使用して、各スパンに属性を追加します。デフォルトでは、これらのスパン属性はすべて X-Ray raw データ内のメタデータに変換されることに注意してください。属性がメタデータではなく注釈に変換されるようにするには、その属性キーを`aws.xray.annotations`属性のリストに追加します。詳細については、「カスタマイズされた X-Ray 注釈を有効にする」を参照してください。

```
# SERVER span will become an X-Ray segment
tracer.in_span('my-server-operation', kind: OpenTelemetry::Trace::SpanKind::SERVER)
 do |span|
    # Your server logic here
    span.set_attribute('attribute.key', 'attribute.value')
    span.set_attribute("metadataKey", "metadataValue")
    span.set_attribute("annotationKey1", "annotationValue")

    # Create X-Ray annotations
    span.set_attribute("aws.xray.annotations", ["annotationKey1"])
end
```

Lambda 手動計測

With X-Ray SDK

Lambda でアクティブトレースを有効にした後、X-Ray SDK を使用するために必要な追加の設定はありません。Lambda は Lambda ハンドラー呼び出しを表すセグメントを作成し、追加の設定なしで X-Ray SDK を使用してサブセグメントまたは計測ライブラリを作成できます。

With OpenTelemetry SDK

次のサンプル Lambda 関数コード (計測なし) を検討してください。

```
require 'json'
def lambda_handler(event:, context:)
    # TODO implement
    { statusCode: 200, body: JSON.generate('Hello from Lambda!') }
end
```

Lambda を手動で計測するには、以下を行う必要があります。

1. Lambda に次の Gem を追加する

```
gem 'opentelemetry-sdk'
gem 'opentelemetry-exporter-otlp'
gem 'opentelemetry-propagator-xray'
gem 'aws-distro-opentelemetry-exporter-xray-udp'
gem 'opentelemetry-instrumentation-aws_lambda'
gem 'opentelemetry-propagator-xray', '~> 0.24.0' # Requires version v0.24.0 or
 higher
```

2. Lambda ハンドラーの外部で OpenTelemetry SDK を初期化します。OpenTelemetry SDK は、以下を使用して設定することをお勧めします。

   1. Lambda の UDP X-Ray エンドポイントにトレースを送信するための X-Ray UDP スパンエクスポーターを備えたシンプルなスパンプロセッサ
   2. X-Ray Lambda プロパゲータ
   3. `service_name` Lambda 関数名に設定する 設定

3. Lambda ハンドラークラスで、次の行を追加して Lambda ハンドラーを計測します。

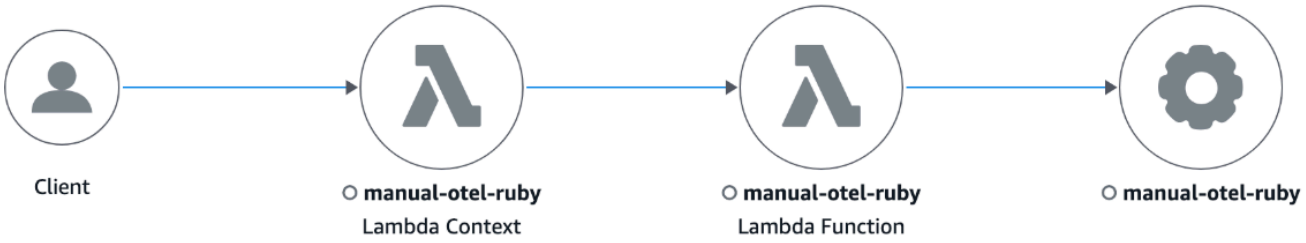
```
  class Handler
        extend OpenTelemetry::Instrumentation::AwsLambda::Wrap
        ...

        instrument_handler :process
    end
```

