Scaling Web Services for the Lightweight Collector

A Major Qualifying Project submitted to the faculty of Worcester Polytechnic Institute in partial fulfillment of the requirements for the degree of Bachelor of Science.

March 19th, 2021

Submitted By:
Ivan Eroshenko
Yaru Gong
Sam Moran
Wenjing Ying

Submitted To:
Professor Mark Claypool
Worcester Polytechnic Institute
John Sadlier
Juniper Networks

This report represents the work of WPI undergraduate students submitted to the faculty as evidence of completion of a degree requirement. WPI routinely publishes these reports on its website without editorial or peer review. For more information about the projects program at WPI, please see http://www.wpi.edu/academics/ugradstudies/project-learning.html
Abstract

To better support enterprise networking customers, Juniper Networks is developing the Lightweight Collector (LWC), a product that automates device monitoring and data collection. The beta version of the LWC is deployed to customer sites, where it gathers data from devices, processes these data in the AWS cloud, and provides an interface for customers to monitor their equipment. We created new Amazon Web Services (AWS) Cloudwatch Dashboard widgets, developed anomaly detection and notification systems, implemented a tool which aims to ensure version consistency of all deployed Stacks, and provisioned our components with AWS CloudFormation to ensure their scalability and maintainability. Our work equips the development team with a set of solutions to monitor and discover underlying issues of the LWC and facilitate the project’s future scalability.
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abbreviations</td>
<td>4</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>5</td>
</tr>
<tr>
<td>2 Background</td>
<td>7</td>
</tr>
<tr>
<td>2.1 Tools and Technologies</td>
<td>7</td>
</tr>
<tr>
<td>2.1.1 Lightweight Collector</td>
<td>7</td>
</tr>
<tr>
<td>2.1.2 AWS Cloudwatch</td>
<td>8</td>
</tr>
<tr>
<td>2.1.3 AWS Lambda</td>
<td>9</td>
</tr>
<tr>
<td>2.1.4 AWS Step Functions</td>
<td>9</td>
</tr>
<tr>
<td>2.1.5 AWS EventBridge</td>
<td>11</td>
</tr>
<tr>
<td>2.1.6 AWS IoT Greengrass</td>
<td>12</td>
</tr>
<tr>
<td>2.1.7 AWS Cloudformation</td>
<td>13</td>
</tr>
<tr>
<td>2.2 Summary</td>
<td>14</td>
</tr>
<tr>
<td>3. Methodology</td>
<td>15</td>
</tr>
<tr>
<td>3.1 Enhancement of Data Collection Monitoring</td>
<td>15</td>
</tr>
<tr>
<td>3.2 Anomaly Detection and Notification Systems</td>
<td>15</td>
</tr>
<tr>
<td>3.3 Maintenance and Scalability</td>
<td>15</td>
</tr>
<tr>
<td>4. Enhancing Cloudwatch Dashboards</td>
<td>16</td>
</tr>
<tr>
<td>4.1 Background</td>
<td>16</td>
</tr>
<tr>
<td>4.1.1 Cloudwatch Insights</td>
<td>16</td>
</tr>
<tr>
<td>4.1.2 Cloudwatch Dashboard</td>
<td>17</td>
</tr>
<tr>
<td>4.1.3 Notation Used in Logs Widgets</td>
<td>17</td>
</tr>
<tr>
<td>4.1.4 Log Groups Used by JSAS Team</td>
<td>18</td>
</tr>
<tr>
<td>4.2 Methodology and Implementation</td>
<td>18</td>
</tr>
<tr>
<td>4.2.1 Scheduled-Request Submit Counter Actual, Manual-Request Submit Counter Actual</td>
<td>21</td>
</tr>
<tr>
<td>4.2.2 Overall Response Distribution By Error Type</td>
<td>25</td>
</tr>
<tr>
<td>4.2.3 Successes and Failures By Target</td>
<td>28</td>
</tr>
<tr>
<td>4.2.4 SFDC-Sender Done Events</td>
<td>31</td>
</tr>
<tr>
<td>4.2.5 S3 Uploads Vs Successful Responses</td>
<td>34</td>
</tr>
<tr>
<td>4.3. Evaluation and Results</td>
<td>38</td>
</tr>
<tr>
<td>4.4. Future work</td>
<td>39</td>
</tr>
<tr>
<td>5. Developing Anomaly Detection Solutions</td>
<td>40</td>
</tr>
<tr>
<td>5.1 Background</td>
<td>40</td>
</tr>
<tr>
<td>5.1.1 Microsoft Teams Webhook</td>
<td>40</td>
</tr>
<tr>
<td>5.1.2 Subscription Filter</td>
<td>40</td>
</tr>
<tr>
<td>5.1.3 AWS Step Function</td>
<td>41</td>
</tr>
<tr>
<td>5.1.4 Cloudwatch Alarm</td>
<td>41</td>
</tr>
<tr>
<td>5.2 Methodology and Implementation</td>
<td>41</td>
</tr>
</tbody>
</table>
5.2.1 Mismatch Between Requested Target Count and Responses 42
5.2.2 Notification When Allocated Memory Consumed 44
5.2.3 Invoke handler error 46
5.3 Evaluation and Results 46
5.4 Future Work 47

6. Designing Scalable Solutions 48
   6.1 Background 48
   6.2 Methodology and Implementation 48
      6.2.1 Provisioning Implemented Components with Cloudformation Template 48
      6.2.2 Version Control and Stack Checking 49
   6.3 Evaluation and Results 53
   6.4 Future Work 53

7. Conclusion 54

8. References 55
Abbreviations

**DEA** Digital Experiences and Automation

**JSAS** Juniper Support Automation Solution

**LWC** Lightweight Collector

**AWS** Amazon Web Services

**IoT** Internet of Things

**GGC** Greengrass Core

**JSON** JavaScript Object Notation

**YAML** Yet Another Markup Language

**S3** Simple Storage Service
1. Introduction

Juniper Networks is a company that provides a wide variety of networking products for its clients, ranging from core networking infrastructure for service providers to enterprise cloud solutions. Providing a good customer support experience is a high priority for Juniper; however, troubleshooting and remediating issues with network equipment is often not an easy task. Juniper’s larger clients have thousands of devices, some running different firmware versions, with varying compatibility levels with other devices. Additionally, equipment is deployed at customer sites, which means that Juniper engineers sometimes need to physically travel to the customer’s location to fix problems that arise, which is costly and inconvenient. Even simple information gathering for troubleshooting, such as discovering exactly which pieces of hardware and firmware versions are deployed at a customer site, is a non-trivial task.

To address these issues, the Juniper Support Automation Solution (JSAS) team is developing the Lightweight Collector (LWC), a product that will automate the network support experience. Currently in beta, the LWC is installed on customer sites and collects diagnostic data from devices on the local network, which it aggregates and sends to the JSAS team’s AWS cloud.

