

Observability in Serverless Application

Ozioma Uzoegwu
Solutions Architect, AWS

 @iam_tessot

What we will cover in this session

What is a serverless application?

What is observability?

AWS services for observability

- Standard and custom metrics
- Structured logging
- Tracing

AWS open source observability services

Serverless applications

Event source



Function



Services
(anything)



Changes in
data state



Changes in
resource state



Requests to
endpoints



Node.js
Python

Java

Go

Ruby

.Net (C# / PowerShell)

Custom (Runtime API)

Traditional application stack

Business

Application + data

Runtime / Middleware

Operating system

VM / Container

Virtualization layer

Server hardware

Network / Storage

Serverless application stack

Business

Application + data

Runtime / Middleware

Operating system

VM / Container

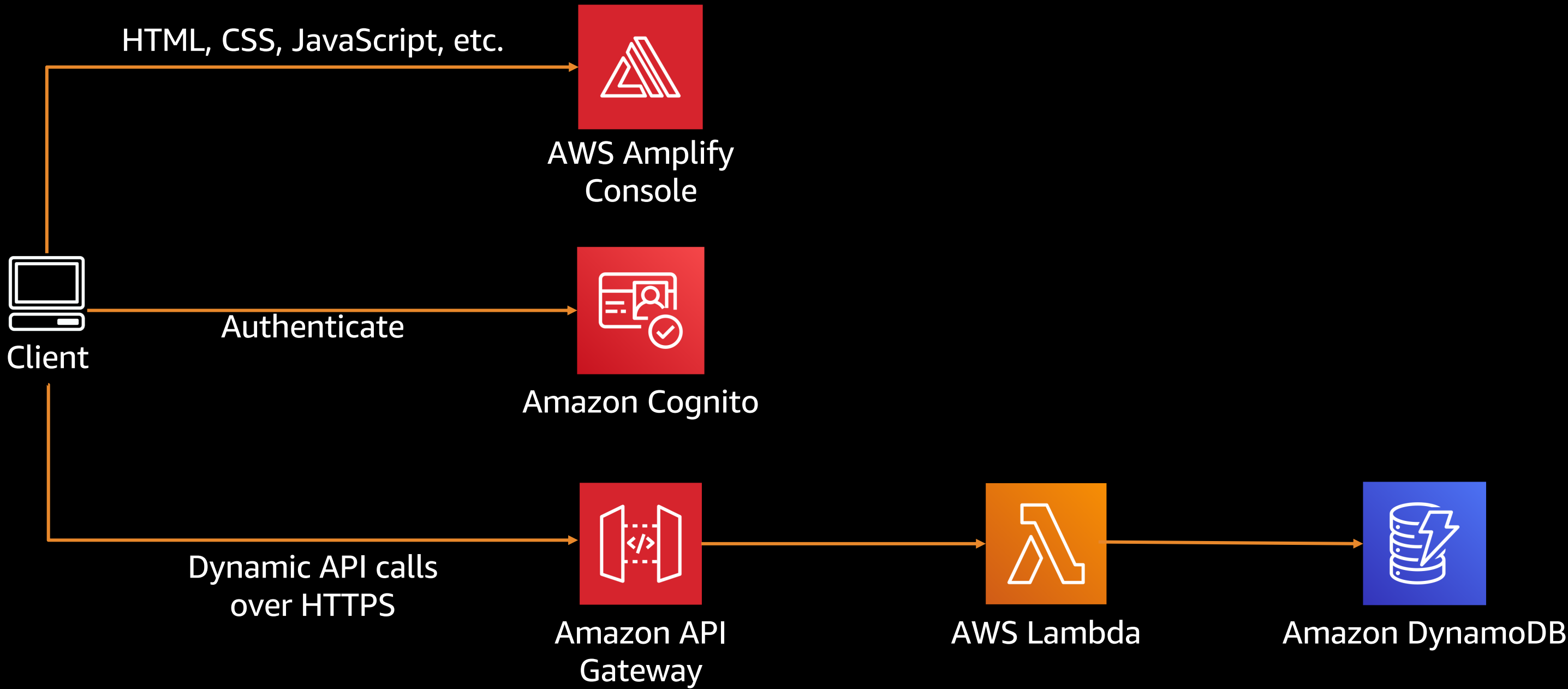
Serverless has you covered!

virtualization layer

Server hardware

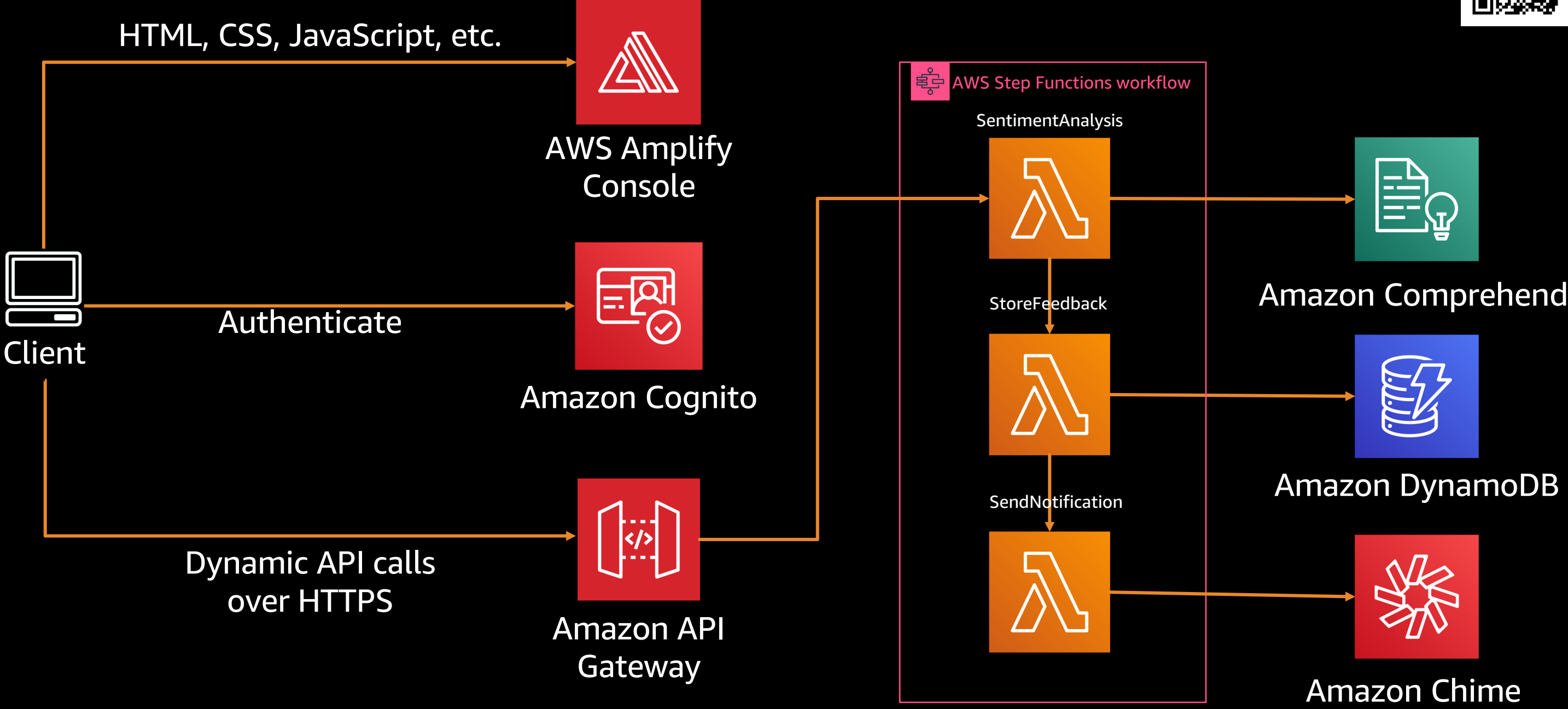
Network / Storage

Serverless Web Application Architecture



Serverless Web Application Architecture

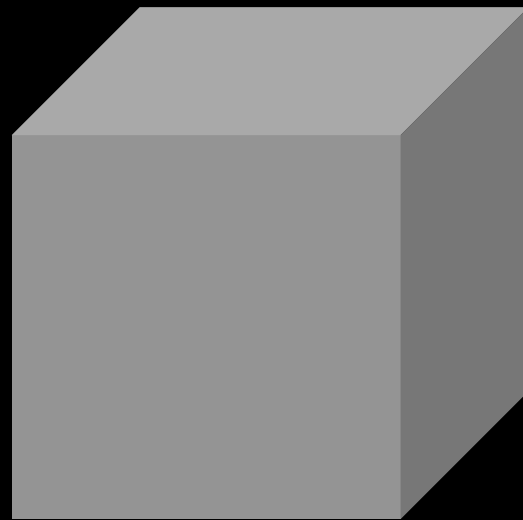
<https://s12d.com/serverless-feedback-app>



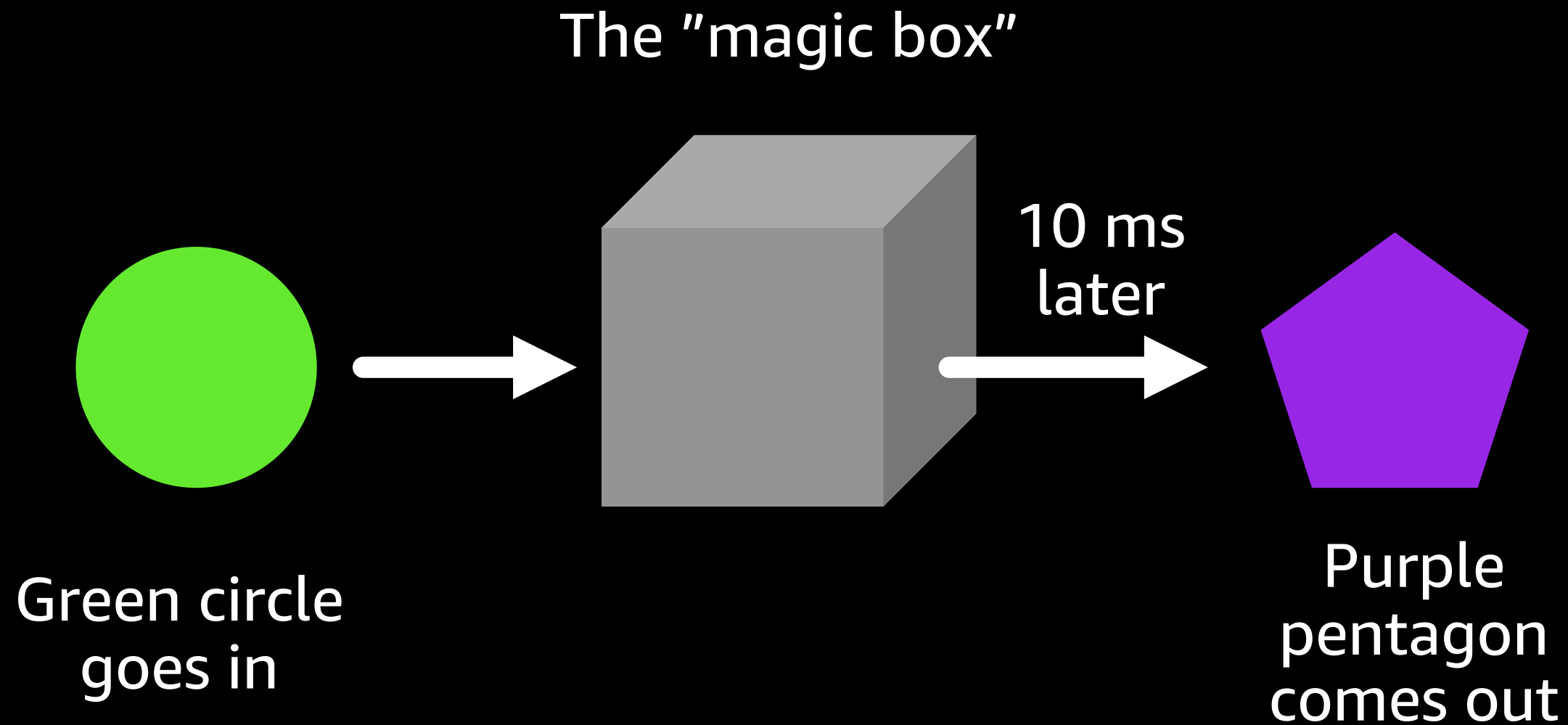
What is observability?