次のコードは、必要な変更後の Lambda 関数を示しています。追加のカスタムスパンを作成して、自動的に提供されるスパンを補完できます。

```
require 'json'
require 'opentelemetry-sdk'
require 'aws/distro/opentelemetry/exporter/xray/udp'
require 'opentelemetry/propagator/xray'
require 'opentelemetry/instrumentation/aws_lambda'

# Initialize OpenTelemetry SDK outside handler
OpenTelemetry::SDK.configure do |c|
```

```
  # Configure the AWS Distro for OpenTelemetry X-Ray Lambda exporter
  c.add_span_processor(
    OpenTelemetry::SDK::Trace::Export::SimpleSpanProcessor.new(
      AWS::Distro::OpenTelemetry::Exporter::XRay::UDP::AWSXRayUDPSpanExporter.new
    )
  )

  # Configure X-Ray Lambda propagator
  c.propagators = [OpenTelemetry::Propagator::XRay.lambda_text_map_propagator]

  # Set minimal resource information
  c.resource = OpenTelemetry::SDK::Resources::Resource.create({
    OpenTelemetry::SemanticConventions::Resource::SERVICE_NAME =>
 ENV['AWS_LAMBDA_FUNCTION_NAME']
  })
  c.use 'OpenTelemetry::Instrumentation::AwsLambda'
end

module LambdaFunctions
  class Handler
    extend OpenTelemetry::Instrumentation::AwsLambda::Wrap
    def self.process(event:, context:)
      "Hello!"
    end
    instrument_handler :process
  end
end
```

以下は、Ruby で記述された計測された Lambda 関数のトレースマップの例です。



Lambda レイヤーを使用して、Lambda の OpenTelemetry を設定することもできます。詳細については、OpenTelemetry AWS-Lambda Instrumentation」を参照してください。

AWS CloudFormationでの X-Ray リソースの作成

AWS X-Ray は と統合されています。これは AWS CloudFormation、 AWS リソースとインフラストラクチャの作成と管理に費やす時間を短縮できるように、リソースのモデル化とセットアップに役立つサービスです。必要なすべての AWS リソースを記述するテンプレートを作成し、それらのリソースを AWS CloudFormation プロビジョニングして設定します。

を使用すると AWS CloudFormation、テンプレートを再利用して X-Ray リソースを一貫して繰り返しセットアップできます。リソースを 1 回記述し、同じリソースを複数の AWS アカウント およびリージョンで何度もプロビジョニングします。

X-Ray と AWS CloudFormation テンプレート

X-Ray および関連サービスのリソースをプロビジョニングして設定するには、AWS CloudFormation テンプレートについて理解しておく必要があります。テンプレートは、JSON や YAML でフォーマットされたテキストファイルです。これらのテンプレートは、 AWS CloudFormation スタックでプロビジョニングするリソースを記述します。JSON または YAML に慣れていない場合は、デザイナー を使用して AWS CloudFormation AWS CloudFormation テンプレートの使用を開始できます。詳細については、「AWS CloudFormation ユーザーガイド」の「AWS CloudFormation Designer とは」を参照してください。

X-Ray は、 での `AWS::XRay::Group`、`AWS::XRay::SamplingRule`、および `AWS::XRay::ResourcePolicy`リソースの作成をサポートしています AWS CloudFormation。JSON テンプレートと YAML テンプレートの例を含む詳細については、AWS CloudFormation ユーザーガイドの「X-Ray リソースタイプのリファレンス」を参照してください。

の詳細 AWS CloudFormation

詳細については AWS CloudFormation、次のリソースを参照してください。

- AWS CloudFormation
- AWS CloudFormation ユーザーガイド
- AWS CloudFormation API リファレンス
- AWS CloudFormation コマンドラインインターフェイスユーザーガイド

X-Ray のサンプリングルールとグループのタグ付け

タグは、 AWS リソースを識別して整理するために使用できる単語またはフレーズです。各 リソースに複数のタグを追加できます。各タグには、ユーザーが定義するキーとオプションの値が含まれます。たとえば、タグキーは **domain**、タグ値は **example.com** などです。追加したタグに基づいて、リソースを検索したりフィルタ処理したりできます。タグの使用方法の詳細については、AWS 一般的なリファレンスの「タグ付け AWS リソース」を参照してください。

タグを使用して、CloudFront ディストリビューションにタグベースのアクセス許可を適用できます。。詳細については、「リソースタグを使用した AWS リソースへのアクセスの制御」を参照してください。

> **Note**
>
> タグエディタおよびAWS リソースグループは現在、X-Ray リソースをサポートしていません。タグを追加および管理するには、 AWS X-Ray コンソールまたは API を使用します。

X-Ray コンソール、API、 SDKs AWS CLI、および を使用して、リソースにタグを適用できます AWS Tools for Windows PowerShell。詳細については、次のドキュメントを参照してください。

- X-Ray API — AWS X-Ray API リファレンスの以下の操作を参照してください。
  - ListTagsForResource
  - サンプリングルールの作成
  - CreateGroup
  - TagResource
  - UntagResource
- AWS CLI – AWS CLI コマンドリファレンスの「Xray」を参照してください。
- SDK – AWS ドキュメントページの該当する SDK ドキュメントを参照

