Performance Testing with AWS X-Ray, CloudWatch, and RDS performance tools

Author: Daniel Kranowski

kranowski.pnsqc.2019@bizalgo.com

Abstract

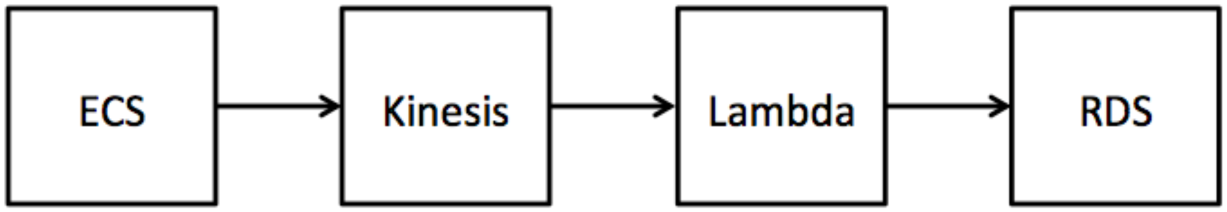
With a distributed, asynchronous system built on AWS, we have a variety of AWS performance measurement tools to use - which one makes the most sense? Our system under test starts with an Elastic Container Service (ECS) Docker app driving high-frequency messages into Kinesis, then to Lambda and a Relational Database Service (RDS) database. We want to measure propagation delay of messages through the system and analyze other parameters that explain or improve the existing performance. We'll look at multiple techniques for measuring performance: AWS X-Ray; AWS CloudWatch Metrics and Logs; and RDS Enhanced Monitoring & Performance Insights. We'll see how to obtain performance data in the AWS console or via command-line, and we'll characterize the pros and cons of each approach.

Biography

Daniel Kranowski is Lead Software Engineer and Principal at Business Algorithms, LLC, a business software services company based in the Portland, Oregon metro area. With over 20 years of software experience, his job roles have included software development, test, devops, and project management, at companies of all sizes and various industries. After working in multiple programming languages, today his preferred "full stack" starts with AWS and statically typed JavaScript. His personal code repositories are at https://github.com/daniel-kranowski. He can be contacted for consulting opportunities at www.bizalgo.com.

1 Introduction

The conversation starts with a question: "What is the propagation delay for a message traversing this distributed, asynchronous system?"



All those system blocks are in Amazon Web Services (AWS):

- AWS Elastic Container Service (ECS), for running Docker containers.
- AWS Kinesis, a messaging service.
- AWS Lambda, a service for running serverless applications.
- AWS Relational Database Service (RDS).

We have a Docker container driving high-frequency messages onto Kinesis. Lambda is listening, it picks up the messages, parses them, and persists the results into the database. We're not showing the larger system, where a thousand users are clicking some button in a website to trigger those messages, and they are sitting patiently in front of their screens waiting for the impact when their specific app can tell it has landed in the database. But that's why we care: users expect quick results, and if the app is slow we lose their business. So we need to do some high-quality performance analysis on this system. We want to know the propagation delay, and if it is not fast enough, we need to dig deeper and identify the specific bottlenecks in the system where performance could be explained and improved.

There are many tools for performance analysis on a distributed system. This is a 100% AWS system, and I can't think of any vendor more qualified to offer performance data on AWS services than AWS itself. So let's consider these:

- AWS X-Ray
- AWS CloudWatch Logs & Metrics
- AWS RDS Enhanced Monitoring & Performance Insights

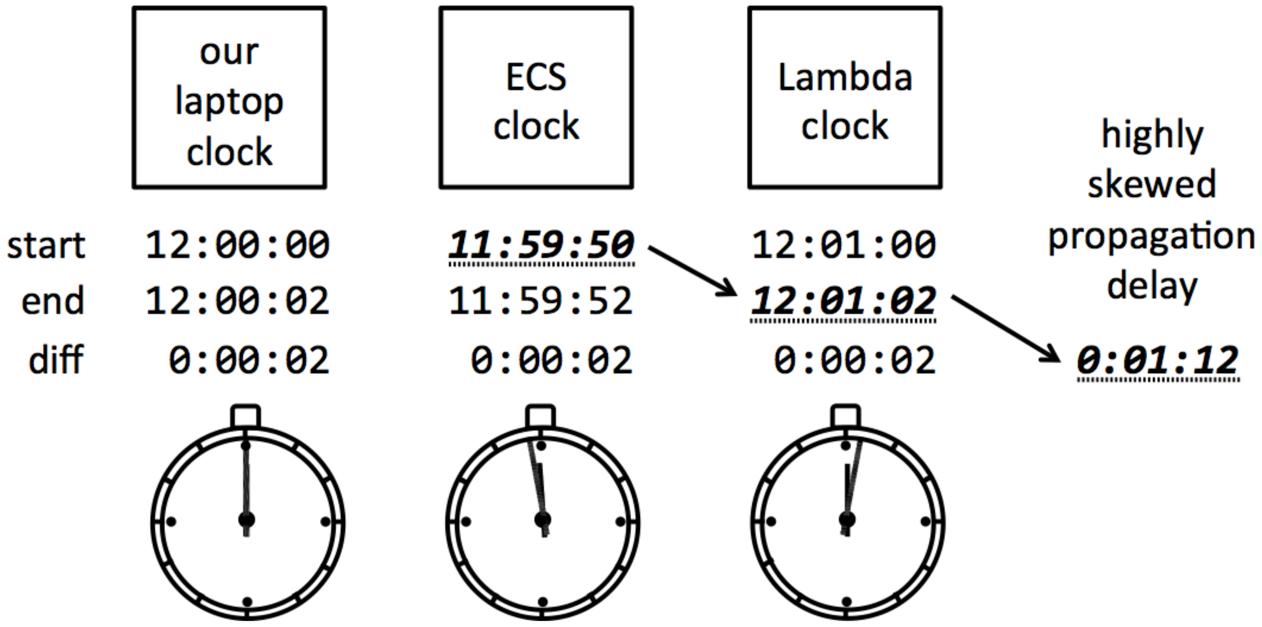
The code for this system lives here: https://github.com/daniel-kranowski/aws-perfmon-01. The programming languages it uses are TypeScript for the ECS app and Lambda function (NodeJS runtime), and Terraform to automate the AWS infrastructure.

In this paper, the important items for discussion are the lessons learned about how to measure performance, not the exact performance of this system.

2 Synchronous clocks in the performance analysis of an asynchronous system

Before we get to the AWS performance tools, let's think a moment on the topic of **clocks**. Imagine someone on our team has done a performance test on this system, and reported to us "It takes 500 milliseconds for a message to get from ECS to RDS." We should have a few specific questions about that number: How exactly did we obtain the start and end timestamps? What clock did we use - was it the local clock on a workstation here in the office, a system timestamp from the VMs we are actually measuring, or time from the clock of a completely separate system? There are other questions to ask, such as whether the test payload fairly represented production data, or whether the tested system omits relevant network hops or other components of the real production system. But here we will focus on the question of the clock.

