Hands-on: Instrumenting AWS Lambda with AWS X-Ray

19 January 2024 - 2 min. read

Application Monitoring | AWS Lambda | Debugging | Microservices

Introduction

Working with microservices has many advantages, but it also has some flaws. One of them undoubtedly is debugging and logging the whole application.

Every microservice may produce logs in a different place, and jumping from one to another while debugging could be time-consuming.

Moreover, knowing what microservices are part of an application and finding bottlenecks or points of failure can be tricky, especially with massive and poorly documented applications.

X-Ray solves both problems, providing a centralized point to access your application logs and consult a map representing your microservices, resources, and how they communicate.

X-Ray Introduction

X-Ray is a service that allows you to gather data from applications you want to monitor so that you can analyze them to find possible issues or ways to improve your application performance. Sending trace data for incoming and outbound requests and other events lets you see which AWS resources, microservices, and web APIs are used.

This process is called **"Instrumentation"**.

There are different types of instrumentations:

- **Auto instrumentation**: This is done via configuration changes and requires zero code changes.

- **Library instrumentation**: it requires minimal application code changes through specific libraries or frameworks, such as the AWS SDK.
- **Manual instrumentation**: you add instrumentation code to your application where you want to send trace information.

Many AWS services provide X-Ray integration, such as AWS Lambda, Amazon API Gateway, Amazon Simple Notification Service, or Amazon Simple Queue Service.

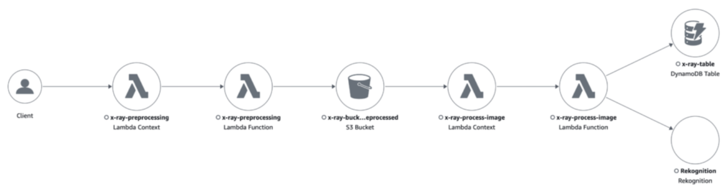
AWS services integration with X-Ray can be split into:

- Active instrumentation: samples and instruments incoming requests
- Passive instrumentation: instruments requests that another service has sampled
- Request tracing: adds a tracing header to all incoming requests
- Tooling: receives segments from the X-Ray SDK running the X-Ray daemon

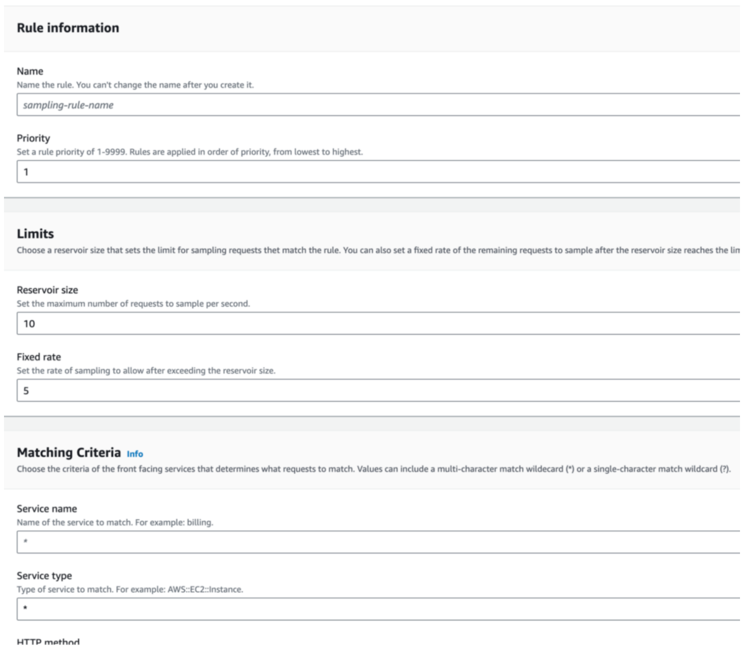
X-Ray Fundamentals

The data sent by your application services are called **segments.** Segments contain different information based on the nature of the request. For instance, an HTTP request will contain information such as the URL, the method, and the response code.