What is observability?

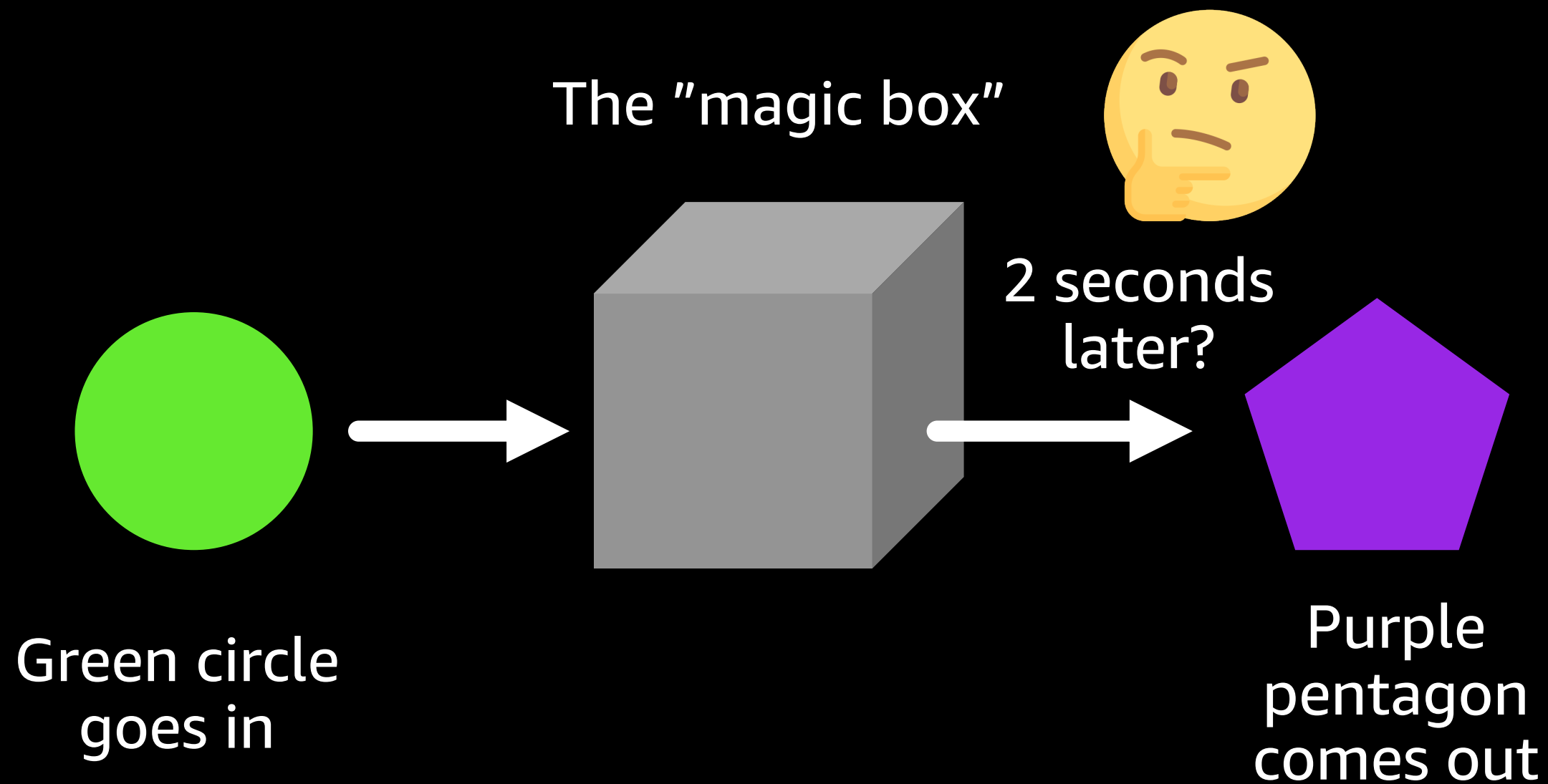
The "magic box"



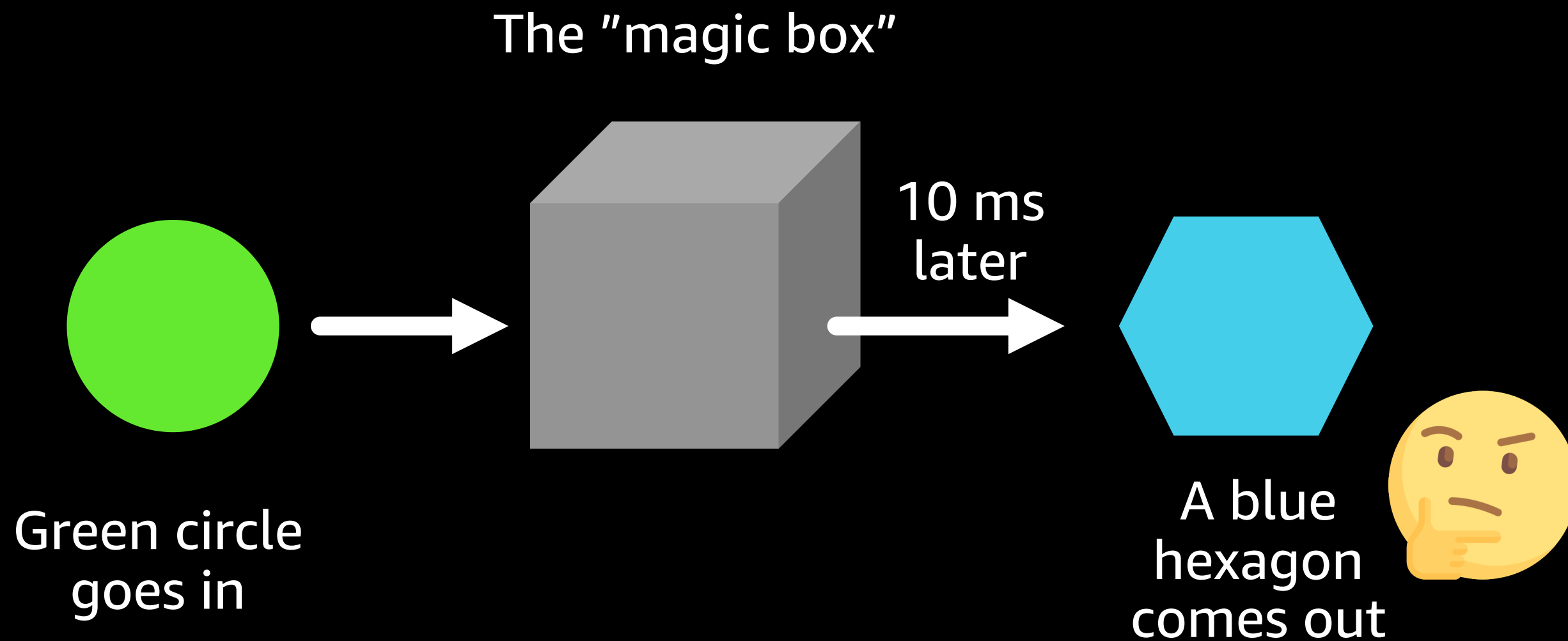
What is observability?



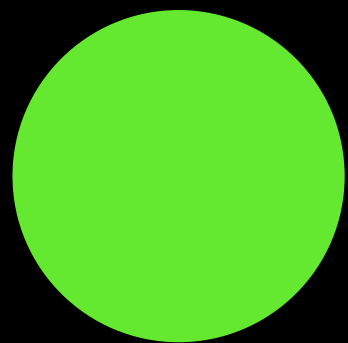
What is observability?



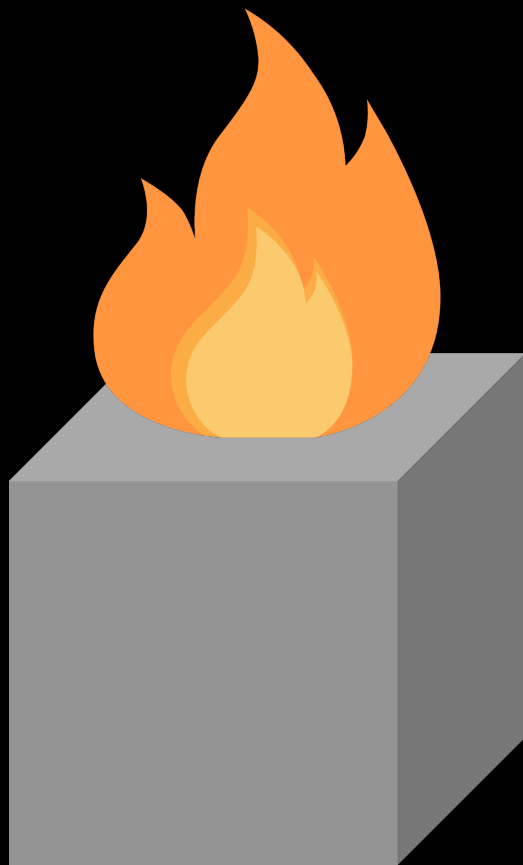
What is observability?



What is observability?



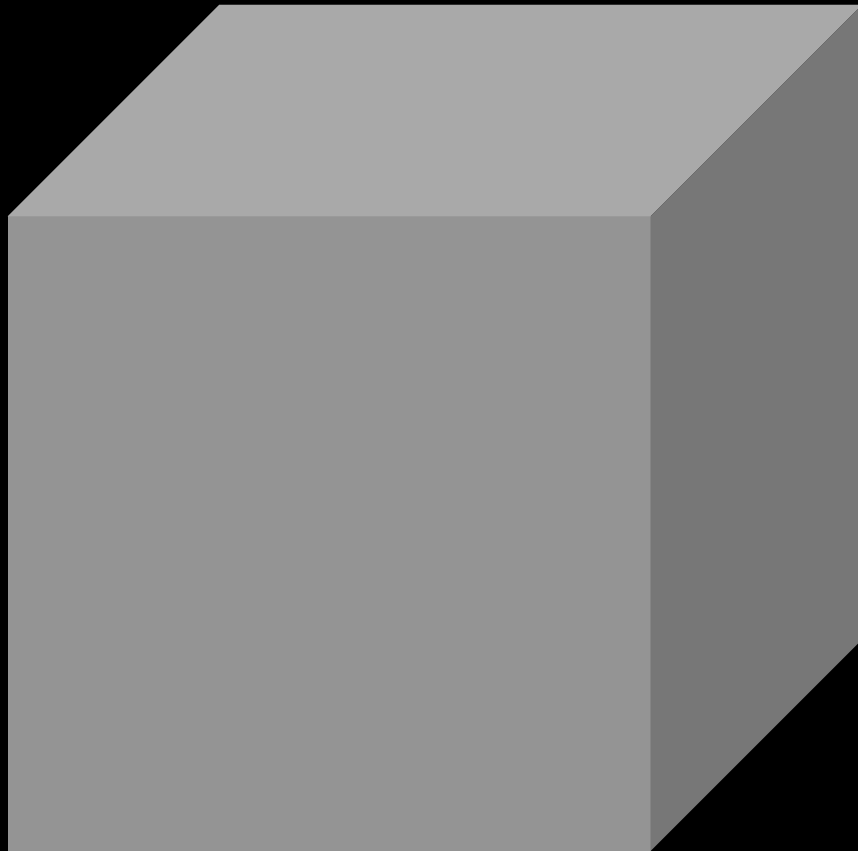
Green circle
goes in



The "magic box"
catches fire and
nothing comes
out!

