

WebRTC Call Debugger

Project Engineering

Year 4

Patrick Feeney

Bachelor of Engineering (Honours) in Software and Electronic Engineering

Atlantic Technological University

2024/2025

Acknowledgment

I would like to thank all my lecturers at ATU for their patience and guidance.

My manager at Genesys also deserves thanks for enabling me to align my college project with my work.

Finally, I am grateful to my parents for their support throughout my education.

Declaration

This project is presented in partial fulfilment of the requirements for the degree of Bachelor of Engineering (Honours) in Software and Electronic Engineering at Atlantic Technological University.

This project is my own work, except where otherwise accredited. Where the work of others has been used or incorporated during this project, this is acknowledged and referenced.

Patrick Feeney

Summary

WebRTC is an opensource web technology that allows peer-to-peer communication. This project focuses on improving the way statistics from these calls are displayed. This was done by querying a log storage service and formatting the data requested to allow it to be plotted. Plotting WebRTC statistics for calls done in the past is important to understand why there were issues in a call such as it ending abruptly. The final product of this project is a dashboard called `webrtc-timeline` that is a functional, flexible and friendly web interface that displays time series data.

Table of contents

1. [WebRTC Call Debugger](#)
2. [Acknowledgment](#)
3. [Declaration](#)
4. [Summary](#)
5. [Table of contents](#)
6. [Intro](#)
7. [Background](#)
 1. [Web Real-Time Communication](#)
 2. [WebRTC Statistics](#)
 3. [Debugging WebRTC Challenges](#)
 4. [Existing tools](#)
 5. [Technologies Used](#)
8. [Project Architecture](#)

1. [Architecture Diagram](#)
2. [Sequence Diagram](#)
3. [Architecture of a Genesys Cloud call](#)
4. [Statistics gathering](#)
9. [Project Plan](#)
10. [Self Learning](#)
11. [Backend development](#)
 1. [API Keys](#)
 2. [Hyperion](#)
 3. [Data Fetching](#)
 4. [Data aggregation](#)
 5. [Meaning of Statistics](#)
 6. [Testing](#)
12. [Frontend development](#)
 1. [Preparing data to be plotted](#)
 2. [Data aggregation and grouping](#)
 3. [Building the UI](#)
 1. [Checkboxes](#)
 2. [Special Hover Effect](#)
 1. [Hover](#)
 2. [Click](#)
 3. [ChartWrapper useEffect](#)
 3. [Performance Issues](#)
 1. [UseMemo](#)
 2. [Progressive Loading](#)
13. [Ethics](#)
14. [Findings](#)
15. [Conclusion](#)
16. [Recommendations](#)
17. [References](#)
18. [Appendix](#)
 1. [Appendix A: Logs](#)
 1. [JSON Data from New Relic excerpt](#)
 2. [Processed data in Lambda](#)
 3. [Data for plotting](#)
 2. [Appendix B: AWS Lambda](#)
 1. [Parser New Relic Search](#)
 2. [Parser Page Builder](#)
 3. [Parser AWS Secrets](#)
 4. [Parser Utils](#)
 3. [Appendix C: React Frontend](#)
 1. [App Component](#)
 2. [User Component](#)
 3. [Track Component](#)
 4. [Extra Info Accordion and Extra Info Components](#)
 5. [ChartWrapper and Chart Components](#)
 6. [Checkbox Component](#)
 7. [Data Aggregator Helper](#)

Intro

Genesys Cloud Services is an American software company that sells customer experience and call centre technology to businesses [1]. One of Genesys's main products is called Genesys Cloud, which leverages Amazon Web Services (AWS) to orchestrate and streamline voice and video communications between agents and customers. Genesys Cloud utilises WebRTC on the agent-side to establish these calls.

The quality, reliability and performance of a call using WebRTC within Genesys Cloud can be influenced by numerous factors, some of which are the responsibility of Genesys to address. To find the cause and troubleshoot these problems, extensive call data is logged and stored. This manual analysis of this extensive logged data is a time-consuming process that can delay troubleshooting of call issues.

The goal of this project was to improve the efficiency of Support Engineers in debugging WebRTC calls by creating a user-friendly interface for visualising logged data from past calls. This was achieved through a full stack project involving Amazon Web Services as the infrastructure for the backend and React for the frontend.

This report details the design, implementation, and testing of the 'WebRTC Call Debugger' dashboard, covering both the backend and frontend technologies used.

Background

Web Real-Time Communication

Web Real-Time communication (WebRTC) is an open-source technology enabling web-powered applications to capture and stream media, such as audio and video, directly between peers without requiring intermediary servers or user installed software. [2] WebRTC allows teleconferencing directly within browsers through standardized JavaScript APIs. [3]

WebRTC Statistics

Debugging WebRTC Challenges

Modern browsers, like Chrome, provide developer tools such as `webrtc-internals` that provide valuable real-time data for ongoing WebRTC communications. The issue at hand is that this tool does not support the analysis of completed calls. Once the call is finished, no more data is shown. This presents a challenge when diagnosing issues that occurred in the past, such as unexpected disconnections or periods of poor call quality. To address this issue, agent-side WebRTC statistics are collected and logged persistently to an external system called New Relic. An example of a useful statistic is `packetLoss` which shows how much data the user is not being able to send or receive. [14] This project aimed to develop an interface by building upon the existing `webrtc-timeline` project to display this data.

Existing tools

There are no tools specialised in plotting WebRTC data in a similar manner to how `webrtc-internals` does it in real-time. The only commercial tool that I found that was related to this is CallStats.io but it seems to no longer be running. [4]

Generic graphing tools such as Excel or Grafana could be used but at that point it becomes easier to write custom code to be able to integrate it better with our existing support systems at Genesys. Integrating the WebRTC call debugger into our support systems such as Supportability allows Support Engineers to query data simply by entering a conversation ID, without needing to specify a time range or other parameters.

Technologies Used

A full-stack approach was required to address both data querying and visualisation requirements. The backend code runs in a serverless environment using AWS lambda. This solution was chosen due to its low maintenance cost, only running when required. The frontend is a React project using both Genesys's own UI elements and

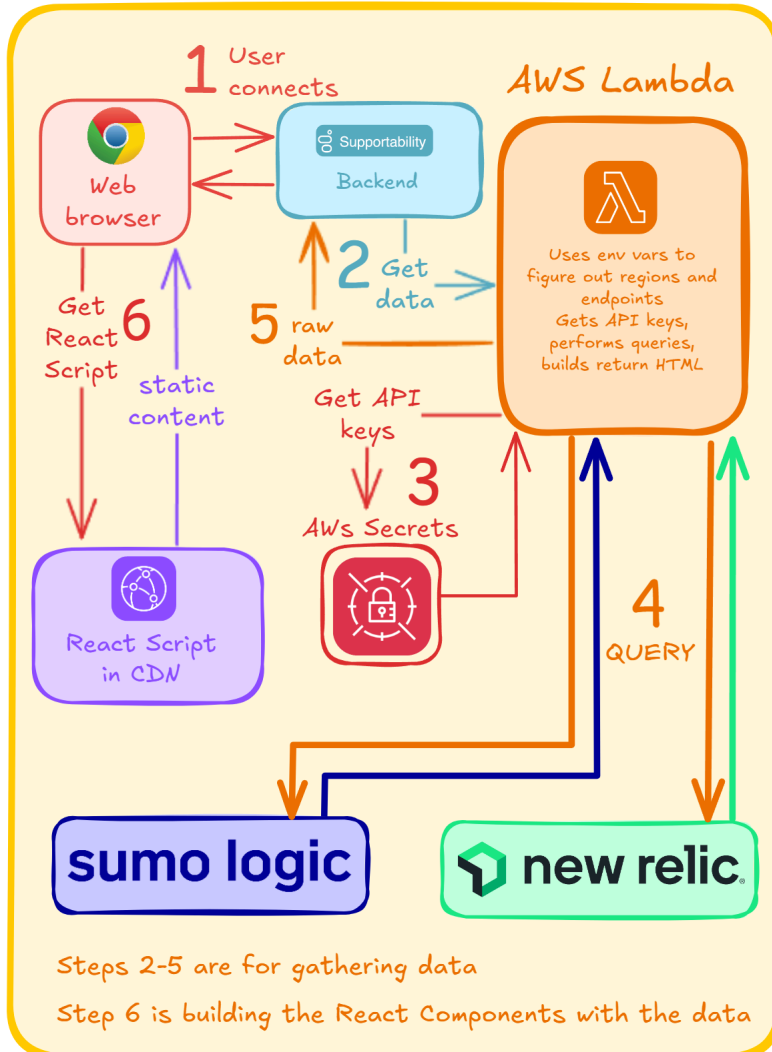
Material UI (MUI) X Charts for plotting data dynamically. The reason for using React is due to the Team's familiarity with this technology and its current use in multiple repositories.

Project Architecture

Architecture Diagram

The following diagram illustrates the high-level architecture of the WebRTC Call Debugger, showing its main components and how they are connected.

The main components of the system include the Supportability service (intermediary platform), the AWS Lambda function (backend for project), the AWS Secrets Manager (API storage), data sources: Sumo Logic and New Relic, and the React Script in AWS CDN.

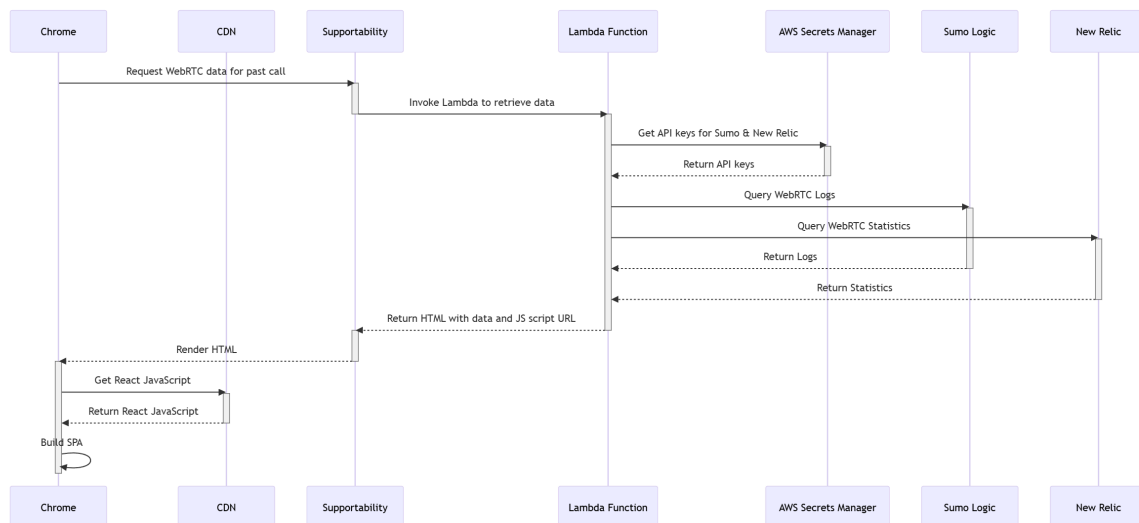


Sequence Diagram

The image below is a sequence diagram displaying what the usual process of the WebRTC Call Debugger looks like.

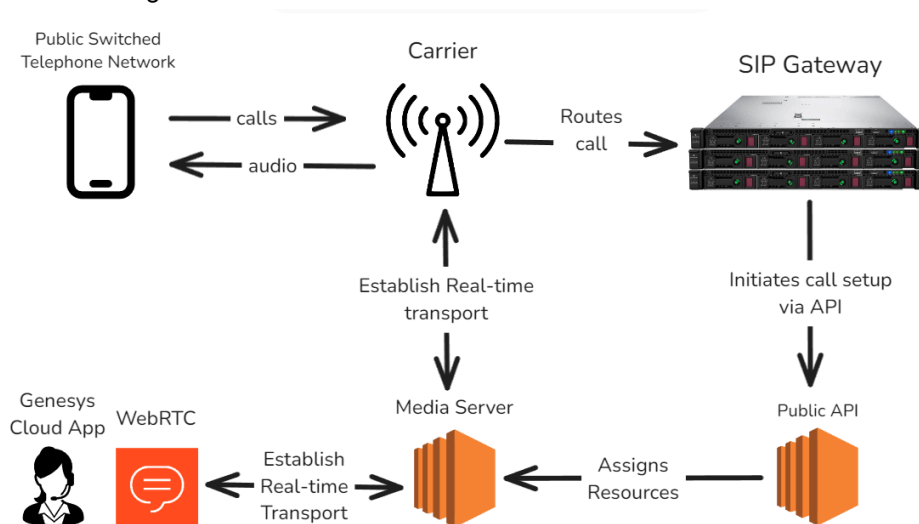
1. First, a Support Engineer, through Chrome, connects to Supportability and performs a query on a specific call.
2. Supportability, using AWS SDK, invokes an AWS Lambda Function, passing the required arguments.
3. The Lambda Function uses the AWS SDK to retrieve saved API keys from AWS Secrets Manager.
4. The Lambda Function uses these API keys to query data from Sumo Logic and New Relic.
5. The Lambda Function embeds the data into a HTML document and returns it to Supportability.
6. Supportability sends back the data to the web browser.

7. The web browser renders the HTML and fetches the required JavaScript from a CDN to build the Single Page Application (SPA).



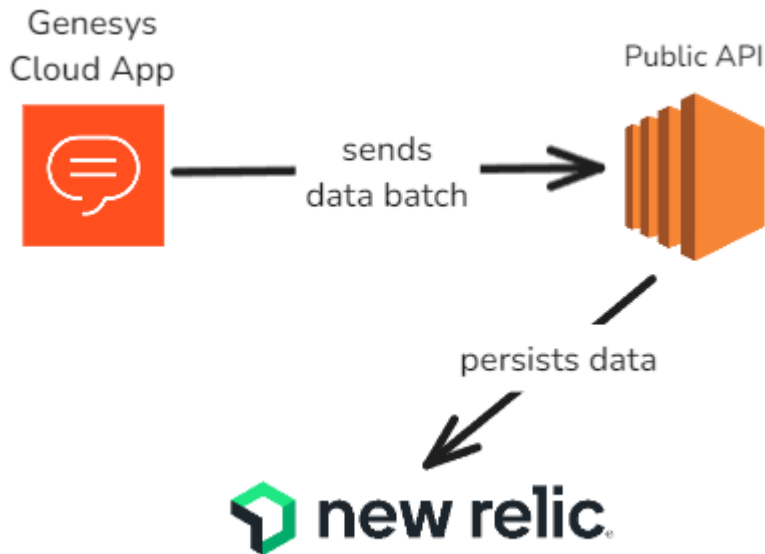
Architecture of a Genesys Cloud call

To understand where the data visualised by the WebRTC Call Debugger originates from, it is helpful to understand the overall architecture of a Genesys Cloud call. The following is the path of a call between a customer and agent



The voice call is initiated by the customer using a smart phone. The carrier receives the call and routes it to Genesys's Session Initiation Protocol (SIP) Gateway. The SIP Gateway initiates the call setup by making an API request to Genesys's Public API. This triggers resource allocation on the media server. The media server uses Session Description Protocol (SDP) to establish Real-time Transport Protocol (RTP) streams with both the carrier and the agent. Once both RTP streams are established, the media server mixes the audio streams from the carrier and the agent, allowing both parties to communicate.

Statistics gathering



Before data can be retrieved from New Relic, it must first be gathered. Call statistics are gathered within the Genesys Cloud App by calling the WebRTC `getStats` API that allows to take a snapshot of the call's statistics. Snapshots are captured every five seconds. Once five snapshots have been recorded, they are sent to Public API as a batch. In Public API, these batches are routed to the corresponding New Relic account to get stored. The logs stored in New Relic serve as this project's data source.

Project Plan

This project followed a Kanban methodology, providing a flexible framework for managing evolving requirements. Notion was set up as a tool for high-level planning and goal tracking. My day-to-day project management and note-taking tool was Obsidian MD. This software served as a dynamic notebook for capturing thoughts, documenting daily progress, and managing tasks. The flexibility of Obsidian MD was valuable for this project due to the evolving and changing requirements as development progressed.

The project followed a two-semester timeline. Semester one primarily focused on the development of the backend AWS Lambda function. Important milestones during this phase included the initial integration with New Relic's API and the implementation of AWS Secrets Manager for secure API key management. Semester two was largely dedicated to frontend development using React. This involved designing the user interface and implementing the data handling logic from the Lambda to enable data visualization using MUI X Charts. The backend development was prioritized in the first semester to ensure the frontend had the necessary data for plotting and testing.

Self Learning

Working on the `webrtc-timeline` repositories, which are written in TypeScript, presented an opportunity to develop new technical skills.

At the start of this project I was not initially familiar with TypeScript, a typed version of JavaScript. I had to learn more about it to be able to contribute effectively. To build a foundational understanding, I began by reading the O'Reilly book "Effective TypeScript," focusing on the initial sections covering fundamental concepts and best practices. I took detailed notes on the most important points to solidify my understanding of the language. However, a significant portion of my learning throughout this project was empirical. This included actively seeking guidance from colleagues through discussions, analyzing existing code within company repositories to understand established patterns and implementations (particularly for interacting with internal services like Hyperion and AWS), and researching solutions by searching public repositories and documentation.

Backend development

This section details the backend implementation of the WebRTC Call Debugger, focusing on the AWS Lambda function responsible for data retrieval and processing.

The project was broken down into two repositories four years ago, in 2021. The reason was that the HTML used to be built directly in the AWS Lambda, and sometimes the data retrieved from Sumo Logic was too large, causing the lambda to fail since it could not return a response because the size of the HTML exceeded the maximum synchronous return size of the AWS Lambda function, which is six megabytes [11]. To address this the codebase was broken down into two separate repositories. The `webrtc-timeline-parser` which gathers the data and returns it to the client through Supportability, and `webrtc-timeline-react` which runs once the user receives the data from Supportability, and builds the website using it.

The backend for this project runs in an AWS Lambda. This is a serverless architecture. Meaning that there is no server constantly running in the background. The reason why the backend used a serverless architecture is that this tool is not meant to be running constantly. It only runs a Genesys Employee wants to look at the WebRTC logs from a call.

The code that runs in the Lambda is JavaScript, in a NodeJS environment. During development TypeScript is used, allowing us to adhere to types while coding, helping us find bugs earlier. Once the development in TypeScript is done, the code is transpiled into JavaScript which is then uploaded to AWS. Transpiling is a process where code from a high-level language is converted into another high-level language. Typescript is in charge of this. This is part of the configuration for transpiling into JavaScript:

```
"target": "es5",  
"module": "commonjs",
```

TypeScript allows us to specify what version of JavaScript to use. For this repository the target is 'es5' which was released in December 2009. [12] The reason for choosing this version of ECMAScript is to ensure compatibility on every browser. The 'module' property set to 'commonjs' ensures that the JavaScript code uses 'require()' instead of 'import()' which is what the NodeJS environment uses for importing dependencies.

The lambda is invoked from a different backend service called Supportability. Supportability is a central point where Quality and Support Engineers can find information about different communications done using Genesys's infrastructure. Supportability invokes the Lambda using AWS's software-development-kit (SDK). Supportability provides the lambda with a 'conversationId' and the start and end times of the call. These arguments are used to find the corresponding information efficiently.