A segment can be split into **subsegments** to provide more information about all the actors involved in the work. The segment of your application can be split into different subsegments for each function it runs to know the functions' order and duration. Subsegments are also used by X-Ray to create "inferred segments" for services that don't support X-Ray, such as DynamoDB. This allows X-Ray to show them in the **trace map.** The trace map shows all the services involved and can be used to analyze the application workflow. The trace map is built using **traces.** A trace is a collection of all segments generated by a single request to your application.



X-Ray does not trace every request, but it follows a sample rate. The sample rate can be customized in the X-Ray settings by creating rules to apply to specific front-facing services.



Rule information

Name
Name the rule. You can't change the name after you create it.
sampling-rule-name

Priority
Set a rule priority of 1-9999. Rules are applied in order of priority, from lowest to highest.
1

Limits
Choose a reservoir size that sets the limit for sampling requests thet match the rule. You can also set a fixed rate of the remaining requests to sample after the reservoir size reaches the lin

Reservoir size
Set the maximum number of requests to sample per second.
10

Fixed rate
Set the rate of sampling to allow after exceeding the reservoir size.
5

Matching Criteria Info
Choose the criteria of the front facing services that determines what requests to match. Values can include a multi-character match wildecard (*) or a single-character match wildcard (?).

Service name
Name of the service to match. For example: billing.
*

Service type
Type of service to match. For example: AWS::EC2::Instance.
*

HTTP method

If your application generates many traces, you can filter them for a more accurate insight using **filter expressions.** Filter expressions allow you to filter information like traces that generated errors in specific services, traces that originated from certain URLs, or traces whose duration exceeded a certain amount of time.

You can also extend filtering by creating a **group.** Groups will contain all traces whose respect the filter expression generated after the creation of the group. In groups, you can activate insights and insight notifications. X-Ray identifies emergent issues in your applications and creates insights when ranges are exceeded. If notifications are enabled, you receive a notification for every insight event.

X-Ray tracks errors that occur in your application and errors that are returned by downstream services. Errors are grouped as follows:

- Error - 4xx HTTP errors
- Fault - 5xx HTTP errors
- Throttle - 429 HTTP errors

X-Ray encrypts traces and data at rest with an AWS-managed encryption key. If you want to turn off encryption or use a key managed by yourself because of compliance requirements, you can configure X-Ray to use a KMS key.

Pricing

You pay based on the number of traces recorded, retrieved, and scanned.

X-Ray provides a minimum guaranteed trace size of 100KB.

Trace data is retained for 30 days from when it is recorded at no additional cost.

X-Ray has a free tier:

- The first 100,000 traces recorded each month are free.
- The first 1,000,000 traces retrieved or scanned each month are free.

You can customize the sample rate to choose the right compromise between costs and the quantity of data to gather.

Example Application

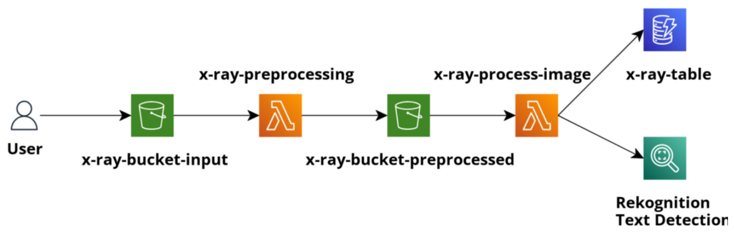
We created a simple example application to show you the setup of X-Ray instrumentation and its outputs.

The purpose of the application consists of pre-processing images, sending them to Rekognition to detect text inside of them, and saving the output to a DynamoDB table for later analysis of the results.

The application comprises an S3 bucket where we upload the images we want to analyze.

The creation of objects inside the bucket triggers a Lambda function, which is in charge of applying some pre-processing transformations to the image.