When calculating propagation delay through a distributed system, *the ideal performance analysis must be made using a single clock*, or clocks that we have reason to believe are completely in sync with each other. Propagation delay is equal to end time minus start time. If we capture the start time on our local machine when the test request is sent out, and we capture the end time on a deployed machine when it detects the arrival of the test message, can we calculate an accurate propagation delay by subtracting these two timestamps? There will be skew between the local clock and the deployed machine clock. Ignoring timezone differences, the two clocks could easily be seconds or minutes apart, which means our calculated result will be too fast or too slow by seconds or minutes. In a measurement where milliseconds count, can we feel confident in the propagation delay calculated using timestamps with an unknown amount of clock skew?



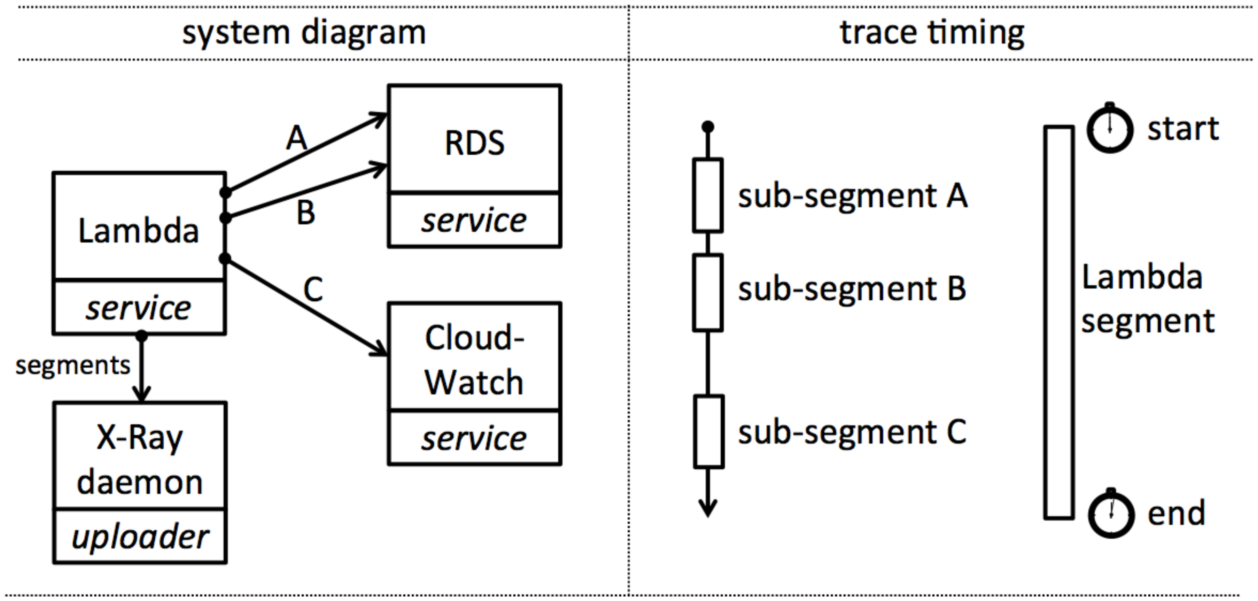
In a round-trip system test, like a synchronous request/response, there is only one clock for measurement purposes, and it lives on the client side. We throw the boomerang, and it comes back to us. No clock problem there. But the system under test described in this paper is a one-way trip, because it involves asynchronous messaging: the Kinesis component. Now we're throwing an envelope in the mailbox, and the clock on the wall where we dropped it off is not the same clock our recipient is looking when it gets received on the other end. The asynchronous aspect of the system is what makes it much harder to measure an accurate propagation delay.

With that concern in mind, now let's consider the AWS-based approaches to performance analysis.

3 AWS X-Ray

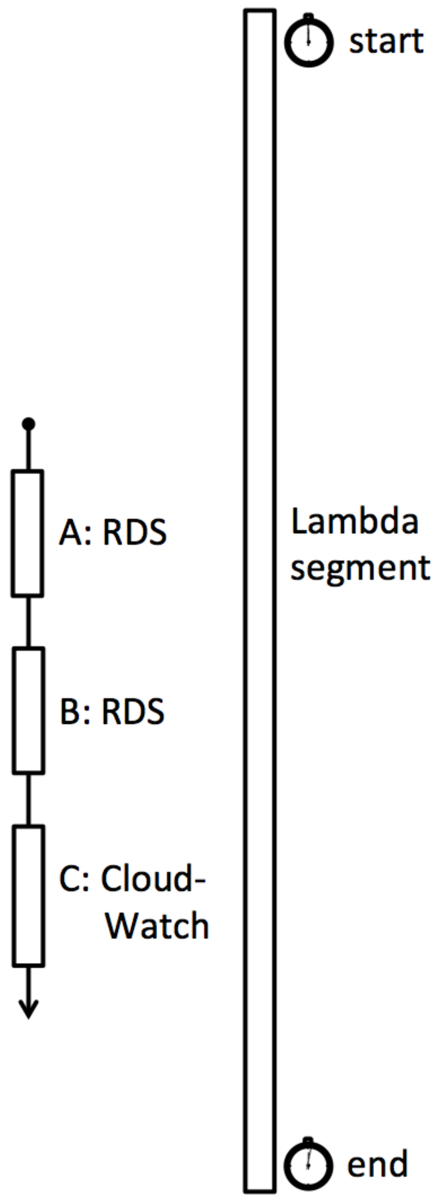
AWS X-Ray is a service for analyzing timing in distributed systems. It is focused on **timing**, and does not venture into other metrics of system performance. Because it is designed for systems with high throughput, it uses a **sampling** approach, which means it records data on a fraction of all the messages passing by, not all of them.

We can understand AWS X-Ray using three basic concepts: **services**, **traces**, and **segment uploader clients**. Below we have the Lambda service making two SQL queries to the RDS service, sending a custom metric to the CloudWatch service, then uploading data segments to the X-Ray service, with the X-Ray daemon as its intermediary:



Except for X-Ray itself, the services are AWS services like Lambda or RDS whose performance we want to measure. Traces are the lines that connect them, and the propagation delay question we're trying to solve here is represented by the duration of an average trace, in milliseconds. A trace is composed of **segments**, such as the main Lambda segment, and sub-segments, such as a single SQL query to RDS. Below is an example X-Ray trace in JSON format for the above system. Many details have been removed from the JSON to make the code snippet shorter:

```
{
  "Duration": 0.202,
  "Id": "1-5d54524b-d8a31d6c6bc1e31072c986d2",
  "Segments": [
    {
      "Document": {
        "start_time": 1565807179.6620595,
        "end_time": 1565807179.851746,
        "origin": "AWS::Lambda::Function",
        "subsegments": [
          {
            "name": "Invocation",
            "start_time": 1565807179.6621196,
            "end_time": 1565807179.8506436,
            "aws": {
              "function_arn": "arn:aws:lambda:us-west-
2:123456789012:function:aws-perfmon-01-msg-consumer"
            },
            "subsegments": [
              {
                "name": "msgdb@aws-perfmon-01-
rdscluster.cluster-asdfasdfasdf.us-west-
2.rds.amazonaws.com",
                "start_time": 1565807179.69,
                "end_time": 1565807179.71
              },
              {
                "name": "msgdb@aws-perfmon-01-
rdscluster.cluster-asdfasdfasdf.us-west-
2.rds.amazonaws.com",
                "start_time": 1565807179.691,
                "end_time": 1565807179.72
              },
              {
                "name": "CloudWatch",
                "start_time": 1565807179.771,
                "end_time": 1565807179.831,
                "aws": {
                  "operation": "PutMetricData"
                }
              }
            ]
          }
        ]
      }
    }
  ]
}
```