As they bring the Lightweight Collector to more customers, the JSAS developers face problems related to scale. Some AWS services that the Lightweight Collector relies on are relatively new technologies that may have not been used at the scale Juniper is using them. Moreover, some stability issues as well as undocumented limitations with these AWS services have been encountered. As Juniper deploys Lightweight Collectors to more clients, they are also discovering additional edge cases and errors in the components of the LWC. Furthermore, the increased load on the LWC infrastructure makes manually troubleshooting such issues more difficult. With hundreds of log messages written every hour, a number rapidly increasing, new automated monitoring solutions are needed to keep up with the pace of information.

The goal of our project was to develop software components that allow the JSAS team to better visualize and understand data received from the Lightweight Collector. We developed AWS CloudWatch dashboard widgets to compile information in a single location to aid the JSAS team in identifying and debugging issues as they occur. We created anomaly detection systems that monitor log messages from the Lightweight Collector and send notifications directly to the JSAS team when issues occur. Finally, we developed a workflow to check deployed AWS infrastructure against the source repository on GitLab to ensure all components are up to date. All of these components were provisioned using AWS CloudFormation templates so that our work can be deployed at a greater scale as the LWC is extended to more customers in the future.
The AWS CloudWatch Dashboard widgets we developed provide the JSAS team with tools that allow them to monitor the status of data collection jobs without needing to sift through hundreds of log messages across different log groups. Additionally, the anomaly detection systems we developed notify the JSAS team directly when problems occur, giving them a better idea of the frequency and type of errors. Insights gained from using both of these tools together have already assisted the JSAS team in identifying and fixing issues with the LWC data collection processes. Finally, the CloudWatch stack checker tool we developed allows the JSAS team to monitor the versions of deployed LWC components and to be informed when application pieces are not up to date. Since we designed each of these solutions with scalability in mind, the JSAS team can continue to use the tools we developed as the LWC project grows to include more customers.

This report covers the tools, technologies and techniques we used, as well as the design process and implementation details. Chapter 2 provides an overview of relevant tools and technologies. Chapter 3 details the overarching requirements and design principles that guided our work. Chapters 4, 5, and 6 discuss in detail how we implemented each component: CloudWatch dashboards, error notification functionality, and scalability work. Finally, Chapter 7 concludes this report with a summary of our work and how it fits into the overall LWC project.
2. Background

This section describes the existing automation solution, and the intentions for the Lightweight Collector (LWC) set forth by the Digital Experience and Automation (DEA) team. This section also outlines the important tools and technologies involved in building the LWC and their relevance to the project.

2.1 Tools and Technologies

The following section describes the tools and technologies relevant to our development of the LWC project, as well as the Lightweight Collector itself. This section covers not only what these tools and technologies are, but also how they relate to the production and deployment of the LWC.

2.1.1 Lightweight Collector

The Lightweight Collector (LWC) is an automated network support solution that is currently in beta, under active development by Juniper’s Digital Experience and Automation team. The beta version of the LWC is deployed on an on-site virtual machine attached to a customer’s existing network equipment. The LWC is also connected to Juniper’s Amazon Web Service (AWS) cloud so it can communicate diagnostic information, though for security reasons, the other devices on a customer’s network may not be. Although a web-based front-end interface, Juniper engineers can remotely access and interact with the LWC to view diagnostic information or schedule data collection tasks on the client’s network devices.

Figure 1 [11]: How the Lightweight Collector gathers data from the customer devices and uploads these data to the AWS Cloud
2.1.2 AWS Cloudwatch

AWS Cloudwatch is an AWS service for developers to monitor and observe their applications. Cloudwatch provides engineers actionable data and insights [1]. CloudWatch can be used to detect anomalous behavior in AWS environments, set alarms, visualize logs and metrics side by side, take automated actions, troubleshoot issues, and discover insights to keep the applications running smoothly.

Cloudwatch Logs is the place where logs received from all AWS services or systems are aggregated. The developers can divide logs from different sources into different log groups. Cloudwatch Logs also enables the users to view, search, filter, query, and sort all the data and logs [2].

In the LWC project, the JSAS team uses AWS Cloudwatch to monitor and analyse the logs received from all the stages of the data collection processes. The relevant results are then displayed in Cloudwatch Dashboards, a tool for aggregating and visualizing relevant metrics and insightful data contained in log messages. Figure 2 below is an example of a single dashboard widget, which shows information about LWC data collection runs extracted from relevant log messages.

![Figure 2: Example of a Dashboard widget](image)

2.1.3 AWS Lambda

AWS Lambda is a serverless computing service that enables developers to run code without managing servers [3]. The developers can upload or edit the code in AWS Lambda, and the Lambda allocates execution power and runs the code in ephemeral containers. AWS Lambda is an event-driven service, which means the Lambda functions are mainly triggered by the occurrence of events [4]. Other AWS resources, like Simple Notification Service (SNS), or EventBridge Subscription Filters, for instance, can be used as triggers to activate the Lambda function. The developers are also able to test and deploy the Lambda functions on the AWS page. AWS Lambda allows the developers to use different languages and runtimes when deploying functions. In our project, Node.js is the primary runtime we use when developing the Lambda functions.
2.1.4 AWS Step Functions

AWS Step Functions is a service that allows for specific workflows to be automatically executed in a series of steps, known as states. A workflow can be defined in the AWS Step Functions console as a “state machine,” a series of states that the workflow can be in during its execution, with connections between different states that define the order in which the workflow proceeds. As the step function executes, only one state is active at a time. Figure 3 below shows a graphical representation of a state machine; the states are the labeled components, and the connections are illustrated by the lines connecting each state:

![Figure 3: An example of an AWS Step Function](image)

The workflow begins at the “Start” state, then progresses through the remaining states in an order defined by the way each state is connected. Some states are connected to multiple other states; decisions for which state to actually transition to can be defined programmatically and can depend on the execution of a previous state. For instance, in Figure 3 above, the states “OrderOK” and “DatabaseError” can both be reached from “CreateOrderB,” but “OrderOk” is the state that was chosen during this execution based on the results of “CreateOrderB.”

Each individual state in a step function can perform one of several actions. From the AWS documentation [10], the possible actions are:

- Do some work in your state machine
- Make a choice between branches of execution
- Stop an execution with a failure or success
- Simply pass its input to its output or inject some fixed data
- Provide a delay for a certain amount of time or until a specified time/date
- Begin parallel branches of execution
- Dynamically iterate steps

AWS Step Functions are useful because they allow for the simple definition and automatic execution of complex workflows, which may need to behave differently or perform different actions each time they run. In the context of the Lightweight Collector project, state machines can be used to address sections of the data collection process that require performing an action after a time delay. By creating a “Wait” state followed by a “Task” state, a workflow (and all the data necessary for it to proceed successfully) can easily be delayed for an arbitrary amount of time.

### 2.1.5 AWS EventBridge

AWS EventBridge is a service that deals with events in the AWS cloud. An event occurs when an application’s environment updates (for instance when a Lambda function executes or a CloudWatch log message is written). AWS EventBridge allows users to receive such messages and configure additional functionality based on certain events happening. EventBridge rules can be used in one of two ways: either to filter messages so that only messages matching the filter criteria are processed, or to create an event on a set schedule regardless of whether an event was received or not. EventBridge targets are useful to quickly and easily receive events from one source and route them to a destination, such as an AWS Lambda function or CloudWatch log group.