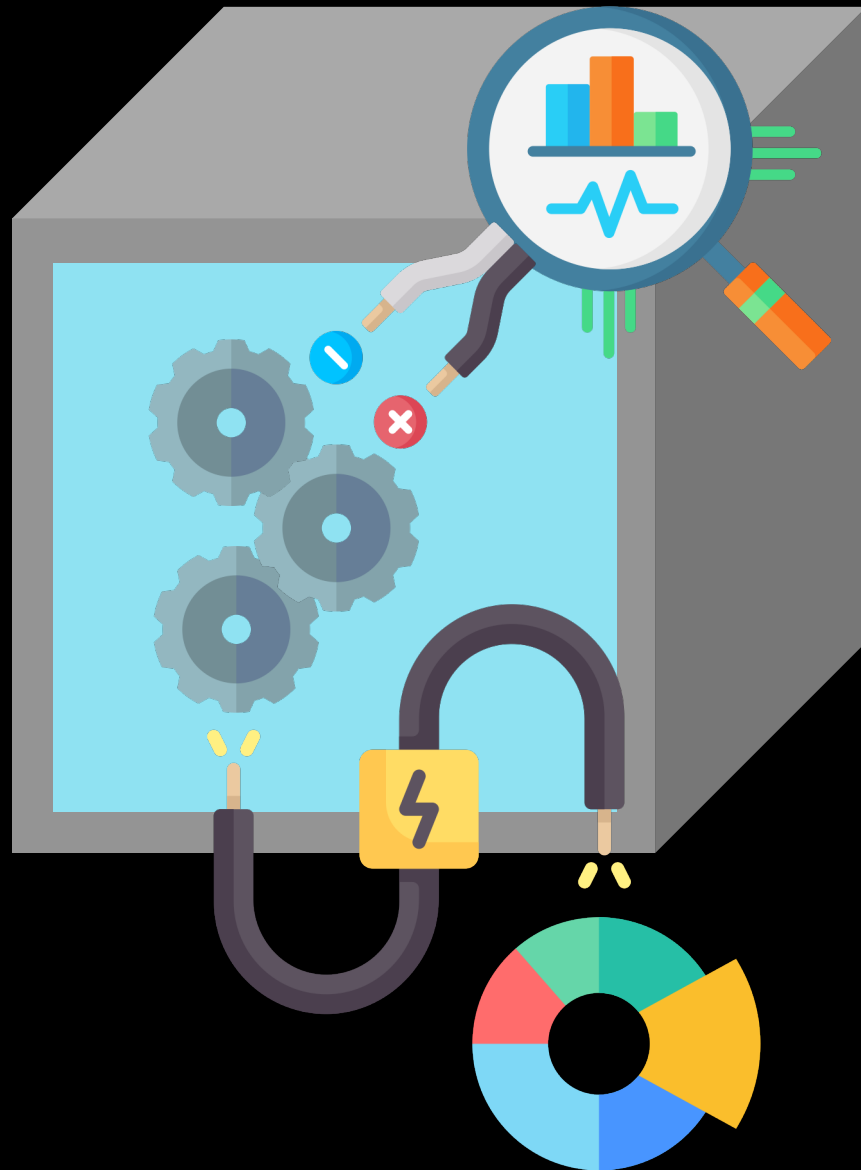


What is observability?



- We have no observability!
- What's in the box?
- Why does it behave the way it does?
- When its behavior changes, why did it change?
- What must be done to make this behavior more consistent?
- What is the usage?
- What is the business impact?

Observability must be proactive



Good observability allows you to answer questions you didn't know you needed to ask

When a problem happens you can access data about the system and understand it

Three pillars of observability tooling

Metrics

Numeric data measured at various time intervals (time series data); SLIs (request rate, error rate, duration, CPU%, etc.)

Logs

Timestamped records of discrete events that happened within an application or system, such as a failure, an error, or a state transformation

Traces

A trace represents a single user's journey across multiple applications and systems (usually microservices)

Definitions from "Distributed Systems Observability," by Cindy Sridharan. Available at:

<https://www.oreilly.com/library/view/distributed-systems-observability/9781492033431/>

Troubleshooting / query workflow

**Notification
/ question**

Identify

Traces

Analyze

Logs

**Receive an alarm
notification**
Ask a question

View service map
**Identify points
of interest to
dive deep**

**View traces,
trace maps,
and requests**
**Look at specific
API / service that
is the current
point of interest**

Trace analysis
**Perform deep
analysis of traces
if necessary**

Query logs
**At specific point
in time for
deeper analysis
and identify
root cause**

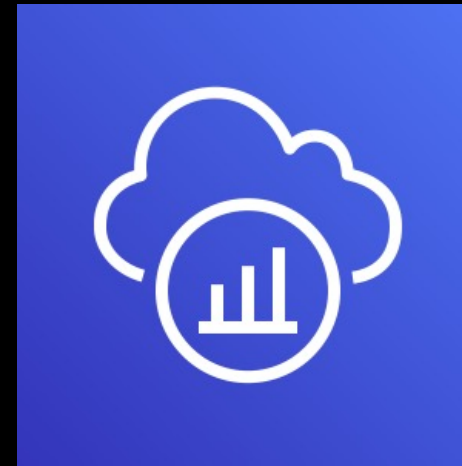
AWS services for observability

AWS services for observability



Amazon
CloudWatch

Dashboards
Logs
Metrics
Alarms
Events



AWS X-Ray

Traces
Analytics
Service map

Amazon CloudWatch

1 quadrillion+

(1,000,000,000,000,000+)

Metric observations each month

3.9 trillion

Events each month

Monitors entire
infrastructure of

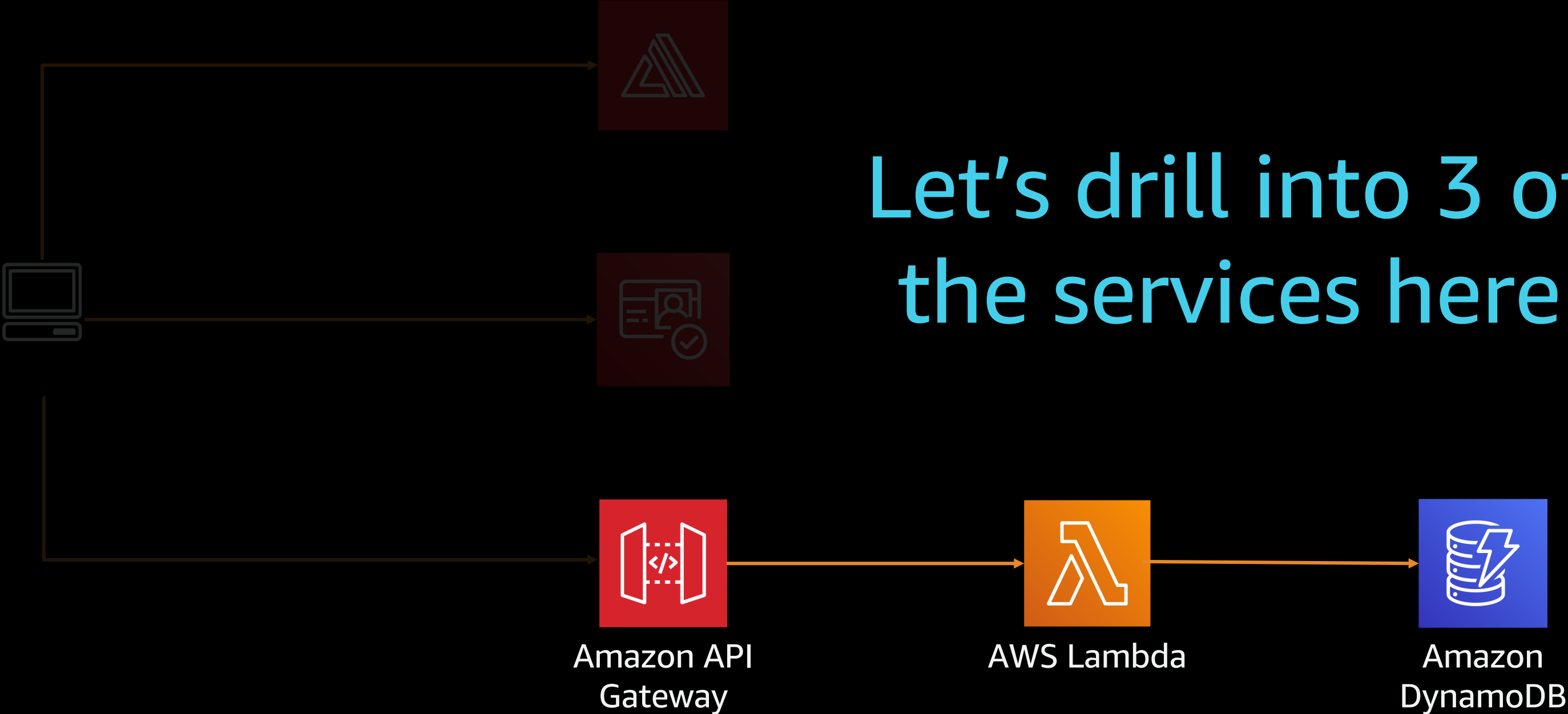
AWS and

Amazon.com

100 PB

Logs ingested each month

Let's drill into 3 of the services here



CloudWatch built-in metrics



AWS Lambda

Invocation metrics

Invocation count, **Invocation errors**,
DeadLetterErrors,
DestinationDeliveryFailures, **Throttles**,
ProvisionedConcurrencyInvocations,
ProvisionedConcurrencySpilloverInvocations

Performance metrics

Duration, IteratorAge

Concurrency metrics

ConcurrentExecutions,
ProvisionedConcurrentExecutions,
ProvisionedConcurrencyUtilization,
UnreservedConcurrentExecutions

Amazon API Gateway

REST APIs

API request count, **Latency**, 4XXs, **5XXs**,
IntegrationLatency, CacheHitCount,
CacheMissCount

HTTP APIs

API request count, **Latency**, 4XXs, **5XXs**,
IntegrationLatency, DataProcessed

WebSocket APIs

ConnectCount, MessageCount,
IntegrationError, ClientError,
ExecutionError, **IntegrationLatency**