Before I started work on this project, the `webrtc-timeline` frontend only showed WebRTC logs from Sumo Logic in tabular format:

Expand All

User - 84f98d08-f7ac-4954-80c7-48c4db7fe290 ^

App - collaborate-ui-sdk:eeb03ae4-48c9-47b4-b521-04277f8442e0 v

App - volt:0f2842c2-5d67-4f02-8668-15c947b0e27d ^

| clientTime | diff (ms) | appName | message | sessionType | extra |
|--------------------------|-----------|------------------|---|------------------|-------------------|
| 2025-04-19T10:14:03.632Z | 0 | streaming-client | [streaming-client] propose received | collaborateVideo | ▶ { ... } 4 items |
| 2025-04-19T10:14:03.632Z | 1 | streaming-client | [streaming-client] sending jingle proceed | collaborateVideo | ▶ { ... } 3 items |
| 2025-04-19T10:14:03.634Z | 0 | webrtc-sdk | [webrtc-sdk] onPendingSession | collaborateVideo | ▶ { ... } 5 items |
| 2025-04-19T10:14:03.635Z | 1 | streaming-client | [streaming-client] sent jingle proceed | collaborateVideo | ▶ { ... } 3 items |
| 2025-04-19T10:14:03.942Z | 307 | streaming-client | [streaming-client] offer received | | ▶ { ... } 2 items |
| 2025-04-19T10:14:03.978Z | 36 | webrtc-sdk | [webrtc-sdk] received webrtc session | collaborateVideo | ▶ { ... } 3 items |
| 2025-04-19T10:14:03.979Z | 1 | volt | [volt] New session incoming: | | ▶ { ... } 2 items |
| 2025-04-19T10:14:04.102Z | 123 | volt | [volt] publishing message to host app | | ▶ { ... } 3 items |
| 2025-04-19T10:14:04.102Z | 0 | webrtc-sdk | [webrtc-sdk] handledPendingSession | | ▶ { ... } 2 items |
| 2025-04-19T10:14:04.362Z | 260 | webrtc-sdk | [webrtc-sdk] Using track based actions | collaborateVideo | ▶ { ... } 3 items |
| 2025-04-19T10:14:04.364Z | 2 | webrtc-sdk | [webrtc-sdk] Incoming track | collaborateVideo | ▶ { ... } 4 items |
| 2025-04-19T10:14:04.364Z | 0 | webrtc-sdk | [webrtc-sdk] Incoming track | collaborateVideo | ▶ { ... } 4 items |
| 2025-04-19T10:14:04.365Z | 1 | webrtc-sdk | [webrtc-sdk] accepting session | collaborateVideo | ▶ { ... } 6 items |
| 2025-04-19T10:14:04.392Z | 27 | streaming-client | [streaming-client] ICE state changed: | collaborateVideo | ▶ { ... } 4 items |
| 2025-04-19T10:14:04.393Z | 1 | streaming-client | [streaming-client] Discovered ice candidate to send to peer | collaborateVideo | ▶ { ... } 4 items |
| 2025-04-19T10:14:04.394Z | 1 | streaming-client | [streaming-client] Connection state changed: | collaborateVideo | ▶ { ... } 4 items |
| 2025-04-19T10:14:04.515Z | 121 | streaming-client | [streaming-client] ICE state changed: | collaborateVideo | ▶ { ... } 4 items |
| 2025-04-19T10:14:04.515Z | 0 | streaming-client | [streaming-client] sending session info: active | collaborateVideo | ▶ { ... } 3 items |

Data coming from this source are WebRTC logs; which are events about the call state. From the call initialisation, to users muting their microphones. These logs do not include any statistics about the call performance. This is where this project comes in. This project adds data from New Relic that shows call statistics from the Genesys Cloud App that the Agent is using. Giving the Engineers more information and tools to debug calls.

API Keys

Since New Relic had not been implemented when I started working on this project, there were no available API keys for me to use. I requested API keys from the corresponding DevOps department by opening a ticket. Once I got the keys, I made HTTP requests manually through Postman to learn what the request structure should look like and what the format of the response was. Here I realised that New Relic was deprecating the conventional REST interface and was replacing it with a GraphQL one. I had to adapt the code to match this.

To store the newly created New Relic API keys and improve the security of the handling of Sumo Logic API keys, my manager suggested using AWS Secrets. Leveraging existing AWS infrastructure was the primary reason for this choice. The previous Lambda setup required passing Sumo Logic API keys as an argument when invoked by Supportability. This method is insecure and led to a previous incident where keys were accidentally leaked through error logging. By fetching both sets of API keys from AWS Secrets Manager, the Lambda function no longer depends on Supportability to provide them as arguments.

The following TypeScript code demonstrates how both the New Relic and Sumo Logic API keys are retrieved from AWS Secrets using the `SecretsManagerClient` in a single batch request:

```
const newRelicSecretId = "WebRTC-Timeline-Parser-NewRelicApiKeys"; // names of the secrets to be retrieved
const sumoLogicSecretId = "WebRTC-Timeline-Parser-SumoLogicApiKey";
const client = new SecretsManagerClient();

export async function getSumoAndNewRelicSecrets() {
  // build the request, specifying the name of the Secrets that we want to retrieve
  const input = {SecretIdList: [newRelicSecretId, sumoLogicSecretId]}
  const command = new BatchGetSecretValueCommand(input);
  const response = await client.send(command); // fetch Secrets

  // ... error handling removed for brevity ...

  // code to parse the keys from the response
  const {apiKey} = JSON.parse(response.SecretValues[0].SecretString || "{}");
```



```
const {accessCode, accessKey} = JSON.parse(response.SecretValues[1].SecretString || "{}");

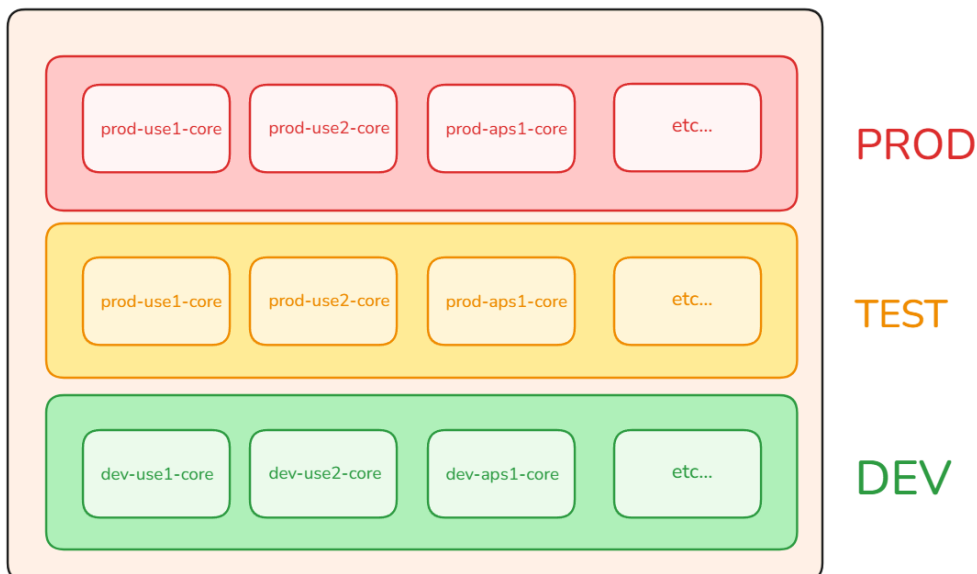
return {
  newRelicApiKey: apiKey,
  sumoAccessCode: accessCode,
  sumoAccessKey: accessKey
}
```

As shown in the code, a `SecretsManagerClient` is created, and the `BatchGetSecretValueCommand` is used with a list of secret IDs. The `client.send()` function executes the command, and the response is then parsed to extract the individual API keys for New Relic and Sumo Logic using object destructuring.

A key consideration was understanding how the AWS SDK authenticates when interacting with AWS services like Secrets Manager. While local development uses credentials from a local configuration file (`~/.aws/credentials`), deployed Lambda functions utilise credentials provided by AWS Identity and Access Management (IAM). IAM is an AWS service that allows for the secure management of access to AWS resources. The Lambda function in this project is configured with an IAM role that grants it permission to retrieve the necessary secrets from AWS Secrets within the same region and account where the Lambda is deployed.

Hyperion

Genesys Cloud operates across multiple AWS regions globally, with over eight different locations around the world. Each of these regions hosts its own instance of major microservices, including the Lambda function for the WebRTC Call Debugger. This distributed architecture means that the endpoints for external services like Sumo Logic and New Relic cannot be hardcoded within the Lambda function, as they vary depending on the specific AWS account, environment (e.g., 'prod', 'test', and 'dev'), and region where the Lambda is running. Below is an image that showcases an example of how different environments have different AWS accounts associated with them. There are three main environments, each having multiple AWS accounts in them.



There are over 15 Genesys Cloud regions. Each region has more than a single AWS account. [20]



To ensure the code can correctly determine the appropriate endpoints and accounts for querying Sumo Logic and New Relic based on its runtime environment, it needs dynamic configuration. For example, a Genesys Cloud server running in Ireland would need to use the European New Relic data center endpoint, not the American one, along with the correct New Relic account ID which is unique to each Genesys Cloud deployment. Figuring out how the code could reliably obtain this information was a significant challenge that involved researching Genesys's internal documentation and consulting with colleagues.

Through this investigation, I learned about Hyperion, an internal Genesys service. Hyperion is designed for dynamic configuration and state management of Genesys Cloud's environmental data within their multi-AWS account infrastructure. This service allows me to inject environment variables into the AWS Lambda function during the deployment process. Hyperion possesses the necessary knowledge to determine the correct account and endpoints for interacting with other services based on the Lambda's region, AWS account, and environment. This allows the Lambda function to use Hyperion as the source of truth for deciding where to make HTTP requests to query data. If a property like the endpoint for New Relic changes, the Lambda's environment variables are updated by Hyperion upon deployment or restart, ensuring the function uses the most current configuration.

Integrating with Hyperion was a tough challenge and took a considerable amount of time to fully understand, particularly understanding Genesys's multi-AWS account structure and the variables involved. However, successfully leveraging Hyperion ensures the Lambda function operates correctly across different regions and environments.

To integrate Hyperion I had to include it as part of the Lambda's serverless configuration. Here the specific environment variables are configured to be injected:

```
environment:
  NR_API_HOST: { Ref: Hyperion.xxx.xxx.newrelic.endpoints.apiHost }
  NR_ACCOUNT_ID: { Ref: Hyperion.xxx.xxx.newrelic.accountId }
  SUMO_API_URL: { Ref: Hyperion.sumo.apiUrl }
```

Then, as part of the data retrieval process, a function is invoked to read these environment variables. This function attempts to read the values from the environment variables and uses default values if a variable is not set or is undefined which is useful to always have a fallback.

```

async function getNewRelicAndSumoConfigs() {
  // Reads the content of the environment variables
  const sumoApiUrl = process.env['SUMO_API_URL'] || SumoEndpointsByRegion['us-west-2'];
  const newRelicHost = process.env['NR_API_HOST'] || 'api.newrelic.com';
  const newRelicAccountId = process.env['NR_ACCOUNT_ID'] || 'xxx';

  const secrets = await getSumoAndNewRelicSecrets(); // Gets API keys from AWS Secrets

  return {
    ...secrets, newRelicHost, newRelicAccountId, sumoApiUrl
  };
}

```

Data Fetching

Once the API keys and environment variables are set up, the Lambda function proceeds to fetch the relevant data from New Relic.

The data retrieval process begins by obtaining the necessary runtime configuration, which includes the API keys and endpoint information. This is done by executing the `getNewRelicAndSumoConfigs` function:

```

const runConfig = await getRunConfig(config);

```

With the `runConfig` object, containing the API keys, the function responsible for fetching New Relic timelines is called:

```

const newRelicTimelinesPromise = getNewRelicTimelines(runConfig);

export async function getNewRelicTimelines(config: RunConfig) {
  let results = await newRelicSearch(config); // fetches data
  return createTimelinesFromResultsNewRelic(results); // extracts what we care about and
  groups
}

```

The `newRelicSearch` function builds and performs the actual HTTP request:

```

export async function newRelicSearch(config: RunConfig) {
  const searchOptions = await getSearchOptions(config);
  const results = await performSearch(searchOptions);
  return results;
}

```

The construction of the GraphQL request, including setting the headers and request body, occurs within the `getSearchOptions` function:

```

export async function getSearchOptions(config: RunConfig): Promise<SearchOptions> {
  const { newRelicHost, newRelicAccountId, conversationIds, from, to } = config;
  const query = getQuery(conversationIds, from, to);
  const headers = { "API-Key": config.newRelicApiKey, "Content-Type": 'application/json' };
  return {
    requestOptions: {
      headers,
      method: 'POST',
      body: JSON.stringify({

```

```

        query: `{actor { account(id: ${newRelicAccountId}) { nrql(query: "${query}") {
results } } } }`
      })
    },
    url: `https://${newRelicHost}/graphql`
  };
}

```

This function dynamically sets the API key in the header and constructs the GraphQL query based on the provided configuration and the `nrql` query built by the `getQuery` function. The `nrql` query language, similar in syntax to SQL, is used to retrieve data from New Relic's database. The `getQuery` function builds the specific query to retrieve all data from the `WebRTCStats` data source for one or more specified `conversationId`s within a defined time range.

```

function getQuery(conversationIds: string[], from: moment.Moment, to: moment.Moment) {
  const limit = 'MAX';
  return `SELECT * FROM WebRTCStats WHERE (${conversationIds.reduce((output, curr, idx) =>
output + 'conversationId = ' + '\'' + curr + '\'' + (conversationIds[idx] !==
conversationIds[conversationIds.length - 1] ? ' OR ' : ''), '')) SINCE ${from.format('x')}
UNTIL ${to.format('x')} LIMIT ${limit}`;
}

```

Once the `searchOptions` are prepared, the actual HTTP request to the New Relic GraphQL endpoint is made by the `performSearch` function:

```

async function performSearch(searchOptions: SearchOptions): Promise<NewRelicSearchResult[]> {
  const start = moment();

  // make the request
  const response = await fetch(searchOptions.url, searchOptions.requestOptions);
  if (!response.ok) { /* ...error handling... */ }

  const responseBody: NewRelicJson = await response.json(); // get the body

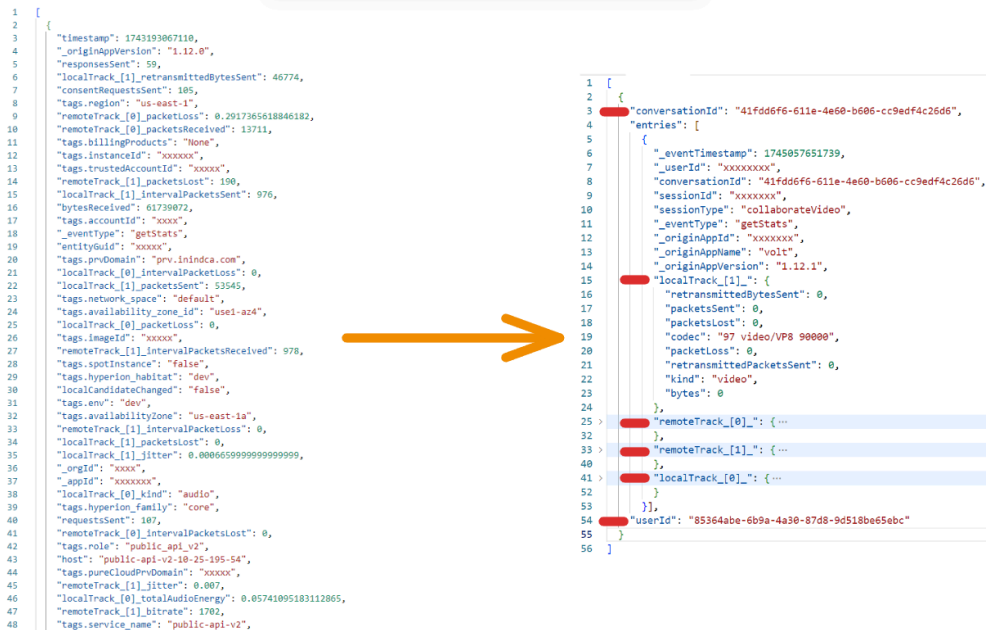
  logger.info(`Search total time: ${moment().diff(start, 'seconds')} seconds`);
  // since the request was made using GraphQL we receive the response in the same format
  return responseBody.data.actor.account.nrql.results;
}

```

This function uses the `fetch` API to send the request, handles potential errors, and logs the time taken for the search. The response body is parsed as JSON, and the relevant results are extracted from the GraphQL response structure. This type of request to New Relic usually takes two seconds.

The next step is the aggregation and processing of the retrieved data.