Filter rules allow users to define patterns to match incoming events against, which means that only the incoming events that match the pattern are passed along by EventBridge to their target. Events are filtered based on the contents of the incoming event, which includes information about the source (typically an AWS service) that the event originated from, as well as additional information about the nature of the environment change that caused the event. For instance, when a CloudWatch log message is written, the event sent to EventBridge will contain the contents of the message and indicate CloudWatch as the source.

EventBridge rules can also be used to create and send events to targets on a schedule, regardless of if an incoming event was received by another source. Rules use CRON (Command Run On, a UNIX software utility for scheduling programs to run) syntax, which has 6 fields, designating minutes, hours, days of the month, month, days of the week, and year. Using CRON expressions, rules can define intervals in minutes, hours, months, or years for events to be sent. Alternatively, events can be scheduled to run on specific minutes of the hour, hours of the day, and days of the week or month. For example, a user could write an EventBridge rule that triggers at 11:08 PM each Thursday every month of 2021, and EventBridge would send an event to a target at each of those times.
Another important feature of AWS EventBridge rules is that the contents of the events being sent to targets by EventBridge are not always necessarily the same as the contents that caused the rule to trigger in the first place. The output of the rule can also be configured to change or replace parts of the incoming event, pass along only parts of the original event, or replace it altogether with a constant event text that will be sent every time.

With regard to the Lightweight Collector, AWS EventBridge allows for event-based design, where different stages of the Lightweight Collector’s data processing pipeline are triggered by the finish of the previous step. For instance, when a function finishes executing, it can write a log message to CloudWatch, which creates an AWS EventBridge event. The event can then be compared against an EventBridge filter, and if applicable, subsequently directed to the next step of the process, which can receive the event as a signal to start further processing.

2.1.6 AWS IoT Greengrass

AWS Greengrass Core is a software component that, upon installing on a host device, extends AWS cloud capabilities to the device. Additionally, AWS Greengrass allows devices to collect and analyze data on the local network, then send these data to a single Greengrass Core device, which aggregates the data. Developers using AWS Greengrass can write serverless code (AWS Lambda functions) in the Cloud and then easily deploy it to on-site devices to execute applications locally. The developers are able to connect their devices or systems to the Greengrass group, and send Greengrass messages locally without the involvement of Cloud. In a Greengrass group, only the Greengrass Core device needs to be connected to the AWS cloud, which means the individual data collection devices do not have to be connected to the Cloud, improving application security [8].

![Figure 4: Structure of a Greengrass Core Group](image)
As shown in Figure 4, several local devices and a Greengrass Core (GGC) make up a Greengrass group. All devices that communicate with the AWS Greengrass Core must be members of a Greengrass group. Each group must contain an AWS Greengrass Core. The Discovery API enables devices to retrieve the information needed to connect to an AWS Greengrass Core (which is in the same Greengrass group as the device).

Local devices and Greengrass Cores communicate over the local network. Local devices are restricted to have access to the Cloud in the LWC project for security reasons. The AWS IoT Device SDK must be installed on the device. Only the collecting device with a Greengrass Core can communicate with the Cloud and needs to have Internet access. Lambda functions can be run on a Greengrass Core device that can communicate with other local devices and with the Cloud. The configuration and Lambda functions to be run locally are installed on Greengrass Core via 'Deploy', a deployment agent on Greengrass Core that fetches the materials to be deployed from the Cloud and then deploys them on Greengrass Core.

![Diagram of LWC using Greengrass](image)

**Figure 5: LWC uses Lambdas deployed with Greengrass**

In the LWC project, the Lightweight Collector, which acts as the AWS Greengrass Core device, will pull information from the Junos devices on-site (where the Lightweight Collector is installed) using AWS Lambda Functions.

### 2.1.7 AWS Cloudformation

AWS CloudFormation is an AWS service which allows developers to easily create a blueprint for a collection of related AWS and third-party resources, provision them quickly and consistently, and manage them throughout their life cycles by treating infrastructure as code. AWS Cloudformation templates allows users to specify the desired resources using code in YAML or JSON format. The template can then be used to create, update or delete an entire stack as a single unit instead of managing the resources
individually. The diagram below depicts the workflow of deploying an AWS Cloudformation template. In the LWC project, AWS Cloudformation is used extensively to provision and deploy all the infrastructure developed by the JSAS team.

![Diagram showing the workflow of deploying an AWS Cloudformation template.](image)

Figure 6: How AWS Cloudformation works

### 2.2 Summary

Our project focuses on building on top of the existing functionality and making improvements to the current iteration of the Lightweight Collector. Juniper is specifically concerned with scaling up the Lightweight Collector to many networks and clients, and they have run into unexpected issues with various Amazon cloud services related to handling such large volumes and high frequencies of data. With the AWS Cloudwatch Dashboard, we are able to query through the log information collected from LWCs that are installed on the customers devices and notify if there exists abnormality in the log data received. Using other AWS services explained above, we develop additional error detection notification systems that notify Juniper developers when problems within the Lightweight Collector occur. Finally, we create CloudFormation templates to allow for our work to be deployed at scale.
3. Methodology

There were two primary objectives of our project. The first objective was to help identify errors and abnormalities in the LWC behavior using the data logs received from all stages of the data collection process. The second objective was to facilitate the scalability of the components of the LWC project. The following sections describe the methodology used to address the three components we focused on as part of our work. Section 3.1 describes the enhancements of data collection monitoring. Section 3.2 describes anomaly detection and notification systems. Section 3.3 describes the maintenance and scalability facilitation of both our solutions and all other components created as part of the LWC project.

3.1 Enhancement of Data Collection Monitoring

An important part of the LWC project is monitoring how data collection processes progress across all the customers. Thus, we focused on enhancing the JSAS team's CloudWatch Dashboard - a tool used for aggregating customized widgets and metrics that display relevant information about the project's behavior. For more context regarding this component's role in the LWC project and the methods used to improve the JSAS team's Cloudwatch Dashboard, see Chapter 4.

3.2 Anomaly Detection and Notification Systems

Anomaly detection solutions are aimed at identifying common errors by analyzing relevant log events and notifying the JSAS developers via a dedicated channel in Microsoft Teams immediately as the problems appear. For an in-depth description of this component, the methodology used for development, and the results of the anomaly detection solutions, see Chapter 5.

3.3 Maintenance and Scalability

As the LWC project grows larger, it is important to maintain scalability of the components. To accomplish this, we described relevant AWS resources for each component we developed in a CloudFormation template - a tool used for provisioning AWS infrastructure. Additionally, we created a solution that checks whether the Git commit IDs of deployed CloudFormation stacks against the versions of the source code, which ensure the relevant components remain up-to-date. For more information regarding this portion of the project, the methodology followed, and results achieved, see Chapter 6.
4. Enhancing Cloudwatch Dashboards

A focus of the JSAS team is on monitoring how the different components of the data collection perform, in order to proactively discover issues or unexpected behavior faster than the customers discover these problems. This is particularly relevant since the project is relatively new and infrastructure is being actively developed. Additionally, since a large amount of data is being monitored, the JSAS team needs tools that make it easier to analyze these data and discover actionable insights which could improve their existing infrastructure and benefit their customers.