CloudWatch built-in metrics



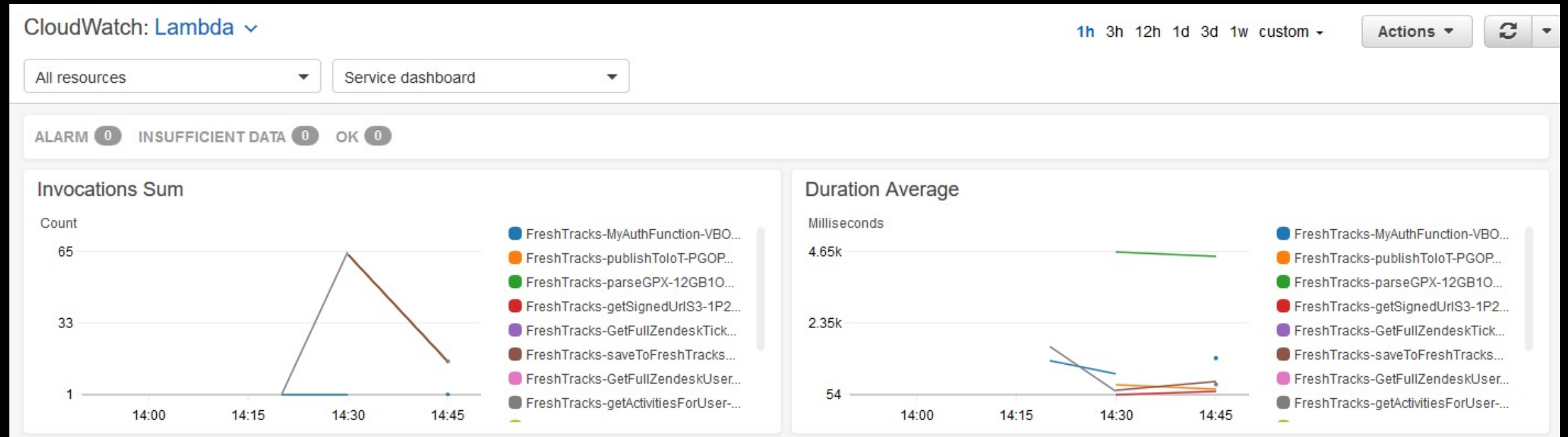
Amazon DynamoDB

AccountMaxReads,
AccountMaxTableLevelReads,
AccountMaxTableLevelWrites,
AccountMaxWrites,
AccountProvisionedReadCapacityUtilization,
AccountProvisionedWriteCapacityUtilization,
ConditionalCheckFailedRequests,
ConsumedReadCapacityUnits,
ConsumedWriteCapacityUnits,
MaxProvisionedTableReadCapacityUtilization,
MaxProvisionedTableWriteCapacityUtilization,
OnlineIndexConsumedWriteCapacity,

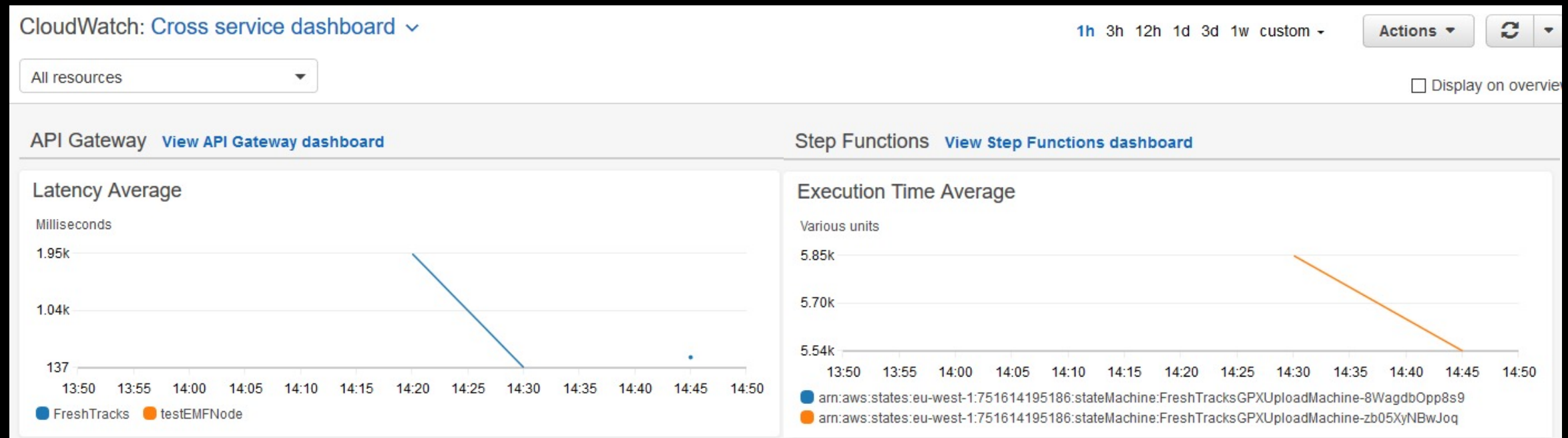
OnlineIndexPercentageProgress,
OnlineIndexThrottleEvents,
PendingReplicationCount,
ProvisionedReadCapacityUnits,
ProvisionedWriteCapacityUnits,
ReadThrottleEvents, ReplicationLatency,
ReturnedBytes, ReturnedItemCount,
ReturnedRecordsCount,
SuccessfulRequestLatency, **SystemErrors**,
TimeToLiveDeletedItemCount,
ThrottledRequests, TransactionConflict,
UserErrors, **WriteThrottleEvents**

Per-service and cross-service dashboards

Per-service metrics dashboard



Cross-service metrics dashboard



Built-in metrics often not enough

What about business / customer metrics?

Measure application performance against business goals

Revenue, signups, pictures uploaded, perceived latency, page views, etc.

How operationally stable is the application?

Continuous integration / deployment feedback time, mean time between failure and recovery, number of on-call pages and time to resolution, etc.

What about caught errors, warnings?

Caught exceptions are not counted as errors on AWS Lambda

What if I want to use other dimensions?

User ID, category, item, tags, environment, etc.

Creating custom metrics

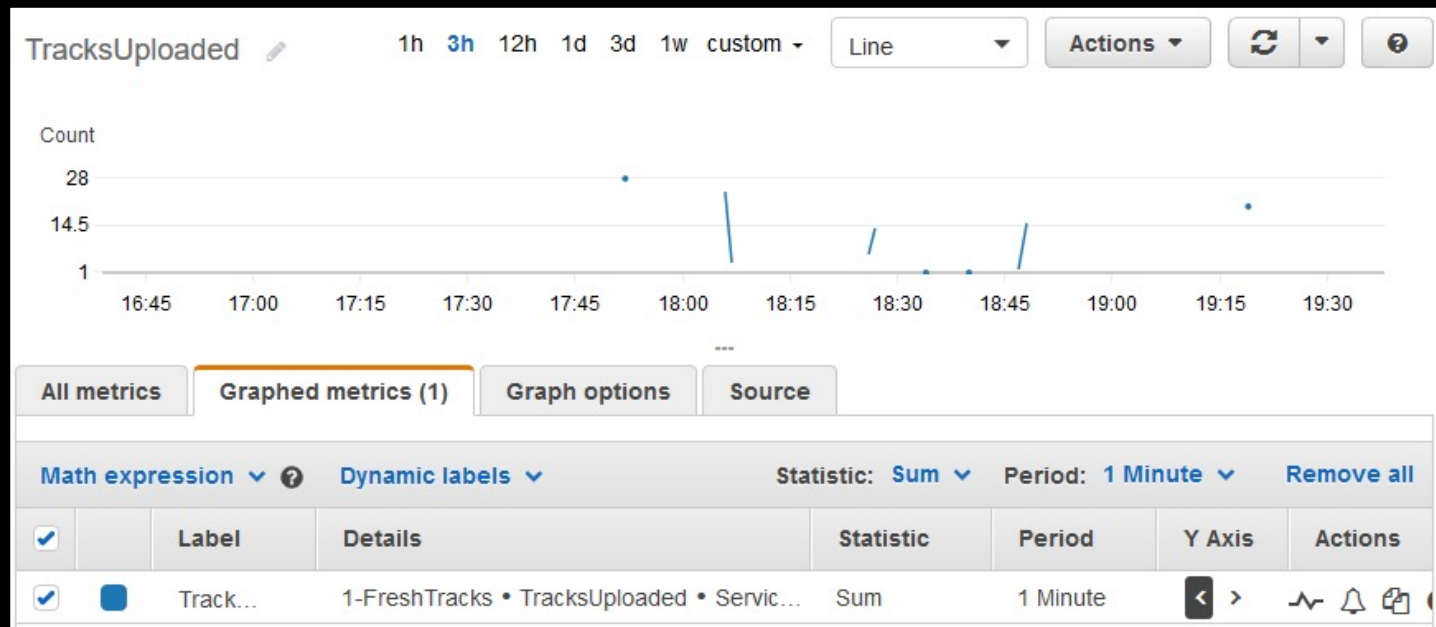
Useful for application, business, and operations metrics

- Use built-in capabilities of the AWS SDK to call the CloudWatch `putMetricData` API call
- Charged by metric and by put call of data into a metric

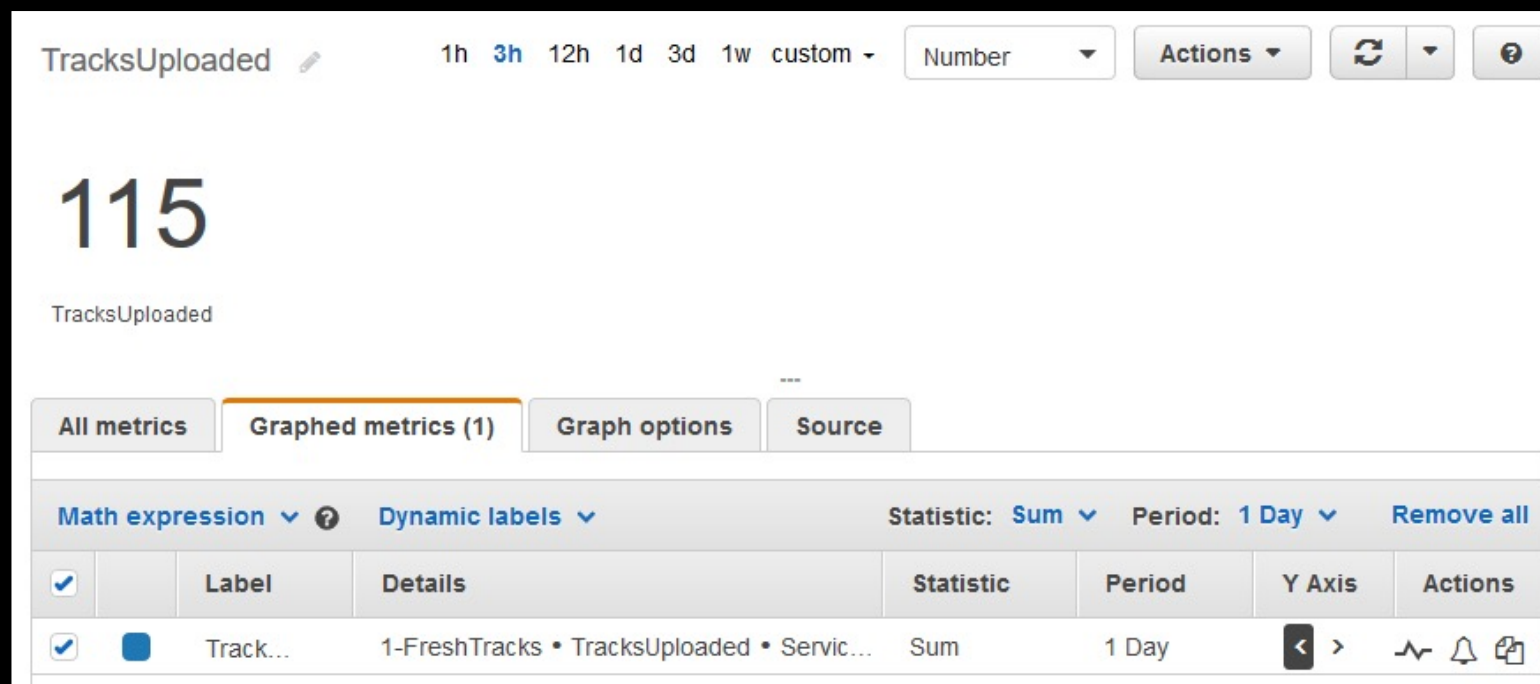
Improved by **embedded metric format** (covering shortly)

```
msg.map((stat) => {  
  params.MetricData.push({  
    'MetricName': 'wait-times',  
    'Dimensions': [  
      {'Name': 'Type', 'Value': 'ride'},  
      {'Name': 'Ride', 'Value': stat.rideId},  
    ],  
    'Unit': 'Seconds', 'Value': (stat.wait * 60)}  
  )  
})  
// Send to CloudWatch  
console.log(await cloudwatch.putMetricData(params).promise())  
}
```

Visualize with CloudWatch metrics graphs



The screenshot shows the 'Metrics' section of the CloudWatch Metrics Explorer. It includes a search bar with the placeholder text 'Enter metric on a resource'. Below the search bar, there are two selected metrics: 'Lambda Function: Duration: p90' and 'Lambda Function: Errors: Sum'. There is also a link to 'Show more chosen options (+2)' and a 'Clear all' button.