Data aggregation



```

1 {
2   "timestamp": 1743193867110,
3   "_originAppVersion": "1.12.0",
4   "responsesSent": 59,
5   "localTrack[1].retransmittedBytesSent": 46774,
6   "consentRequestsSent": 105,
7   "tags.region": "us-east-1",
8   "remoteTrack[0].packetsLost": 0,
9   "remoteTrack[0].packetsReceived": 13711,
10  "tags.billingProducts": "None",
11  "tags.instanceId": "xxxxxx",
12  "tags.trustedAccountId": "xxxxxx",
13  "remoteTrack[1].packetsLost": 190,
14  "localTrack[1].intervalPacketsSent": 976,
15  "bytesReceived": 61739072,
16  "tags.accountId": "xxxx",
17  "event": "getStats",
18  "entityId": "xxxx",
19  "tags.prdomain": "prv.inidca.com",
20  "localTrack[0].intervalPacketsLost": 0,
21  "localTrack[1].packetsSent": 53545,
22  "tags.network_space": "default",
23  "tags.availability_zone_id": "us-east-1a",
24  "localTrack[0].packetsLost": 0,
25  "tags.imageId": "xxxx",
26  "remoteTrack[1].intervalPacketsReceived": 978,
27  "tags.spotInstance": "false",
28  "tags.hyperion_habitat": "dev",
29  "localCandidateChanged": "false",
30  "tags.env": "dev",
31  "tags.availabilityZone": "us-east-1a",
32  "remoteTrack[1].intervalPacketsLost": 0,
33  "localTrack[1].packetsLost": 0,
34  "localTrack[1].jitter": 0.0006599999999999999,
35  "orgId": "xxxx",
36  "appId": "xxxxxx",
37  "localTrack[0].kind": "audio",
38  "tags.hyperion_family": "core",
39  "requestsSent": 107,
40  "remoteTrack[0].intervalPacketsLost": 0,
41  "tags.role": "public-api-v2",
42  "host": "public-api-v2-10-25-195-54",
43  "tags.pureCloudPrdDomain": "xxxxxx",
44  "remoteTrack[1].jitter": 0.007,
45  "localTrack[0].totalAudioEnergy": 0.05741095183112865,
46  "remoteTrack[1].bitrate": 1702,
47  "tags.service_name": "public-api-v2",
48 }

```

```

1 {
2   "conversationId": "41fdd6f6-611e-4e60-b606-cc9edf4c26d6",
3   "entries": [
4     {
5       "eventTimestamp": 1745057651739,
6       "userId": "xxxxxxxx",
7       "conversationId": "41fdd6f6-611e-4e60-b606-cc9edf4c26d6",
8       "sessionId": "xxxxxxxx",
9       "sessionType": "collaborateVideo",
10      "eventType": "getStats",
11      "originAppId": "xxxxxxxx",
12      "originAppName": "volt",
13      "originAppVersion": "1.12.1",
14      "localTrack[1]": {
15        "retransmittedBytesSent": 0,
16        "packetsSent": 0,
17        "packetsLost": 0,
18        "codecs": "97 video/VP8 90000",
19        "packetLoss": 0,
20        "retransmittedPacketsSent": 0,
21        "kind": "video",
22        "bytes": 0
23      },
24      "remoteTrack[0]": {
25        "packetsSent": 0,
26        "packetsLost": 0,
27        "codecs": "97 video/VP8 90000",
28        "packetLoss": 0,
29        "retransmittedPacketsSent": 0,
30        "kind": "video",
31        "bytes": 0
32      },
33      "remoteTrack[1]": {
34        "packetsSent": 0,
35        "packetsLost": 0,
36        "codecs": "97 video/VP8 90000",
37        "packetLoss": 0,
38        "retransmittedPacketsSent": 0,
39        "kind": "video",
40        "bytes": 0
41      },
42      "localTrack[0]": {
43        "packetsSent": 0,
44        "packetsLost": 0,
45        "codecs": "97 video/VP8 90000",
46        "packetLoss": 0,
47        "retransmittedPacketsSent": 0,
48        "kind": "video",
49        "bytes": 0
50      },
51      "remoteTrack[1]": {
52        "packetsSent": 0,
53        "packetsLost": 0,
54        "codecs": "97 video/VP8 90000",
55        "packetLoss": 0,
56        "retransmittedPacketsSent": 0,
57        "kind": "video",
58        "bytes": 0
59      }
60     ]
61   }
62 }

```

After fetching the data from New Relic, it requires processing before being sent to the user. The raw data is received as a single, unorganized array of objects `[{...}, {...}, ...]`, which is shared among all users involved in the call. To prepare this data for visualization, the backend iterates over each object in this array to separate the data by individual user. The properties related to the media tracks, those starting with `localTrack` and `remoteTrack`, are grouped together. This grouping and restructuring simplifies frontend development by nesting properties from the same track under the track's name, allowing the frontend to easily iterate over properties like 'packetsLost' or 'jitter' and use them directly as Chart titles, eliminating the need to parse complex keys like `localTrack[0]_packetsLost`. These tracks represent the audio or video sent and received by each user through the WebRTC connection, and they contain the statistics that will be used for plotting.

The main function responsible for aggregating the data is `createTimelinesFromResultsNewRelic`. This function takes the raw results from New Relic and transforms them into a structured format, as shown in the following code snippet:

```

export function createTimelinesFromResultsNewRelic(results: NewRelicSearchResult[]):
NewRelicTimeline[] {
  // create the object to store data, where the key is the userId
  const newRelicTimelinesByUserId: { [userId: string]: NewRelicTimeline } = {};

  results.forEach(result => {
    const userId = result['_userId']; // extract the 'userId' for current 'result'
    const conversationId = result['conversationId']; // extract 'conversationId' for current
    'result'
    /* ... skipping error handling for brevity ... */

    // get the value for 'userId'. The first time this for any 'userId' it will evaluate to
    false
    let timelineObj = newRelicTimelinesByUserId[userId];
    // If 'timelineObj' evaluates to false, we create a new object for the 'userId' and save
    it in 'newRelicTimelinesByUserId'
    if (!timelineObj) {
      timelineObj = newRelicTimelinesByUserId[userId] = {
        conversationId: conversationId,
        entries: [],
        userId,
      };
    }

    const entries = timelineObj.entries; // get the array of entries for the 'userId'
    // push data from the current 'result' from the 'forEach' loop and extract the values we

```

```

care about and push them into the 'entries' array as a single object.
entries.push({
  _eventTimestamp: result._eventTimestamp,
  bytesReceived: result.bytesReceived,
  bytesSend: result.bytesSend,
  _userId: result._userId,
  conversationId: result.conversationId,
  requestsReceived: result.requestsReceived,
  responsesReceived: result.responsesReceived
  /* ... etc ... */
  ...aggregateTracks(result) // This groups localTracks and mediaTracks
});
});
return Object.values(newRelicTimelinesByUserId);
}

```

This function iterates over each unformatted log entry received from New Relic. If an object for the current `userId` does not yet exist in the `newRelicTimelinesByUserId` object, a new object is created to store the data for that user. Properties from the current log entry, including general call statistics and the aggregated track data (from the `aggregateTracks` function), are then pushed into the `entries` array for that user.

The `aggregateTracks` function is responsible for grouping the media track properties based on their name:

```

export function aggregateTracks(result: NewRelicSearchResult) {
  const tracks: aggregatedTracks = {}; // object to store the entries
  const regex = /((?:local|remote)track_\d+)(.+)/i; // i for case insensitive to match
  'track' and 'Track'

  // iterates over the 'entries' for the current 'result'.
  Object.entries(result).forEach(([key, value]) => {
    const match = key.match(regex);
    // check if the current 'key' is a 'track'
    if (match) {
      const trackKey = match[1]; // name of track
      const trackProperty = match[2]; // the property of the track
      // create object inside 'tracks' for the specific 'trackKey' i.e. 'localTrack_[0]_'
      if (!tracks[trackKey]) {
        tracks[trackKey] = {};
      }
      // nest inside the 'trackKey', inside the 'property' name, the value for that property.
      // under the 'property' (i.e. packetsLoss) under its 'trackKey' (i.e. localTrack_[0]_)
      set the value for it for the current 'entry'
      tracks[trackKey][trackProperty] = value;
    }
  });
  return tracks;
}

```

This function iterates through the properties of each raw log entry. It applies a Regular Expression (RegEx) to identify properties that correspond to media tracks (those starting with 'local' or 'remote' followed by 'track'). For each matching property, it extracts the track name (e.g., `localTrack_[0]_`) and the property name (e.g., `packetLoss`). It then creates a nested structure within the object `tracks`, grouping properties that share the same media track name. The `tracks` object is returned, containing the aggregated data for each media track, each with a single log entry. This is what the aggregated track data looks like for a single log entry after this process:

```

16 |         "localTrack_[1]_": {
17 |             "retransmittedBytesSent": 0,
18 |             "packetsSent": 0,
19 |             "packetsLost": 0,
20 |             "codec": "97 video/VP8 90000",
21 |             "packetLoss": 0,
22 |             "retransmittedPacketsSent": 0,
23 |             "kind": "video",
24 |             "bytes": 0
25 |         },
26 |         "remoteTrack_[0]_": {
27 |             "packetLoss": 0,
28 |             "packetsReceived": 0,
29 |             "kind": "audio",
30 |             "jitter": 0,
31 |             "packetsLost": 0,
32 |             "bytes": 0
33 |         },
34 |         "remoteTrack_[1]_": {
35 |             "packetsLost": 0,
36 |             "jitter": 0,
37 |             "bytes": 0,
38 |             "packetLoss": 0,
39 |             "packetsReceived": 0,
40 |             "kind": "video"
41 |         },
42 |         "localTrack_[0]_": {
43 |             "packetLoss": 0,
44 |             "kind": "audio",
45 |             "totalAudioEnergy": 4.683907106381113e-8,
46 |             "audioLevel": 0.0003967406231879635,
47 |             "packetsSent": 0,
48 |             "packetsLost": 0,
49 |             "retransmittedPacketsSent": 0,
50 |             "bytes": 0,
51 |             "retransmittedBytesSent": 0,
52 |             "codec": "96 audio/opus 48000"
53 |         }

```

This structured format, with properties nested under their respective tracks, significantly simplifies processing on the frontend. It is important to note that a typical call would generate hundreds or thousands of these aggregated log entries.

Once the data has been aggregated and processed, it is embedded directly into the window object of the HTML document that will be returned to the client.

```

html = `
<html lang="en">
<head>
  <title>Results</title>
  <script>
    %% Embed the data into the window object to be used to build the SPA %%
    window.sumoData = ${JSON.stringify(sumoLogicTimelines)};
    window.newRelicData = ${JSON.stringify(newRelicTimelines)};
  </script>
  <link rel="stylesheet" href="${reactCDNBase}/index.css" />
</head>
<body>
  <div id="root"></div>
  <script src="${reactCDNBase}/index.js"></script>
</body>
</html>
`;

```

Embedding the data in the window object allows other scripts running on the same page, such as `index.js`, which is the JavaScript code responsible for building the Single Page Application (SPA), to access the data directly. The HTML document containing the embedded data is then returned as a string to the caller (Supportability), ready to be rendered and iframed within Supportability's user interface.

Meaning of Statistics

I would like to describe what some of the most important statistics mean.

- `packetsReceived` : The total number of packets received from connection. Includes retransmissions. [14]
- `packetsLost` : The total number of packets lost from connection. [14] Occurs when packets do not reach destination. Critical indicator of network errors.
- `packetLoss` : Percentage of `packetsLost` with respect to `packetsSent` . [14]
- `jitter` : The variation in the delay of received packets. Caused by network congestion and route changes. [19]
- `roundTripTime` : The time taken for data to travel to another member of the call and back. [16]
- `bitrate` : The rate at which data is being transmitted or received. Usually in kbit/s. [18]
- `bytesSent` and `bytesReceived` : Total number of bytes transferred. [17]
- `_eventTimestamp` : Specific moment in time when a snapshot of stats was taken. [15]

Testing

To ensure the correct functionality of this projects backend testing was done using Jest. Jest is a JavaScript testing framework that focuses on being simple. It allows mocking function calls to properly isolate functionality [13]

Here is a test that simulates building the request for New Relic. The `expect` functions help us assert values.

```
it('should build a request', async function () {
  const runConfig = { /* arguments passed to lambda */ }
  const searchOptions = await getSearchOptions(runConfig);
  expect(Object.keys(searchOptions)).toEqual(['requestOptions', 'url']);
  expect(/* ... */)
  // ...
});
```

The following test uses dummy data to test the grouping of data by `userId` and aggregation by media track. We create three logs, each one with a different `userId`. The function `createNewRelicSearchResult` adds media tracks to all logs.

```
it('createTimelineFromResults for New Relic. Multiple user Ids', async function () {
  const results = [
    // helper function to add localTrack and remoteTrack dummy data
    createNewRelicSearchResult({ ['_userId']: 'userId1', /* ... */ }),
    createNewRelicSearchResult({ ['_userId']: 'userId2', /* ... */ }),
    createNewRelicSearchResult({ ['_userId']: 'userId3', /* ... */ })
  ];

  // function that extracts relevant data from 'results' and groups by media track
  const timelines = createTimelinesFromResultsNewRelic(results);

  expect(timelines[1]).toEqual({
    "conversationId": "123",
    "entries": [{ /* ... */ }],
    "userId": "userId2"
  });
  /* more assertions */
});
```


In the code above, we pass the dummy results into the `createTimelinesFromResultsNewRelic` function which will iterate over every log and group them by `userId`. Finally, we run some assertions to ensure that the results are as expected.

The test below is an alternate version of the one above. We pass multiple logs but this time they share the same `userId`, meaning that they should be grouped all under one object:

```
it('createTimelineFromResults for New Relic. Same user Id', async function () {
  const results = [
    // same 'userId'
    createNewRelicSearchResult({ ['_userId']: 'userId1', /* ... */ }),
    createNewRelicSearchResult({ ['_userId']: 'userId1', /* ... */ }),
    createNewRelicSearchResult({ ['_userId']: 'userId1', /* ... */ }),
  ]
  const timelines = createTimelinesFromResultsNewRelic(results);
  expect(timelines.length).toBe(1); // timelines is length 1 because there was only 1
  distinct 'userId'
  /* more assertions */
});
```

On the snippet above, the result of `createTimelinesFromResultsNewRelic` is one object containing three entries, all containing the same `userId`.

Finally, the test below tests the data aggregation:

```
it('aggregates tracks in an object', async function () {
  const result: NewRelicSearchResult = {
    _userId: '123',
    conversationId: 'XXX',
    ['localTrack_[0]_jitter']: '123',
    ['localTrack_[0]_bitrate']: '55555',
    ['localTrack_[1]_jitter']: '456',
    ['localTrack_[1]_bitrate']: '55555',
    ['localTrack_[3]_jitter']: '456',
    ['localTrack_[3]_bitrate']: '55555',
    ['localTrack_[3]_codec']: '55555',
    ['localTrack_[3]_kind']: '55555',
    ['remoteTrack_[0]_jitter']: '55555',
  }

  // function that uses RegEx to match track names starting with localTrack or remoteTrack
  and groups them
  const res = aggregateTracks(result);

  expect(Object.keys(res)).toEqual(['localTrack_[0]_', /* ... */]);
  expect(res['localTrack_[0]_']).toEqual({ 'jitter': '123', 'bitrate': '55555' });
  /* more assertions */
});
```

On the snippet above we pass a single result into the `aggregateTracks` function. We then assert the response of this function. As it can be seen, the response has four keys, one for each media track, and the properties inside of each track is the summarised name, i.e. 'jitter' instead of `localTrack_[0]_jitter`.

Frontend development

The frontend of the WebRTC Call Debugger was developed using React, a JavaScript library for building interactive user interfaces using reusable components. React was chosen due to the team's familiarity with this

technology and its existing use across multiple repositories within Genesys, facilitating consistent development practices.

During the development phase, the frontend code is executed in a Node.js environment to enable rapid iterations and testing. To facilitate local testing, a dummy HTML document that contains a script which adds example call data to the window object is used. This approach allows frontend development and testing to proceed independently of the backend Lambda function, streamlining the development workflow.

When the frontend code is ready for deployment, ParcelJs is used as a build tool. ParcelJs compiles, minimizes, and packages the React project into static JavaScript and CSS files. These static assets are then uploaded to AWS's Content Delivery Network (CDN). Serving the frontend script from a CDN offers significant advantages [5], including geographic distribution of assets closer to end-users for quicker loading times, and caching to improve performance and reduce the Lambda's load. This deployment model eliminates the need for a dedicated Node.js server running the frontend in the background and avoids the constraints of serving the assets directly from the Lambda function. This approach allows us to re deploy and change the JavaScript asset in the CDN without having to re deploy or modify the AWS Lambda.

To align the dashboard's look and feel with existing Genesys products, components from Genesys's own UI library were utilized where possible, such as accordions for managing content visibility. For data visualization, Material UI (MUI) X charts was selected. This open-source React component library follows Google's Material Design principles [6]. While Genesys has its own line chart component, it is still in beta. MUI's widespread adoption and comprehensive documentation also played a role in its selection, significantly assisting development, particularly as I had limited prior experience with React and no prior experience plotting data in a web environment. The primary component used for visualization is MUI's `LineChart`.

An important design consideration for the frontend was to ensure flexibility regarding the data received from the Lambda function. The frontend is designed to be agnostic to the specific set of statistics passed through the window object. This means that if the Lambda function is modified in the future to gather more or fewer statistics from a call, the frontend can adapt without requiring code changes, provided the statistics are numeric. The frontend is capable of dynamically plotting data for any number of users, tracks, and their associated numeric statistics.

Preparing data to be plotted

The data had to be processed and converted into arrays with every element in the array corresponding to a timestamp. The logs that were sent from the Lambda were not in this format. Each log from the lambda was an individual log. It only has its own data for that point in time. To be able to plot data I had to reduce all of these logs based on the `userId` and the statistic name into an array. Then, I would pass this object containing the arrays into the Charts.

Grouping the data was a tough task. I had to write multiple helper function to extract the plottable data into groups from the logs. To extract and aggregate the data that was injected into the window object by the AWS Lambda parser, the function `prepareDataForGraphing_NewRelic` is executed:

```
export function prepareDataForGraphing_NewRelic() {
  const windowData = window.newRelicData;

  return windowData.map((user) => { // 'windowData' is an array of objects
    // sort the entries since they are not in order when we receive them
    const timestampSortedEntries = sortNewRelicEntriesByEventTimeStamp(user["entries"]);
    return {
      "userId": user["userId"],
      "conversationId": user["conversationId"],
      "entries": flattenUserEntry(timestampSortedEntries) // the data
    };
  });
}
```

```
});
}
```

The function above loops over all the users that were part of the call. Usually the amount of users would be two. If a customer is redirected between different agents in the same call, all the agents that were part of the call would appear also appear on the `windowData` object. The data that comes from the AWS Lambda is not in order, so before doing any kind of work on it, I sort it calling the `sortNewRelicEntriesByEventTimeStamp` function and pass the entries for the current user. This function will use the JavaScript `sort` function to sort the entries from newest to oldest based on `_eventTimeStamp` which is a UNIX timestamp (i.e '1745749017'):

```
function sortNewRelicEntriesByEventTimeStamp(entries) {
  return entries.sort((a, b) => {
    const aTime = new Date(a["_eventTimeStamp"]);
    const bTime = new Date(b["_eventTimeStamp"]);
    return aTime.getTime() - bTime.getTime();
  });
}
```

It is crucial to sort the data based on time or the following could happen when Charting the data. On the left is the unsorted Chart and on the right the sorted Chart.



It can be seen how the line of the first Chart seems to be crossing itself over and over. This is caused by using unsorted data. The Chart processes the data entry but since the current entry could have happened before the previous one it has to go back, not displaying data properly. On the other hand, the second graph is correct.

After sorting by timestamp, we can proceed with the actual data aggregation.

Data aggregation and grouping

At the start of this project I was a beginner in React, MUI and JavaScript. The following functionality took me many iterations to get working.