This chapter describes how the goals mentioned above are achieved, the tools used and how we contributed to improving the existing setup.

4.1 Background

In order to monitor how the data collection progresses across the current customers, log data are aggregated from all the stages of the data collection processes. Then, tools, described below, are used to analyze the logs, perform queries to extract insightful information and to display the results in a convenient way for the JSAS team in AWS Cloudwatch Dashboards.

4.1.1 Cloudwatch Insights

Cloudwatch Insights is a tool which allows users to write and execute queries on selected log groups using a SQL-like language to manipulate the log data and extract relevant insights.

A single query contains one or more query commands, which are separated using the “|” character. At the time of writing, there are only seven query commands which are supported: fields, filter, parse, stats, display, sort, and limit. Additionally, a number of different functions and operations, including date and time functions, string functions, numeric functions and regular expressions, are supported [9].

If a log event contains a JSON object as part of the log message, Cloudwatch Insights automatically extracts the fields of the JSON object, which can then be referenced in the query. For example, if a log message has the following format: “{customer_id: "12345"}”, the customer_id field could be accessed directly in the Cloudwatch query without having to parse it manually.

Figure 7 below shows an example of a Cloudwatch Insights query which uses some of the commands mentioned above, as well as regular expressions, in order to count how many log events matching particular filters are present. The results are aggregated by the group_name, jsasuniqueid, nfxsn fields, which uniquely identify a particular LWC data collection job.
4.1.2 Cloudwatch Dashboard

Cloudwatch Dashboards is a tool which allows developers to aggregate customized widgets and metrics on a single page. Thus, developers get a single access-point for monitoring all the necessary components in one place.

Cloudwatch Logs Insights queries can be added as widgets to AWS Cloudwatch Dashboards. An example of a dashboard containing widgets generated from Logs Insights queries is shown below:

Figure 8: Example of Cloudwatch Dashboard Log Insights widgets

4.1.3 Notation Used in Logs Widgets

Here are the terms that are often used in the Dashboard Widgets of the JSAS team:

- **group_name** - the name of a group of network devices, configured by the Juniper customer
- **jsasuniqueid** - unique ID string generated by JSAS for every data collection job
- **nfx_sn** - the serial number of a light-weight collector device
- **site_id** - ID of a customer’s site
- **customer_id** - ID of a customer

### 4.1.4 Log Groups Used by JSAS Team

Below are some of the log group names used by the JSAS team and which are mentioned in this chapter. The `<stage>` suffix at the end of the log group names can be either “dev” (development stage) or “prod” (production stage), depending on the type of account where the log group is present.

- `/aws/lambda/jsas-control-scheduled-request-<stage>` - the log group which generally receives log events related to regular, automatically-scheduled data collection jobs
- `/aws/lambda/jsas-control-config-update-<stage>` - the log group which generally receives log events related to manually scheduled data collection jobs
- `/aws/lambda/ig04-l-snspublish-<stage>` - the log group which generally receives logs indicating that data collected from a particular target device was uploaded to S3
- `/aws/lambda/jsas-control-response-handler-<stage>` - the log group which generally receives log responses from the LWC, indicating the status of the data collection on a particular device
- `/aws/lambda/jsas-control-sfdc-sender-<stage>` - the log group which generally receives log events indicating that a particular data collection job was completely finished

### 4.2 Methodology and Implementation

The JSAS team implemented a number of AWS Cloudwatch Dashboards and Logs widgets right before the beginning of our project. The Dashboards proved to be useful in fulfilling the goal of monitoring how the data collection proceeds on the beta customers.

In order to provide better context for the rest of the chapter, here are some of the Dashboard widgets that are commonly used by the LWC team and which were implemented prior to when we started working on the project:

**Scheduled Request Submit Counter** (log group: `/aws/lambda/jsas-control-scheduled-request-<stage>`) - this widget displays the number of target devices for a particular data collection job as seen in Figure 9 and Figure 10 below. Each job is uniquely identified by a combination of **groupname**, **jsasuniqueid**, **nfxsn**, **siteid** and **customerid**. The **totalTargetCount** field indicates the number of target devices. Additionally, the timestamp indicating when each job was scheduled is displayed.
Manual Request Submit Counter (log group: aws/lambda/jsas-control-config-update--<stage>) - this widget is similar to Scheduled Request Submit Counter, with the only difference being that it displays the jobs that were scheduled manually by the LWC team. The widget can be seen in Figure 11 below.

S3 Upload Counter (log group: /aws/lambda/ig04-l-snspublish--<stage>) - this widget counts the number of target devices for which the data collection ran successfully and the data were uploaded to AWS S3. The query aggregates the results by group_name, jsasuniqueid and nfxsn in order to count the number of
successful file uploads for each unique job that was scheduled. The widget can be seen in Figure 12 below.

Response-Handler Success Counter and Response-Handler Failure Counter (log group: /aws/lambda/jsas-control-response-handler-*stage*) - these two widgets are responsible for counting the number of responses received as successes and failures respectively. The “success” status of a response log event means that the data collection for a particular target device ran successfully, but it does not necessarily imply that the data collected was successfully uploaded to AWS S3. The “Fail” status means that data collection for a particular target device failed. The results are aggregated by group_name, jsasuniqueid, nfx_sn, site_id and customer_id in order to display relevant data (success_count and fail_count) for uniquely identified jobs. Additionally, the timestamp of the event is displayed. The widgets can be seen in Figure 13 below.
In addition to the Logs Widgets mentioned above, there are several other widgets, including:

- **LWC Runtime Errors, LWC Runtime Warnings** - display any and all errors and warnings, respectively, that were logged by the LightWeight Collector
- **Response-Handler Failures** - displays distinct log events, indicating the responses with “fail” status
- **Response-Handler All Logs** - displays all the log events that were sent to /aws/lambda/jsas-control-response-handler-<stage> log group
- **Response-Handler Duplicate(Cloud-initiated) Counter** - displays the number of duplicate responses received for a particular job
- **S3-Upload Duplicate Counter** - displays the number of duplicate S3 file uploads for a particular job

At the beginning of the development work on enhancing the AWS Cloudwatch Dashboards, we did not have a singular vision for developing particular widgets. Rather, with guidance from the JSAS team, we worked on implementing new Dashboard widgets in a reactive way as the need arose, keeping in mind the goal of making it easier to proactively identify any issues with the data collection processes.

Below is a description of the widgets that we developed as well as some details on how and why we created them.