CloudWatch Metrics Explorer
<https://s12d.com/cw-me>



CloudWatch logging

Built-in logging

API Gateway logging

- REST: Two levels of logging, ERROR and INFO
 - Set globally in stage, or override per method
 - Optionally log method request / body content
- HTTP APIs and WebSocket APIs with logging variables

Lambda logging

- Logging directly from your code with your language's equivalent of `console.log()` – basic request information included
- JSON structured logging via `PutMetricData` API or embedded metric format, which includes invocation information

Export logs to Amazon Opensearch or Amazon S3

- Explore with Kibana or Amazon Athena / Amazon QuickSight



CloudWatch embedded metric format

AUTOMATICALLY GENERATE METRICS FROM STRUCTURED CLOUDWATCH LOGS

Embed custom metrics alongside detailed log event data

Can send structured format in `PutLogEvents` API call with specific format

Asynchronous

Open-source client libraries available for

- Node.js
- Python
- Java



<https://s12d.com/cwl-emf-client>

Installation

```
npm install aws-embedded-metrics
```

Usage

To get a metric logger, you can either use the `metricScope` decorator with the `Unit` parameter.

```
const { metricScope, Unit } = require('aws-embedded-metrics');

const myFunc = metricScope({
  metricNamespace: 'Service',
  unit: Unit.NONE
})(async () => {
  metrics.putDimensions({ Service: 'my-service' });
  metrics.putMetric("ProcessingLatency", 100);
  metrics.setProperty("RequestId", "422b1234");
  // ...
});

await myFunc();
```

Installation

```
pip3 install aws-embedded-metrics
```

Usage

To get a metric logger, you can do the following:

```
from aws_embedded_metrics import metric_scope

@metric_scope
def my_handler(metrics):
    metrics.put_dimensions({"Service": "my-service"})
    metrics.put_metric("ProcessingLatency", 100)
    metrics.set_property("AccountId", "123456789012")
    metrics.set_property("DevEnv", "dev")

    return {"message": "Hello World"}
```

Usage

To use a metric logger, you need to manually create and flush the `MetricsLogger` object.

```
import software.amazon.cloudwatchlogs.emf.logger.MetricsLogger
import software.amazon.cloudwatchlogs.emf.model.DimensionSet

class Example {
    public static void main(String[] args) {
        MetricsLogger logger = new MetricsLogger("my-namespace");
        logger.putDimensions(DimensionSet.of("Service", "my-service"));
        logger.putMetric("ProcessingLatency", 100);
        logger.setProperty("RequestId", "422b1234");
        logger.flush();
    }
}
```

CloudWatch embedded metric format

```
message = {  
  PriceInCart    100  
  QuantityInCart 2  
  ProductId     a23390f3  
  CategoryId    bca4cec1  
  UserId        31ba3930  
  CartId        58dd189f  
  Environment    prod  
  LogLevel       INFO  
  Timestamp      2019-12-11 12:44:40.300473  
  Message       Added 2 items 'a23390f3' to cart  
  '58dd189f'
```

```
[...]  
  "_aws": {  
    "functionVersion": "$LATEST",  
    "Timestamp": 1576064416496,  
    "CloudWatchMetrics": [{  
      "Namespace": "ecommerce-cart",  
      "Dimensions": [  
        "Environment", "CategoryId"  
      ]  
    },  
    "Metrics": [  
      {"Name": "PriceInCart", "Unit": "None"},  
      {"Name": "QuantityInCart", "Unit": "None"}  
    ]  
  }  
}
```

Amazon CloudWatch Logs Insights

Amazon CloudWatch Logs Insights

Interactively search and analyze your log data in Amazon CloudWatch Logs

Drive actionable intelligence to address operational issues without needing to provision servers or manage software

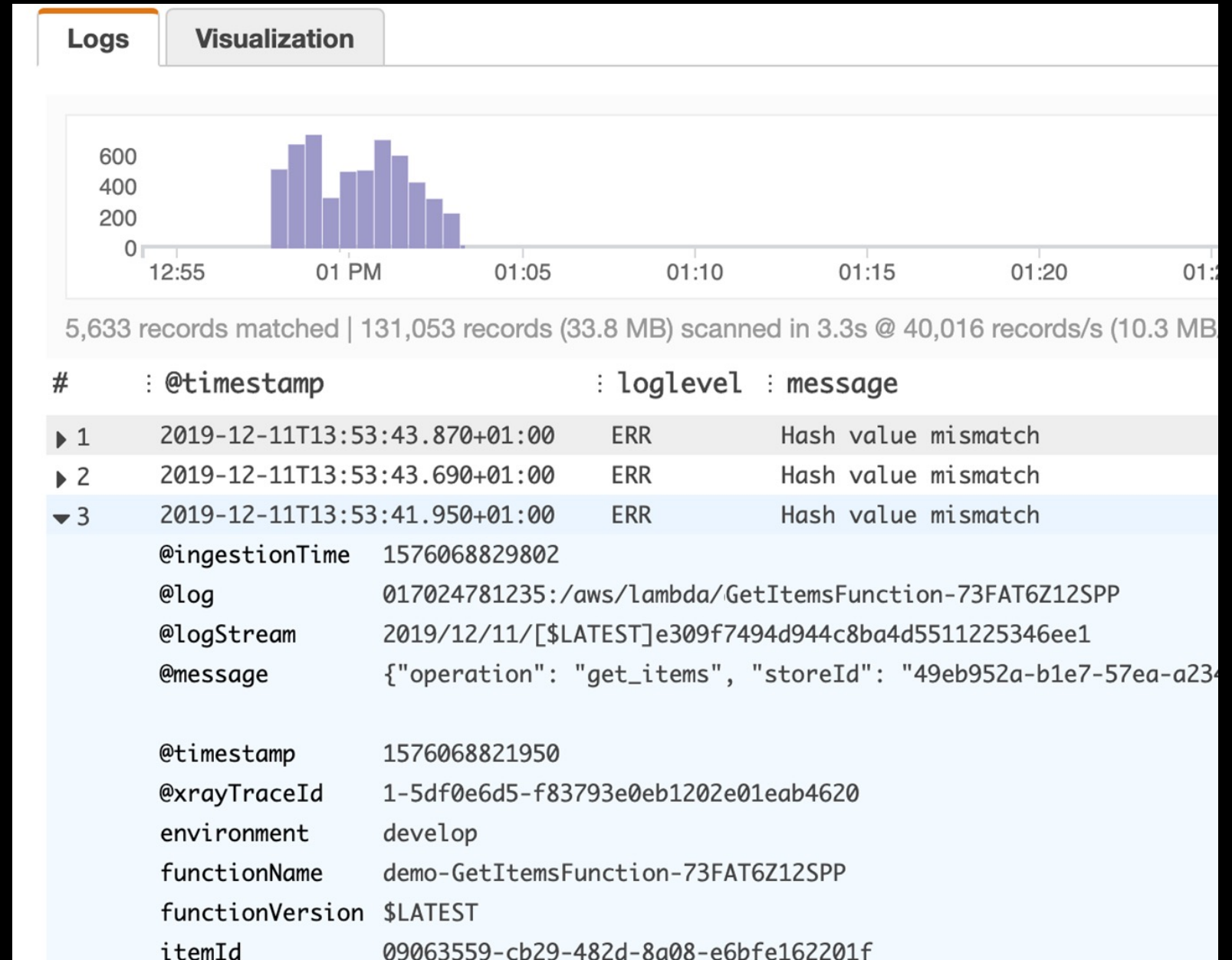
- Processes structured log data
- Flexible purpose-built query language
- Query up to 20 log groups
- Save queries

fields `Timestamp`, `LogLevel`, `Message`

```
| filter LogLevel == "ERR"
```

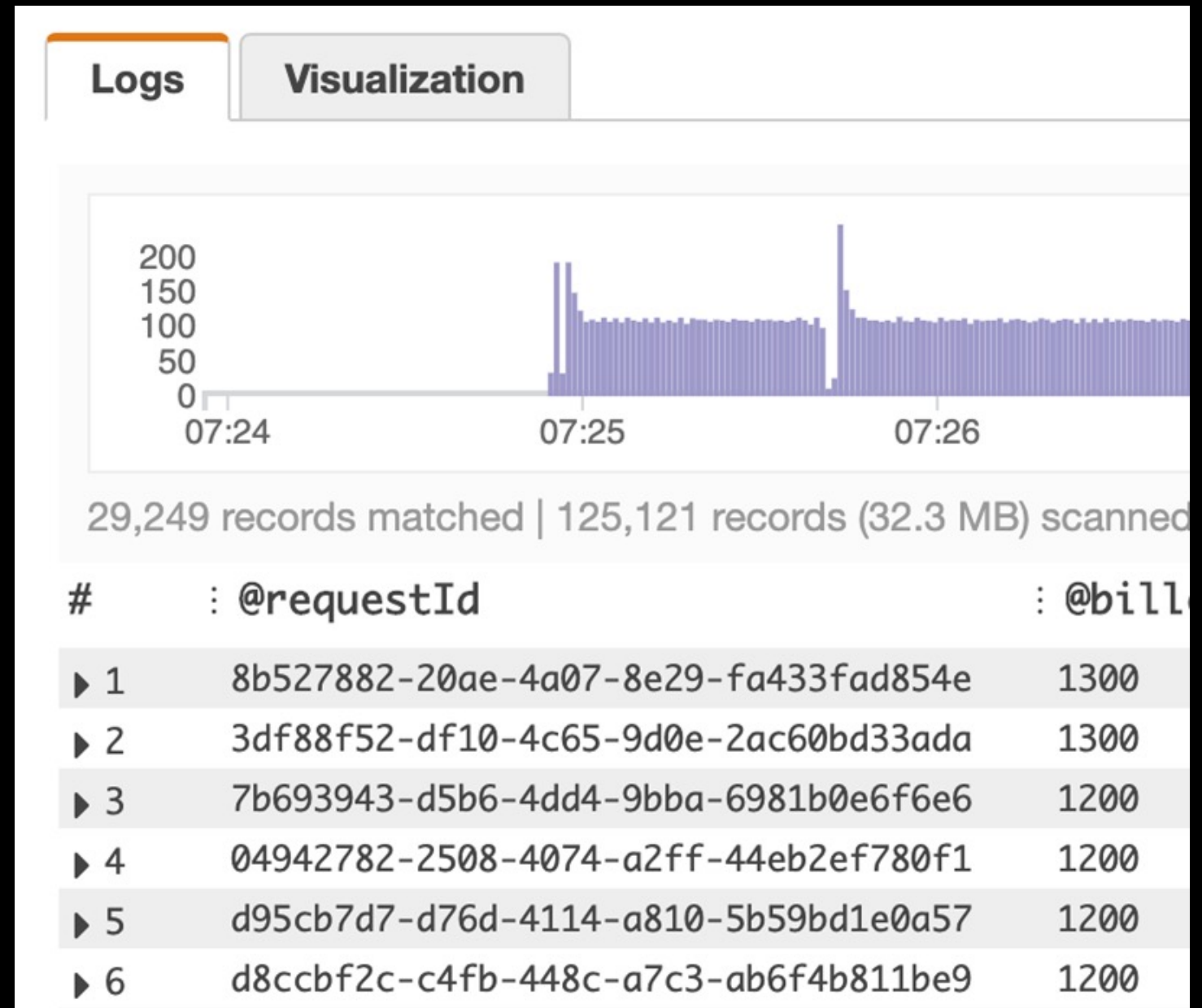
```
| sort @timestamp desc
```

```
| limit 100
```