On the previous step we left at sorting the entries by time. The next step is to call `flattenUserEntry` for every user in the `windowData` object. `flattenUserEntry` is in charge of building the new object where the statistics are grouped by media track name.

```
function flattenUserEntry(userEntries: { string: object }[]) {
  const global = {}; // object to store formatted data for 'userEntries'
  global['General call data'] = {};
  userEntries.forEach((logEntries) => {
    // Object.entries returns the keys and values of every property in 'logEntries'
    const nrEntryProperties = Object.entries(logEntries);
    const timestamp = logEntries["_eventTimestamp"];

    addGeneralCallData(global, nrEntryProperties);

    // 'logEntries' that are media track entries
    const trackProperties = nrEntryProperties.filter((entry) =>
entry[0].toLowerCase().includes('Track'));
    // using 'trackProperties' get the data and aggregate it
    findTracksAndAddTheirProperties(trackProperties, global, timestamp);
  });
  return global;
}
```

The function above will iterate over every log entry for the current user. It uses the JavaScript function `Object.entries` to get the keys and values for the current entry. Before grouping by track, we call the `addGeneralCallData` function that will group generic statistics from the call. This is what this function looks like:

```
function addGeneralCallData(global, nrEntryProperties) {
  // properties whose value is a number
  const numericalNonTrackStats = nrEntryProperties.filter((entry) => typeof entry[1] ===
'number');
  // Check if there is more than 1 in the 'numericalNonTracksStats' array since some times we
wont have any data but only the timestamp. But we can't do anything with just the timestamp
so we just skip the code then.
  if (numericalNonTrackStats.length < 2) return;
  numericalNonTrackStats.forEach(([key, value]) => {
    if (global['General call data'][key] === undefined) {
      global['General call data'][key] = [];
    }
    // push the 'value' into the array of the 'key' inside 'General call data'.
    global['General call data'][key].push(value);
  });
}
```

The function above applies a filter to get an array of properties whose value is a number, such as `bytesReceived`, `requestsReceived` and `responsesReceived` in the image below:

```

{
  "_eventTimestamp": 1745059587098,
  "bytesReceived": 432981504,
  "_userId": "85364abe-6b9a-4a30-87d8-9d518be65ebc",
  "conversationId": "41fdd6f6-611e-4e60-b606-cc9edf4c26d6",
  "requestsReceived": 391,
  "responsesReceived": 732,
  "localTrack_[1]_": { ...
},
  "remoteTrack_[0]_": { ...
},
  "remoteTrack_[1]_": { ...
},
  "localTrack_[0]_": { ...
}
},

```

It then iterates over these properties and creates an array for each of these properties. It then pushes the current data into their respective array. These arrays are saved inside of an object called `General call data`. If we go back to the caller function `flattenUserEntry`, the next step is to aggregate the actual data from the tracks. The following code in `flattenUserEntry` filters the entries for the ones that include the substring 'Track'. It passes these `trackProperties` to a function called `findTracksAndAddTheirProperties`.

```

const trackProperties = nrEntryProperties.filter((entry) =>
entry[0].toLowerCase().includes('Track'));
findTracksAndAddTheirProperties(trackProperties, global, timestamp);

```

The function `findTracksAndAddTheirProperties` is the following:

```

function findTracksAndAddTheirProperties(newRelicEntryProperties: [string, object][], global:
{}, timeStampOfEntry) {
  newRelicEntryProperties.forEach((logEntriesKV) => {
    // build the 'trackName' to be displayed in the UI later
    const trackName = `Track: ${logEntriesKV[0]} Kind ${logEntriesKV[1]['kind']}`;

    // create an object for the specific track i.e. localhost_[1]_
    if (global[trackName] === undefined) {
      global[trackName] = {};
    }

    // each track has a copy of the timestamps
    if (global[trackName]["_eventTimestamp"] === undefined) {
      global[trackName]["_eventTimestamp"] = [];
    }
    global[trackName]["_eventTimestamp"].push(timeStampOfEntry);
    // we use 'numberOfEntriesWeShouldHaveCurrently' to handle missing data
    const numberOfEntriesWeShouldHaveCurrently = global[trackName]["_eventTimestamp"].length
    // 'entryTrackEntries' holds key-value pair properties for the current track.
    const entryTrackEntries = Object.entries(logEntriesKV[1]);
    addPropertiesFromTrack(entryTrackEntries, global, trackName,
numberOfEntriesWeShouldHaveCurrently);
  });
}

```

The function above will iterate over the media track log entries. It builds the complete name of the track used for displaying it. If the current media track has no entry in the global object a new empty object is made. The same is done with `_eventTimestamp`. Each track has a copy of the time stamp. Then, we push the current time stamp to the corresponding array. Now that we have an empty object set up for the track, we call the `addPropertiesFromTrack` function. Finally, this is the function that adds the data into the object:

```
function addPropertiesFromTrack(entryTrackEntries, global, trackName, timestampEntriesAmount)
{
  entryTrackEntries.forEach((entryTrackEntry) => {
    // 'trackProperty' is the name of the property/statistic
    const trackProperty = entryTrackEntry[0];
    // create an array for the 'trackProperty' if it does not exist
    if (global[trackName][trackProperty] === undefined) {
      global[trackName][trackProperty] = [];
    }
    // this if statement will add an extra '0' if we are missing data for the current
    'trackProperty'
    if ((global[trackName][trackProperty].length < timestampEntriesAmount - 1) &&
      !isNaN(entryTrackEntry[1])) {
      global[trackName][trackProperty].push(0);
    }
    global[trackName][trackProperty].push(entryTrackEntry[1]);
  });
}
```

The code above will iterate over the entries for the specific track, such as the ones below:

```
{
  "packetLoss": 0,
  "kind": "audio",
  "totalAudioEnergy": 0.000011309237359902249,
  "audioLevel": 0,
  "bitrate": 12,
  "packetsSent": 24250,
  "roundTripTime": 0.12074299999999999,
  "packetsLost": 0,
  "intervalPacketsSent": 250,
  "bytes": 776470,
  "jitter": 0.006999999999999999,
  "codec": "96 audio/opus 48000"
}
```

If the `global[trackName][trackProperty]` has not been initialised yet, it creates an empty array. Then, it runs a check to ensure that if data is missing it gets filled out with a 0, and finally it pushes the value of the property into the array for the property key.

This is what a single log looked like:

```
"_eventTimestamp": 1745332347079,
"_userId": "85364abe-6b9a-4a30-87d8-9d518be65ebc",
"localTrack_[1]_": {
  "retransmittedBytesSent": 0,
  "packetsSent": 0,
  "packetsLost": 0,
  "codec": "97 video/VP8 90000",
  "packetLoss": 0,
```

```

    "retransmittedPacketsSent": 0,
    "kind": "video",
    "bytes": 0
  },
  "remoteTrack_[0]_": {
    "packetLoss": 0,
    "packetsReceived": 1,
    "codec": "96 audio/opus 48000",
    "kind": "audio",
    "jitter": 0,
    "packetsLost": 0,
    "bytes": 27
  },
  "remoteTrack_[1]_": {...},
  "localTrack_[0]_": {...}

```

On the other hand, after all of this process, this is what the data looks like. Way more friendly and aggregated into arrays ready to be plotted! As you can see everything has been grouped. In a real call there would be hundreds of items in each array. On this case, to keep it clean, I made a 15 second call.

```

"userId": "85364abe-6b9a-4a30-87d8-9d518be65ebc",
"conversationId": "7f1bdf42-c5cb-437b-ac76-6d44e493b7d4",
"entries": {
  "General call data": {...},
  "Track: localTrack_[1]_ Kind video": {
    "_eventTimestamp": [1745332347079,1745332352274,1745332357075],
    "packetsSent": [0,653,1560],
    "bytes": [0,677938,1657614],
    "intervalPacketsSent": [0,653,907],
    "jitter": [0,0.001077,0.0009],
    "bitrate": [0,1043,1631],
    "roundTripTime": [0,0.12013199999999999,0.126526]
  },
  "Track: remoteTrack_[0]_ Kind audio": {...},
  "Track: remoteTrack_[1]_ Kind video": {...},
  "Track: localTrack_[0]_ Kind audio": {...}
}

```

We also receive some non-plottable but still useful data from the lambda, so we group this using a different function called `prepareExtraData`:

```

export function prepareExtraData() {
  const windowData = window['newRelicData'];

  const usersArr = windowData.map((user) => {
    return {
      userId: user.userId,
      data:
        // extra data is anything that is not 'getStats'
        [ ...user.entries.filter((log) => log._eventType !== "getStats") ]
    }
  });

  // convert from array to obj to be able to select data by 'userId'
  return usersArr.reduce((accumulator, currUser) => {
    accumulator[currUser.userId] = currUser.data;
    return accumulator;
  });
}

```

```
    }, {}));
  }
}
```

The function above will grab the same data from the window object. Then it will iterate over the users from the object and it will create a new object using the information from the current `user` object. We set the `data` array to hold the logs whose `_eventType` is not `getStats`. After this, to make it easier to interact and retrieve data from `usersArr`, I converted the array of objects into just an object whose keys are the user ids.

Building the UI

Once the data has been aggregated into the proper format, we build the components. This following snippet maps over `nrData` which contains the users of the call:

```
<div>
  <h1>New Relic Charts</h1>
  <div>
    {nrData.length === 0 ? (noResults) : (
      nrData.map((userData, idx) => {
        return (
          <User
            userData={userData}
            extraData={extraData}
            key={userData.userId}
            expandAllNr={expandAllNr}
            initialChartDelayBetweenUsers={idx * 1000}
          />
        );
      })
    )}
  </div>
</div>
```

We pass `userData` and `extraData` to the `User` component as props. This is what the return statement of the `User` component looks like:

```
return (
  <div>
    %% Accordion to close and open User's content %%
    <GuxAccordion>
      <GuxAccordionSection
        open={isAccordionOpen}
        onGuxopened={() => setIsAccordionOpen(true)}
        onGuxclosed={() => setIsAccordionOpen(false)}
      >
        %% Title of the Accordion %%
        <h2 slot={"header"}>User {userData.userId}</h2>
        %% The 'content', where the 'Tracks' and 'General call data' will go and the
        'extraInfo' %%
        <div slot="content">
          {buildUserTracks(keysFromUserEntries)}
          <ExtraInfoAccordion userExtraInfoLogs={extraData[userData.userId]} expandAllNr=
            {expandAllNr}/>
        </div>
      </GuxAccordionSection>
    </GuxAccordion>
  </div>
)
```



```

    </div>
  );

```

The code above returns an accordion which allows to hide or show content. We have to keep track of the Accordion's state because if a child or sibling component changes, the accordion would close since when it is redrawn it has no track of its current state. To keep track we use `setIsAccordionOpen` which is a React `useState` Hook. In the `content` section of the accordion we call the `buildUserTracks` function. This function looks like the following:

```

// 'keysFromUserEntries' are the names for displaying the Tracks. i.e 'Track: localTrack_[0]_
Kind audio'
const buildUserTracks = (keysFromUserEntries) => {
  return (
    // we call 'sort' to have the same order between different 'User' accordions.
    keysFromUserEntries.sort().map((trackName, idx) => {
      // we use the 'trackName' to get it's corresponding data from the 'userEntries' object
      const trackEntries = userEntries[trackName];
      return (
        <Track
          userAccordionOpen={isAccordionOpen}
          entriesFromTrack={trackEntries}
          key={`_${userEntry.userId}_${trackName}`}
          expandAllNr={expandAllNr}
          trackName={trackName}
          userId={userEntry.userId}
          extraDelayForChart={idx * 200 + initialChartDelayBetweenUsers}
        />
      );
    })
  );
}

```

The function above is in charge of iterating over the media tracks that the user has. It creates a `Track` component for every track and passes required data as props. Before mapping the data, we call the `sort` function to ensure some order between different users; local tracks first, remote second.

This is what the return statement of the `Track` component looks like:

```

return (
  <GuxAccordion>
    %% 'event.stopPropagation' is needed to prevent closing parent Accordion %%
    <GuxAccordionSection
      open={isAccordionOpen}
      onGuxopened={(event) => {
        event.stopPropagation();
        setIsAccordionOpen(true);
      }}
      onGuxclosed={(event) => {
        actionType.current = 'all';
        event.stopPropagation();
        setIsAccordionOpen(false);
      }}>
      <h2 slot="header">{trackName}</h2>
      <div slot="content" style={{ display: "flex", flexDirection: "row" }}>
        <div style={{ display: "flex", flexDirection: "column", flexShrink: 0 }}>
          %% 'buildCheckboxes' will create the Checkboxes for selecting which Charts to see
          {buildCheckboxes(sortedEntriesKeysFromTrack)}

```

```

</div>
<div style={{ display: "flex", flexDirection: "row", overflowX: "scroll",
paddingBottom: "20px" }}>
  %% iteration over 'charts' array, only displaying the visible ones %%
  {charts.map(chart => (chart.visible &&
    <ChartWrapper
      hoveredDataValue={hoveredDataValue}
      stickyValue={stickyValue}
      onHover={setHoveredDataValue}
      onSticky={setStickyValue}
      trackPropertyName={chart.chart.trackPropertyName}
      formattedTime={chart.chart.formattedTime}
      data={chart.chart.data}
      isAccordionOpen={isAccordionOpen}
      key=
{container-`${chart.chart.trackPropertyName}:${chart.chart.formattedTime}:${userId}}
      />
    )})}
</div>
</GuxAccordionSection>
</GuxAccordion>
);

```

The Track component is the longest component on this project. It returns another accordion for every Track, to be able to hide individual Tracks if the user desires to do this. We also use `useState` to hold the state of the accordion. In addition, the `stopPropagation` function had to be used since the click event bubbled up to the parent Accordion closing it when clicking on the child Accordion. To prevent this, `stopPropagation` was used. The content of the accordion is the Charts. To generate the Charts we use the `map` function over the Charts array which is held in a `useState` React Hook. When iterating we check if the current Chart should be visible, if it is we continue and create the corresponding component and pass the required data as props. We use `useEffect` to initialise the state of the Charts, when the Track component loads. This is what this `useEffect` looks like:

```

useEffect(() => {
  const allCharts = sortedEntriesKeysFromTrack.reduce((acc: any, propertyName: string) => {
    // using the current 'propertyName' get its corresponding data from the
    'entriesFromTrack' prop object
    // 'entriesFromTrack' only contains data for the current track
    const aggregatedDataArray = entriesFromTrack[propertyName];
    if (!isNumericalData(aggregatedDataArray, propertyName)) {
      return acc; // we don't want to plot properties that are not numbers. We can't
    }
    // convert the timestamps into JavaScript Date objects
    const unformattedTime = entriesFromTrack["_eventTimestamp"];
    const formattedTime = formatTime(unformattedTime);
    // 'newChart' is an object that holds the data to create the Chart and the Chart's
    visibility status.
    const newChart = {
      visible: false,
      propertyName,
      chart: {
        trackPropertyName: propertyName,
        formattedTime: formattedTime,
        data: aggregatedDataArray,
      }
    }
    return [...acc, newChart];
  }, []); // everything gets saved into this empty array

```

```
setCharts(allCharts);
}, []); // runs on mount, once
```

The `useEffect` above triggers the JavaScript `reduce` function which iterates over the names of each property for the current track. It checks if the type of data for that property is numerical. If it is numerical, it continues and formats the unix-timestamp by converting it to a JavaScript Date Object. Then, it creates an object with the required data to create a Chart and a property indicating if the Chart should be visible or not.

The `ChartWrapper` is a Component that I made to add a hover effect to the Charts. I will talk more about this on the Special Hover Effect Heading. But for now the `ChartWrapper` returns this:

```
<Chart
  trackPropertyName={trackPropertyName} // the title for the 'Chart'
  formattedTime={formattedTime} // the timestamps. Goes on the x axis
  data={data} // the data to plot. Goes on the y axis
/>
```

The `Chart` component which is another wrapper but this time around MUI's `LineChart` component. This is the return statement of the `Chart` component:

```
return (
  <div>
    <LineChart // MUI's Component
      xAxis={[{
        label: "Time",
        data: formattedTime, // data for x axis
        scaleType: 'time',
        valueFormatter: (date) => date.toLocaleTimeString() // tells MUI how to format the
date
      }]}
      yAxis={[{
        valueFormatter: formatLargeNumber // formats numbers on the y axis. Useful when
numbers are too large or too small
      }]}
      series={[
        {
          label: trackPropertyName, // title
          data: data, // the data to plot!
          showMark: false
        },
      ]}
      width={400}
      height={400}
      grid={{ vertical: true, horizontal: true }}
      skipAnimation={true}
    />
  </div>
);
```

Thanks to MUI we can pass the data and format it as time using the `valueFormatter` property on the x-axis. The values that are plotted are sometimes too large or too small. MUI allows us to pass our own function that formats the numbers. On this case I wrote a simple function to add metric prefixes such as 'k' for thousand and 'M' for million. The data is passed to the series property as `data`. To improve performance the `skipAnimation` property is set to true.

Checkboxes

To decide which Charts to render we use Checkboxes. The user can interact with these Checkboxes. The website uses the Checkbox component provided by Genesys. I added an `onChange` listener to these Checkboxes to detect when they are pressed. I then set the state for the Checkboxes accordingly.

A `useEffect` is listening for any change of the Checkboxes state. One of the things that the callback of the `useEffect` does is set the visibility of the Charts from the Charts `useState` to match the Checkboxes state. The following snippet represents this. It will iterate over the Charts and it will set the `visible` property of each Chart to match the Checkboxes state.

```
setCharts(prev => // 'prev' is the current Charts
  prev.map(chart => ({
    ...chart, // we expand i.e 'copy' the Chart contents
    // but we change the visibility state to match the one from the 'checkboxesState' that
    // has the same name
    visible: !!checkboxesState[chart.propertyName]
  })));
```

I added a Button under the Checkboxes to allow the user to select or deselect all Checkboxes for a specific Track with one click:

- ☒ audioLevel
- ☒ bitrate
- ☒ bytes
- ☒ intervalPacketLoss
- ☒ intervalPacketsLost
- ☒ intervalPacketsSent
- ☒ jitter
- ☒ packetLoss
- ☒ packetsLost
- ☒ packetsSent
- ☒ retransmittedBytesSent
- ☒ retransmittedPacketsSent
- ☒ roundTripTime
- ☒ totalAudioEnergy

Reset

I thought adding this functionality was going to be simple but it was not as straight forward as I thought. The following is the `onClick` handler for this Button.