X-Ray trace

X-Ray doesn't record traces unless we configure the services to upload segments using the X-Ray API. There are many ways to do it:

- *AWS CLI*: Call `aws xray put-trace-segments` on the command-line, with the segment as a JSON string argument.
- *X-Ray Daemon*: Run a central daemon process, as an intermediary between our clients and the X-Ray service. Clients send the segment JSON to the daemon, which forwards them to X-Ray.
- *AWS JavaScript SDK*: The JavaScript SDK has an X-Ray wrapper that instruments the core AWS client object, so that the client sends a segment for every AWS service our code connects to.
- *AWS Java SDK*: The Java SDK has several options, including a servlet filter for webapps that starts a trace on receiving a request (the round-trip measurement model); and a general purpose X-Ray recorder that sends a segment for every AWS service call, when added as an interceptor on the Java client object for that service.

Instantiating our own Daemon is the most labor-intensive solution. Using the CLI eliminates the need to set up a Daemon, but we will still need to manage X-Ray segment formatting, and integrate the CLI calls into the application under test. The SDKs are easiest, because they take care of the low-level details for us.

Here is an example of instrumenting a NodeJS Lambda function, using the AWS JS SDK in TypeScript:

```
import * as AWS from 'aws-sdk';
import * as AWSXRayCore from 'aws-xray-sdk-core';

AWSXRayCore.captureAWS(AWS);  // AWS object is now instrumented for X-Ray.

export async function handler(event, content) { ..... }
```

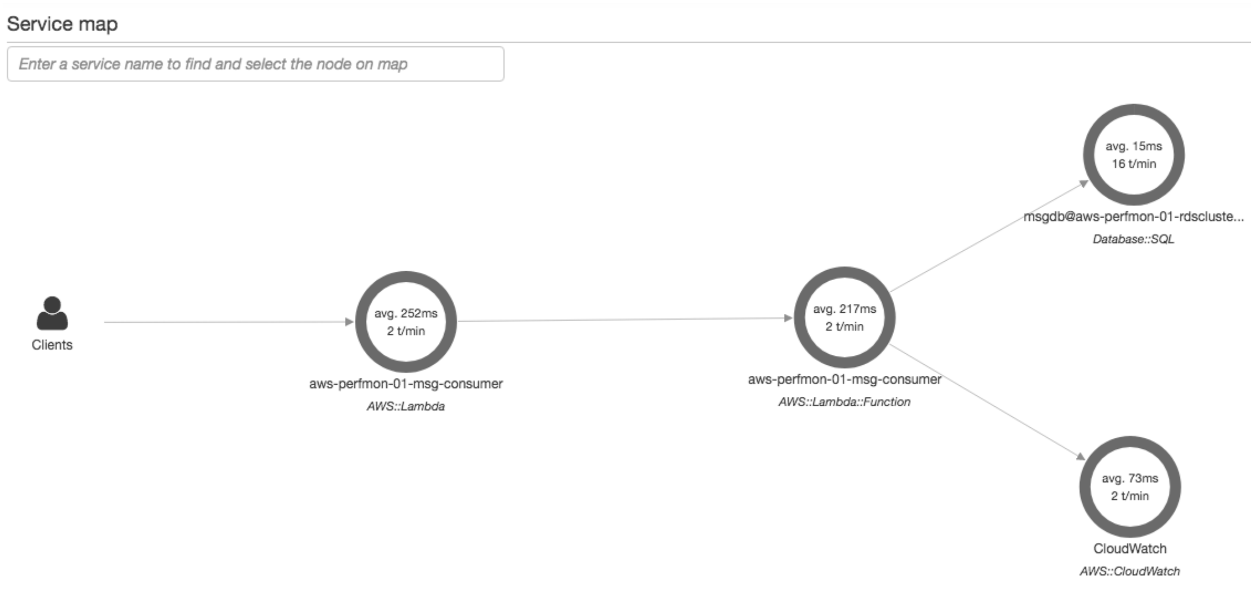
Simply by having called `captureAWS()`, this Lambda function handler will now automatically send a segment to X-Ray to record the start and end of the Lambda function's invocation. Additionally, if the handler uses the `AWS` client object in an expression to communicate with another AWS service, that will automatically send a sub-segment. Behind the scenes, the Lambda service runs an X-Ray daemon as intermediary to receive these segments and forward them to the X-Ray service.

SQL queries to an RDS database are performed over a database connection. The AWS client object is not involved in the connection or the query, so our Lambda function needs additional instrumentation in order to record time spent in the database. In our example, we use the `mysql` JavaScript client library to make connections and perform queries to an RDS Aurora MySQL database. The X-Ray SDK has a wrapper for that library, which can capture it just like it captured the AWS object:

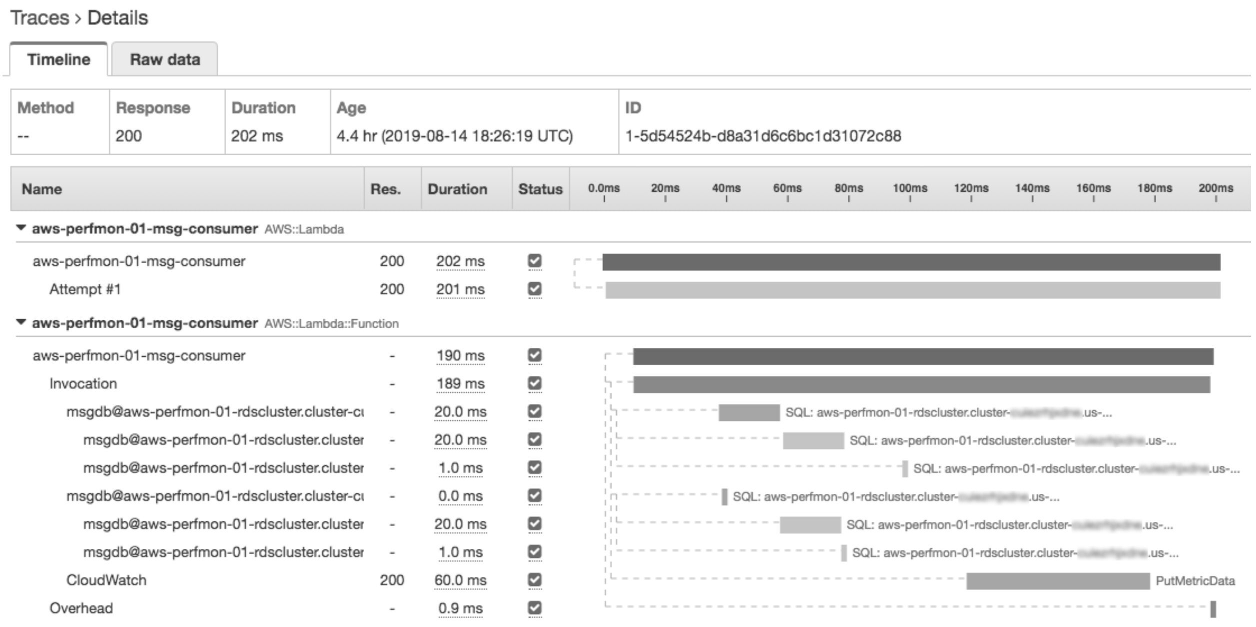
```
import * as mysql from 'mysql';
import captureMySQL = require('aws-xray-sdk-mysql');
const capturedMySQL = captureMySQL(mysql);
const { createPool } = capturedMySQL;
const pool: Pool = createPool({
    host: 'aws-perfmon-01-rdscluster.cluster-asdfasdfasdf.us-west-
2.rds.amazonaws.com',
    port: 3306,
    user: 'username',
    password: 'password',
    database: 'msgdb'
});
```