### 4.2.1 Scheduled-Request Submit Counter Actual, Manual-Request Submit Counter Actual

As described above, Scheduled-Request Submit Counter and Manual-Request Submit Counter widgets already existed. They displayed the number of target devices for which data collection was supposed to be scheduled for a particular job. However, in order to provide better granularity, the data collection process, including scheduling the data collection for target devices, was further split into sequences of three target devices. Thus, for example, if the total target count of devices for a particular job is 15, the data collection process is split into 5 sequences of 3 devices. Below is the diagram displaying how the updated data collection process would proceed:
Thus, to accommodate for this change and to count the actual number of devices for which the data collection was attempted, we created two new widgets: Scheduled-Request Submit Counter Actual and Manual-Request Submit Counter Actual. Figure 15 below contains screenshots of these two widgets:
The only change compared to the existing Scheduled/Manual-Request Submit Counter widgets was that the `totalTargetCount` field was replaced with the `actualTargetCount`.

In order to create these new widgets, we needed to account for the modified structure of the log events. Each relevant log event indicates a particular sequence in the data collection process and includes a list of target devices. An example of the log event can be seen in Figure 16 below:
Figure 16: Example of a log event from /aws/lambda/jsas-control-scheduled-request-<stage> log group, displaying the new format containing the fields: sequenceId and targets

As is evident in Figure 16 above, the fields sequenceId and targets are present in these log events. Thus, our task was to write a query that would count the number of the actual targets across all the sequences aggregated by jsasuniqueid, group_name, customer_id, site_id and nfx_sn.

We started writing the query by filtering out all irrelevant log events using the following Cloudwatch Insights query command:

```
filter (@message like /Caching chunk to Redis/)
```

Next, from the filtered log events we retrieved a group_name field and, since there is no way to get the length of a JSON array (groups[0].targets) using the Cloudwatch Queries, we summed the results of multiple “ispresent(...)” query functions, which return 1 if a given field is present and 0 otherwise, to get the number of targets contained in each log event:
In the query statement above, we summed the results of three “ispresent(...)” functions since, at the time of writing, a maximum number of targets any given sequence of devices can have is three.

Next, in order to get the actual target count for each unique data collection job, we simply used “stats sum(num_targets)” command, aggregating the results by jsasuniqueid, group_name, nfx_sn, customer_id and site_id - the fields which Cloudwatch automatically extracted from the JSON objects of each log event. Additionally, we recorded the timestamp of the earliest event in the series in order to store a reference to when the data collection started for a particular job. Afterwards, we displayed the results using the “display” query command:

```
| stats sum(num_targets) as actualTargetCount, min(@timestamp) as earliest_event_time by jsasuniqueid, group_name, nfx_sn, customer_id, site_id
| display earliest_event_time, group_name, actualTargetCount, jsasuniqueid, nfx_sn, site_id, customer_id
```

4.2.2 Overall Response Distribution By Error Type

The next widget which we created queries over the /aws/lambda/jsas-control-response-handler-<stage> log group and shows the distribution of responses received by error type as seen in Figure 17 below.

The idea behind the implementation of the widget shown above is that a log event with “fail” status contains additional information about the error received. An example of a response log event with “fail” status can be seen in Figure 18.
Thus, if a response is successful, there will be no error type.

Implementation of the "Overall Response Distribution By Error Type" was straightforward: all we had to do was to extract the error type field from all the relevant log events and use the statement “\texttt{stats count(*) as num_targets by error_type}” to get the distribution of all responses aggregated by error type as shown in the complete query below:

\begin{verbatim}
fields payload.error.type as e_type |
filter (@message like /LWC response:/) |
# concat to avoid strange grouping issues with visualization |
fields concat(e_type, " ") as error_type |
stats count(*) as num_targets by error_type |
display num_targets, error_type
\end{verbatim}

Since we used the “\texttt{stats ...}” command, Cloudwatch allows us to visualize the results of the query. In order to do that, the “Visualization” tab needs to be selected in the Cloudwatch Insights view, as shown in Figure 19 below.
Figure 19: The results of executing the query. In the top left corner the "Visualization" button is located

After proceeding to the visualization section, and opening up the drop down tab as shown in Figure 20, a number of visualization options were presented: “Line”, “Stacked area”, “Bar” and “Pie”.

Since visualizing the query results as a pie chart would be the most convenient way for the JSAS team, we selected the “Pie” option. Thus, Figure 21 was rendered, which was added to the Cloudwatch Dashboards.
4.2.3 Successes and Failures By Target

Another widget that we implemented shows the number of successful responses and failed responses for a particular target device. The widget queries over the /aws/lambda/jsas-control-response-handler-<stage> log group. The widget can be seen in Figure 22 below.

This widget allows identifying potentially problematic target devices, if for example there are more failed responses than successful responses, or only failed responses. After identifying such targets, the members of the Juniper team can further investigate into what could possibly cause such behaviour.

The implementation of the query was straightforward: at first we retrieved the fields we needed, which were jsasuniqueid, nfx_sn, site_id, customer_id, target, status and group_name. Next, we filtered out all the irrelevant log messages, which should not be considered, using the “filter” command. Then, we simply used a series “stats sum(...)” functions to count the number of successes and failures for each target. After which, we displayed the query results. The complete widget query is as follows:
In addition to the “Successes And Failures By Target” widget, we also implemented two complementary widgets: “Successes And Failures By Target_status_over_time_MANUAL” and “Successes And Failures By Target_count_by_error_type_MANUAL”. The widget queries provide more detailed information about a specific target device. The “_MANUAL” suffix means that in order to use these widgets, one has to navigate to the query-view and enter some information manually. In case of the two widgets mentioned prior, group_name, nfx_sn, site_id, customer_id and target would need to be entered to display the relevant query results for the specified target.

In order to transition from the Dashboard widget into the query view for that particular widget, a “query” icon in the top right corner of the widget, as highlighted below, needs to be clicked.

Figure 23: Successes and Failures By Target_status_over_time_MANUAL widget with the “query-view” icon highlighted as green in the top right corner

The “Successes And Failures By Target_status_over_time_MANUAL” widget provides, as the name suggests, a distribution of the statuses of the responses received for a particular target device over time. After proceeding to the query view of the widget, entering the identification information for a particular target device, running the query and rendering the results of the query as a bar chart, the visualization will look similar to what is seen in Figure 24.
Figure 24: Results of executing the query, which displays distribution of response-statuses over time. On the left side most recent events are rendered.

The complete query for the widget above is as follows:

```plaintext
fields payload.jsasuniqueid as jsasuniqueid, payload.status as status, payload.customer_id as customer_id, payload.site_id as site_id, payload.nfx_sn as nfx_sn, payload.sequenceId as sequenceId, payload.target as target, payload.error.type as error_type
| parse payload.redis_data_handoff_key
/\^.*\([-\w\{36}\)]\<(?group_name>.*)\._.*\([-9]{1,3}.\{3}\{0-9}{1,3}\_\_.*$/

# enter the group_name, nfx_sn, site_id, customer_id and target
# of the device, to execute the query
| filter (group_name="" and nfx_sn="" and site_id="" and customer_id="" and target="")
| stats sum(status="success") as success_count, sum(status="fail") as fail_count by bin(1s), error_type
```

When writing the query, we used a “`bin(...)`” function in order to separate the results of the “`stats sum(...)`” function into distinct time bins, which have a span of 1 second. We ended up using such an approach because Cloudwatch would not let us visualize the results if the “`stats ...`” command is not used in the query.