> **Note**
>
> X-Ray リソースでタグを追加または変更できない場合、または特定のタグを持つリソースを追加できない場合は、この操作を実行する権限がない可能性があります。アクセスをリクエ

ストするには、X-Ray で管理者権限を持つエンタープライズの AWS ユーザーにお問い合わせください。

トピック

タグの制限

タグには次の制限があります。

- リソースあたりのタグの最大数 – 50
- キーの最大長 – 128 文字 (Unicode)
- 値の最大長 – 256 文字 (Unicode)
- キーと値の有効な値 – a ~ z、A ~ Z、0 ~ 9、スペース、特殊文字 (_ . : / = + - @)
- タグのキーと値では、大文字と小文字が区別されます。
- `aws:` をキーのプレフィックスとしてを使用しないでください。 AWS 用に予約済みです。

> **Note**
>
> システムタグを編集または削除することはできません。

コンソールでのタグの管理

X-Ray グループまたはサンプリングルールの作成時に、オプションのタグを追加できます。タグは、後でコンソールで変更または削除することもできます。

次の手順では、X-Ray コンソールでグループおよびサンプリングルールのタグを追加、編集、削除する方法について説明します。

トピック

新しい グループにタグを追加する (コンソール)

新しい X-Ray グループを作成するときに、オプションのタグをグループの作成ページで追加できます。

1. にサインイン AWS Management Console し、https://console.aws.amazon.com/xray/home:// www.com」で X-Ray コンソールを開きます。
2. ナビゲーションペインで、設定を展開し、Groupsを選択します。
3. [グループの作成] を選択してください。
4. グループの作成ページで、グループの名前とフィルタ式を指定します。これらのプロパティの詳細については、「グループの設定」を参照してください。
5. タグで、タグキー、およびオプションでタグ値を入力します。たとえば、**Stage**のタグキー、および**Production**のタグ値を入力して、このグループが生産用途であることを示します。タグを追加すると、必要に応じて別のタグを追加するための新しい行が表示されます。タグの制限については、このトピックで タグの制限 を参照してください。
6. タグの追加が完了したら、[Create group (グループの作成)] を選択します。

新しいサンプリングルールにタグを追加する (コンソール)

新しい X-Ray のサンプリングルールを作成する際、サンプリングルールの作成ページでタグを追加することができます。

1. にサインイン AWS Management Console し、https://console.aws.amazon.com/xray/home:// www.com」で X-Ray コンソールを開きます。
2. ナビゲーションペインで、設定を展開し、サンプリングを選択します。
3. サンプリングールの作成を選択します。
4. サンプリングールの作成ページで、名前、優先度、制限、一致条件、および一致する属性を指定します。これらのプロパティの詳細については、「サンプリングルールの設定」を参照してください。

5. タグで、タグキー、およびオプションでタグ値を入力します。たとえば、**Stage**のタグキー、および**Production**のタグ値を入力して、このサンプリングルールが本番用途であることを示します。タグを追加すると、必要に応じて別のタグを追加するための新しい行が表示されます。タグの制限については、このトピックで タグの制限 を参照してください。
6. タグの追加が完了したら、[サンプリングルールの作成] を選択します。

グループのタグを編集または削除する (コンソール)

X-Ray グループのタグをグループの編集ページで変更または削除できます。

1. にサインイン AWS Management Console し、https://console.aws.amazon.com/xray/home:// www.com」で X-Ray コンソールを開きます。
2. ナビゲーションペインで、設定を展開し、Groupsを選択します。
3. Groups テーブルで、グループの名前を選択します。
4. グループの編集ページ、タグで、タグキーと値を編集します。重複するタグキーを使用することはできません。タグ値はオプションです。必要に応じて値を削除できます。グループの編集ページでの他のプロパティの詳細については、「グループの設定」を参照してください。タグの制限については、このトピックで タグの制限 を参照してください。
5. タグを削除するには、タグの右側にあるXを選択します。
6. タグの編集または削除を完了したら、[グループの更新] を選択します。

サンプリングルールのタグを編集または削除する (コンソール)