By creating a database connection through the `capturedMySQL` object, the subsequent query will automatically send a segment to X-Ray to record the start and end time inside RDS.

If we run our performance test, we can log into the AWS console and see the X-Ray Service Graph:



And here is an example trace, where the Lambda function made six calls to the database and then called CloudWatch:


Traces › Details

Timeline | Raw data

Method	Response	Duration	Age	ID
--	200	202 ms	4.4 hr (2019-08-14 18:26:19 UTC)	1-5d54524b-d8a31d6c6bc1d31072c88

Name	Res.	Duration	Status	0.0ms 20ms 40ms 60ms 80ms 100ms 120ms 140ms 160ms 180ms 200ms
aws-perfmon-01-msg-consumer AWS::Lambda				
aws-perfmon-01-msg-consumer	200	202 ms	✓	
Attempt #1	200	201 ms	✓	
aws-perfmon-01-msg-consumer AWS::Lambda::Function				
aws-perfmon-01-msg-consumer	-	190 ms	✓	
Invocation	-	189 ms	✓	
msgdb@aws-perfmon-01-rdscluster.cluster-cu	-	20.0 ms	✓	SQL: aws-perfmon-01-rdscluster.cluster-[redacted].us-...
msgdb@aws-perfmon-01-rdscluster.cluster	-	20.0 ms	✓	SQL: aws-perfmon-01-rdscluster.cluster-[redacted].us-...
msgdb@aws-perfmon-01-rdscluster.cluster	-	1.0 ms	✓	SQL: aws-perfmon-01-rdscluster.cluster-[redacted].us-...
msgdb@aws-perfmon-01-rdscluster.cluster-cu	-	0.0 ms	✓	SQL: aws-perfmon-01-rdscluster.cluster-[redacted].us-...
msgdb@aws-perfmon-01-rdscluster.cluster	-	20.0 ms	✓	SQL: aws-perfmon-01-rdscluster.cluster-[redacted].us-...
msgdb@aws-perfmon-01-rdscluster.cluster	-	1.0 ms	✓	SQL: aws-perfmon-01-rdscluster.cluster-[redacted].us-...
CloudWatch	200	60.0 ms	✓	PutMetricData
Overhead	-	0.9 ms	✓	

If we want a copy of our traces for offline analysis, we can download them with the AWS CLI:

```
aws xray get-trace-summaries \
    --start-time "2019-08-14T11:25:00.000-07:00" \
    --end-time   "2019-08-14T11:30:00.000-07:00" \
    --filter-expression 'service(id(name: "aws-perfmon-01-msg-consumer",
                                    type: "AWS::Lambda::Function"))'

aws xray batch-get-traces \
    --trace-ids 1-5d54524b-d8a31d6c6bc1e31072c986d2 \
                1-5d34910e-71883e81bd6506cefb7768af
```

This all sounds great so far, but there is a fly in the ointment. Notice we have no X-Ray coverage for the Kinesis system block that feeds messages into our Lambda. There's a reason for this: X-Ray works great for the round-trip request/response model, but not so great for a one-way asynchronous model. X-Ray could capture the time to put a message on the Kinesis stream, but measuring the time *across* the stream is another matter. Kinesis does not have its own X-Ray daemon like Lambda, and we cannot spin up our own daemon behind the AWS curtain where Kinesis receives PUT messages and delivers GET messages. To build up a trace that covers our system path from start to finish, the best we can do is to leverage the compute nodes on either end of the Kinesis stream, which in our case are the ECS Docker container and the Lambda.

To make that work, the ECS and Lambda compute nodes would need to manage the trace id, so that segments are uploaded and attached to the right trace. The trace starts in the ECS app, which must produce an id and put it somewhere on the message. Upon pushing that message into Kinesis, the ECS app must send a segment to X-Ray saying "For trace id 123, the ECS service started work at time1, and at time2 it finished." When the Lambda function receives the message, it must parse out the trace id, do its work, and send its own segment to X-Ray saying "For trace id 123, Lambda received the message at time3, and finished processing it at time4." There is no way to make note of Kinesis in these trace segments.

It might be worth going down that road, but for one other issue: the compute nodes would need to individually fill in the start time and end time of the segment. That's right, two clocks used in distributed performance analysis! We've identified that as a risk, but there's a chance this still could work out. ECS and Lambda are both inside the AWS ecosystem, and the VMs hosting them are all probably using the same Network Time Protocol (NTP) servers, and if we stick with a single AWS region then the network latency between them is minimized. So it's possible the two clocks are "reasonably" in sync, as reasonable as it gets when we are measuring delays on the order of milliseconds. But ECS and Lambda are unreliable candidates for this because their internal supporting VMs are constantly popping in and out of existence, which is a basic part of the design and sales pitch for these AWS services. Docker containers and especially Lambda functions are short-lived, and their capacity to scale in or out is strong evidence that the AWS internal VMs which host them will also be short-lived. In a word, they are *ephemeral*. This matters because NTP sync is not instantaneous when a VM spins up, and I can tell you I see Lambdas that do not know what time it is right after a Lambda cold start. We will see evidence of this in the next section, using a CloudWatch Custom Metric for system propagation delay.

When a compute node's supporting VM has been running for a minute, then we can be sure NTP sync has occurred, however good that may be. But we have no direct visibility into the VMs underpinning ECS and Lambda, and evidence shows that the new instances start out with unsynchronized clocks.

4 AWS CloudWatch

AWS CloudWatch encompasses a handful of features, and for performance analysis I'll consider Logs, Dashboards, and Metrics. I think it's fair to say more people have used CloudWatch than, say, X-Ray, so there is perhaps nothing new and shiny on the surface here. But humor me for a moment.