Top 100 most expensive invocations

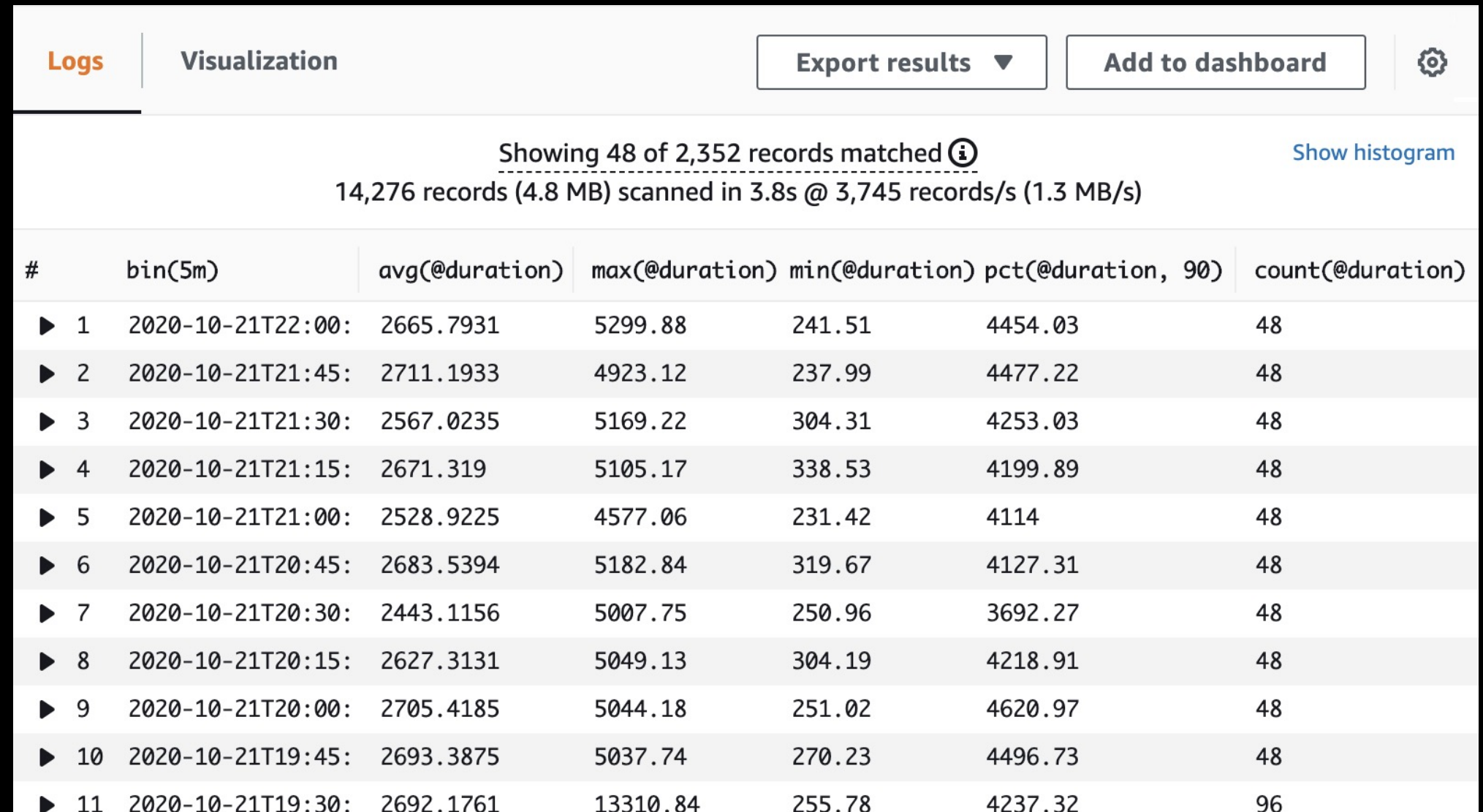
```
filter @type = "REPORT"  
  
| fields @requestId,  
@billedDuration  
  
| sort by @billedDuration  
desc  
  
| limit 100
```



Checking Lambda performance

P90 latency, total invokes, and max latency for each 5-minute window

```
filter @type =  
"REPORT"  
| stats  
avg(@duration),  
max(@duration),  
min(@duration),  
pct(@duration,  
90),  
count(@duration)  
by bin(5m)
```



The screenshot shows the AWS CloudWatch Logs console interface. At the top, there are tabs for 'Logs' and 'Visualization', along with buttons for 'Export results' and 'Add to dashboard'. Below the tabs, it indicates 'Showing 48 of 2,352 records matched' and '14,276 records (4.8 MB) scanned in 3.8s @ 3,745 records/s (1.3 MB/s)'. A 'Show histogram' link is also present. The main content is a table with the following columns: '#', 'bin(5m)', 'avg(@duration)', 'max(@duration)', 'min(@duration)', 'pct(@duration, 90)', and 'count(@duration)'. The table displays 11 rows of data, each representing a 5-minute window.

#	bin(5m)	avg(@duration)	max(@duration)	min(@duration)	pct(@duration, 90)	count(@duration)
▶ 1	2020-10-21T22:00:	2665.7931	5299.88	241.51	4454.03	48
▶ 2	2020-10-21T21:45:	2711.1933	4923.12	237.99	4477.22	48
▶ 3	2020-10-21T21:30:	2567.0235	5169.22	304.31	4253.03	48
▶ 4	2020-10-21T21:15:	2671.319	5105.17	338.53	4199.89	48
▶ 5	2020-10-21T21:00:	2528.9225	4577.06	231.42	4114	48
▶ 6	2020-10-21T20:45:	2683.5394	5182.84	319.67	4127.31	48
▶ 7	2020-10-21T20:30:	2443.1156	5007.75	250.96	3692.27	48
▶ 8	2020-10-21T20:15:	2627.3131	5049.13	304.19	4218.91	48
▶ 9	2020-10-21T20:00:	2705.4185	5044.18	251.02	4620.97	48
▶ 10	2020-10-21T19:45:	2693.3875	5037.74	270.23	4496.73	48
▶ 11	2020-10-21T19:30:	2692.1761	13310.84	255.78	4237.32	96

Creating CloudWatch alarms

All metrics | **Graphed metrics (1)** | Graph options | Source

All > 1-FreshTracks > LogGroup, ServiceName, ServiceType

LogGroup (14)	ServiceName	ServiceType	Metric Name
FreshTracks-parseGPX	FreshTracks-parseGPX	AWS::Lambda::Function	WarmStart-
FreshTracks-parseGPX	FreshTracks-parseGPX	AWS::Lambda::Function	TracksUploaded

Graph
This alarm will trigger when the blue line goes below the red line for 1 datapoints within 5 minutes.

Count: 60, 40, 20

Time: 12:00, 13:00, 14:00, 15:00

Namespace: 1-FreshTracks

Metric name: TracksUploaded

ServiceName: FreshTracks-parseGPX-12GB1OHD2LKJX

LogGroup: FreshTracks-parseGPX-12GB1OHD2LKJX

ServiceType: AWS::Lambda::Function

Statistic: Sum

Period: 5 minutes

Conditions

Threshold type

- Static**
Use a value as a threshold
- Anomaly detection
Use a band as a threshold

Whenever TracksUploaded is...
Define the alarm condition.

- Greater
> threshold
- Greater/Equal
>= threshold
- Lower/Equal
<= threshold
- Lower**
< threshold

than...
Define the threshold value.

5
Must be a number

Notification

Alarm state trigger
Define the alarm state that will trigger this action.

- In alarm**
The metric or expression is outside of the defined threshold.
- OK
The metric or expression is within the defined threshold.
- Insufficient data
The alarm has just started or not enough data is available.

Select an SNS topic
Define the SNS (Simple Notification Service) topic that will receive the notification.

- Select an existing SNS topic**
- Create new topic
- Use topic ARN

Send a notification to...

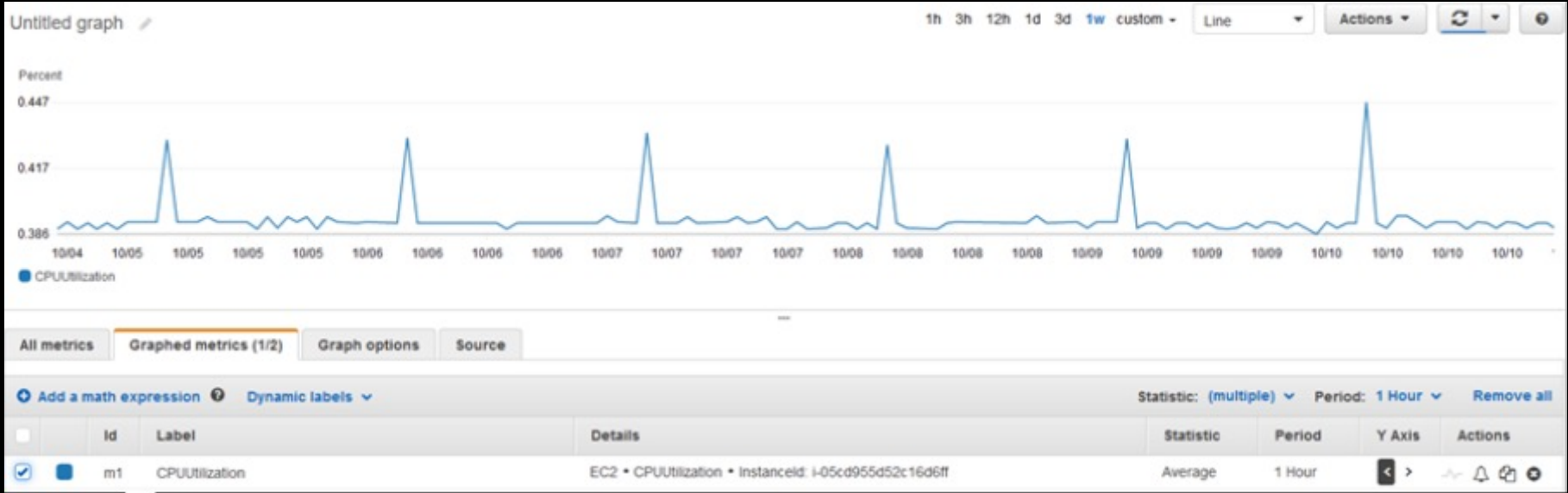
FT_CWL_Topic

Only email lists for this account are available.