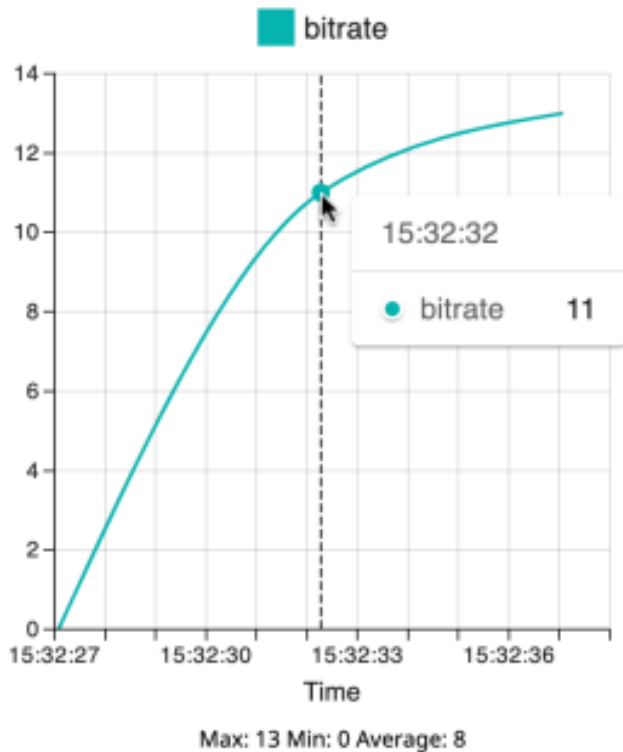
```
onClick={() => {
  // If there are no values set to true, we set them all to true
  const noValueIsTrue = Object.values(checkboxesState).filter((val) => val).length === 0;
  if (noValueIsTrue) {
    setCheckboxesState(() =>
      // Iterate over the keys and create an object where we set the keys to be true
      Object.keys(checkboxesState).reduce((acc, key) => ({
        ...acc,
        [key]: true
      }), {}));
  } else { // At least one value was true, so we set all to false.
    setCheckboxesState(() =>
      // Iterate over the keys but this time we set them to false
      Object.keys(checkboxesState).reduce((acc, key) => ({
        ...acc,
        [key]: false
      }), {}));
  }
}
```

```
}
}}
```

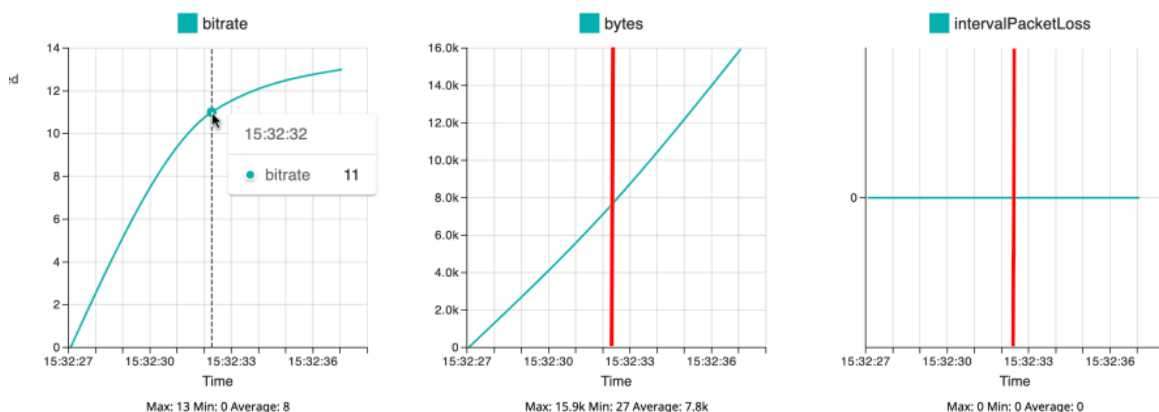
Thanks to this code, when this 'Reset' button is clicked, it will cause all Charts to hide or show instead of forcing the user to click the Checkboxes one by one.

Special Hover Effect

The `LineChart` component provided by MUI has a hover effect. It shows a vertical dotted line at the same x position where the user's mouse is and a box displaying the exact value at that hover location:



The functionality that MUI provides is great. I came up with the idea of sharing this 'hover state' across the Charts of a single Track. So when the user hovered over a Chart triggering the hover effect, the same effect would appear in the rest of the Charts that are part of the same Track. On the image below the user is hovering over the first chart and the vertical line also appears on the other Charts. This was an example sketch that I made:



I came across two major problems that limited my ability to develop this functionality. The first one was that the MUI `LineChart` does not provide a `onHover` Hook that I could use to trigger code execution. There was no way of knowing where the mouse was. The `LineChart` component does have this functionality internally to allow its own hover effect to work, but it does not let me to interact with it. It is a private implementation. The `LineChart` component provides an `onAxisClick` hook but this was not enough for what I wanted to achieve. The second problem that I came up against was that MUI does not allow me to trigger the hover effect with code,

programmatically. I can not tell a Chart to simulate that a mouse is hovering at x y position. Even if I were to use the `onAxisClick` hook I had no way of interacting with the rest of the Charts without having to reinvent the wheel, so that's what I did.

I decided to 'hack' my way through this. My idea was to have a vertical div that would move over the Charts, as an overlay. This div would only appear when the mouse was hovering any of the Charts. I had to leverage Amazon Q which is AWS's AI chatbot to give me ideas and examples on how to solve this since this was not a trivial problem. This was my solution. I created a component called `ChartWrapper` that returns the original Chart and a div which works as the overlay. This is the most important part of the return:

```
return (
  // The Hooks are set on the div that holds the components
  <div ref={containerRef} onMouseMove={onMouseMoveHandler} onMouseLeave=
{onMouseLeaveHandler} onClick={onClickHandler}
    style={{...}}>
    <div>
      {memoizedChart} // The Chart
      {relativeOverlayX !== null && ( // The Overlay
        <div style={{
          position: "absolute", pointerEvents: "none",
          // The Charts for a track are beside each other so the left property does
need to change
          left: clipArea.current.x! + relativeOverlayX,
          // The Charts for a Track are always at the same height
          top: clipArea.current.y,
          height: clipArea.current.height, // The height is the same for all Charts
          width: "1px", backgroundColor: "black"
        }}
        />)}
    </div>
    // Shows Max Min and Average
    <span style={{...}}>Max: ... Min: ... Average: ...</span>
  </div>
)
```

Hover

When the `<div>` detects that a mouse is hovering over it, it calls the `onMouseMoveHandler` function and passes and event that contains the x and y position of the mouse. This function will get the size of the Chart that triggered the callback and figure out where in the screen the Chart is. It then checks if the position where the mouse is is directly over the line of the `LineChart` or if its just ove the title or axes. It calls the `onHover` function which sets the state in the parent component, which is shared among the other Charts.

```
const onMouseMoveHandler = (event: { clientX: number; clientY: number; }) => {
  if (!clipArea.current || !containerRef.current) return; // Check if the ChartWrapper has
mounted properly

  const containerRect = containerRef.current.getBoundingClientRect(); // Size of the parent
div that holds the Chart and the hover div effect
  const relX = event.clientX - containerRect.left; // Relative position of the mouse inside
the parent div. Find pos with respect to the div and not the whole screen
  const relY = event.clientY - containerRect.top;
  // This checks if the relative coordinates are at a certain threshold in the Chart, i.e.
between 50 and 250 in the X axis.
  if (plottableArea.current && positionIsInside(relX, relY)) {
    const xInsideClip = relX - clipArea.current.x!; // Get a new relative position, this time
with respect plottable area grid rather than the whole Chart. This is the area that will
trigger the hover effect.
```

```

const snappedData = snapToDataPoint(xInsideClip); // explained after this
onHover(snappedData); // sets state in parent
} else {
  onHover(null); // when mouse is still hovering over the parent div but not over the
specific area that we care about
}
}

```

The function `snapToDataPoint` helps us mimic the default functionality that MUI provides, which is clipping the vertical line to the closest data point, instead of the line moving smoothly across the x axis. Achieving this functionality was really challenging. First, we have to map the timestamps to an x position with respect to the current Chart. This value will be different for every Chart in the Track.

```

// Maps the formatted time to an x-axis value
const positions = formattedTime.map((value) => (
  mapValueToPositionInChart(value, clipArea.current.width)
));

```

To work out the x-axis position for each timestamp I wrote this function:

```

const mapValueToPositionInChart = (value, width) => { // value is the timestamp, width is the
Chart's width i.e. 200
  const min = formattedTime[0]; // the formattedTime is already ordered
  const max = formattedTime[formattedTime.length - 1];
  return ((value - min) / (max - min)) * width; // Calculates the relative position of the
timestamp value in terms of pixels of the Chart
};
/* Example of formula in action
min = 1745761111, max = 1745768888, value = 1745765555, width = 200
(1745765555 - 1745761111) / (1745768888 - 1745761111) * 200 => 114.28
The timestamp 1745765555 would be mapped to 114.28px
*/

```

So then at this point we know where each timestamp would go in the Chart. What is left is to find the timestamp that is closest to the current hover position of the user's mouse. To do that we just iterate over the array `positions` that we created on the previous two blocks and compare the values to find the one whose difference against `xInsideClip` is the smallest i.e. the closest.

```

let closest = positions[0];
let minDiff = Math.abs(xInsideClip - positions[0]);
positions.forEach(pos => {
  const diff = Math.abs(xInsideClip - pos);
  if (diff < minDiff) {
    minDiff = diff;
    closest = pos;
  }
});
return closest;

```

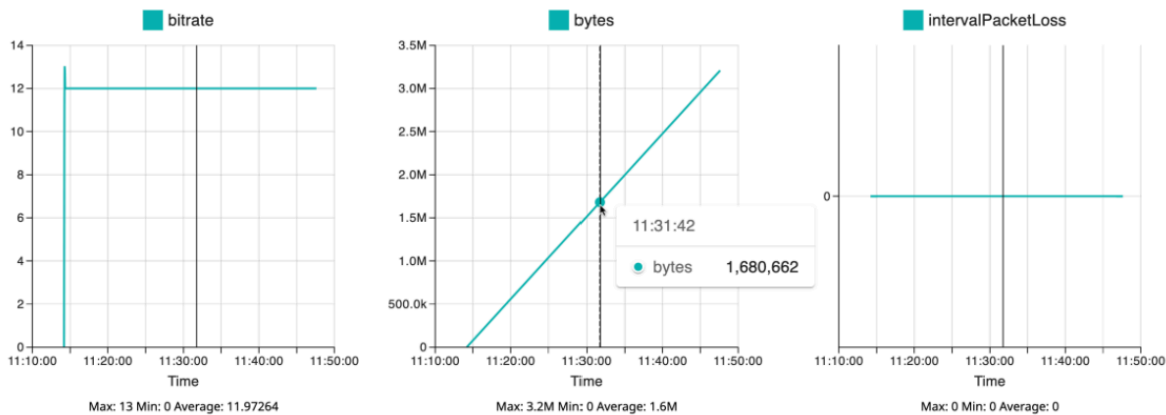
Then, when the user's mouse leaves the parent div completely, the `onMouseLeave` hook calls the `onMouseLeaveHandler` function which will set the hover state to null getting rid of the hover div effect.

```

onHover(null);

```

This is what the hover effect looks like. The user is hovering over the middle Chart and the same line appears on the two other Charts at the same relative position. The track the user's mouse position.



Click

I also added the option to click the Chart when hovering over it causing the hover div effect to essentially 'stick' in place allowing the user to move the mouse somewhere else while the vertical div line stays in place. This is useful for allowing the user to scroll between the Charts of the same Track or to scroll down or up to visualise other Tracks. To get rid of this 'sticky' state the user just needs to click any of the Charts of the track again. When the parent div is clicked it calls the `onClickHandler` function. This function is very similar to the `onMouseMoveHandler` function.

```
const onClickHandler = (event: { clientX: number; clientY: number; }) => {
  if (!clipArea.current || !containerRef.current) return; // Check if the ChartWrapper has
  mounted properly
  if (stickyValue !== null) { // If the stickyValue was already set, 'clear' it and return.
    onSticky(null);
    return;
  }

  const containerRect = containerRef.current.getBoundingClientRect(); // Size of the parent
  div that holds the Chart and the hover div effect
  const relX = event.clientX - containerRect.left; // Relative position inside the parent div
  const relY = event.clientY - containerRect.top;
  if (plottableArea.current && positionIsInside(relX, relY)) { // Same as with hover
    const xInsideClip = relX - clipArea.current.x!; // Same as with hover
    const snappedData = snapToDataPoint(xInsideClip); // Same as with hover
    onSticky(snappedData); // This time we set a different state on the parent
  }
}
```

ChartWrapper useEffect

Before being able to run all of these callbacks properly I have to get the position and dimensions of the Chart. Maybe I should've talked about this first, but I find it useful to understand the final product first. This is the `useEffect` function that runs when the component mounts to setup the `useRef` values.

```
useEffect(() => {
  if (!containerRef.current) return; // references the parent div. Does not run if the parent
  div did not load
  const timer = setTimeout(() => { // sets a timeout to allow DOM to fully load
    if (!containerRef.current) return;
  });
}, [containerRef]);
```



```

// The DOM querying is necessary to find the exact size of the Chart's Line.
const svgElement = containerRef.current.querySelector('.css-1evyvmv-MuiChartsSurface-root');
const clipPathFromSvg = svgElement.querySelector(":scope > clipPath:last-of-type");
if (!clipPathFromSvg) return;
const plottableAreaGrid = clipPathFromSvg.querySelector("rect");
if (!plottableAreaGrid) return;
const y = +plottableAreaGrid.getAttribute("y"); // using the + symbol converts a string to
a number
const height = +plottableAreaGrid.getAttribute("height");
const wholeX = +plottableAreaGrid.getAttribute("x");
const wholeWidth = +plottableAreaGrid.getAttribute("width");
// plottableArea is the 300x300 grid area where the Line can be drawn
// this is the threshold for enabling the Hover ever.
// If the position of the mouse is not inside this plottableArea we do nothing
plottableArea.current = {
  y, height,
  topY: y + height,
  x: wholeX,
  width: wholeWidth,
  topX: wholeX + wholeWidth
}

// here we get the specific size of the Line, not of the whole Grid or plottable area
// we require the specific size to be able to clip the mouse position adequately in
equally spaced steps
const xValueForHoverLine = containerRef.current.querySelector('g[clip-path]').getBBox();
if (!xValueForHoverLine) return;
const x = +xValueForHoverLine.x;
const width = +xValueForHoverLine.width;
// this is the height and most importantly the width of the line in the Chart.
// X represents the width of the Line exactly, allowing us to 'snap' to the closest point
clipArea.current = { x, y, width, height };
}, 25);
return () => clearTimeout(timer);
}, [isAccordionOpen]);

```

The code above may seem very unconventional, and it is. This is the 'hack' that I had to come up with. The problem with the `LineChart` component is that it does not tell us where exactly the line is rendered, or anything else. And the position is not constant. Depending on the amount of data points in the x-axis, MUI will leave a larger margin at the beginning and end of the Line, not allowing me to hardcode values. That's why I have to manually interact with the rendered HTML generated by the `LineChart` component to retrieve the dimensions and locations for the Line and for the `plottableArea` elements. This was not part of the documentation. This is something I came up with. I know this is not a great coding practice, but it allowed me to learn a lot about React Hooks and the JavaScript queue. It is not a good coding practice because we should only interact with the virtual DOM that React provides to prevent getting unexpected behaviour [7].

Performance Issues

After adding all of this functionality I realised about a critical problem. The performance was not good. When hovering over the Charts the line would not move smoothly.

UseMemo

I had to learn about a new React Hook called `useMemo` which allows you to specify when to re render a component instead of letting React re render the component on every draw. I used the `useMemo` hook on the Chart component to prevent it from re rendering every time I triggered the hover effect by hovering or clicking on the Chart. This is what that looks like:

```
const memoizedChart = useMemo(() => chart(), []); // 'memoising' the Chart component to
prevent re renders
const chart = () => {
  return (
    <Chart
      trackPropertyName={trackPropertyName}
      formattedTime={formattedTime}
      data={data}
    />
  );
}
```

This 'memoised' component is used like any other component. By implementing `useMemo`, moving the mouse over a Chart no longer lags the website.

Progressive Loading

There was still another problem present. Chart generation was slow, not because of the hover effect that I added, but because of the `LineChart` itself. MUI's `LineChart` component is heavy and we are graphing a lot of data in many Charts. For a normal call, each user would have, at least, four media tracks. Each track would have on average ten properties. That is 80 Charts. If the call is ten minutes long, there would be around 120 data point for every chat. That is over 9000 plotted data points. Because of this, generating the Charts i.e. looping using a conventional JavaScript map and building them sequentially made the website freeze for some seconds. I did not like this. I first tried using the Suspend API and Lazy loading but because generating the Chart components does not return a Promise, we can not asynchronously wait for it.

My solution to this problem was, once again, 'hacky'. I decided to use timeouts to progressively create the `ChartWrapper` components. By using timeouts I prevent blocking the main JavaScript thread, allowing JavaScript's main thread to catch up on other tasks.

Like I explained before under the 'Building the UI' header, when the Track component first loads, the callback for a `useEffect` runs to setup up the initial state for the different `useState` Hooks.

```
useEffect(() => {
  // loops over keys
  const allCharts = sortedEntriesKeysFromTrack.reduce((acc: any, propertyName: string) => {
    // already explained under Building the UI. The code here initialises the 'useState' of
    the Charts
  }, []);
  setCharts(allCharts);

  // iterates over the newly created Charts and grabs the propertyNames and initialises the
  buttons state by setting all buttons to be checked
  setCheckboxesState(() => {
    return allCharts.reduce((acc, chart) => {
      const propertyName = chart.propertyName;
      acc[propertyName] = true;
      return acc;
    }, {});
  });
}, []);
```

Setting up two distinct `useState` Hooks for the Charts and Checkboxes may seem excessive, but this is needed to be able to progressively load in the Charts only when they are visible. I could have just used a single `useState` for the Checkboxes, and as the return of the Track component I could just have iterated over the Checkboxes names and retrieved that data from the main data object and created the `ChartWrapper` with that directly. The problem with this approach is that when the page first loads, it tries to generate all of the Charts in one go,

delaying the initial page paint significantly, by many seconds, which decreases the perceived performance significantly. Specially since I should not affect the current functionality of the page.

By having two states, I can detach the current state which is represented by the Checkboxes from the state of the Charts that are currently displayed, to be able to progressively set the visible Charts when the Checkboxes update all together.

So after the setup of both `useState` hooks, another `useEffect` hook that listens for changes in the Checkboxes state triggers since we just initialised its state in the other `useEffect`. The callback from this Hook will first check if the Accordions that contain the Track are open. If they are not open, the Charts wont be visible, so we set the visibility to false for each Chart and make an early return. The next if block is in charge of setting the visibility when the user interacts. There are two ways of interacting with the Checkboxes, by either clicking each one individually or by clicking on the 'Reset' and 'Select All' button that is under the Checkboxes. If this button is pressed, the body of the if statement runs, causing the Charts to progressively load. If a Checkbox was pressed individually, the else if block runs, updating the visibility state directly.

```
useEffect(() => {
  // Charts are not visible on this state
  if (!isAccordionOpen || !userAccordionOpen) {
    setCharts(prev =>
      // Sets visibility to false for all Charts
      prev.map(chart => ({ ...chart, visible: false })));
    return
  }
  // The actionType is set to "all" if the "Reset" button is clicked
  if (actionType.current === "all") {
    loadChartsProgressively(charts.filter(chart => checkboxesState[chart.propertyName]), 50);
  } else if (actionType.current === "individual") {
    setCharts(prev =>
      // Updates the state for the Charts but since only one has changed state the page does
      not lag
      prev.map(chart => ({
        ...chart,
        visible: !!checkboxesState[chart.propertyName]
      })));
  }
}, [checkboxesState, isAccordionOpen, userAccordionOpen]);
```

Below is the `loadChartsProgressively` function. What this function does is iterate over the `checkedCheckboxes` and set a timeout for each one. Each timeout is set to expire progressively later than the previous one. This can be seen on the second argument of the `setTimeout` function where the delay is `idx * delay`. When the callback for any timeout runs, it will update the state of a specific Chart from the Charts `useState` to true. The `useRef` called `initialAddedDelay` is a value that is used when all Tracks are expanded at the same time to offset the Chart visibility change, to prevent Charts from different Tracks from appearing at the same time causing lag. I only want to use this `initialAddedDelay` when all Tracks are opened at the time, not when only a single Track is interacted with. So after this added delay is used, it is reset to 0. To prevent this from affecting the callbacks of the timeouts, I copy its value into a new variable called `initialDelay` at the beginning of the function. The value of `initialAddedDelay` is updated back to its original value when the 'expand all' button is pressed.

```
const loadChartsProgressively = (checkedCheckboxes: {...}[], delay: number) => {
  const initialDelay = initialAddedDelay.current; // copy value to be able to reset
  'initialAddedDelay' after scheduling timeouts
  checkedCheckboxes.forEach((checkbox, idx) => { // loop over the checked Checkboxes
    setTimeout(() => { // set a timeout that will set the 'visible' property of a specific
      Chart to true
      setCharts(prevCharts =>
```

```

    prevCharts.map(prevChart =>
      prevChart.propertyName === checkbox.propertyName ? { ...prevChart, visible:
true } : prevChart
    ));
  }, idx * delay + initialDelay); // different delay for each Chart.
});
// after iterating, set 'initialAddedDelay' to 0.
initialAddedDelay.current = 0;
}

```

Ethics

While this project does not involve the collection of any new user data, a key consideration is the secure management of API keys when implementing AWS Secrets. A common mistake is unintentionally logging API keys as part of error messages, leading to their exposure and persistence within the logging system. This can potentially allow unauthorized individuals to access the keys, which is particularly critical in production environments. In the event of such an incident, the affected API key would need to be immediately revoked and regenerated to prevent unauthorized access and maintain security. This situation is analogous to sending your email password to a family group chat.