4.1 AWS CloudWatch Logs

AWS CloudWatch Logs are an available route for troubleshooting other AWS services. For example, ECS and Lambda put their standard output into CloudWatch Logs by default. Logs are organized in a three-level hierarchy of Log Groups, each of which contains Log Streams, each of which contains Log Events (the actual log texts). Looking at logs is an indispensable tool for understanding *why* a compute service is slow: we can find stacktraces or other clues in the output that explain what is happening.

More pertinent to our propagation delay question is the fact that CloudWatch Log events have a timestamp. In fact, they have two of them. The reference API for `GetLogEvents` shows:

- `timestamp`: The time the event occurred.
- `ingestionTime`: The time the event was ingested.

The first one, `timestamp`, is the same one on the `PutLogEvents` operation, which means it is set by the compute node or service that uploads the log event. This is the same clock sync situation we were in with X-Ray, where we were trying to decide whether ECS and Lambda were a reliable source of clock data. However the `ingestionTime` *is set by the CloudWatch service itself*, and even though it is behind the AWS curtain, we can be sure its supporting VMs are long-lived and not ephemeral like ECS and Lambda VMs. CloudWatch is under constant request load from all accounts in the entire AWS ecosystem, unlike our little ECS container or our particular Lambda function, where traffic varies. Moreover, CloudWatch is a single service, which means one clock, not many clocks. Within a given AWS region there may be some network latency missed between ECS and CloudWatch, or between Lambda and CloudWatch, but weigh that against the more robust premise of using `ingestionTime` as a central clock to measure a distributed, asynchronous system path. This advantage is not to be ignored. We could conceivably design our distributed system to write log events for "start" and "end," then use the API to parse out the start/end ingestion times and calculate the propagation delay.

On the other hand, there is a significant disadvantage with basing our performance analysis on CloudWatch Logs, which can be summarized neatly by saying the API is a royal pain in the neck. Searching for text in CloudWatch Logs is very hard, because it is not an indexed system like Elasticsearch, or other SaaS indexed logging systems like Splunk and Loggly. The CloudWatch Logs API does have an operation for global searching, `FilterLogEvents`, but let's be careful using it or we could be sitting for hours while it scans every log event in the filtered log streams. To deal efficiently with log events from the CloudWatch API, we need to be prepared to process every log event in a log stream.

4.2 AWS CloudWatch Metrics

AWS CloudWatch Metrics are the starting point for performance diagnostics on an individual block in the system. AWS is the infrastructure provider, and they are the only ones who can truly supply us with accurate information about infrastructure metrics. We don't have to take any action for CloudWatch to record these metrics on all our resources; it is always recording them, which is handy for morning-after troubleshooting.

A **metric** is simply a time-series dataset that gives the numerical value of some system indicator over time. We can download the numbers using the API, or we can use the AWS console to plot metrics as a graph in the browser. CloudWatch Dashboards are a feature for remembering the metric graphs we like to see most often. These console graphs work very well, and we can even export their configuration in JSON to automatically recreate our desired Dashboards in an infrastructure automation language like AWS CloudFormation or Terraform.

Every AWS service has a developer documentation page with a list of metrics specifically recorded for that service. Here is a brief selection:

- ECS CloudWatch Metrics
  - `CPUUtilization`, `MemoryUtilization`
- Kinesis CloudWatch Metrics
  - `GetRecords.IteratorAgeMilliseconds`, `IncomingBytes`
- Lambda CloudWatch Metrics
  - `Duration`, `IteratorAge`, `Invocations`, `Errors`
- RDS CloudWatch Metrics
  - `CPUUtilization`, `DatabaseConnections`, `ReadIOPS`

That's where we go when we are digging into specific performance of a particular service or resource. X-Ray can't give us that information, only CloudWatch. But what about the total propagation delay in our distributed, asynchronous system? Drumroll please: we present CloudWatch Custom Metrics, by which anyone can create an arbitrary metric, even across a distributed system.

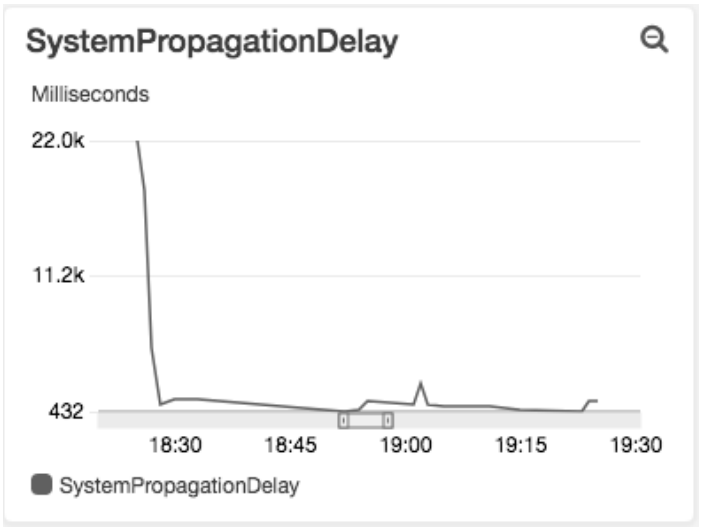
To create a Custom Metric, use the `PutMetricData` operation in the CloudWatch API. It is up to our creativity how to apply this, but one approach for our system under test is for the first compute node in the chain (the ECS Docker container) to take note of its start time, and make that an attribute of the message sent over Kinesis; then the last compute node in the chain (for us that is Lambda) calculates propagation time by subtracting the start time from the current time, and the Lambda then calls `PutMetricData` to store that propagation delay number in CloudWatch. Here is how it could be written in TypeScript for a NodeJS Lambda:

```
import {AWSError} from 'aws-sdk';
import * as CloudWatch from 'aws-sdk/clients/cloudwatch';
const cloudwatch = new CloudWatch();
const now: Date = new Date();
const propDelay: number = now.getTime() - ecsStartTime;
cloudwatch.putMetricData(
    {
        MetricData: [
            {
                MetricName: "SystemPropagationDelay",
                Timestamp: now,
                Unit: 'Milliseconds',
                Value: propDelay,
            }
        ],
        Namespace: "my-namespace",
    })
    .promise()
    .catch((err: AWSError) => console.log('Error', JSON.stringify(err)));
```

We can download metrics later for offline analysis using `GetMetricData` from the CloudWatch API:

```
aws cloudwatch get-metric-data --metric-data-queries '[
  {
    "Id": "m1",
    "MetricStat": {
      "Metric": {
        "Namespace": "my-namespace",
        "MetricName": "SystemPropagationDelay"
      },
      "Period": 60,
      "Stat": "Average",
      "Unit": "Milliseconds"
    }
  }
]' --start-time "2019-08-14T18:25:00Z" --end-time "2019-08-14T18:30:00Z"
```

And of course we will be able to view the Custom Metric using the CloudWatch Metrics console GUI, which works very well:



As noted strenuously above, there is some concern with the robustness of an approach that uses both the ECS clock and the Lambda clock. It is a weakness of the Custom Metrics approach. In fact, notice how the above SystemPropagationDelay metric is initially 22 sec, which seems unreasonably large, then normalizes to a more reasonable delay around 432 msec, presumably because the Lambda has finally synchronized its system clock to NTP.