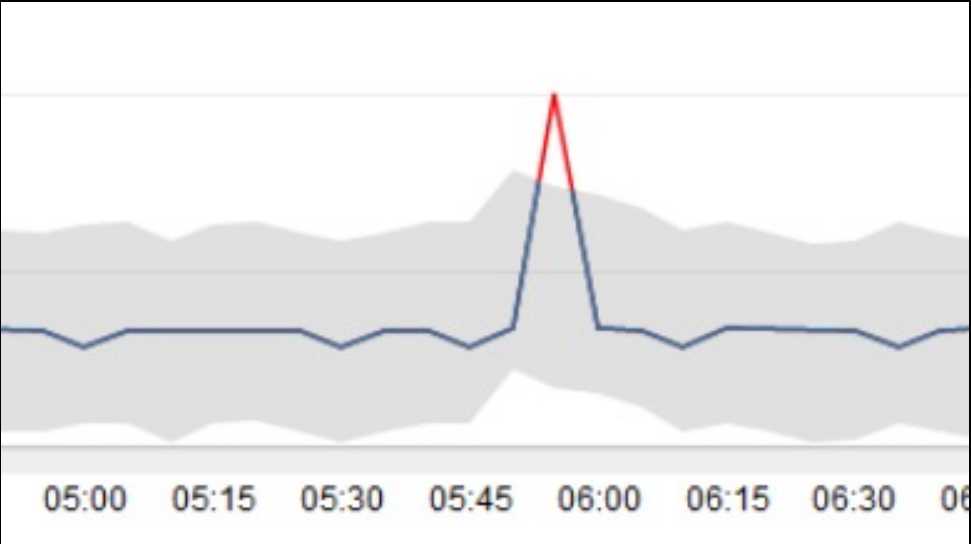
Email (endpoints)
[redacted]@amazon.com - View in SNS Console

Add notification

Using CloudWatch anomaly detection alarms



This close-up shows the configuration options for a metric in the CloudWatch console. At the top, it says "Statistic: Average" and "Period: 5 Minutes". Below this is a table with columns: "Statistic", "Period", "Y Axis", and "Actions". The table contains one row: "Average", "5 Minutes", and icons for actions. A tooltip points to the "Anomaly Detection" icon (a waveform with a bell) and says "Enables anomaly detection for this metric and statistic".



This close-up shows the configuration options for a metric in the CloudWatch console. At the top, it says "Statistic: Average" and "Period: 5 Minutes". Below this is a table with columns: "Statistic", "Period", "Y Axis", and "Actions". The table contains one row: "Average", "5 Minutes", and icons for actions. At the bottom right, there is a link "Edit model" with a close icon.



AWS X-Ray

End-to-end view of requests flowing through an application

- **Lambda** – instruments incoming requests for all supported languages and can capture calls made in code

Enable X-Ray Tracing

Globals:
Function:
Tracing: Active

- **API Gateway** – inserts a tracing header into HTTP calls as well as reports data back to X-Ray itself

Enable active tracing [Info](#)

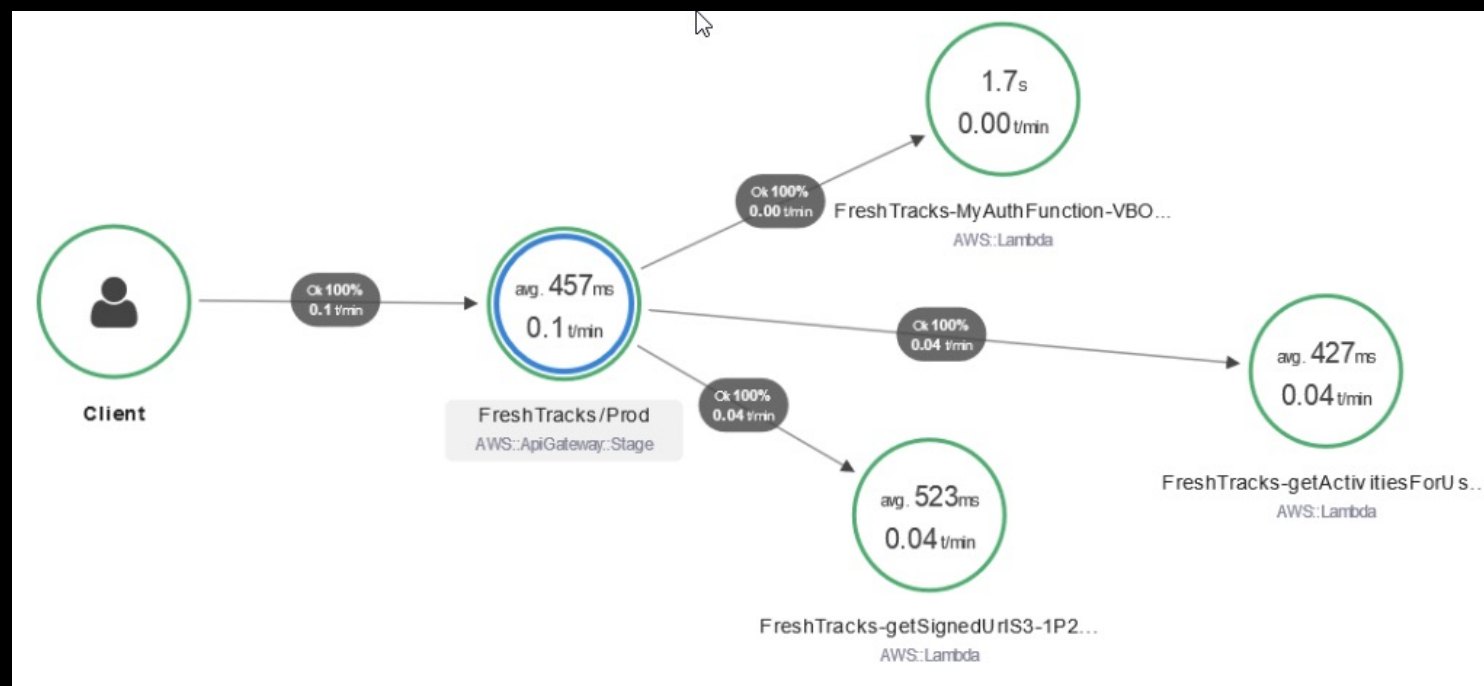


Globals:
Api:
TracingEnabled: True

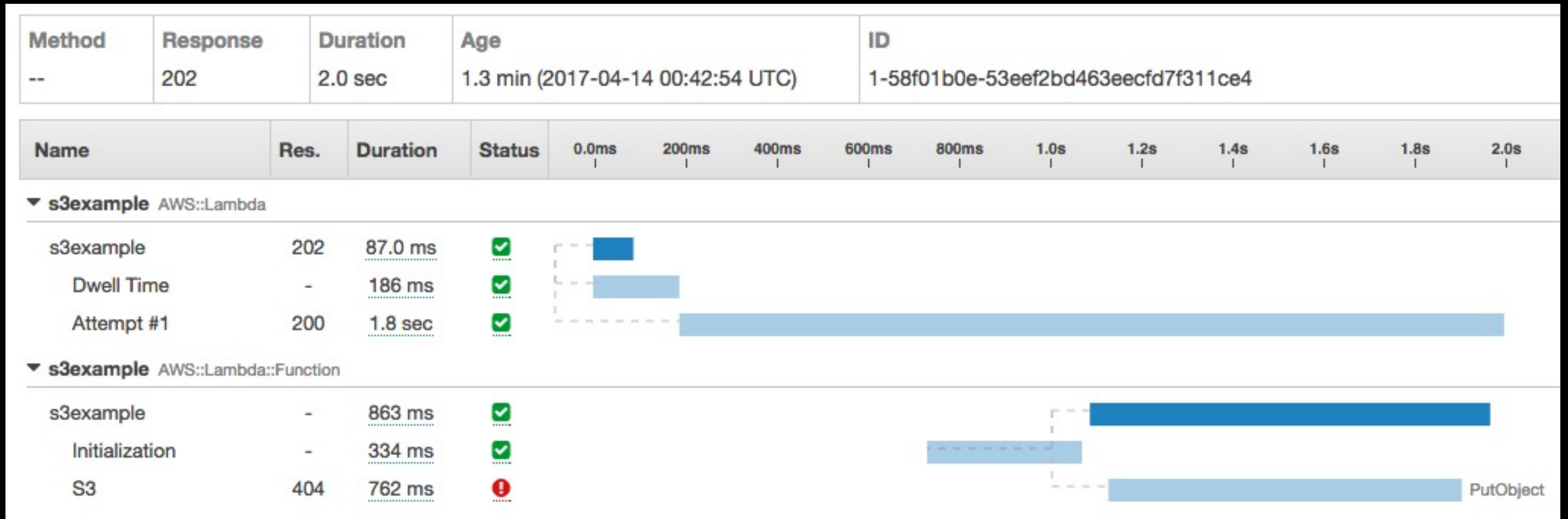
○ ○ ○

```
const AWSXRay = require('aws-xray-sdk-core');
const AWS = AWSXRay.captureAWS(require('aws-sdk'));

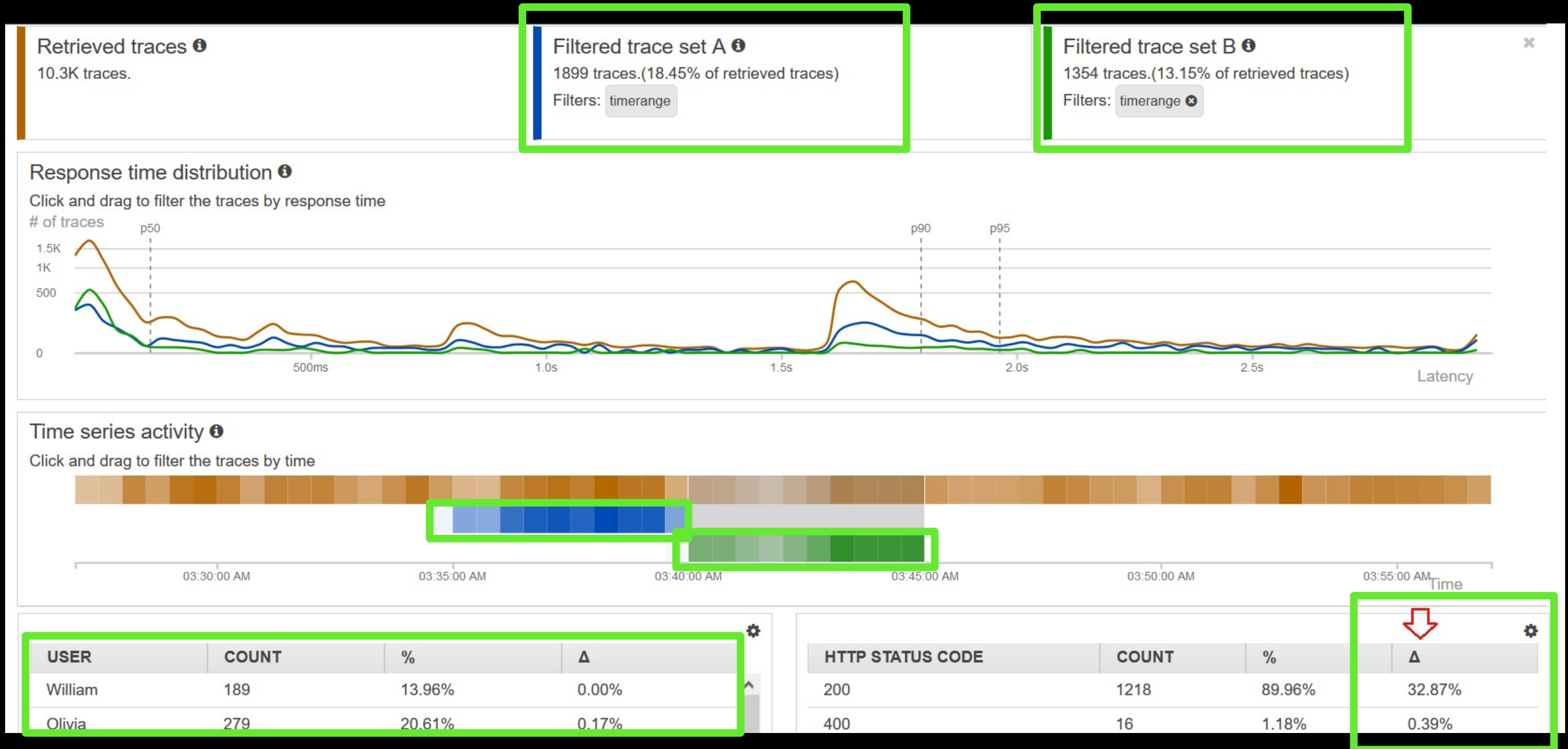
const documentClient = new AWS.DynamoDB.DocumentClient();
```