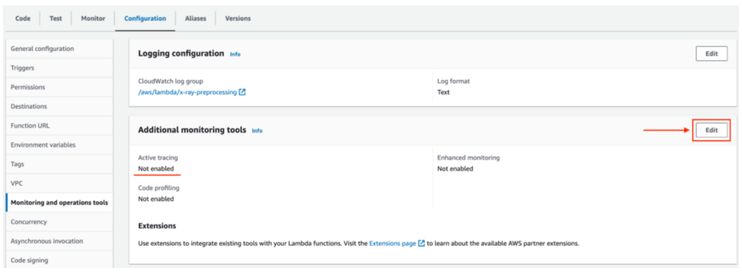
The output of the pre-processing Lambda function will be saved in another S3 bucket, which will trigger another Lambda function to send the image to Rekognition and save the results of the “Text Detection” functionality into a DynamoDB table.



First, to gather data from our application, we must activate tracing in its Lambda functions.

It can be done:

- From the AWS console:



Logging configuration

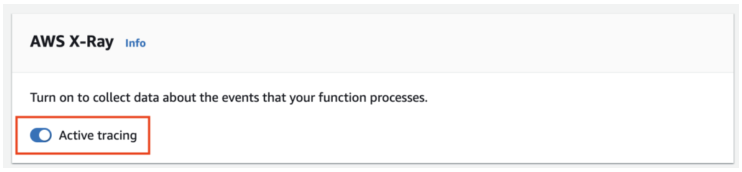
CloudWatch log group: /aws/lambda/x-ray-preprocessing
Log format: Text

Additional monitoring tools

Active tracing: Not enabled
Enhanced monitoring: Not enabled
Code profiling: Not enabled

Extensions

Use extensions to integrate existing tools with your Lambda functions. Visit the Extensions page to learn about the available AWS partner extensions.



AWS X-Ray Info

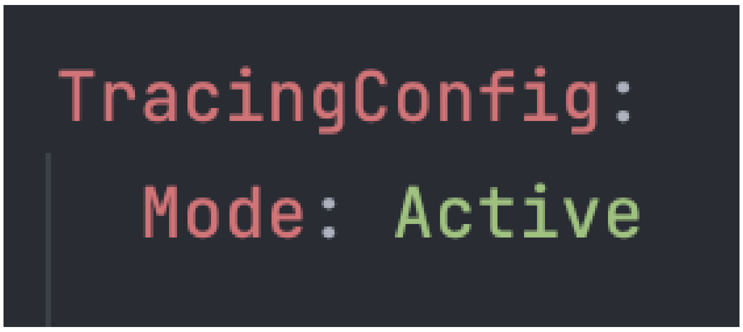
Turn on to collect data about the events that your function processes.

Active tracing

- Through the AWS CLI:

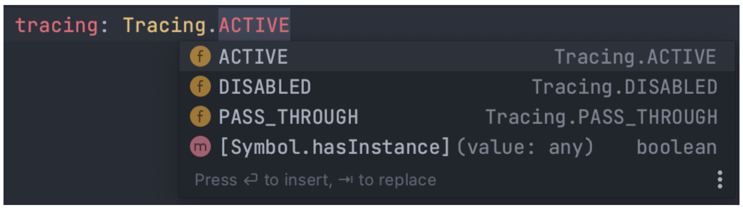
```
aws lambda update-function-configuration --function-name my-function --tr
acing-config Mode=Active
```

- As a configuration of the IaC resources:
  - CloudFormation



```
TracingConfig:
  Mode: Active
```