The second complimentary widget is “Successes And Failures By Target_count_by_error_type_MANUAL”. It is very similar to the “Overall Response Distribution By Error Type” widget described above. The only difference being that it displays the response distribution by error type for a particular device, information about which needs to be entered manually by navigating to the query mode of the widget.
4.2.4 SFDC-Sender Done Events

Another widget we implemented is “SFDC-Sender Done Events”, which displays the log events received in the /aws/lambda/jsas-control-sfdc-sender-<stage> log group. The widget can be seen in Figure 25 below.

![Figure 25: SFDC-Sender Done Events widget](image)

The log events received in that particular log group indicate finished data collection jobs. The start_time represents the time when the data collection job was scheduled, the end_time represents when it was completed and the field took_to_complete_h_m_s displays the time it took to complete the data collection job in the following format “hours : minutes : seconds.milliseconds”.

Here is an example of what the main part of a typical log event in the /aws/lambda/jsas-control-sfdc-sender-<stage> log group looks like:

```json
{
  "sfdcuniqueid": "NA",
  "jsasuniqueid": "34e1293e-e484-d8af-4123-b7b9ad598667-1",
  "totalTargetCount": 21,
  "timestamp": "2021-03-12T03:06:08.355Z",
  "version": "1",
  "typename": "newmeasurementupdate",
  "customerid": "0100272020",
  "siteid": "0100272020",
  "nfxsn": "DD1120AN0135",
  "groupname": "Bastion",
  "targetname": "Bastion",
  "result": "targetfailure",
  "details": []
}
```

![Figure 26: Example of a main part of a typical log event in the /aws/lambda/jsas-control-sfdc-sender-<stage> log group. The log event indicates that a particular data collection job was finished](image)
When writing the query for the widget, the delicate part was creating the derived field “took_to_complete_h_m_s”. We used the \texttt{@timestamp} of when the log event was received as an \texttt{end\_time}. This \texttt{@timestamp} field is derived by the Cloudwatch itself and is represented by the number of milliseconds since the Unix Epoch (January 1, 1970) elapsed when the log message was written. However, for the start time of the data collection, we are using the \texttt{timestamp} field from the log message itself, which is represented as a string. Thus, at first we considered converting the string \texttt{timestamp} into the number of milliseconds since the Epoch. However, the algorithm for making the computation was too complicated to implement in the Cloudwatch Insights query. That led us to a different idea, based on the following observation: whenever a log event is sent to a log group, the string representation of \texttt{@timestamp} of when the message was logged is also displayed as part of the \texttt{@message} field of the log event as seen in Figure 27 below.

Thus, our approach was to parse it from the log event as a string and then convert both the string representations of end timestamp and start timestamp of log events into respective number of milliseconds.

The following query filters out all irrelevant log events and parses out the \texttt{end\_time} from the \texttt{@message} field of each log event:

\begin{verbatim}
filter @message like /tokenRequest sqsBody/ and typename like /newmeasurementupdate/
| fields timestamp as start_time # store the timestamp variable as start_time to have meaningful naming conventions
| parse @message "*Z" as end_time
\end{verbatim}

Next, we extracted years, months, days, hours, minutes, seconds and milliseconds from both the \texttt{start\_time} and \texttt{end\_time} using the following query statements:

\begin{verbatim}
| parse start_time "*-*-T*:.*Z" as start_year, start_month, start_day, start_hours, start_minutes, start_seconds, start_milliseconds
| parse end_time "*-*-T*:.*" as end_year, end_month, end_day, end_hours, end_minutes, end_seconds, end_milliseconds
\end{verbatim}
Next, in order to calculate the difference between these two dates, we converted both the times into the millisecond representation, such that we would easily be able to find their difference:

```sql
# convert the extracted fields into milliseconds for easier calculations
| fields start_year * 31556926000 + start_month * 2629743000 + start_day * 86400000 +
  start_hours * 3600000 + start_minutes * 60000 + start_seconds * 1000 + start_milliseconds as
  start_time_milliseconds
| fields end_year * 31556926000 + end_month * 2629743000 + end_day * 86400000 + end_hours *
  3600000 + end_minutes * 60000 + end_seconds * 1000 + end_milliseconds as
  end_time_milliseconds
```

Next, we subtracted `start_time_milliseconds` from `end_time_milliseconds` and converted the resulting number of milliseconds into number of hours, minutes, seconds and milliseconds, after which we displayed the results using the following query:

```sql
| fields start_year * 31556926000 + start_month * 2629743000 + start_day * 86400000 +
  start_hours * 3600000 + start_minutes * 60000 + start_seconds * 1000 + start_milliseconds as
  start_time_milliseconds
| fields end_year * 31556926000 + end_month * 2629743000 + end_day * 86400000 + end_hours *
  3600000 + end_minutes * 60000 + end_seconds * 1000 + end_milliseconds as
  end_time_milliseconds
| fields end_time_milliseconds - start_time_milliseconds as difference, floor(difference /
  3600000) as diff_hours, difference - diff_hours * 3600000 as hour_reminder,
  floor(hour_reminder / 60000) as diff_minutes, hour_reminder - (diff_minutes * 60000) as
  minute_reminder, minute_reminder / 1000 as diff_seconds
| fields concat(diff_hours, " : ", diff_minutes, " : ", diff_seconds) as
  took_to_complete_h_m_s
| display start_time, end_time, took_to_complete_h_m_s, groupname, totalTargetCount,
  jsasuniqueld, nfxsn, siteId, customerId, @message
```

One issue that we missed out on with the approach mentioned above is that the difference between `end_time` and `start_time` may end up being incorrect due to the generalizations we made when converting the number of years and months into milliseconds. Since Cloudwatch Insights does not support conditional logic, we assumed that one month contains an average of 30.44 days and one year contains an average of 365.24 days. Thus, for example, when the `end_time` of a log event is “2021:06:01:00:00:00” and `start_time` is “2021:05:31:23:00:00” the `took_to_complete_h_m_s` value will be negative.

After researching the possible solutions for the issue mentioned above, we concluded that with the current functionality of Cloudwatch queries it seems impossible to solve the problem with the current format of the log events. However, if the string representation of a `timestamp` field in a log event was
replaced with the number of milliseconds since the Epoch on the back-end by the JSAS team, the problem of finding the difference between the two timestamps would become trivial.

### 4.2.5 S3 Uploads Vs Successful Responses

The JSAS team noticed that in certain cases, the number of S3 uploads was different from the number of successful responses received for particular data collection jobs. We were tasked to create a widget that would show the number of successful responses alongside the number of S3 uploads for a particular job, as well as calculate a difference between the two columns.

![Figure 28: S3 Uploads Vs Successful Responses widget](image)

The interesting point about creating this widget is that we had to query over two different log groups: `/aws/lambda/jsas-control-response-handler-<stage>` and `/aws/lambda/ig04-l-snspublish-<stage>`, which both contained log events having the necessary information for uniquely identifying data collection jobs, but the format of the respective log events was different as seen in Figure 29 and 30 below.