Now is a good moment to switch gears. We have obtained the overall propagation delay number we were looking for. But we are still wearing our performance engineering hat, and we should ask a new question: How do we make the propagation delay quicker? Is there an obvious bottleneck in the system that we could optimize? Our system has a relational database, and AWS offers additional performance tools that are specific to RDS which can give us actionable information about the database that we cannot obtain from X-Ray or CloudWatch.

5 AWS RDS Enhanced Monitoring & Performance Insights

Let me start by saying the names are just too long. AWS does not do us the honor of abbreviating them, so I'll make the acronyms myself:

- RDS Enhanced Monitoring (RDS-EM)
- RDS Performance Insights (RDS-PI)

These are optional, added-cost features we can enable on a new or existing RDS database instance to get more advanced diagnostics. They won't tell us the propagation delay of our entire system under test, but they will help us diagnose more deeply when we have indications that the relational database is a source of slowness. The time to enable these features is *before* the performance problem happens, because, unlike CloudWatch, with RDS-EM and RDS-PI they only record performance data after we explicitly tell them to do so.

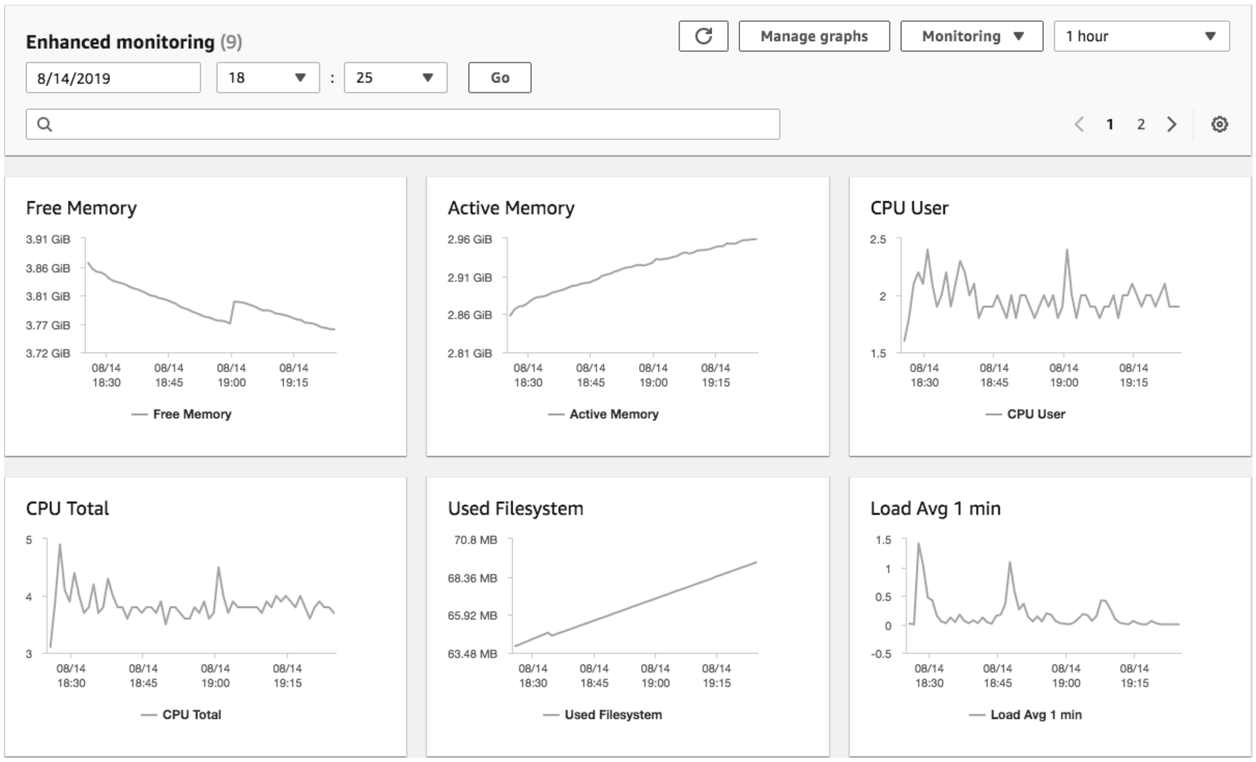
These features operate at the database instance level, not an entire database cluster. To enable RDS-EM requires an instance restart, as does RDS-PI for MySQL databases.

5.1 AWS RDS Enhanced Monitoring

RDS-EM extracts data from the operating system (OS) of the internal supporting VM on which the RDS instance runs. With RDS-EM, we get two new datasets: OS-level numeric metrics, and an OS process list. The process list is like what we would see with Linux "top" or "ps", where each process is shown with its memory and CPU usage. I have not yet encountered a "smoking gun" situation where I used RDS-EM and saw a specific database process hogging all the memory or CPU, but this is the only place where such an event could be discovered.

As for the OS-level numeric metrics, we get about sixty of them, and we can either graph them in the RDS console, or download them as JSON. There are some CloudWatch Metrics that appear redundant with RDS-EM metrics, but the latter offer much more detail. For example, CloudWatch Metrics for RDS offers `CPUUtilization`, while RDS-EM offers `cpuUtilization.{guest, idle, irq, nice, user, wait, ....}`, and `loadAverageMinute.{one, five, fifteen}`. CloudWatch Metrics for RDS offers `FreeableMemory` and `SwapUsage`, while RDS-EM Metrics offers `memory.{free, active, cached, hugePagesFree, mapped, ....}`. RDS-EM gives more granular metrics.

Notice I said we graph them in the RDS console, not the CloudWatch console. That is unfortunate, because the RDS-EM graphs are not as configurable or as easy to use as CloudWatch graphs. CloudWatch graphs let us customize the axes, specify a precise historical time window, plot arbitrary metrics on the same graph, perform functions on the data, and in my humble opinion they look better too. Also we can set CloudWatch Alarms on a CloudWatch Metric, to be notified whenever the metric exceeds a threshold, and that isn't supported with RDS-EM metrics. Finally, it would just be simpler UX to have just one data graphing functionality in AWS rather than multiple.



I see several reasons why RDS-EM graphs don't integrate with CloudWatch Metrics graphs. First, the time units don't always align: RDS-EM numeric metrics can be recorded on a granularity of 1, 5, 10, 15, 30, or 60 seconds, while CloudWatch Metrics are graphed on periods of 1, 5, 10, 30, or multiples of 60 seconds. The RDS-EM metrics are recorded on just one granularity, while CloudWatch Metrics are available on all periods for 15 days, 5 minute periods for 63 days, and 1 hour periods for 15 months. Also, the RDS-EM data is stored separately in CloudWatch Logs, rather than behind the AWS curtain, wherever CloudWatch Metrics stores its data. RDS-EM writes a JSON datafile to a CloudWatch Log Group called `RDSOSMetrics`, once per granularity interval. I am not privy to AWS internals, but conceptually those reasons do not appear to be utter blockers from AWS someday offering the ability to graph RDS-EM data using the CloudWatch Metrics and Dashboards widgets, which are superior for graphing. AWS, if you are listening, think about integrating RDS-EM graphs with CloudWatch Metrics graphs!