X-Ray サンプリングルールのタグをサンプリングルールの編集ページで変更または削除できます。

1. にサインイン AWS Management Console し、https://console.aws.amazon.com/xray/home:// www.com」で X-Ray コンソールを開きます。
2. ナビゲーションペインで、設定を展開し、サンプリングを選択します。
3. サンプリングルールテーブルで、サンプリングルールの名前を選択します。
4. タグで、タグキーと値を編集します。重複するタグキーを使用することはできません。タグ値はオプションです。必要に応じて値を削除できます。サンプリングルールの編集ページでの他のプロパティの詳細については、「サンプリングルールの設定」を参照してください。タグの制限については、このトピックで タグの制限 を参照してください。
5. タグを削除するには、タグの右側にあるXを選択します。
6. タグの編集または削除を完了したら、[サンプリングルールの更新] を選択します。

でのタグの管理 AWS CLI

X-Ray グループまたはサンプリングルールの作成時に、タグを追加できます。を使用してタグ AWS CLI を作成および管理することもできます。既存のグループまたはサンプリングルールのタグを更新するには、 AWS X-Ray コンソール、または TagResource または UntagResource APIsを使用します。

トピック

- 新しい X-Ray グループまたはサンプリングルールにタグを追加する (CLI)
- 既存のリソースにタグを追加する (CLI)
- リソースのタグを一覧表示する (CLI)
- リソースのタグを削除する (CLI)

新しい X-Ray グループまたはサンプリングルールにタグを追加する (CLI)

新しい X-Ray グループまたはサンプリングルールの作成時にオプションのタグを追加するには、次のいずれかのコマンドを使用します。

- 新しいグループにタグを追加するには、以下のコマンドを実行し、*group_name*をグループ名に、*mydomain.com*をサービスのエンドポイントに、*key_name*をタグキーに、オプションで、*#*をタグ値に置き換えてください。グループの作成方法の詳細については、「グループ」を参照してください。

```
aws xray create-group \
   --group-name "group_name" \
   --filter-expression "service(\"mydomain.com\") {fault OR error}" \
   --tags [{"Key": "key_name","Value": "value"},{"Key": "key_name","Value": "value"}]
```

以下に例を示します。

```
aws xray create-group \
   --group-name "AdminGroup" \
   --filter-expression "service(\"mydomain.com\") {fault OR error}" \
   --tags [{"Key": "Stage","Value": "Prod"},{"Key": "Department","Value": "QA"}]
```

- 新しいサンプリングルールにタグを追加するには、以下のコマンドを実行し、*key_name*をタグキーに、オプションで、*#*をタグ値に置き換えてください。このコマンドは、`--sampling-rule`

パラメータの値をJSON ファイルとして指定します。サンプリングルールの作成方法の詳細については、「サンプリングルール」を参照してください。

```
aws xray create-sampling-rule \
   --cli-input-json file://file_name.json
```

--cli-input-json パラメーターで指定された JSON ファイル*file_name.json*の内容は以下の通りです。

```
{
    "SamplingRule": {
        "RuleName": "rule_name",
        "RuleARN": "string",
        "ResourceARN": "string",
        "Priority": integer,
        "FixedRate": double,
        "ReservoirSize": integer,
        "ServiceName": "string",
        "ServiceType": "string",
        "Host": "string",
        "HTTPMethod": "string",
        "URLPath": "string",
        "Version": integer,
        "Attributes": {"attribute_name": "value","attribute_name": "value"...}
    }
    "Tags": [
           {
               "Key":"key_name",
               "Value":"value"
           },
           {
               "Key":"key_name",
               "Value":"value"
           }
          ]
}
```

コマンドの例を次に示します。

```
aws xray create-sampling-rule \
   --cli-input-json file://9000-base-scorekeep.json
```

--cli-input-json パラメーターで指定されたサンプル 9000-base-scorekeep.json ファイルの内容は以下の通りです。

```
{
    "SamplingRule": {
        "RuleName": "base-scorekeep",
        "ResourceARN": "*",
        "Priority": 9000,
        "FixedRate": 0.1,
        "ReservoirSize": 5,
        "ServiceName": "Scorekeep",
        "ServiceType": "*",
        "Host": "*",
        "HTTPMethod": "*",
        "URLPath": "*",
        "Version": 1
    }
    "Tags": [
           {
               "Key":"Stage",
               "Value":"Prod"
           },
           {
               "Key":"Department",
               "Value":"QA"
           }
          ]
}
```

既存のリソースにタグを追加する (CLI)

tag-resourceコマンドを実行して既存の X-Ray グループまたはサンプリングルールにタグを追加するこの方法は、update-group や update-sampling-rule を実行してタグを追加するよりも簡単な場合があります。