- CDK



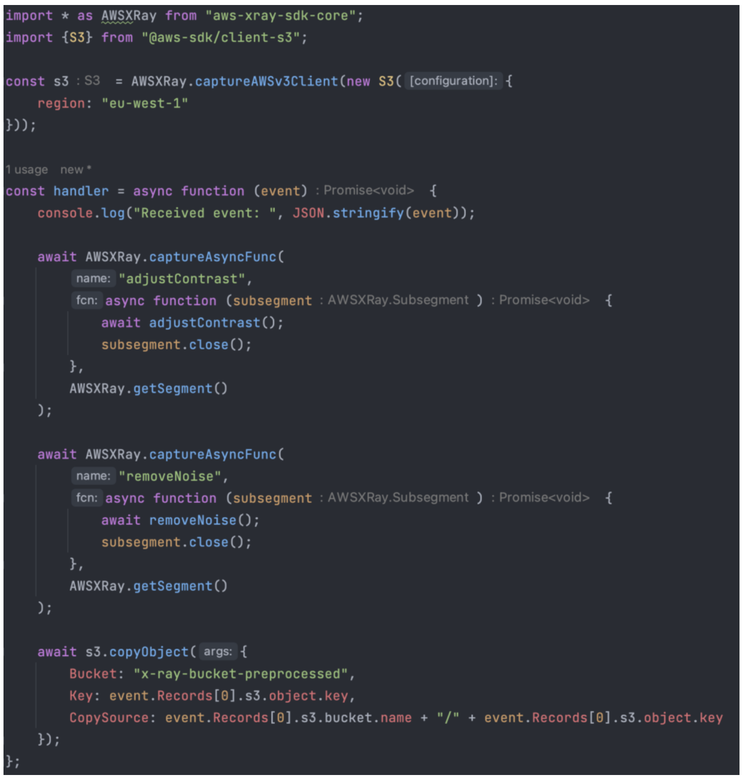
```
tracing: Tracing.ACTIVE
              ACTIVE                          Tracing.ACTIVE
              DISABLED                        Tracing.DISABLED
              PASS_THROUGH                    Tracing.PASS_THROUGH
              [Symbol.hasInstance](value: any)     boolean
              Press ⏎ to insert, ⇥ to replace
```

Now that the Lambda function’s tracing is enabled, we must configure X-Ray in our code.

One way of doing so is by manual instrumentation, thanks to the X-Ray SDK.

Our Lambda functions run on Node.js and use SDK v3. With the AWS SDK v2, you can activate the capture of all outgoing requests made by the library. Since the AWS SDK V3 has a modular structure, you need to capture each client the application uses. By doing so, every time the clients send a command, a subsegment will be extracted from the Lambda invocation segment, giving us a more precise insight into what happened while executing that command and how much time it required to complete.

You can do the same thing with functions and methods inside your code to know how much every invocation lasted compared to the total duration of the Lambda function execution.

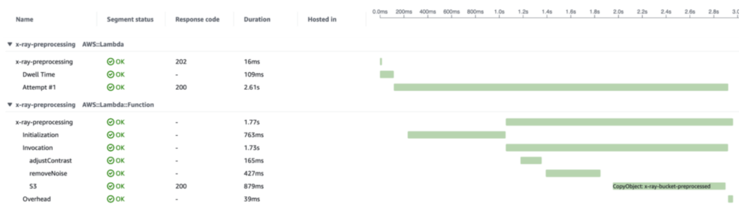


```
import * as AWSXRay from "aws-xray-sdk-core";
import {S3} from "@aws-sdk/client-s3";

const s3 : S3  = AWSXRay.captureAWSv3Client(new S3( [configuration]: {
    region: "eu-west-1"
}));

1 usage   new *
const handler = async function (event) : Promise<void>  {
    console.log("Received event: ", JSON.stringify(event));

    await AWSXRay.captureAsyncFunc(
        name: "adjustContrast",
        fcn: async function (subsegment : AWSXRay.Subsegment ) : Promise<void>  {
            await adjustContrast();
            subsegment.close();
        },
        AWSXRay.getSegment()
    );

    await AWSXRay.captureAsyncFunc(
        name: "removeNoise",
        fcn: async function (subsegment : AWSXRay.Subsegment ) : Promise<void>  {
            await removeNoise();
            subsegment.close();
        },
        AWSXRay.getSegment()
    );

    await s3.copyObject( args: {
        Bucket: "x-ray-bucket-preprocessed",
        Key: event.Records[0].s3.object.key,
        CopySource: event.Records[0].s3.bucket.name + "/" + event.Records[0].s3.object.key
    });
};
```