Findings

Working on the WebRTC Call Debugger project provided many personal learning outcomes and resulted in several key accomplishments.

Learning Outcomes:

- Gained practical experience developing with React, including a deeper understanding of Hooks, Component lifecycle, and managing asynchronous behaviour in user interfaces.
- Acquired proficiency in handling, formatting, and transforming complex JSON data for visualisation.
- Learned and applied various JavaScript data manipulation techniques, such as `map`, `reduce`, and iteration methods, optimising their use for data processing.
- Developed knowledge of AWS cloud architecture, specifically focusing on AWS Lambda (serverless backend processing) and AWS Secrets Manager (secure credential storage).
- Understood Genesys's multi-AWS account infrastructure and the role of internal services like Hyperion in dynamic environmental configuration.
- Gained experience interacting with external APIs (Sumo Logic and New Relic).
- Developed skills in TypeScript through a combination of structured learning (book study, note-taking) and practical application.
- Gained insight into Single Page Application (SPA) architecture and the deployment of static assets via Content Delivery Networks (CDN).
- Understood the importance of secure credential management in a production environment, including the risks of logging sensitive information and the benefits of using services like AWS Secrets Manager.
- Developed experience writing unit tests for backend logic using Jest.
- Implemented frontend performance optimisation techniques, including memoisation (`useMemo`) to reduce unnecessary rendering and a progressive loading strategy using timeouts.
- Gained experience troubleshooting and developing workarounds for library limitations (e.g., the hover effect), deepening understanding of DOM manipulation.

Project Accomplishments:

- Integrated New Relic data into the dashboard, adding crucial call statistics alongside existing Sumo Logic event logs.
- Implemented a secure method for retrieving API keys by migrating from insecure argument passing to using AWS Secrets Manager.

- Integrated the internal Hyperion service to enable dynamic configuration for the Lambda function.
- Developed robust data fetching logic capable of querying both Sumo Logic and New Relic APIs (including the New Relic GraphQL interface).
- Implemented sophisticated data aggregation and transformation logic to restructure complex raw data for frontend visualisation.
- Designed and built an interactive and user-friendly frontend interface utilizing React, Genesys UI components, and MUI X charts.
- Created dynamic data visualisation components capable of plotting statistics for any number of users and media tracks.
- Addressed data consistency issues to ensure accurate and meaningful data plotting.
- Implemented interactive controls (checkboxes, "Select All/Reset" button) for selective chart display.
- Developed a custom shared hover effect across charts to facilitate visual correlation of data points over time.
- Successfully implemented frontend performance optimisations (memoisation, progressive loading) to ensure a smooth user experience even with large datasets.
- Wrote unit tests for the backend Lambda function to ensure correctness and reliability.
- Delivered a functional WebRTC Call Debugger dashboard successfully integrated within the Supportability tool.

Conclusion

This project successfully achieved its goal: adding call statistics from New Relic to the `webrtc-timeline` frontend repository in the form of Charts. The final product is a dashboard that visualises data from both New Relic and Sumo Logic using interactive components. This work directly addresses the challenge of time-consuming manual analysis of historical call data, eliminating the need for manual data querying and sorting. By using this tool, Support Engineers can now more easily access and utilise relevant call data to streamline their debugging process. Key technical improvements implemented included enhancing API key management security by integrating AWS Secrets Manager and adding Hyperion for the dynamic configuration of the Lambda function to ensure data is fetched from the correct sources across different environments.

This project provided valuable practical experience, deepening my understanding of how different services interact with each other in a distributed system. It also offered significant insight into professional practice, allowing me to experience working within a professional environment and participate in activities such as weekly team stand-ups and check-ins with my manager and colleagues, preparing me for my future endeavours.

Recommendations

I am proud of the work I have accomplished on this project. Nevertheless, there are always opportunities for further enhancement.

The following recommendations are proposed for future work on the WebRTC Call Debugger:

- **Implement a Loading Indicator within Supportability:** The current implementation lacks visual feedback when the AWS Lambda function is processing a data request, resulting in a blank screen within the Supportability interface. Adding a loading screen or progress indicator would enhance the user experience and provide clearer information on the tool's status during data retrieval.
- **Explore Chart Zoom Functionality:** Adding 'zooming' functionality to the charts would be a quality-of-life improvement to help in the interpretation and analysis of longer calls with numerous data points. This feature is not supported by the free tier of MUI X Charts, necessitating exploration of alternative charting libraries or investigating options within MUI's paid tier.

References

- [1] Wikipedia. "Genesys (company)". Accessed: April 20, 2025. [Online]. Available: [https://en.wikipedia.org/wiki/Genesys_\(company\)](https://en.wikipedia.org/wiki/Genesys_(company))
- [2] MDN. "WebRTC API". Accessed: April 22, 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API
- [3] WebRTC. "Real-time communication for the web". Accessed: April 23, 2025. [Online]. Available: <https://webrtc.org/>
- [4] LinkedIn. "callstats.io: Overview". Accessed: April 16, 2025. [Online]. Available: <https://www.linkedin.com/company/callstatsio/>
- [5] Cloudflare. "What is a content delivery network (CDN)? | How do CDNs work?". Accessed: April 24, 2025. [Online]. Available: <https://www.cloudflare.com/learning/cdn/what-is-a-cdn/>
- [6] MUI. "Ready to use Material Design components". Accessed: April 24, 2025. [Online]. Available: <https://mui.com/material-ui/>
- [7] Gearon. "# Why not to use document.querySelector()". Github. Accessed: April 21, 2025. [Online]. Available: <https://github.com/reactjs/react.dev/issues/4626>
- [11] AWS. "Lambda quotas". Accessed: April 20, 2025. [Online]. Available: <https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html>
- [12] Ecma International. "ECMA-262". Accessed: April 20, 2025. [Online]. Available: <https://ecma-international.org/publications-and-standards/standards/ecma-262/>
- [13] Jest. "Delightful JavaScript Testing". Accessed: April 21, 2025. [Online]. Available: <https://jestjs.io/>
- [14] W3. "Identifiers for WebRTC's Statistics API". Accessed: April 21, 2025. [Online]. Available: <https://www.w3.org/TR/webrtc-stats/#dom-rtcreceivedrtptime>
- [15] MDN. "RTCIceCandidatePairStats: timestamp property". Accessed: April 24, 2025. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/API/RTCIceCandidatePairStats/timestamp>
- [16] MDN. "RTCIceCandidatePairStats: currentRoundTripTime property". Accessed: April 24, 2025. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/API/RTCIceCandidatePairStats/currentRoundTripTime>
- [17] MDN. "RTCIceCandidatePairStats: bytesSent property". Accessed: April 24, 2025. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/API/RTCIceCandidatePairStats/bytesSent>
- [18] H. Zelaya. "WebRTC Video Optimization: The Crucial Balance Between Bitrate and Frame Rate". WebRTC.ventures. Accessed: April 21, 2025. [Online]. Available: <https://webrtc.ventures/2024/08/webrtc-video-optimization-the-crucial-balance-between-bitrate-and-frame-rate/>
- [19] C. Sandanshiv. "Understanding Latency in WebRTC? How can VideoSDK Fix Latency?". Videosdk. Accessed: April 21, 2025. [Online]. Available: <https://www.videosdk.live/blog/what-is-latency-in-webrtc>
- [20] Genesys. "AWS regions for Genesys Cloud". Accessed: April 20, 2025. [Online]. Available: <https://help.mypurecloud.com/articles/aws-regions-for-genesys-cloud-deployment/#tab1>

Appendix

The appendices contain supplementary material providing additional technical detail on the project's data structures at various processing stages and code listings for key backend and frontend components. This information expands upon the concepts discussed in the main body of the report for readers seeking deeper technical understanding.

Appendix A: Logs

JSON Data from New Relic excerpt

This is the data that comes from New Relic without any processing.

```
{
  "timestamp": 1745332371347,
  "_originAppVersion": "11.52.0+21-release/11.52.0-b20a4f826f3e160a0a696b961b6cd95c3badd94",
  "tags.region": "us-east-1",
  "tags.billingProducts": "None",
  "tags.instanceId": "i-xxxxx",
```

```

"tags.trustedAccountId": "xxxxx",
"tags.accountId": "xxxxx",
"_eventType": "firstPropose",
"entityGuid": "Mzxxxxxxxxxxxxxxxx58MTA00Tc0MjY5NA",
"tags.prvDomain": "prv.inindca.com",
"tags.network_space": "default",
"tags.availability_zone_id": "use1-az2",
"tags.imageId": "ami-xxxxxxx",
"tags.spotInstance": "false",
"tags.hyperion_habitat": "dev",
"tags.env": "dev",
"tags.availabilityZone": "us-east-1d",
"_orgId": "xxxxxxxx",
"_appId": "82xxxxxxe",
"tags.hyperion_family": "core",
"tags.role": "public_api_v2",
"host": "public-api-v2-xxxxxxxx",
"tags.pureCloudPrvDomain": "prvxxxxxxx",
"tags.service_name": "public-api-v2",
"tags.privateIp": "10xxxxxx",
"tags.account": "gc-dev-use1",
"tags.hyperion_ou": "lower",
"tags.kernelId": "None",
"tags.instanceType": "c6i.xxx",
"tags.devpayProductCodes": "None",
"_userId": "xxxexxxxxc",
"tags.hyperion_region": "use1",
"_appName": "streamingclient",
"tags.version": "9172",
"tags.marketplaceProductCodes": "None",
"_originAppName": "collaborate-ui",
"sdpViaXmppRequested": "true",
"tags.hyperion_accountName": "dev",
"tags.hyperion_biome": "dev-use1",
"sessionId": "xxxxxxxx",
"_originAppId": "xxxxxx",
"appName": "public_api_v2 (Development)",
"tags.domain": "inindca.com",
"sessionType": "collaborateVideo",
"realAgentId": "xxxxx",
"_eventTimestamp": 1745332346167,
"tags.hyperion_environment": "dev",
"tags.ecosystem": "pc",
"_appVersion": "19.0.1",
"tags.architecture": "x86_64",
"tags.pureCloudPubDomain": "use1.dev-pure.cloud",
"tags.ramdiskId": "None",
"conversationId": "7f1bdf42-c5cb-437b-ac76-6d44e493b7d4",
"tags.hyperion_compliance_fipsRequired": "true",
"appId": "xxxxxxx"
}

```

Processed data in Lambda

This is the properties that have been picked from the data coming form New Relic.

```

{
  "_eventTimestamp": 1745332347079,
  "_userId": "85364abe-6b9a-4xxxxxxxxxx",

```

```

"conversationId": "7f1bdf42-c5cb-437b-ac76-6d44e493b7d4",
"sessionId": "8021014592865428481",
"sessionType": "collaborateVideo",
"_eventType": "getStats",
"_originAppId": "7d3b05ab-2abb-467a-xxxxxx",
"_originAppName": "volt",
"_originAppVersion": "1.12.1",
"localTrack_[1]_": {
  "retransmittedBytesSent": 0,
  "packetsSent": 0,
  "packetsLost": 0,
  "codec": "97 video/VP8 90000",
  "packetLoss": 0,
  "retransmittedPacketsSent": 0,
  "kind": "video",
  "bytes": 0
},
"remoteTrack_[0]_": {
  "packetLoss": 0,
  "packetsReceived": 1,
  "codec": "96 audio/opus 48000",
  "kind": "audio",
  "jitter": 0,
  "packetsLost": 0,
  "bytes": 27
},
"remoteTrack_[1]_": {
  "packetsLost": 0,
  "jitter": 0,
  "bytes": 0,
  "packetLoss": 0,
  "packetsReceived": 0,
  "kind": "video"
},
"localTrack_[0]_": {
  "packetLoss": 0,
  "kind": "audio",
  "totalAudioEnergy": 6.28681108929658e-9,
  "audioLevel": 0.00015259254737998596,
  "packetsSent": 0,
  "packetsLost": 0,
  "retransmittedPacketsSent": 0,
  "bytes": 0,
  "retransmittedBytesSent": 0,
  "codec": "96 audio/opus 48000"
}
},

```

Data for plotting

This is an example of a Track with data aggregated as observed in the arrays instead of individual values.

```

"Track: localTrack_[1]_ Kind video": {
  "_eventTimestamp": [1745057651739, 1745057656734, 1745057661733, 1745057666732],
  "retransmittedBytesSent": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
  "packetsSent": [0, 599, 1511, 2480, 3476, 4453, 5426, 6409, 7393],

```



```

"packetsLost": [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
"codec": ["97 video/VP8 90000", "97 video/VP8 90000"],
"packetLoss": [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
"retransmittedPacketsSent": [33,33,33,33,33,33,33,33,33,33,33],
"kind": ["video", "video", "video", "video", "video", "video"],
"bytes": [0,611525,1584844,2634976,3706257,4766409,5819994,6893648],
"intervalPacketsSent": [0,599,912,969,996,977,973,983,984,983,],
"jitter": [0,0.002466,0.0007549999999999999,0.001411,0.001144,0.001011],
"intervalPacketsLost": [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
"intervalPacketLoss": [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
"bitrate": [0,978,1557,1680,1714,1696,1685,1717,1707],
"roundTripTime": [0,0.11673,0.130112,0.110703,0.11445599999999999],
},

```

Appendix B: AWS Lambda

Parser New Relic Search

This is the code that performs the API request to New Relic to fetch call statistics.

```

import logger from '../logger';
import moment from 'moment';
import {
  NewRelicJson,
  NewRelicSearchResult,
  RunConfig
} from '../interfaces';

function getQuery(conversationIds: string[], from: moment.Moment, to: moment.Moment) {
  const limit = 'MAX';
  return `SELECT * FROM WebrtcStats WHERE (${conversationIds.reduce((output, curr, idx) =>
output + 'conversationId = ' + '\'' + curr + '\'' + (conversationIds[idx] !==
conversationIds[conversationIds.length - 1] ? ' OR ' : ''), '')) SINCE ${from.format('x')}
UNTIL ${to.format('x')} LIMIT ${limit}`;
}

export async function getSearchOptions(config: RunConfig): Promise<SearchOptions> {
  const { newRelicHost, newRelicAccountId, conversationIds, from, to } = config;

  const query = getQuery(conversationIds, from, to);

  const headers = {
    "API-Key": config.newRelicApiKey,
    "Content-Type": 'application/json'
  };

  return {

```

```

requestOptions: {
  headers,
  method: 'POST',
  body: JSON.stringify({
    query: `{
      actor {
        account(id: ${newRelicAccountId}) {
          nrql(query: "${query}") {
            results
          }
        }
      }
    }`
  }),
  url: `https://${newRelicHost}/graphql`
};
}

interface SearchOptions {
  url: string;
  requestOptions: RequestInit
}

async function performSearch(searchOptions: SearchOptions): Promise<NewRelicSearchResult[]> {
  const start = moment();

  const response = await fetch(searchOptions.url, searchOptions.requestOptions);
  if (!response.ok) {
    throw new Error(`New Relic fetch response was not an Ok with error:
    ${JSON.stringify(await response.json())}`);
  }

  const responseBody: NewRelicJson = await response.json();

  logger.info(`Search total time: ${moment().diff(start, 'seconds')} seconds`);
  return responseBody.data.actor.account.nrql.results;
}

export async function newRelicSearch(config: RunConfig) {
  const searchOptions = await getSearchOptions(config);
  console.log('SEARCH OPTIONS *****')
  const results = await performSearch(searchOptions);
  return results;
}

```

Parser Page Builder

This code will build the HTML string that will be returned to the front end. It inserts the data into the `window` object.

```

import { JobConfig, NewRelicSearchResult, NewRelicTimeline, RunConfig, SumoSearchResult,
SumoTimeline } from "../interfaces";
import { getReactCDNUrl, getRunConfig } from "../utils/utils";
import { search } from '../sumo-search/sumo-search';
import { newRelicSearch } from '../newrelic-search/newrelic-search';
import logger from '../logger';

const commonComponentsCDN = 'https://dhqbrvplips7x.cloudfront.net/common-ui-docs/genesys-
webcomponents/2.41.0-352/genesys-webcomponents';

export function createTimelinesFromResults(results: SumoSearchResult[]): SumoTimeline[] {
  const timelinesObjByUserId: { [userId: string]: SumoTimeline } = {};

  results.forEach(result => {

```

```

const userId = result.map.userid;

let timelineObj = timelinesObjByUserId[userId];
if (!timelineObj) {
  timelineObj = timelinesObjByUserId[userId] = {
    conversationId: result.map.conversationid,
    entries: [],
    userId,
  };
}

const entries = timelineObj.entries;

entries.push({
  message: result.map.timelinemessage,
  loggerClientId: result.map.clientid,
  appName: result.map.app,
  originAppName: result.map.originappname,
  originAppVersion: result.map.originappversion,
  originAppId: result.map.originappid,
  extra: result.map.extra,
  clientTime: result.map.clienttime,
});
});

return Object.values(timelinesObjByUserId);
}

export function createTimelinesFromResultsNewRelic(results: NewRelicSearchResult[]):
NewRelicTimeline[] {
  const newRelicTimelinesByUserId: { [userId: string]: NewRelicTimeline } = {};

  results.forEach(result => {
    const userId = result['_userId'];
    if (!userId) {
      throw new Error('User id is null');
    }
    const conversationId = result['conversationId'];
    if (!conversationId) {
      throw new Error('Conversation id is null');
    }

    let timelineObj = newRelicTimelinesByUserId[userId];
    if (!timelineObj) {
      timelineObj = newRelicTimelinesByUserId[userId] = {
        conversationId: conversationId,
        entries: [],
        userId,
      };
    }

    const entries = timelineObj.entries;

    entries.push({
      localCandidateType: result.localCandidateType,
      remoteCandidateType: result.remoteCandidateType,
      _eventTimestamp: result._eventTimestamp,
      bytesReceived: result.bytesReceived,
      bytesSend: result.bytesSend,
      _userId: result._userId,
      conversationId: result.conversationId,
    });
  });
}