グループまたはサンプリングルールにタグを追加するには、次のコマンドを実行し、ARN をリソースの ARN に置き換え、追加したいタグのキーとオプションの値を指定します。

```
aws xray tag-resource \
   --resource-arn "ARN" \
```

```
   --tag-keys [{"Key":"key_name","Value":"value"}, {"Key":"key_name","Value":"value"}]
```

以下に例を示します。

```
aws xray tag-resource \
   --resource-arn "arn:aws:xray:us-east-2:01234567890:group/AdminGroup" \
   --tag-keys [{"Key": "Stage","Value": "Prod"},{"Key": "Department","Value": "QA"}]
```

リソースのタグを一覧表示する (CLI)

`list-tags-for-resource`コマンドを実行して、X-Ray グループまたはサンプリングルールのタグを一覧表示できます。

グループまたはサンプリングルールに関連付けられているタグを一覧表示するには、次のコマンドを実行し、ARN をリソースの ARN に置き換えます。

```
aws xray list-tags-for-resource \
   --resource-arn "ARN"
```

以下に例を示します。

```
aws xray list-tags-for-resource \
   --resource-arn "arn:aws:xray:us-east-2:01234567890:group/AdminGroup"
```

リソースのタグを削除する (CLI)

`untag-resource`コマンドを実行して、X-Ray グループまたはサンプリングルールからタグを削除できます。

グループまたはサンプリングルールからタグを削除するには、次のコマンドを実行し、ARN をリソースの ARN に置き換え、削除したいタグのキーを指定します。

`untag-resource`コマンドでタグ全体のみを削除できます。タグ値を削除するには、X-Ray コンソールを使用するか、タグを削除し、同じキーで異なる値または空の値で新しいタグを追加します。

```
aws xray untag-resource \
   --resource-arn "ARN" \
   --tag-keys ["key_name","key_name"]
```

以下に例を示します。

```
aws xray untag-resource \
   --resource-arn "arn:aws:xray:us-east-2:01234567890:group/group_name" \
   --tag-keys ["Stage","Department"]
```

タグに基づいて X-Ray リソースへのアクセスを制御する

X-Ray グループまたはサンプリングルールにタグをアタッチしたり、リクエストでタグを X-Ray に渡すことができます。タグに基づいてアクセスを制御するには`xray:ResourceTag/key-name`、`aws:RequestTag/key-name`、または `aws:TagKeys` の条件キーを使用して、ポリシーの条件要素でタグ情報を提供します。これらの条件キーの詳細については、AWS 「リソースタグを使用したリソースへのアクセスの制御」を参照してください。

リソースのタグに基づいてリソースへのアクセスを制限するためのアイデンティティベースポリシーの例を表示するには、「タグに基づいて X-Ray グループおよびサンプリングルールへのアクセスを管理する」を参照してください。

トラブルシューティング AWS X-Ray

このトピックでは、X-Ray、API、コンソール、または SDK を使用する際に発生する可能性のある一般的なエラーと問題を示します。ここに記載されていない問題が見つかった場合は、このページの [Feedback] ボタンを使用して報告することができます。

セクション

- X-Ray トレースマップおよびトレース詳細ページ
- X-Ray SDK for Java
- Node.jsに使われる X-Ray SDK
- X-Ray デーモン

X-Ray トレースマップおよびトレース詳細ページ

以下のセクションは、X-Ray トレースマップおよびトレース詳細ページの使用に次のような問題がある場合に役立ちます。

CloudWatch ログがすべて表示されない

X-Ray トレースマップおよびトレース詳細ページにログが表示されるようにログを設定する方法は、サービスによって異なります。

- API Gateway でログが有効になっている場合は、API Gateway のログが表示されます。

すべてのサービスマップのノードが関連ログの表示をサポートしているわけではありません。次のノードタイプのログを表示します。

- Lambda コンテキスト
- Lambda function
- API Gateway のステージ
- Amazon ECS クラスター
- Amazon ECS インスタンス
- Amazon ECS サービス
- Amazon ECS タスク

- Amazon EKS クラスター
- Amazon EKS 名前空間
- Amazon EKS ノード
- Amazon EKS ポッド
- Amazon EKS サービス