In the case of our pre-processing Lambda Function, we expect the invocation segment to be split into three subsegments:

- **adjustConstrast** subsegment
- **removeNoise** subsegment
- S3 **copyObject** subsegment

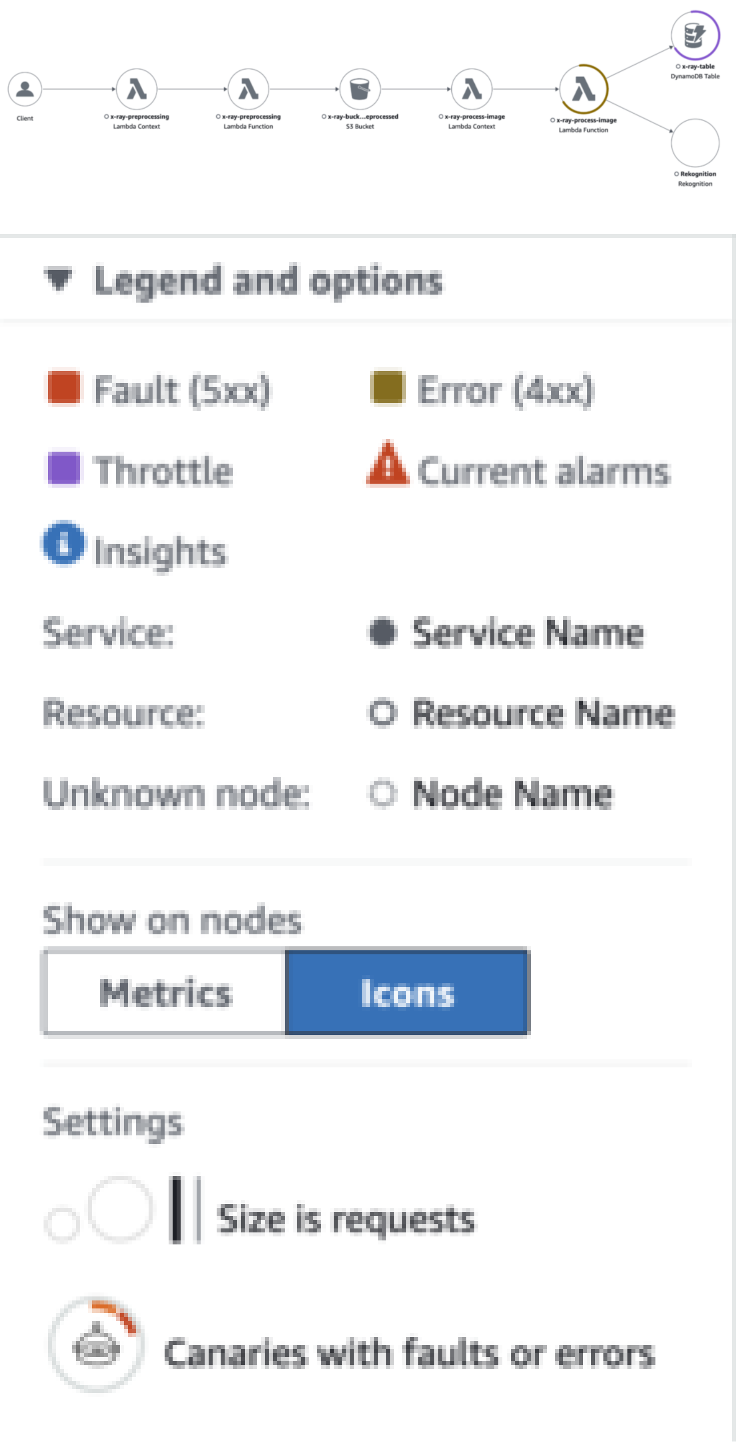
In the following there is the result of one invocation of the function:



Name	Segment status	Response code	Duration	Hosted in
▼ x-ray-preprocessing AWS::Lambda				
x-ray-preprocessing	✓ OK	202	16ms	
Dwell Time	✓ OK	-	109ms	
Attempt #1	✓ OK	200	2.61s	
▼ x-ray-preprocessing AWS::Lambda::Function				
x-ray-preprocessing	✓ OK	-	1.77s	
Initialization	✓ OK	-	763ms	
Invocation	✓ OK	-	1.73s	
adjustContrast	✓ OK	-	165ms	
removeNoise	✓ OK	-	427ms	
S3	✓ OK	200	879ms	
Overhead	✓ OK	-	39ms	

0.0ms 200ms 400ms 600ms 800ms 1.0s 1.2s 1.4s 1.6s 1.8s 2.0s 2.2s 2.4s 2.6s 2.8s 3.0
CopyObject: x-ray-bucket-preprocessed

By uploading some images to the input S3 bucket, we will have an overview of how the application manages the requests' workload.



▼ Legend and options

🟥 Fault (5xx) 🟫 Error (4xx)

🟪 Throttle ⚠ Current alarms

ℹ Insights

Service: ● Service Name

Resource: ○ Resource Name

Unknown node: ○ Node Name

Show on nodes

Metrics [✓ Icons]

Settings

Size is requests

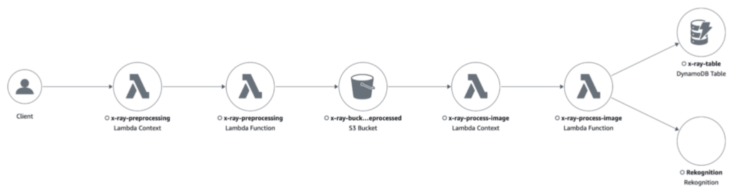
Canaries with faults or errors

The trace map shows that the process-image Lambda function is responding with some 4xx errors caused by the throttling of the DynamoDB table.

Clicking on the icon of a resource opens a side menu with all kinds of information about the interest resource. I could quickly see what was causing the error by clicking on the "Exceptions" tab, which showed the following error: "*The level of configured provisioned throughput for the table was exceeded. Consider increasing your provisioning level with the UpdateTable API*".

I changed the DynamoDB capacity mode from "provisioned" with low-capacity units to "on-demand".

With the new configuration, the application sustained the traffic without any errors.



Conclusion

In a world where microservices applications are growing in number and complexity, X-Ray provides powerful tools for debugging and analysis.

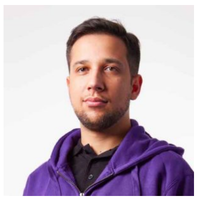
We wanted to create a small guide to its fundamentals and how to enable it in AWS Lambda functions.

Our application used manual instrumentation since it was small and written as an example, but in the case of significant applications, automatic instrumentation is a faster and easier option.

Let us know if you want to learn more about it!

About Proud2beCloud

Proud2beCloud is a blog by beSharp, an Italian APN Premier Consulting Partner expert in designing, implementing, and managing complex Cloud infrastructures and advanced services on AWS. Before being writers, we are Cloud Experts working daily with AWS services since 2007. We are hungry readers, innovative builders, and gem-seekers. On Proud2beCloud, we regularly share our best AWS pro tips, configuration insights, in-depth news, tips&tricks, how-tos, and many other resources. Take part in the discussion!



Daniele Papa

DevOps Engineer and backend developer @ beSharp. I like playing video games and board games in my free time. In the last few years, I approached the Cloud environment, and now I switch between IAM roles and role-play games.