```

```

        requestsReceived: result.requestsReceived,
        responsesReceived: result.responsesReceived,
        sessionId: result.sessionId,
        sessionType: result.sessionType,
        totalRoundTripTime: result.totalRoundTripTime,
        consentRequestsSent: result.consentRequestsSent,
        _eventType: result._eventType,
        _originAppId: result._originAppId,
        _originAppName: result._originAppName,
        _originAppVersion: result._originAppVersion,
        ...aggregateTracks(result)
    });
});
return Object.values(newRelicTimelinesByUserId);
}

interface aggregatedTracks {
    [trackKey: string]: {
        [completeTrackKey: string]: string;
    }
}

export function aggregateTracks(result: NewRelicSearchResult) {
    const tracks: aggregatedTracks = {};
    const regex = /((?:local|remote)track_\[\d+\])(.+)/i;

    Object.entries(result).forEach(([key, value]) => {
        const match = key.match(regex);
        if (match) {
            const trackKey = match[1];
            const trackProperty = match[2];
            if (!tracks[trackKey]) {
                tracks[trackKey] = {};
            }
            tracks[trackKey][trackProperty] = value;
        }
    });
    return tracks;
}

export async function getTimelines(config: RunConfig) {
    const { results } = await search(config);

    return createTimelinesFromResults(results);
}

export async function getNewRelicTimelines(config: RunConfig) {
    let results = await newRelicSearch(config);
    return createTimelinesFromResultsNewRelic(results);
}

export async function run(config: JobConfig, errorInHtml = true) {
    let html = '';

    try {
        const runConfig = await getRunConfig(config);
        console.log('**** DEBUG', { baseUrl: runConfig.cdnBaseUrl, env:
process.env['ENVIRONMENT'] });
        console.log('CDN BASE URL', runConfig.cdnBaseUrl);
        const reactCDNBase = runConfig.cdnBaseUrl || getReactCDNUrl();
        console.log('CDN BASE URL', reactCDNBase);
    }

```

```

const newRelicTimelinesPromise = getNewRelicTimelines(runConfig);
const sumoLogicTimelinesPromise = getTimelines(runConfig);

await Promise.allSettled([newRelicTimelinesPromise,
sumoLogicTimelinesPromise]).then((promiseResults) => {
  const [newRelicPromiseResult, sumoLogicPromiseResult] = promiseResults;

  let sumoLogicTimelines: SumoTimeline[] = [];
  let newRelicTimelines: NewRelicTimeline[] = [];

  if (newRelicPromiseResult.status === 'fulfilled') {
    newRelicTimelines = newRelicPromiseResult.value;
  } else {
    logger.error('Failed to get NewRelic results', newRelicPromiseResult.reason);
  }

  if (sumoLogicPromiseResult.status === 'fulfilled') {
    sumoLogicTimelines = sumoLogicPromiseResult.value;
  } else {
    logger.error('Failed to get SumoLogic results', sumoLogicPromiseResult.reason);
  }

  html = `
    <html lang="en">      <head>          <title>Results</title>          <script>
window.sumoData = ${JSON.stringify(sumoLogicTimelines)};
      window.newRelicData = ${JSON.stringify(newRelicTimelines)};
    </script>      <link rel="stylesheet" href="${reactCDNBase}/index.css" />
    </head>      <body>          <div id="root"></div>          <script
src="${reactCDNBase}/index.js"></script>
    </body>
    </html>      `;
  });
} catch (err: any) {
  if (!errorInHtml) {
    throw err;
  }

  logger.error('Failed to parse timeline', err);

  html = `
    <html lang="en">      <head>          <title>Results</title>          </head>      <body>
<h3>Failed to get results</h3>      <div>          ${err.stack}
    </div>      </body>      </html>      `;
  }
  return html;
}

```

Parser AWS Secrets

This code uses the AWS SDK to get the API keys to fetch data from Sumo Logic and New Relic.

```

import {SecretsManagerClient, GetSecretValueCommand, BatchGetSecretValueCommand} from "@aws-
sdk/client-secrets-manager";
import logger from "../logger";

const newRelicSecretId = "WebRTC-Timeline-Parser-NewRelicApiKeys";
const sumoLogicSecretId = "WebRTC-Timeline-Parser-SumoLogicApiKey";

```

```

const client = new SecretsManagerClient();

export async function getSumoAndNewRelicSecrets() {
  const input = {SecretIdList: [newRelicSecretId, sumoLogicSecretId]}
  const command = new BatchGetSecretValueCommand(input);
  const response = await client.send(command);

  if (response.SecretValues === undefined || response.SecretValues.length == 0) {
    throw new Error("Could not retrieve API keys");
  } else if (response.SecretValues.length < 2) {
    logger.error("Could only retrieve one API key")
  }

  const {apiKey} = JSON.parse(response.SecretValues[0].SecretString || "{}");
  const {accessCode, accessKey} = JSON.parse(response.SecretValues[1].SecretString || "{}");

  return {
    newRelicApiKey: apiKey,
    sumoAccessCode: accessCode,
    sumoAccessKey: accessKey
  }
}

```

Parser Utils

This code is a helper file used for building the run configuration. It gets environment variables and provides defaults. It calls the function to retrieve API keys from AWS.

```

import moment from "moment";
import { Moment } from "moment";
import { JobConfig, RunConfig, SumoEndpointsByRegion } from "../interfaces";
import logger from "../logger";
import { getSumoAndNewRelicSecrets } from "../aws/secrets";

export async function getRunConfig(jobConfig: JobConfig): Promise<RunConfig> {
  const { to, from, days, cdnBaseUrl } = validateAndFormatConfig(jobConfig);
  const sumoAndNewRelicCreds = await getNewRelicAndSumoConfigs();

  return { ...jobConfig, to, from, days, cdnBaseUrl, ...sumoAndNewRelicCreds };
}

function validateAndFormatConfig(jobConfig: JobConfig) {
  let to: Moment;
  let from: Moment;
  let days = jobConfig.days;

  if (!jobConfig.to) {
    to = moment();
  } else {
    to = moment(jobConfig.to).add(5, 'minutes');
  }

  if (!jobConfig.from) {
    days = days || 1;
    from = moment().subtract(days, 'day');
  } else {
    from = moment(jobConfig.from).subtract(5, 'minutes');
  }
}

```

```

    if (from.isAfter(to) || !to.isValid() || !from.isValid()) {
      throw new Error(`Search time frame is invalid. from: ${from.toString()} to:
${to.toString()}`);
    }

    const cdnBaseUrl = jobConfig.cdnBaseUrl;
    return { to, from, days, cdnBaseUrl };
  }

  async function getNewRelicAndSumoConfigs() {
    const sumoApiUrl = process.env['SUMO_API_URL'] || SumoEndpointsByRegion['us-west-2'];
    const newRelicHost = process.env['NR_API_HOST'] || 'api.newrelic.com';
    const newRelicAccountId = process.env['NR_ACCOUNT_ID'] || '3805541';

    const secrets = await getSumoAndNewRelicSecrets();

    return {
      ...secrets, newRelicHost, newRelicAccountId, sumoApiUrl
    };
  }

  export function getReactCDNUrl(): string {
    const defaultEnv = 'prod';

    const baseByEnv: any = {
      dev: 'inindca.com',
      test: 'inintca.com',

      prod: 'mypurecloud.com',
      'prod-usw2': 'usw2.pure.cloud',
      'prod-cac1': 'cac1.pure.cloud',
      'prod-euc1': 'mypurecloud.de',
      'prod-euw1': 'mypurecloud.ie',
      'prod-euw2': 'euw2.pure.cloud',
      'prod-apne1': 'mypurecloud.jp',
      'prod-apne2': 'apne2.pure.cloud',
      'prod-apse2': 'mypurecloud.com.au',
      'prod-sae1': 'sae1.pure.cloud'
    };

    const env = process.env.ENVIRONMENT as string;

    const base = baseByEnv[env] || baseByEnv[defaultEnv];

    const url = `https://apps.${base}/webrtc-timeline-react`;

    logger.info(`Getting CDN url for react env: ${env} url: ${url}`);

    return url;
  }

```

Appendix C: React Frontend

App Component

This is the frontend's entry point. There were existing functions on this file to handle data from Sumo Logic. I added my own code.

```

import {aggregateByAppId, prepareDataForGraphing_NewRelic, prepareExtraData} from
'../services/data-aggregator';
import UserTimelineComponent from '../user-timeline/UserTimeline';
import '@fontsource/urbanist/latin.css';
import '@fontsource/noto-sans/latin.css';
import {useState} from 'react';
import {UserTimelineByApp} from '../interfaces';
import {GuxButton} from 'genesys-spark-components-react';
import User from "../new-relic/User";

import './app.scss'

const App = () => {
  const [timelines, setTimelines] = useState(aggregateByAppId());
  const [expandAll, setExpandAll] = useState(false);
  const [expandAllNR, setExpandAllNR] = useState(false);

  const userTimelineChanged = (userTimeline: UserTimelineByApp) => {
    const index = timelines.findIndex(timeline => timeline.userId === userTimeline.userId);
    timelines.splice(index, 1, userTimeline);
    setTimelines([ ... timelines]);
  }

  const userTimelines = () => {
    const tl = timelines.map((timeline) => {
      return (
        <div key={timeline.userId}>
          <UserTimelineComponent
            timeline={timeline}
            onUserTimelineChanged={userTimelineChanged}
            expandAll={expandAll}
          />
        </div>
      );
    });
    return (
      <>
        <h1>Sumo Logic</h1>
        {tl.length ? tl : noResults}
      </>
    );
  };

  const newRelicCharts = () => {
    const nrData = prepareDataForGraphing_NewRelic();
    const extraData = prepareExtraData();
    return (
      <div>
        <h1>New Relic Charts</h1>
        <div>
          {nrData.length === 0 ? (noResults) : (
            nrData.map((userData, idx) => {
              return (
                <User
                  userData={userData}
                  extraData={extraData}
                  key={userData.userId}
                  expandAllNr={expandAllNR}
                  initialChartDelayBetweenUsers={idx * 1000}
                />
              );
            })
          )}
        </div>
      </div>
    );
  };
};

```



```

    );
  })
}
</div>
</div>
);
})

const noResults = (
  <div>
    <h2 style={{ fontWeight: "bold" }}>No results found</h2>
  </div>
);

function popout() {
  const { search, pathname, host } = window.location;
  const params = new URLSearchParams(search);

  params.append('popout', 'true');

  const url = `https://${host}${pathname}?${params.toString()}`;
  window.open(url);
}

function renderPopoutButton() {
  const params = new URLSearchParams(window.location.search);
  if (params.has('popout')) {
    return;
  }
  return (
    <GuxButton
      className={"top-buttons"}
      onClick={popout}>
      <span>Popout To New Window</span>
    </GuxButton>
  );
}

function expandAllButton() {
  const text = expandAll ? 'Collapse All Sumo Logs' : 'Expand All Sumo Logs';
  return (
    <GuxButton
      className={"top-buttons"}
      onClick={() => setExpandAll(!expandAll)}>
      <span>{text}</span>
    </GuxButton>
  );
}

function expandAllNRButton() {
  const text = expandAllNR ? 'Collapse All New Relic Charts' : 'Expand All New Relic Charts';
  return (
    <GuxButton
      className={"top-buttons"}
      onClick={() => setExpandAllNR(!expandAllNR)}>
      <span>{text}</span>
    </GuxButton>
  );
}

```

```

    return (
      <>
        {renderPopoutButton()}
        {expandAllButton()}
        {expandAllNRButton()}
        {userTimelines()}
        {newRelicCharts()}
      </>
    );
  }

  export default App;

```

User Component

This component is used for representing the Users of a call.

```

import {GuxAccordion, GuxAccordionSection} from "genesys-spark-components-react";
import {useEffect, useState} from "react";
import ExtraInfoAccordion from "../ExtraInfoAccordion";
import Track from "../Track";

export default function User({ initialChartDelayBetweenUsers, userData, expandAllNr,
extraData }) {
  const [isAccordionOpen, setIsAccordionOpen] = useState(false);

  useEffect(() => {
    setIsAccordionOpen(expandAllNr);
  }, [expandAllNr]);

  const userEntries = userData.entries;
  const keysFromUserEntries = Object.keys(userEntries);

  const buildUserTracks = (keysFromUserEntries) => {
    return (
      keysFromUserEntries.sort().map((trackName, idx) => {
        const trackEntries = userEntries[trackName];
        return (
          <Track
            userAccordionOpen={isAccordionOpen}
            entriesFromTrack={trackEntries}
            key={`_${userData.userId}_${trackName}`}
            expandAllNr={expandAllNr}
            trackName={trackName}
            userId={userData.userId}
            extraDelayForChart={idx * 200 + initialChartDelayBetweenUsers}
          />
        );
      })
    );
  };

  return (
    <div>
      <GuxAccordion>
        <GuxAccordionSection
          open={isAccordionOpen}
          onGuxopened={() => setIsAccordionOpen(true)}
          onGuxclosed={() => setIsAccordionOpen(false)}

```

```

    >
    <h2 slot={"header"}>User {userData.userId}</h2>
    <div slot="content">
      {buildUserTracks(keysFromUserEntries)}
      <ExtraInfoAccordion userExtraInfoLogs={extraData[userData.userId]} expandAllNr=
{expandAllNr}/>
    </div>
  </GuxAccordionSection>
</GuxAccordion>
</div>
);
}

```

Track Component

This component is used for representing individual Media Tracks from a User.

```

import {GuxAccordion, GuxAccordionSection, GuxButton} from "genesys-spark-components-react";
import {useEffect, useRef, useState} from "react";
import Checkbox from "../Checkbox";
import ChartWrapper from "../ChartWrapper";

export default function Track({
  userAccordionOpen,
  extraDelayForChart,
  entriesFromTrack,
  expandAllNr,
  trackName,
  userId
}) {
  const sortedEntriesKeysFromTrack = Object.keys(entriesFromTrack).sort();

  const [isAccordionOpen, setIsAccordionOpen] = useState(false);
  const [checkboxesState, setCheckboxesState] = useState({});
  const [charts, setCharts] = useState<{ propertyName: string, chart: any, visible: boolean }
[]>([]);

  // States for hover effect
  const [hoveredDataValue, setHoveredDataValue] = useState(null);
  const [stickyValue, setStickyValue] = useState(null);

  const actionType = useRef<'all' | 'individual'>('all');
  const initialAddedDelay = useRef(0);

  // Handle button to Expand All Accordions
  useEffect(() => {
    setIsAccordionOpen(expandAllNr);
    if (expandAllNr) initialAddedDelay.current = extraDelayForChart;
  }, [expandAllNr]);

  // Prepare the data to plot later. Set the data state.
  useEffect(() => {
    const allCharts = sortedEntriesKeysFromTrack.reduce((acc: any, propertyName: string) => {
      const aggregatedDataArray = entriesFromTrack[propertyName];
      if (!isNumericalData(aggregatedDataArray, propertyName)) {
        return acc; // we don't want to plot properties that are not numbers
      }
      const unformattedTime = entriesFromTrack["_eventTimestamp"];
      const formattedTime = formatTime(unformattedTime);
    });
  });

```

```

    const newChart = {
      visible: false,
      propertyName,
      chart: {
        trackPropertyName: propertyName,
        formattedTime: formattedTime,
        data: aggregatedDataArray,
      }
    }
    return [ ... acc, newChart];
  }, []);

  setCharts(allCharts);

  setCheckboxesState(() => {
    return allCharts.reduce((acc, chart) => {
      const propertyName = chart.propertyName;
      acc[propertyName] = true;
      return acc;
    }, {});
  })
}, []);

// Change the Visibility property of Charts when checkboxes change
useEffect(() => {
  if (!isAccordionOpen || !userAccordionOpen) { // set charts to not visible
    setCharts(prev =>
      prev.map(chart => ({ ... chart, visible: false })));
    return
  }
  if (actionType.current === "all") {
    loadChartsProgressively(charts.filter(chart => checkboxesState[chart.propertyName]),
50);
  } else if (actionType.current === "individual") {
    setCharts(prev =>
      prev.map(chart => ({
        ... chart,
        visible: !!checkboxesState[chart.propertyName]
      })));
  }
}, [checkboxesState, isAccordionOpen, userAccordionOpen]);

function formatTime(timestamps: number[]) {
  return timestamps.map((timestamp) => {
    return new Date(timestamp);
  });
}

function isNumericalData(aggregatedData: any[], propertyName: string) {
  if (propertyName === "_eventTimestamp") return false;
  if (typeof (aggregatedData) === null || typeof (aggregatedData) === "object" &&
    typeof (aggregatedData[0]) !== "number") return false;
  return true;
}

const buildCheckboxes = (sortedEntriesKeysFromTrack: string[]) => {
  return (
    <div>
      {sortedEntriesKeysFromTrack.map((propertyName) => {
        const aggregatedDataArray = entriesFromTrack[propertyName];

```

```

    if (!isNumericalData(aggregatedDataArray, propertyName)) return;
    return (
      <Checkbox
        trackProperty={propertyName}
        setCheckboxesState={setCheckboxesState}
        checkboxesState={checkboxesState}
        actionType={actionType}
        key={` ${propertyName}${userId}`}
      />
    );
  }
}

const resetSelectAllButton = () => {
  return (
    <GuxButton
      style={{ marginTop: '15px' }}
      onClick={() => {
        const noValueIsTrue = Object.values(checkboxesState).filter((val) => val).length
        if (noValueIsTrue) {
          actionType.current = 'all';
          setCheckboxesState(() =>
            Object.keys(checkboxesState).reduce((acc, key) => ({
              ...acc,
              [key]: true
            }), {}));
        } else {
          actionType.current = 'individual';
          setCheckboxesState(() =>
            Object.keys(checkboxesState).reduce((acc, key) => ({
              ...acc,
              [key]: false
            }), {}));
        }
      }}
    >
      {Object.values(checkboxesState).filter(val => val).length === 0 ? 'Select all' :
'Reset'}
    </GuxButton>
  );
}

// This sets a timeout to update the visibility of a chart to true if the button state says
so.
const loadChartsProgressively = (checkedCheckboxes: {
  propertyName: string,
  chart: any,
  visible: boolean
}[], delay: number) => {
  const initialDelay = initialAddedDelay.current;
  checkedCheckboxes.forEach((checkbox, idx) => {
    setTimeout(() => {
      setCharts(prevCharts =>
        prevCharts.map(prevChart =>
          prevChart.propertyName === checkbox.propertyName ? { ...prevChart, visible: true
} : prevChart
        ));
    }, idx * delay + initialDelay);
  });
}