X-Ray トレースマップに自分のすべてのアラームが表示されない

あるノードに関連付けられたアラームが ALARM 状態にある場合、X-Ray ではそのノードの警告アイコンだけが表示されます。

トレースマップでは、次のロジックを使用してアラームがノードに関連付けられます。

- ノードが AWS サービスを表す場合、そのサービスに関連付けられた名前空間を持つすべてのアラームがノードに関連付けられます。例えば、タイプ `AWS::Kinesis` のノードは、CloudWatch 名前空間 `AWS/Kinesis` のメトリクスに基づくすべてのアラームにリンクされています。
- ノードが AWS リソースを表す場合、その特定のリソースのアラームがリンクされます。例えば、名前が「MyTable」のタイプ `AWS::DynamoDB::Table` のノードは、名前空間の `AWS/DynamoDB` メトリクスに基づいており、`TableName` ディメンションが `MyTable` に設定されているすべてのアラームにリンクされます。
- ノードのタイプが不明で、名前の周囲に破線で囲まれている場合、そのノードにはアラームが関連付けられていません。

トレースマップに一部の AWS リソースが表示されない

すべての AWS リソースが専用ノードで表されるわけではありません。一部の AWS サービスは、サービスへのすべてのリクエストに対して 1 つのノードで表されます。次のリソースタイプは、リソースごとのノードとともに表示されます。

- `AWS::DynamoDB::Table`
- `AWS::Lambda::Function`

  Lambda 関数は 2 つのノードで表されます - 1 つは Lambda コンテナ用で、もう 1 つは関数用です。これは、Lambda 関数のコールドスタートの問題を特定するのに役立ちます。Lambda コンテナノードは、Lambda 関数ノードと同じ方法でアラームとダッシュボードに関連付けられます。
- `AWS::ApiGateway::Stage`

- `AWS::SQS::Queue`
- `AWS::SNS::Topic`

トレースマップにノードが多すぎる

X-Ray グループを使用して、マップを複数のマップに分割します。詳細については、「グループでフィルター式を使用する」を参照してください。

X-Ray SDK for Java

エラー: "Thread-1" com.amazonaws.xray.exceptions.SegmentNotFoundException スレッドの例外: 'AmazonSNS' という名前のサブセグメントの開始に失敗しました: セグメントが見つかりません。

このエラーは、X-Ray SDK が への発信通話を記録しようとしたが AWS、開いているセグメントを見つけられなかったことを示します。これは次の場合に発生する可能性があります。

- サーブレットフィルタが設定されていない X–Ray、 SDK は、 という名前のフィルターを使用して、着信リクエストのセグメントを作成しますAWSXRayServletFilter。着信リクエストを計測するためのサーブレットフィルターを設定します。
- サーブレットコード以外の計測クライアントを使用しています 計測クライアントを使用して、スタートアップコードまたは着信リクエストに応答して実行されないその他のコードで呼び出しを行う場合は、セグメントを手動で作成する必要があります。例については、「スタートアップコードの作成」を参照してください。
- ワーカースレッドで計測クライアントを使用しています 新しいスレッドを作成すると、X-Ray レコーダーはオープンセグメントへの参照を失います。`getTraceEntity` および `setTraceEntity`メソッドを使用して、現在のセグメントまたはサブセグメント (`Entity`) への参照を取得し、スレッド内のレコーダーに戻すことができます。例については、「実装されたクライアントをワーカースレッドで使用する」を参照してください。

Node.jsに使われる X-Ray SDK

問題: Sequelize で CLS が機能しない

`cls`の方法で Node.js の名前空間で使われるX-Ray SDKをSequelizeへ渡します。

```
var AWSXRay = require('aws-xray-sdk');
```

```
const Sequelize = require('sequelize');
Sequelize.cls = AWSXRay.getNamespace();
const sequelize = new Sequelize(...);
```

問題: Bluebird で CLS が機能しない

`cls-bluebird` を使用して、Bluebird が CLS で動作するようにします。

```
var AWSXRay = require('aws-xray-sdk');
var Promise = require('bluebird');
var clsBluebird = require('cls-bluebird');
clsBluebird(AWSXRay.getNamespace());
```

X-Ray デーモン

問題: デーモンが間違った認証情報を使用する

デーモンは AWS SDK を使用して認証情報をロードします。認証情報を提供する複数の方法を使用する場合は、優先順位が最も高い方法が使用されます。詳細については「デーモンを実行する」を参照してください。

のドキュメント履歴 AWS X-Ray

次の表に、 ドキュメントの重要な変更点を示します AWS X-Ray。このドキュメントの更新に関する通知を受け取るには、RSS フィードにサブスクライブできます。