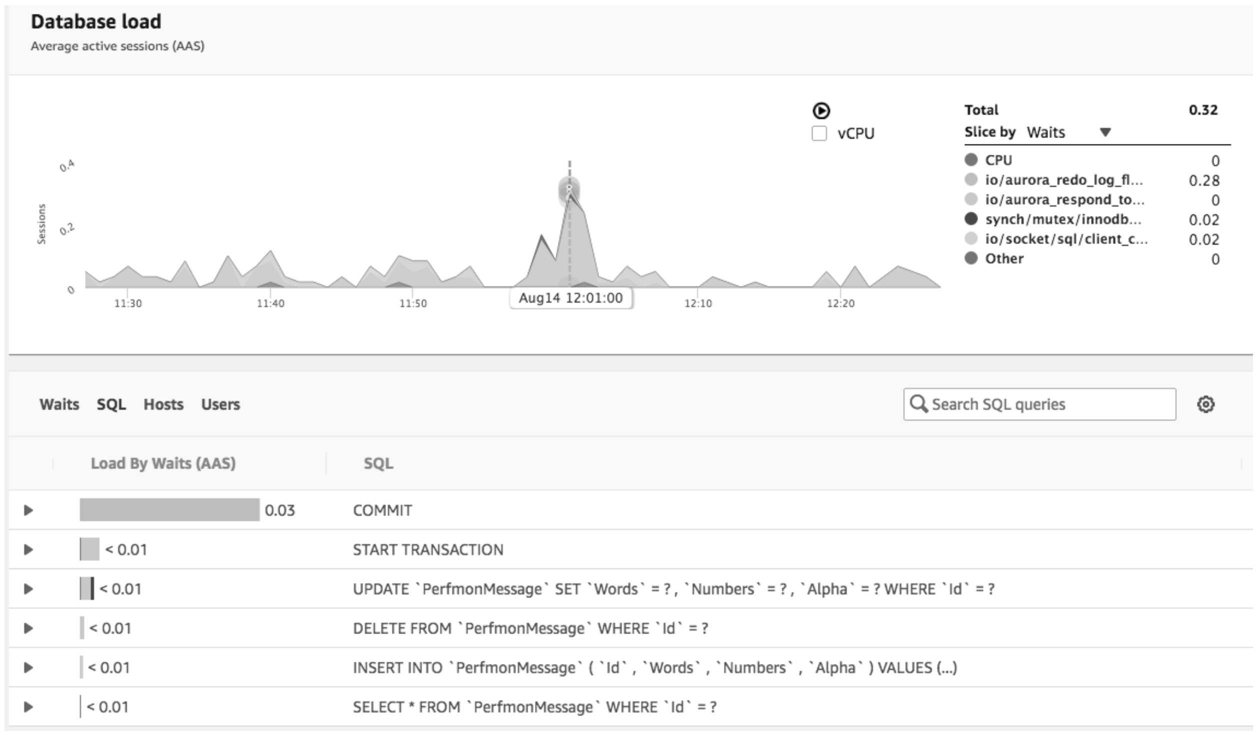
To amplify one last point: RDS-EM data is stored in CloudWatch Logs, and our cost for using RDS-EM is simply the cost of using CloudWatch Logs.

5.2 AWS RDS Performance Insights

RDS-PI is a grab bag of additional metrics and the colorful PI Dashboard. RDS-PI metrics overlap a lot with the RDS-EM metrics, but the color charts are unique to RDS-PI.

The key concept in the RDS-PI Dashboard charts is the new **`DBLoad`** metric, which is the count of active sessions on the database instance. An active session is "a connection that has submitted work to the DB engine and is waiting for a response from it," which I interpret to mean any session that has transmitted manipulation SQL like INSERT, UPDATE, SELECT; or definition SQL like CREATE, ALTER. If `DBLoad` is 5, then the database instance is actively processing SQL statements for 5 concurrent clients during the entire 60 second measurement interval. RDS-PI integrates with CloudWatch Metrics here by offering three new metrics, which we can graph alongside any other CloudWatch metric or Dashboard: `DBLoad`, `DBLoadCPU`, `DBLoadNonCPU`. By itself, that's not major news, because there is already a CloudWatch Metric for RDS called `DatabaseConnections`, which appears to have the same meaning.

Where RDS-PI brings its unique value is the PI Dashboard, which is shown on the RDS console, not the CloudWatch console. This is the diagnostics tool where we can slice up the contributors to `DBLoad` in two dimensions, obtaining the "performance insights" that might just pinpoint the culprit of a database slowness problem. This is the colorful chart I was talking about, and it looks simply fabulous in a screenshot, unless we happen to use dull grayscale like I did here:



Total	0.32
Slice by Waits	
● CPU	0
● io/aurora_redo_log_fl...	0.28
● io/aurora_respond_to...	0
● synch/mutex/innodb...	0.02
● io/socket/sql/client_c...	0.02
● Other	0

Load By Waits (AAS)	SQL
0.03	COMMIT
< 0.01	START TRANSACTION
< 0.01	UPDATE `PerfmonMessage` SET `Words` = ? , `Numbers` = ? , `Alpha` = ? WHERE `Id` = ?
< 0.01	DELETE FROM `PerfmonMessage` WHERE `Id` = ?
< 0.01	INSERT INTO `PerfmonMessage` (`Id` , `Words` , `Numbers` , `Alpha`) VALUES (...)
< 0.01	SELECT * FROM `PerfmonMessage` WHERE `Id` = ?

But hang on a minute, we won't get any value from the PI Dashboard until we learn its operating model. The Average Active Sessions (AAS) chart plots Sessions (aka `DBLoad`) versus time, and the area under the waveform is stratified into color-coded layers, according to the active **dimension**: Waits, SQL, Hosts, or Users. When we change the *Slice-by dimension* in the chart legend, the top line of the waveform stays the same but the color-coded layers change. If we slice-by SQL, the chart shows how many database sessions are occupied with the various types of SQL statement, and we could think about trying to optimize how those statements are used in the client application. If we slice-by Waits, we will see the session allocation to the engine-specific types of database wait states, which for Aurora MySQL might include `io/aurora_redo_log_flush` (waiting on a write to disk storage) or `synch/mutex/innodb/aurora_lock_thread_slot_futex` (waiting on a MySQL record lock). If we move the mouse left or right along the X-axis of time, the legend shows the exact number of sessions allocated to each value of the current slice-by dimension. In the above screenshot, at the selected point in time, the `io/aurora_redo_log_flush` Wait type has the largest contribution to `DBLoad`, compared to other Wait types, and it contributes 0.28 of the 0.32 total number of sessions. Since data granularity is fixed at 60 seconds, it means the database is in that wait state for 0.28 x 60 = 16.8 sec, from a total of 0.32 x 60 = 19.2 sec processing time, during this one-minute measurement interval.