![Figure 29: The log event from /aws/lambda/jsas-control-response-handler-<stage>, indicating a successful response received from LWC for a particular target](image)
As we can see from the two examples of the log events, the values `jsasuniqueid`, `nfx_sn` and `group_name` are present in both types of log events, but are stored differently. In the `/aws/lambda/jsas-control-response-handler-<stage>`, these fields, including `status`, which is specific to the response-handler log group, are stored as part of the payload object. Thus, in the query, the fields can be accessed as `payload.jsasuniqueid`, `payload.status` and `payload.nfx_sn`. The `group_name` would have to be parsed out from the `payload.redis_data_handoff_key` field. However, in the format for log events in the `/aws/lambda/ig04-l-snspublish-<stage>` log group, all the fields which allow to identify individual data collection jobs would have to be parsed out from `Records[0].s3.object.key`.

When writing the query, we needed to keep in mind some of the limitations of the Cloudwatch Insights: the logic of Cloudwatch queries is not similar to typical SQL logic. It is not possible to run a subquery on the first log group to aggregate the results in the format that we want, then run a different subquery on the second log group and then join the results of the two queries. The log events coming from multiple log groups cannot be processed separately, but rather uniformly as a single aggregation of logs despite the fact that they have different formats.
With the Cloudwatch limitations in mind, we started writing the query by filtering out all the irrelevant log events:

```
| filter @message like /LWC response:/ or (@message like /object/ and @message like /eventSet/)
```

Next, we needed to parse out the `jsasuniqueid`, `group_name` and `nfx_sn` from the aggregation of the log events from both of the log groups. Additionally, the `status` field, which only log events from the response-handler log group contain, was parsed out. Since, as mentioned above, the log events had different formats, we had to parse the fields separately and store them in different columns. The following query snippet shows how `jsasuniqueid`, `group_name` and `nfx_sn` and `status` were parsed out from response handler log events:

```
fields payload.jsasuniqueid as jsasuniqueid_unparsed,payload.status as status,payload.nfx_sn as r_nfx_sn,payload.redis_data_handoff_key as redis_data_handoff_key
# parse group_name from response handler logs
| parse redis_data_handoff_key
/^.*([-\w]{36})(?<r_group_name>.*).*$/$
```

Next, we retrieved `jsasuniqueid`, `group_name` and `nfx_sn` from the log events coming from the “S3 uploads” log group:

```
# parse information from s3 upload counter logs
| parse Records.0.s3.object.key
/\s__\([<s3_jsasuniqueid][-\w]{36}]\)\([<s3_group_name>.*]\)\([<s3_nfxsn>.*]\)\([<s3_time>.*].xml/\)
```

And since the `jsasuniqueid` field coming from the response-handler log events contained "-<number>" in the end and the S3 upload log events didn’t, we used a regular expression to extract the main part of the `jsasuniqueid` field, which was obtained from response handler log events:

```
# make sure the jsasuniqueid does not contain -<number> at the end to have same format as jsasuniqueid field in S3 log events
| parse jsasuniqueid_unparsed /(?[r_jsasuniqueid][-\w]{36})/)
```

After executing the created query, we ended up having the following results:
Figure 31: The query results after parsing out the fields that we need from log events coming from the “response-handler” and “S3 uploads” log groups

At this point, in order to use the `stats sum(status='success') as success_count, sum(status='') as s3_uploads` command, which would count the number of items with successful status and count the number of items with blank status (indicating the successful S3 uploads), we need to make sure that the results can be aggregated by `jsasuniqueid`, `group_name` and `nfx_sn`. However, with the fields displayed as they are in the Figure 31 above, this would be impossible. Thus, we needed to merge the relevant fields to achieve uniformity of data. In order to achieve this task, we used the “concat(...)” function, which concatenates the given strings or fields. Thus, by running “concat(r_jsasuniqueid, s3_jsasuniqueid) as jsasuniqueid”, the fields `s3_jsasuniqueid` and `r_jsasuniqueid` of each row of the query results shown above would be concatenated and stored in the `jsasuniqueid` column. Using this approach we would be able to convert the fields we generated to the same naming format. Here is the query which performs the concatenation:

```
| fields concat(r_jsasuniqueid, s3_jsasuniqueid) as jsasuniqueid, concat(r_group_name, s3_group_name) as group_name, concat(r_nfx_sn, s3_nfxsn) as nfx_sn, concat(status, "") as merged_status
```

After adding the statement to the previously created query and displaying the newly derived variables, we got the results shown in Figure 32.
Figure 32: The results of executing the query to concatenate respective fields: jsasuniqueid, group_name, nfx_sn and status of log events from two different log groups.

Finally, we used the query command “stats sum(...)” in order to sum the number of items with merged_status = “success” (successful responses) and merged_status = “” (S3 uploads) by jsasuniqueid, group_name and nfx_sn. After which, a filter statement is applied to make sure all the results presented are relevant:

```
| stats sum(merged_status='success') as response_success_count, sum(merged_status='') as s3_uploads by jsasuniqueid, group_name, nfx_sn |
| filter (response_success_count > 0 or s3_uploads > 0) |
```

This way, we obtain the number of successful responses and the number of S3 uploads for each job. Then, we simply subtract the number of s3 uploads from the number of successful responses to get the newly derived field which would indicate the difference. Finally, we display the results:

```
fields response_success_count - s3_uploads as response_upload_mismatch
| display group_name, jsasuniqueid, nfx_sn, s3_uploads, response_success_count, response_upload_mismatch |
```

Thus, we were able to run a single query on two log groups, which contained differently formatted log events. The technique of running a single query on multiple log groups, containing differently formatted messages can be extensively used by the Juniper team in the future to create more complex widgets.

4.3. Evaluation and Results

Despite the fact that during development of the widgets, we encountered some limitations in the functionality of Cloudwatch Queries, we implemented a number of powerful widgets which are already being used by the JSAS team to proactively identify issues. Additionally, we found a relatively unique way
to write Cloudwatch Queries, which processes log events from multiple log groups, even if the respective log events have different formats. This may end up being a powerful technique for creating new widgets or extending the functionality of existing ones, which would allow more convenient means of discovering insights about the data collection processes.

4.4. Future work

There are three areas of work we identified for further development.

The first area of work is related to using the technique of running a query on multiple log groups, in which log events have different formats. The JSAS team can apply the methodology to aggregate some of the existing widgets such as “Scheduled-Request Submit Counter”, “Manual-Request Submit Counter”, “Scheduled-Request Submit Counter Actual”, “Manual-Request Submit Counter Actual”, “S3 Upload Counter”, “Response-Handler Success Counter”, “Response-Handler Failure Counter” into a single widget, which would provide a single access-point for viewing the data that is currently displayed by the six widgets mentioned above.