ドキュメントの最終更新日: 2024 年 3 月 7 日

変更	説明	日付
追加された機能	X-Ray から OpenTelemetry への移行。詳細については、「X-Ray 計測から OpenTelemetry 計測への移行」を参照してください。	2025 年 6 月 13 日
追加された機能	AWS X-Ray がトランザクション検索をサポートするようになりました。詳細については、「Transaction Search」を参照してください。	2024 年 11 月 21 日
追加された機能	AWS X-Ray は OpenTelemetry Protocol (OTLP) エンドポイントをサポートするようになりました。詳細については、「OpenTelemetry」を参照してください。	2024 年 11 月 21 日
追加された機能	X-Ray は、、`PutTraceSegments` 、 `GetTraceSummaries` などのデータイベント`BatchGetTraces` をに記録するようになりました AWS CloudTrail。X-Ray は `GetSamplingStatisticSummaries` 管理イベントを CloudTrail のログに記録	2024 年 3 月 7 日

	するようになりました。詳細については、「 を使用した X-Ray API コールのログ記録 AWS CloudTrail」を参照してください。	
追加された機能	W3C 形式のトレース ID が X-Ray で確実に受け入れられるようにするため、バージョン 0.86.0 以降の X-Ray は、OpenTelemetry、または W3C トレースコンテキスト仕様に準拠するその他のフレームワークを介して作成されたトレース ID をサポートするようになりました。詳細については、「X-Ray へのトレースデータの送信」を参照してください。	2023 年 10 月 25 日
追加された機能	Amazon SNS アクティブトレースを設定して、Amazon SNS トピックを通過するリクエストをトレースして分析できるようになりました。詳細については、Amazon SNSと AWS X-Ray」を参照してください。	2023 年 2 月 8 日
X-Ray SDK for Node.js のトピックを更新	AWS SDK for JavaScript V3 を使用したクライアントの計測に関する詳細を追加しました。詳細については、「X-Ray AWS SDK for Node.js を使用した SDK 呼び出しのトレース」を参照してください。	2023 年 2 月 7 日

IAM 管理ポリシーの詳細を更新	`AWSXRayReadOnlyAccess` 、`AWSXRayFullAccess` 、`AWSXrayCrossAccountSharingConfiguration` の管理ポリシーにクロスアカウントオブザーバビリティ用の IAM アクセス許可を追加しました。詳細については、「X-Ray の IAM 管理ポリシー」を参照してください。	2023 年 2 月 7 日
追加された機能	AWS X-Ray はクロスアカウントオブザーバビリティをサポートするようになりました。これにより、 内の複数のアカウントにまたがるアプリケーションをモニタリングおよびトラブルシューティングできます AWS リージョン。詳細については、「クロスアカウントトレーシング」を参照してください。	2022 年 11 月 27 日
追加された機能	メッセージプロデューサー、Amazon SQS キュー、コンシューマー間のリンクされたトレースを表示できるようになりました。これにより、イベント駆動型アプリケーションから送信されたトレースを連結して表示できます。詳細については、「イベント駆動型アプリケーションのトレース」を参照してください。	2022 年 11 月 20 日

IAM 管理ポリシーの詳細を更新	リソースポリシーを一覧表示するための IAM アクセス許可を AWSXRayReadOnlyAccess 管理ポリシーに追加しました。詳細については、「X-Ray の IAM 管理ポリシー」を参照してください。	2022 年 11 月 15 日
IAM コンソールのアクセス許可と管理ポリシーの詳細を更新	X-Ray コンソールが使用する IAM アクセス許可のセットが、AWSXRayReadOnlyAccess 管理ポリシーの説明とともに更新されました。詳細については、「X-Ray コンソールの使用」を参照してください。	2022 年 11 月 11 日
AWS Distro for OpenTelemetry Ruby を追加	AWS Distro for OpenTelemetry (ADOT) は、分散トレースとメトリクスを収集するためのオープンソース APIs、ライブラリ、エージェントの単一のセットを提供します。ADOT Ruby では、X-Ray やその他のトレースバックエンドの Ruby アプリケーションを計測できます。詳細については、「AWS Distro for OpenTelemetry Ruby」を参照してください。	2022 年 2 月 7 日
追加された機能	CloudWatch コンソールからトレースを表示し、X-Ray を設定できるようになりました。詳細については、「X-Ray コンソール」を参照してください。	2022 年 1 月 24 日