X-Ray trace example



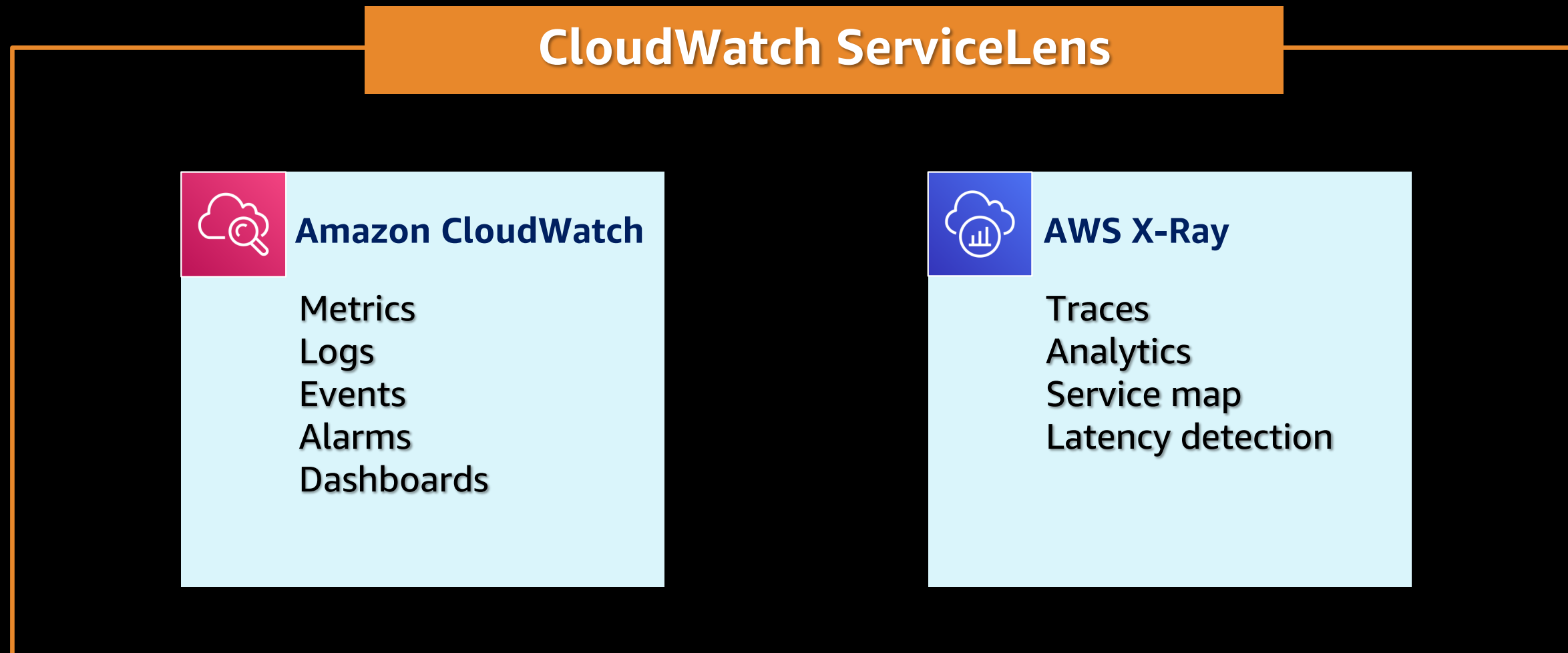
AWS X-Ray Analytics example



CloudWatch ServiceLens

CloudWatch ServiceLens

- Ties together CloudWatch metrics and logs, in addition to traces from AWS X-Ray
- Gives you a complete view of your applications and their dependencies



CloudWatch
Dashboards
Alarms
ALARM 0
INSUFFICIENT 0
OK 0
Billing
Logs
Log groups
Insights
Metrics
Explorer BETA
Events
Rules
Event Buses
ServiceLens
Service Map
Traces
Container Insights NEW
Resources
Performance Monitoring
Lambda Insights NEW

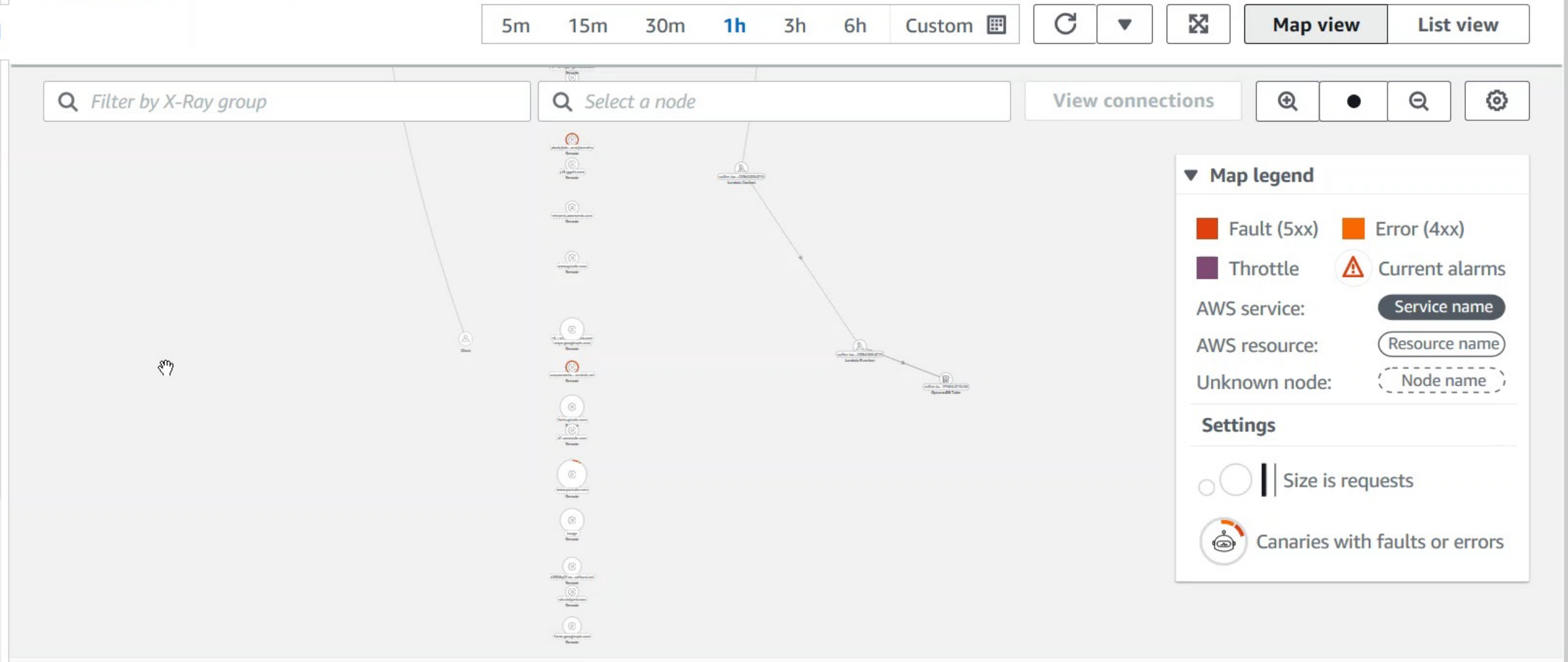
CloudWatch > Service Map

5m 15m 30m 1h 3h 6h Custom [Calendar icon] [Refresh] [Dropdown] [Fullscreen] Map view List view

Filter by X-Ray group

Select a node

View connections [Zoom in] [Zoom out] [Settings]



Map legend

- Fault (5xx)
- Error (4xx)
- Throttle
- Current alarms
- AWS service: Service name
- AWS resource: Resource name
- Unknown node: Node name

Settings

- Size is requests
- Canaries with faults or errors

No node selected

View logs View traces View dashboard

Troubleshooting / query workflow

Notification
/ question

Identify
ServiceLens

Traces
X-Ray traces

Analyze
X-Ray
Analytics

Logs
Logs
Insights

Receive a
CloudWatch
alarm
notification
Ask a question

View **Service**
Lens service map
Identify point of
interest to dive
deep

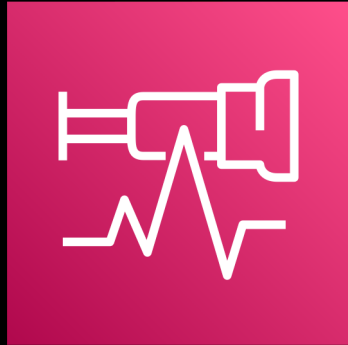
View **X-Ray**
traces, maps, and
requests
Look at specific
API / service that
is the current
point of interest

Trace analysis
with **X-Ray**
Analytics
Perform deep
analysis /
comparison of
traces if
necessary

Query **Logs**
Insights
At specific point
in time for
deeper analysis
and identify root
cause

AWS open source observability services

AWS Open Source Observability Services



AWS Distro for
OpenTelemetry



Amazon Managed Service
for Prometheus



Amazon OpenSearch
Service



Amazon Managed Service
for Grafana

Collection

Metrics

Logs and Traces

Visualisation

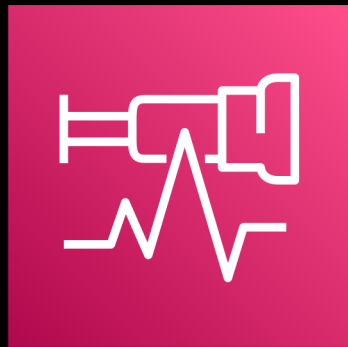
What is OpenTelemetry?

- 50% of companies use at least 5 observability tools
- 33% of the companies used more than 10 observability tools
- Challenges
 - Using different SDK and agents
 - Increase in resource consumption
 - Manual correlation is error-prone
- OpenTelemetry is an opensource observability framework for cloud-native software. It is a collection of tools, APIs, and SDKs.
- You can use it to instrument, generate, collect, and export telemetry data (metrics, logs, and traces) for analysis in order to understand your software's performance and behaviour



AWS Distro for OpenTelemetry?

A SECURE, PRODUCTION-READY OPEN SOURCE DISTRIBUTION SUPPORTED BY AWS



AWS Distro for
OpenTelemetry

- Upstream-first distro of OpenTelemetry
- Certified by AWS for security and predictability
- Backed by AWS Support
- One-click deploy and configure from AWS container and AWS Lambda consoles
- The AWS Distro for OpenTelemetry collector is added as a layer to the lambda function
- Exporters for AWS monitoring solutions including – CloudWatch, X-Ray, Amazon Managed Service for Prometheus, OpenSearch Service and Partner Solutions

Resources

- Sample Serverless Application (Serverless Feedback App)
<https://github.com/aws-samples/aws-serverless-feedback-app>
- AWS Observability Workshop
<https://observability.workshop.aws>
- AWS Distro for OpenTelemetry
<https://aws-otel.github.io>
- Lambda Powertools - Python
<https://github.com/aws-labs/aws-lambda-powertools-python>
- CloudWatch Embedded Metric Format
<https://s12d.com/cwl-emf-client>
- CloudWatch Metrics Explorer
<https://s12d.com/cw-me>
- Tracing AWS Lambda functions in AWS X-Ray with OpenTelemetry
<https://s12d.com/tracing-lambda-otel>
- Monitoring and observability – AWS Lambda Operator Guide
<https://s12d.com/lambda-op-guide-obs>
- Getting started with Trace Analytics in Amazon Elasticsearch Service
<https://s12d.com/trace-analytics-es>

Thank you!

Ozioma Uzoegwu
Solutions Architect, AWS

 @iam_tessot