```

```

    });
    initialAddedDelay.current = 0;
  }

  return (
    <GuxAccordion>
      <GuxAccordionSection
        open={isAccordionOpen}
        onGuxopened={(event) => {
          event.stopPropagation();
          setIsAccordionOpen(true);
        }}
        onGuxclosed={(event) => {
          actionTypes.current = 'all';
          event.stopPropagation();
          setIsAccordionOpen(false);
        }}>
        <h2 slot="header">{trackName}</h2>
        <div slot="content" style={{ display: "flex", flexDirection: "row" }}>
          <div style={{ display: "flex", flexDirection: "column", flexShrink: 0 }}>
            {buildCheckboxes(sortedEntriesKeysFromTrack)}
          </div>
          <div style={{ display: "flex", flexDirection: "row", overflowX: "scroll",
paddingBottom: "20px" }}>
            {charts.map(chart => (chart.visible &&
              <ChartWrapper
                hoveredDataValue={hoveredDataValue}
                stickyValue={stickyValue}
                onHover={setHoveredDataValue}
                onSticky={setStickyValue}
                trackPropertyName={chart.chart.trackPropertyName}
                formattedTime={chart.chart.formattedTime}
                data={chart.chart.data}
                isAccordionOpen={isAccordionOpen}
                key=
{`container-${chart.chart.trackPropertyName}:${chart.chart.formattedTime}:${userId}`}
              </>
            ))}
          </div>
        </div>
      </GuxAccordionSection>
    </GuxAccordion>
  );
}

```

Extra Info Accordion and Extra Info Components

These components are used for showing non plottable information that we receive from the Lambda.

```

import {GuxAccordion, GuxAccordionSection} from "genesys-spark-components-react";
import ExtraInfo from "../ExtraInfo";
import {useEffect, useState} from "react";

export default function ExtraInfoAccordion({ userExtraInfoLogs, expandAllNr }) {
  const [isAccordionOpen, setIsAccordionOpen] = useState(false);

  useEffect(() => {
    setIsAccordionOpen(expandAllNr);
  });
}

```

```

}, [expandAllNr]);

return (
  <div>
    <GuxAccordion>
      <GuxAccordionSection>
        open={isAccordionOpen}
        onGuxopened={(event) => {
          event.stopPropagation();
          setIsAccordionOpen(true);
        }}
        onGuxclosed={(event) => {
          event.stopPropagation();
          setIsAccordionOpen(false);
        }}
      >
        <h2 slot="header">Extra Data</h2>
        <div slot="content">
          <div>
            <div style={{
              fontWeight: "bold",
              color: "var(--gse-ui-accordion-header-default-foreground-labelColor)"
            }}>
              <p>Session Type: {userExtraInfoLogs.find(log => log.sessionType)?.sessionType
?? 'No data'}</p>
              <p>Session Id: {userExtraInfoLogs.find(log => log.sessionId)?.sessionId ??
'No data'}</p>
              <p>LocalCandidate
                Type: {userExtraInfoLogs.find(log =>
log.localCandidateType)?.localCandidateType ?? 'No data'}</p>
              <p>RemoteCandidate
                Type: {userExtraInfoLogs.find(log =>
log.remoteCandidateType)?.remoteCandidateType ?? 'No data' } </p>
            </div>
          </div>
          <ExtraInfo userExtraInfoLogs={userExtraInfoLogs}></ExtraInfo>
        </div>
      </GuxAccordionSection>
    </GuxAccordion>
  </div>
);
}

```

```

import {GuxTable} from "genesys-spark-components-react";

export default function ExtraInfo({ userExtraInfoLogs }) {
  return (
    <div>
      <GuxTable>
        <table slot="data">
          <thead>
            <tr>
              <th data-column-name="event-timestamp">Event Timestamp</th>
              <th data-column-name="event-type">Event Type</th>
              <th data-column-name="origin-app-id">Origin App Id</th>
              <th data-column-name="origin-app-name">Origin App Name</th>
              <th data-column-name="origin-app-version">Origin App Version</th>
            </tr>
          </thead>
          <tbody>

```

```

    {userExtraInfoLogs.map((log) => {
      return <tr key={`extra-
    throw${log._eventTimestamp}${log._eventType}${log._userId}`}>
      <td>{new Date(log._eventTimestamp).toISOString()}</td>
      <td>{log._eventType}</td>
      <td>{log._originAppId}</td>
      <td>{log._originAppName}</td>
      <td>{log._originAppVersion}</td>
    </tr>
    }}}
  </tbody>
</table>
</GuxTable>
</div>
);
}

```

ChartWrapper and Chart Components

The `ChartWrapper` component contains the `Chart` Component with the `div` to allow the hover effect to work. It manages the hover and click interactions of the user. The `Chart` component is a wrapper around MUI's `LineChart` Component.

```

import {MutableRefObject, useEffect, useMemo, useRef} from "react";
import {formatLargeNumber} from "../../services/data-aggregator";
import Chart from "./Chart";

export default function ChartWrapper({
  trackPropertyName,
  formattedTime,
  data,
  stickyValue,
  onHover,
  onSticky,
  hoveredDataValue,
  isAccordionOpen
}) {
  const containerRef: MutableRefObject<any> = useRef(null);
  const plottableArea: MutableRefObject<any> = useRef(null);
  const clipArea: MutableRefObject<any> = useRef(null);

  useEffect(() => {
    if (!containerRef.current) return;
    const timer = setTimeout(() => {
      if (!containerRef.current) return;
      const svgElement = containerRef.current.querySelector('.css-1evyvmv-MuiChartsSurface-root');
      const clipPathFromSvg = svgElement.querySelector(":scope > clipPath:last-of-type");
      if (!clipPathFromSvg) return;
      const plottableAreaGrid = clipPathFromSvg.querySelector("rect");
      if (!plottableAreaGrid) return;
      const y = +plottableAreaGrid.getAttribute("y");
      const height = +plottableAreaGrid.getAttribute("height");
      const wholeX = +plottableAreaGrid.getAttribute("x");
      const wholeWidth = +plottableAreaGrid.getAttribute("width");
      plottableArea.current = {
        y, height,
        topY: y + height,
        x: wholeX,

```



```

    width: wholeWidth,
    topX: wholeX + wholeWidth
  }

  const xValueForHoverLine = containerRef.current.querySelector('g[clip-
path]').getBBox();
  if (!xValueForHoverLine) return;
  const x = +xValueForHoverLine.x;
  const width = +xValueForHoverLine.width;
  clipArea.current = { x, y, width, height };
}, 25);
return () => clearTimeout(timer);
}, [isAccordionOpen]);

const mapValueToPositionInChart = (value, width) => {
  const min = formattedTime[0];
  const max = formattedTime[formattedTime.length - 1];
  return ((value - min) / (max - min)) * width;
};

const snapToDataPoint = (xInsideClip) => {
  if (!clipArea.current) return 0;
  const positions = formattedTime.map((value) => (
    mapValueToPositionInChart(value, clipArea.current.width)
  ));

  let closest = positions[0];
  let minDiff = Math.abs(xInsideClip - positions[0]);
  positions.forEach(pos => {
    const diff = Math.abs(xInsideClip - pos);
    if (diff < minDiff) {
      minDiff = diff;
      closest = pos;
    }
  });
  return closest;
}

const onClickHandler = (event: { clientX: number; clientY: number; }) => {
  if (!clipArea.current || !containerRef.current) return;

  if (stickyValue !== null) {
    onSticky(null);
    return;
  }

  const containerRect = containerRef.current.getBoundingClientRect();
  const relX = event.clientX - containerRect.left;
  const relY = event.clientY - containerRect.top;
  if (plottableArea.current && positionIsInside(relX, relY)) {
    const xInsideClip = relX - clipArea.current.x!;
    const snappedData = snapToDataPoint(xInsideClip);
    onSticky(snappedData);
  }
}

const onMouseMoveHandler = (event: { clientX: number; clientY: number; }) => {
  if (!clipArea.current || !containerRef.current) return;

  const containerRect = containerRef.current.getBoundingClientRect();
  const relX = event.clientX - containerRect.left;

```

```

const relY = event.clientY - containerRect.top;
if (plottableArea.current && positionIsInside(relX, relY)) {
  const xInsideClip = relX - clipArea.current.x!;
  const snappedData = snapToDataPoint(xInsideClip);
  onHover(snappedData);
} else {
  onHover(null);
}
}

function positionIsInside(relX: number, relY: number) {
  return relX >= plottableArea.current.x - 1 && relX <= plottableArea.current.topX &&
    relY >= plottableArea.current.y - 1.5 && relY - 1 <= plottableArea.current.topY;
}

const onMouseLeaveHandler = () => {
  onHover(null);
};

const chart = () => {
  return (
    <Chart
      trackPropertyName={trackPropertyName}
      formattedTime={formattedTime}
      data={data}
    />
  );
}

const memoizedChart = useMemo(() => chart(), []);
const memoizedMaxAndMin = useMemo(
  () => ({
    max: formatLargeNumber(Math.round(Math.max(... data) * 100_000) / 100_000),
    min: formatLargeNumber(Math.round(Math.min(... data) * 100_000) / 100_000),
    avg: formatLargeNumber(Math.round(data.reduce((sum, curr) => sum + curr) / data.length
* 100_000) / 100_000)
  }), []);

const finalValue = stickyValue !== null ? stickyValue : hoveredDataValue;
const relativeOverlayX = clipArea.current && finalValue !== null ?
  finalValue :
  null;

return (
  <div
    ref={containerRef}
    onMouseMove={onMouseMoveHandler}
    onMouseLeave={onMouseLeaveHandler}
    onClick={onClickHandler}
    style={{ position: "relative", width: 500, display: "flex", flexDirection: "column",
alignItems: "center" }}
  >
    <div>
      {memoizedChart}
      {
        relativeOverlayX !== null && (
          <div
            style={{
              position: "absolute",
              left: clipArea.current.x! + relativeOverlayX,
              top: clipArea.current.y,
              height: clipArea.current.height,

```

```

        width: "1px",
        backgroundColor: "black",
        pointerEvents: "none"
      }}
    />
  )
}
</div>
<span style={{
  textAlign: "center",
  margin: "0 auto"
}}>Max: {memoizedMaxAndMin.max} Min: {memoizedMaxAndMin.min} Average:
{memoizedMaxAndMin.avg}</span>
</div>
)
}

```

```

import {LineChart} from "@mui/x-charts";
import {formatLargeNumber} from "../../services/data-aggregator";

export default function Chart({ trackPropertyName, formattedTime, data }) {

  return (
    <div>
      <LineChart
        xAxis={[{
          label: "Time",
          data: formattedTime,
          scaleType: 'time',
          valueFormatter: (date) => date.toLocaleTimeString()
        }]}
        yAxis={[{
          valueFormatter: formatLargeNumber
        }]}
        series={[
          { label: trackPropertyName, data: data, showMark: false },
        ]}
        width={400}
        height={400}
        grid={{ vertical: true, horizontal: true }}
        skipAnimation={true}
      />
    </div>
  );
}

```

Checkbox Component

This Component is a wrapper around the `GuxFormFieldCheckbox`. It is used for selecting which `Charts` to show to the User.

```

import {GuxFormFieldCheckbox} from "genesys-spark-components-react";

export default function Checkbox({ actionType, trackProperty, setCheckboxesState,
checkboxesState }) {
  return (
    <GuxFormFieldCheckbox>
      <input slot="input" checked={!!checkboxesState[trackProperty]} type="checkbox" name=

```

```

{trackProperty}
    onChange={(event) => {
        actionType.current = 'individual';
        event.stopPropagation();
        setCheckboxesState(prevState => ({
            ...prevState, [trackProperty]: event.target.checked
        }));
    }};
  </GuxFormFieldCheckbox>
  <label slot="label">{trackProperty}</label>
</GuxFormFieldCheckbox>
)
}

```

Data Aggregator Helper

The frontend's data aggregation occurs in this file. Where we aggregate data from multiple logs into arrays. There were existing functions on this file to handle data from Sumo Logic. I added my own code.

```

import {
  AppTimeline,
  EnhancedTimelineEntry,
  SessionTypes,
  SortColumn,
  SortDirection,
  SumoTimeline,
  TimelineEntry,
  UserTimelineByApp
} from "../interfaces";

let messageCounter = 0;

declare global {
  interface Window {
    sumoData: SumoTimeline[];
  }
}

function addGeneralCallData(global, nrEntryProperties) {
  const numericalNonTrackStats = nrEntryProperties.filter(
    (entry) => typeof entry[1] === 'number');
  if (numericalNonTrackStats.length < 2) return;
  numericalNonTrackStats.forEach(([key, value]) => {
    if (global['General call data'][key] === undefined) {
      global['General call data'][key] = [];
    }
    global['General call data'][key].push(value);
  });
}

function flattenUserEntry(userEntries: { string: object }[]) {
  const global = {};
  global['General call data'] = {};
  userEntries.forEach((logEntries) => {
    const nrEntryProperties = Object.entries(logEntries);
    const timestamp = logEntries["_eventTimestamp"];

    addGeneralCallData(global, nrEntryProperties);

    const trackProperties = nrEntryProperties.filter((entry) =>

```

```

entry[0].toLowerCase().includes('track'));
    findTracksAndAddTheirProperties(trackProperties, global, timestamp);
  });
  return global;
}

function findTracksAndAddTheirProperties(newRelicEntryProperties: [string, object][], global:
{}, timeStampOfEntry) {
  newRelicEntryProperties.forEach((logEntriesKV) => {
    const trackName = `Track: ${logEntriesKV[0]} Kind ${logEntriesKV[1]['kind']}`;

    if (global[trackName] === undefined) {
      global[trackName] = {};
    }
    if (global[trackName]["_eventTimestamp"] === undefined) {
      global[trackName]["_eventTimestamp"] = [];
    }
    global[trackName]["_eventTimestamp"].push(timeStampOfEntry);
    const numberOfEntriesWeShouldHaveCurrently = global[trackName]["_eventTimestamp"].length
    const entryTrackEntries = Object.entries(logEntriesKV[1]);
    addPropertiesFromTrack(entryTrackEntries, global, trackName,
numberOfEntriesWeShouldHaveCurrently);
  });
}

function addPropertiesFromTrack(entryTrackEntries, global, trackName, timestampEntriesAmount)
{
  entryTrackEntries.forEach((entryTrackEntry) => {
    const trackProperty = entryTrackEntry[0];
    if (global[trackName][trackProperty] === undefined) {
      global[trackName][trackProperty] = [];
    }
    // Some track properties are missing the first data point. These properties seem to take
    longer to start sending data.
    // So I just fill the missing entry with a 0.    if ((global[trackName]
[trackProperty].length < timestampEntriesAmount - 1) &&
      !isNaN(entryTrackEntry[1])) { // only add 0's for things that have numbers. Not for
    codec for example.
      global[trackName][trackProperty].push(0);
    }
    global[trackName][trackProperty].push(entryTrackEntry[1]);
  });
}

export function prepareExtraData() {
  const windowData = window['newRelicData'];

  const usersArr = windowData.map((user) => {
    return {
      userId: user.userId,
      data:
        [
          ... user.entries.filter((log) => log._eventType !== "getStats")
        ]
    }
  });
};

// convert from array to obj
return usersArr.reduce((accumulator, currUser) => {
  accumulator[currUser.userId] = currUser.data;
  return accumulator;

```

```

    }, {}));
  }

export function prepareDataForGraphing_NewRelic() {
  const windowData = window['newRelicData'];

  return windowData.map((user) => {
    const timestampSortedEntries = sortNewRelicEntriesByEventTimeStamp(user.entries);
    return {
      "userId": user.userId,
      "conversationId": user.conversationId,
      "entries": flattenUserEntry(timestampSortedEntries)
    };
  });
}

function sortNewRelicEntriesByEventTimeStamp(entries) {
  return entries.sort((a, b) => {
    const aTime = new Date(a["_eventTimeStamp"]);
    const bTime = new Date(b["_eventTimeStamp"]);
    return aTime.getTime() - bTime.getTime();
  });
}

export function aggregateByAppId(): UserTimelineByApp[] {
  return window.sumoData.map((userTimeline) => {
    const aggregatedTimeline: UserTimelineByApp = {
      conversationId: userTimeline.conversationId,
      userId: userTimeline.userId,
      apps: []
    };

    const apps: { [appName: string]: AppTimeline } = {};

    userTimeline.entries.forEach((timelineEntry) => {
      // if there's a secondary app, use that so logs get grouped together
      let key = `${timelineEntry.appName}:${timelineEntry.loggerClientId}`;

      if (timelineEntry.originAppName) {
        key = `${timelineEntry.originAppName}:${timelineEntry.originAppId}`;
      }

      let appTimeline = apps[key];
      if (!appTimeline) {
        appTimeline = apps[key] = { name: key, entries: [] };
      }

      let extra: any;
      let sessionType: SessionTypes = SessionTypes.unknown;
      if (typeof timelineEntry.extra === 'string' && timelineEntry.extra.length > 0) {
        extra = JSON.parse(timelineEntry.extra);
        sessionType = extra?.sessionType
        // if the extra data is just an empty array, just ignore it
        if (Array.isArray(extra) && !extra.length) {
          extra = undefined;
        }
      } else if (timelineEntry.extra) {
        console.error('expected \'extra\' to be stringified json', { timelineEntry });
      }
    });
  });
}

```

```

    if (timelineEntry.message.includes('Initiating incoming session')) {
      if (sessionType === SessionTypes.softphone) {
        appTimeline.softphoneSessionInit = true;
      } else if (sessionType === SessionTypes.collaborateVideo) {
        appTimeline.videoSessionInit = true;
      } else if (sessionType === SessionTypes.screenRecording) {
        appTimeline.screenRecordingSessionInit = true;
      } else {
        appTimeline.unknownSessionInit = true;
      }
    }
  }

  appTimeline.entries.push({ ... timelineEntry, id: `${messageCounter++}`, extra,
sessionType });
});

const appsArr = Object.values(apps);

appsArr.forEach((appTimeline) => addTimeDiffsToEntries(appTimeline.entries));

aggregatedTimeline.apps = appsArr;
return aggregatedTimeline;
});
}

function addTimeDiffsToEntries(entries: EnhancedTimelineEntry[]) {
  entries.forEach((entry, index) => {
    let timeDiff = 0;
    if (index !== 0) {
      const previousItem = entries[index - 1];
      timeDiff = new Date(entry.clientTime).getTime() - new
Date(previousItem.clientTime).getTime();
    }

    entry.timeDiff = timeDiff;
  });
}

export function formatLargeNumber(value: number) {
  if (value >= 1_000_000_000) {
    return `${(value / 1_000_000_000).toFixed(1)}B`;
  }
  if (value >= 1_000_000) {
    return `${(value / 1_000_000).toFixed(1)}M`;
  }
  if (value >= 1_000) {
    return `${(value / 1_000).toFixed(1)}k`;
  }
  if (value <= 0.001 && value > 0) {
    return `${(value * 1_000).toFixed(1)}m`
  }
  return value.toString();
}

export function sortAppTimeline(timeline: AppTimeline, sortColumn: SortColumn, sortDirection:
SortDirection): AppTimeline {
  const sortedTimeline: AppTimeline = { ... timeline, sortColumn, sortDirection };

  const sortFn = (entry1: TimelineEntry, entry2: TimelineEntry) => {
    const v1 = entry1[sortColumn];
    const v2 = entry2[sortColumn];

```

```
const ret = v1 < v2 ? 1 : -1;
if (sortDirection === 'asc') {
  return ret * -1;
}

return ret;
}

sortedTimeline.entries.sort(sortFn);
addTimeDiffsToEntries(sortedTimeline.entries);
sortedTimeline.entries = [ ... sortedTimeline.entries];

return sortedTimeline;
}
```