CloudWatch RUM の統合	AWS X-Ray と CloudWatch RUM を使用すると、ダウンストリームの AWS マネージドサービスを通じて、アプリケーションのエンドユーザーからのリクエストパスを分析およびデバッグできます。詳細については、CloudWatch RUM と AWS X-Ray」を参照してください。	2021 年 12 月 3 日
OpenTelemetry 用の統合 AWS Distro	AWS Distro for OpenTelemetry (ADOT) は、分散トレースとメトリクスを収集するためのオープンソース APIs、ライブラリ、エージェントの単一のセットを提供します。ADOT では、X-Ray やその他のトレースバックエンドのアプリケーションを計測できます。詳細については、「」を参照してください。アプリケーションの作成。	2021 年 9 月 23 日
追加された機能	AWS X-Ray は Amazon Virtual Private Cloud と統合され、Amazon VPC 内のリソースがパブリックインターネットを経由せずに X-Ray サービスと通信できるようになりました。詳細については、「VPC エンドポイント AWS X-Ray での の使用」を参照してください。	2021 年 5 月 20 日

追加された機能	AWS X-Ray は と統合され AWS CloudFormation、X-Ray リソースのプロビジョニングと設定が可能になりました。詳細については、CloudFormation での X-Ray リソースの作成を参照してください。	2021 年 5 月 6 日
追加された機能	AWS X-Ray は Amazon EventBridge と統合され、EventBridge を通過するイベントを追跡するようになりました。これにより、ユーザーはシステムのより完全なビューを表示できます。詳細については、「Amazon EventBridge」および AWS X-Ray「」を参照してください。	2021 年 3 月 2 日
ECR にデーモンを追加しました	デーモンは Amazon ECR からダウンロードできるようになりました。詳細については、「 daemon のダウンロード」ページを参照してください。	2021 年 3 月 1 日
追加された機能	AWS X-Ray は、Amazon EventBridge へのインサイト関連の通知をサポートするようになりました。これにより、EventBridge を使用してインサイトに対して自動アクションを実行できます。詳細については、インサイト通知を参照してください。	2020 年 10 月 15 日

ダウンロード可能なデーモンを追加	AWS X-Ray では、Linux ARM64 のサポートデーモンが導入されました。詳細については、「」を参照してください。AWS X-Ray daemonBrazil WS	2020 年 10 月 1 日
追加された機能	AWS X-Ray は、Amazon CloudWatch Synthetics とのアクティブな統合をサポートするようになりました。これにより、応答時間やステータスなど、Synthetics canaryアクライアントノードの詳細を表示できます。Synthetics canaryアクライアントノードからの情報に基づいて Analytics コンソールで分析を実行することもできます。詳細については、「」を参照してください。X-Ray を使用した CloudWatch 合成 canaryアのデバッグ。	2020 年 9 月 24 日

追加された機能	AWS X-Ray はend-to-endワークフローのトレースをサポートするようになりました AWS Step Functions。ステートマシンのコンポーネントの視覚化、パフォーマンスのボトルネックの特定、およびエラーの原因となったリクエストのトラブルシューティングを行うことができます。詳細については、AWS Step Functions「」および AWS X-Ray「」を参照してください。	2020 年 9 月 14 日
追加された機能	AWS X-Ray では、アカウント内のトレースデータを継続的に分析して、アプリケーションの緊急の問題を特定するためのインサイトが導入されています。Insights はインシデントを記録し、解決するまでインシデントの影響を追跡します。詳細については、「AWS X-Ray コンソールでのインサイトの使用」を参照してください。	2020 年 9 月 3 日

追加された機能	AWS X-Ray では Java 自動計測エージェントが導入されており、既存の Java ベースのアプリケーションを変更することなくトレースデータを収集できます。Java Web およびサーブレットベースのアプリケーションを、最小限の構成変更でコードの変更なしでトレースできるようになりました。詳細については、「」を参照してください。AWS X-Ray Java 用自動インストルメンテーションエージェント。	2020 年 9 月 3 日
追加された機能	AWS X-Ray は、トレースのグループの作成と管理を容易にするために、X-Ray コンソールに新しいグループページを追加しました。詳細については、「X-Ray コンソールでのグループの設定」を参照してください。	2020 年 8 月 24 日

追加された機能	AWS X-Ray では、グループとサンプリングルールにタグを追加できるようになりました。タグに基づいてグループおよびサンプリングルールへのアクセスを制御することもできます。詳細については、「」を参照してください。X-Ray のサンプリングルールとグループの作成そしてタグに基づいて X-Ray グループおよびサンプリングルールへのアクセスを管理する。	2020 年 8 月 24 日

翻訳は機械翻訳により提供されています。提供された翻訳内容と英語版の間で齟齬、不一致または矛盾がある場合、英語版が優先します。