RDS-PI gives us a second dimension of insight on the same displayed time window. In the *Load-by table* below the chart, choose a second dimension (again: Waits, SQL, Hosts, or Users) and we are rewarded with a table of color-coded bar graphs. In our screenshot, the COMMIT SQL command has the largest contribution to `DBLoad`, compared to other SQL queries, and it contributes 0.03 sessions, or 0.03 x 60 = 1.8 sec of processing time, on average, during all the 60 second intervals currently displayed in the AAS chart. The X axis spans one hour from 11:25 AM to 12:25 AM, so there are 60 measurement intervals of one minute each. The total sessions at the selected time point is 0.32, but the average sessions over the entire hour is a smaller number, 0.03.

When we put those two dimensions together, this tells us the `io/aurora_redo_log_flush` Wait type has the most database load, and the COMMIT SQL query is most responsible for that Wait type. We can get similar information directly from the database system tables, such as Aurora MySQL's table `performance_schema.events_waits_current`, but the RDS-PI color charts make it visually easier to identify.

There's no way the RDS-PI slice-by/load-by model could reasonably integrate its two-dimensional time series data with CloudWatch Metrics and Dashboards. It will always be a separate data graphing tool. So we won't be able to set CloudWatch Alarms on RDS-PI dimensionally-sliced datapoints, like the AAS chart and the Load-by bar charts. Also, just like with RDS-EM, the graphing features of RDS-PI are not as configurable or as easy to use as in CloudWatch.

In addition to the dimensional AAS chart and bar charts, enabling RDS-PI gives us access to dozens of "Performance Insight Counters." The **PI Counters** are plain old time-series data, and cannot be sliced like `DBLoad`. About half of these counters are engine-specific datapoints, like `Innodb_rows_read` and `Table_locks_waited` for MySQL, and the rest are the same OS-level numeric metrics provided by RDS-EM. Unlike RDS-EM, with RDS-PI we cannot download PI Counters over the AWS API, and the datapoint granularity is limited to just one point per 60 seconds, so for access to OS-level metrics we see that RDS-EM does a better job. The engine-specific data are helpful, because although we could obtain them separately by a select statement into the appropriate system table, like the `events_waits_current` table, RDS-PI offers the convenience of selecting them iteratively and storing the time-series. Again, none of these PI Counters can be graphed in CloudWatch, only in the less desirable graphing UX on the RDS console.

RDS-PI is unique for the way it provides dimensionally-sliced insight into the `DBLoad` metric, but it does not provide that capability for any other indicator. If the `DBLoad` metric is a large number, that is clearly a problem and we would want to delve into the dimensional analysis to see why so many database sessions have accumulated, but I could imagine a situation of database slowness caused by a small number of very bad user sessions, causing a small `DBLoad`, and the AAS chart would not draw our attention to that. Don't get me wrong, the slicing of `DBLoad` is very helpful, but I think it would add huge additional value and "actionable intelligence" if we could use the RDS-PI dimensions to slice other existing RDS CloudWatch Metrics, like CPU and memory usage, network throughput, IOPs, and operation latency. Maybe that's very hard to do, behind the AWS curtain, but I can still dream.

6 Conclusions

The use of AWS Kinesis messaging made our system asynchronous. Calculating the propagation delay through an asynchronous, distributed system is much more difficult than timing a synchronous, round-trip request, because it is hard to find a single clock to measure an asynchronous activity from start to finish. If the start/end clocks are separate, their skew becomes our timing error, and when we are measuring time on the order of milliseconds, even small error is significant. Ignorance of the precise skew casts doubt on the validity of the result. Our system under test also included AWS ECS and AWS Lambda, which are designed for ephemeral infrastructure, and since their supporting VMs are likely also ephemeral, those services are more susceptible to clock error when a fresh VM has not yet synced with NTP.

AWS X-Ray is for measuring the timing of a distributed trace through the nodes of an AWS service graph. It collects trace segments from specific integrated AWS services, and from compute nodes that use the X-Ray API. Because X-Ray relies on the client to supply start/end times, the trace result in our asynchronous system is subject to clock error. Additionally, X-Ray cannot directly represent the propagation delay across the Kinesis service, although the delay can be implied by the start/end times of X-Ray segments transmitted by the services on either end of the Kinesis stream. If the task is to measure system propagation delay, X-Ray is best used with a system based on synchronous requests. X-Ray is also helpful to get a general sense of the timing health of an isolated service node, or to drill into sub-segment timing of an isolated node. X-Ray is for timing measurements and not other kinds of metrics.

AWS CloudWatch Logs is a plausible answer to the clock error problem on an asynchronous system, because it records log event ingestion time using its own clock. CloudWatch VMs should be long-lived, not ephemeral, therefore presenting a stable clock. If the compute nodes at both ends of the system under test write to CloudWatch Logs, we can calculate propagation delay as time2 minus time1 because the ingestion times are based on a central source of timing truth. The pitfall of using CloudWatch Logs for performance measurement is its lack of search indexing, and the subsequent difficulty in locating the necessary log events.

After the initial collection of propagation delays through the entire system, we will still need to drill down and ask where the bottlenecks are, and how they can be explained and improved. X-Ray measures the typical delay in isolated nodes. To explain the delay, we need AWS CloudWatch Metrics. Its graphing capability in the console is very mature, and we do not need to take any action to enable data recording of predefined service metrics. Using Custom Metrics, we have another solution for measuring propagation delay through an asynchronous system, although like X-Ray it is subject to clock error.

Because the system under test uses an AWS RDS relational database, we considered RDS Enhanced Monitoring and RDS Performance Insights. They add greater detail on OS-level numbers and engine-specific datapoints that are not available in CloudWatch Metrics. However, RDS-EM, RDS-PI, and CloudWatch Metrics are mutually incompatible systems. Except for three `DBLoad` metrics, we cannot graph RDS-EM and RDS-PI metrics in CloudWatch Metrics (nor use them to trigger CloudWatch Alarms), and instead must use separate graphing widgets on the RDS console that are not as good. The strength of RDS-PI is not its counter metrics, but rather its two-dimensional approach to slicing up the contributing factors in database load.

To sum it all up, we have multiple tools at our disposal in AWS for measuring performance and drilling down for more details. Our strategy for using one or more of these tools will be guided by the types of AWS services in our system, like Kinesis or RDS, which have different levels of support for performance analysis. Finally, when reporting propagation delay, we will always remember to be very clear about how start/end times are measured, including an honest acknowledgement where timing error is unknown.

References

AWS Developer Guides:

- CloudWatch
  https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html

- RDS Enhanced Monitoring
  https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_Monitoring.OS.html

- RDS Performance Insights
  https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_PerfInsights.html

- AWS X-Ray
  https://docs.aws.amazon.com/xray/latest/devguide/aws-xray.html

X-Ray Daemon on GitHub:

- https://github.com/aws/aws-xray-daemon

JavaScript 'mysql' client library:

- https://www.npmjs.com/package/mysql

The code to stand up this AWS system and run the performance test:

- https://github.com/daniel-kranowski/aws-perfmon-01