Additionally, the potential inconsistent behavior with the took_to_complete_h_m_s field in the “SFDC-Sender Done Events” widget may need to be fixed. The current approach is prone to causing inconsistent behavior because of generalizations we had to make when converting the string representations of the start and end timestamps into the respective number of milliseconds manually due to some of the limitations of the Cloudwatch queries. A potential solution would be to change the format of the log events, which are sent to the /aws/lambda/jsas-control-sfdc-sender-<stage> log group, to instead have the millisecond representation of the timestamp of when the data collection job was scheduled.

The last area of future development is related to the “Scheduled-Request Submit Counter Actual” and “Manual-Request Submit Counter Actual” widgets. When implementing the respective widgets, we assumed that the maximum number of target devices any sequence can have is three. However, if the maximum number of target devices in a sequence increases, the widget queries will need to be updated accordingly. Thus, we recommend the JSAS team to update the widget queries to account for the maximum potential number of target devices any given sequence could have (e.g. ten), such that the possible issue mentioned previously would never be encountered.
5. Developing Anomaly Detection Solutions

Due to the complexity and scale of the LWC project, anomaly detection is a difficult task. With hundreds of log messages written per hour, errors can easily go unnoticed and difficult to debug even when discovered. To better discover errors and display relevant debug information, we developed a series of tools to automatically collect data, distill the important information, and send notifications to the JSAS team when errors occur.

5.1 Background

We used a number of tools and technologies described above in the background section of this report to develop anomaly detection solutions. This section provides a brief overview of technologies not previously described in detail.

5.1.1 Microsoft Teams Webhook

Microsoft Teams Incoming Webhooks are a feature of Microsoft Teams that allow users to send notifications to Teams channels from external sources [6]. In the LWC project, the JSAS team uses Microsoft Teams to communicate. As a result, Microsoft Teams are a sound media for the whole team to receive notifications. There is a separate channel named “Notification Test” in the JSAS team’s workspace specifically for receiving messages related to issues found in device logs. As can be inferred from the name, the channel was created for testing the notifications in development mode.

5.1.2 Subscription Filter

Subscription Filters is a feature that Cloudwatch Log Groups provides to detect the log events matching configured filter patterns. Once there is a match, a subscription filter can trigger other AWS services, such as Lambda functions with the log events matching predefined filter patterns as parameters. When developing anomaly detection solutions, we used Subscription filters to trigger the execution of respective anomaly detection components. An example can be seen in Figure 33 below.
5.1.3 AWS Step Function

AWS Step Functions are useful for defining a series of actions to be performed automatically. This makes them a natural choice for use as part of a workflow for detecting anomalies in the LWC data collection pipeline. Specifically, we use Step Functions to configure and specify the order of events as well as the timing between them.

5.1.4 Cloudwatch Alarm

Cloudwatch Alarm is a Cloudwatch service which can notify the developers when a log fits the pattern of a specific metric filter. Developers can set the threshold of metrics. If the number of the log events matching the filter pattern exceeds the threshold in a given time period, the Alarm warns the developers and transitions from “OK” to “In alarm” state [5]. Thus, the service helps developers monitor problematic behavior. For some of the components we developed, we not only configured sending relevant notifications to Microsoft Teams but also set up alarms to provide an additional way for the JSAS team to get notified about relevant issues.

5.2 Methodology and Implementation

We were asked to notify the JSAS team of three types of errors through Microsoft Teams. In section 5.2.1, we introduce the first issue, the mismatch between requested target count and responses. In section
5.2.2, we outline the second issue, the runtime warning due to allocated memory being consumed. In section 5.2.3, we discuss the last issue, the invoke handler error.

### 5.2.1 Mismatch Between Requested Target Count and Responses

One common failure that occurs while running LWC data collection jobs is that target machines fail to respond to the Lightweight Collector request for data. This can occur for a variety of different reasons, common ones being that a target machine is offline, or that the network is configured in such a way that the machine is unreachable from the Lightweight Collector. As a result, it is common for some targets in LWC data collection jobs to fail to respond, meaning that the number of responses received by the LWC is lower than the number of requests sent. On the other hand, it is also possible to receive more responses than requests; this is a different failure scenario that typically means one or more targets responded multiple times.

It is important to note that either type of failure can be difficult to detect or interpret. Some targets take longer to respond to queries than others - response times can range from fractions of a second all the way up to several hours. Therefore, it can be difficult to distinguish a situation in which a target is actually unreachable by the LWC and one where a response is on its way, but taking a long time to reach the LWC. For this reason, a lengthy delay is needed between when a data collection job is initiated and when the results are reported.

If either of the above scenarios occurs, the JSAS team wants to know about it so they can inspect the cause and impact of the error. To accomplish this, we developed an application, illustrated in Figure 35 below, that monitors for mismatches in the number of requested targets and responses received:
Figure 35: The process that checks for mismatches in the number of requests sent to vs. responses received from target machines.

The first step of the process is an AWS EventBridge filter rule that looks for a specific pattern in a CloudWatch log group that indicates when an LWC data collection run is initiated. When such a log message is written, the EventBridge rule triggers the first Lambda function in this workflow, passing the log message that matched the rule as input. This log message contains information about the LWC data collection job, such as the JSAS unique ID, which uniquely identifies the job. The Lambda function then invokes an AWS Step Function, passing along the contents of the log message as well as an amount of time in seconds that the step function should spend in a “wait” state before continuing.

The next part of the process is the AWS Step Function state machine, which takes as input a log message and the amount of time it should spend waiting. This delay exists so that target machines have time to respond to a query from the LWC before the number of responses is counted and reported. This amount of time is configurable via an environment variable in the Lambda function depending on how long the JSAS team wants to allow for responses to be received. The structure of the step function, alongside an input event containing the contents of a log message, is shown in Figure 36 below.
This Step Function is simple; its main function is to add a configurable delay before the execution of the Lambda function that does the actual comparisons between the number of requests sent and responses received. The reason we chose to use a Step Function rather than simply using a single function activated by a CloudWatch log message with a timeout to programmatically implement this delay is that Lambda functions incur cost based on the time spent executing. Using a timeout in code would result in hours of idle time being charged to the JSAS team’s account while waiting for responses to be received. In contrast, the cost of using Step Functions is based on the number of transitions between states, which makes it a much more cost-effective solution for implementing lengthy delays.

The final step of this workflow is the AWS Lambda function invoked by the Step Function that actually parses the log events in order to verify the number of targets requested versus the number of responses received.

### 5.2.2 Notification When Allocated Memory Consumed

Among all runtime warnings received from the Lightweight Collectors installed at customers’ sites, some warnings exist because the local Lambda functions running in the Lightweight Collectors use up all allocated memory while executing data collection jobs. In this situation, the customers’ devices return logs specifying that the allocated memory was consumed. Those logs are gathered in the /aws/greengrass/GreengrassSystem/runtime log group. Because of the relative newness of Greengrass, a software component installed on Lightweight Collector devices allowing execution of local lambda functions, not all of the errors are documented. Thus, the JSAS team would like to get notified about any such warnings, so that they can investigate what caused them in a timely manner.
To identify this type of error, we set up a metric filter under the /aws/greengrass/GreengrassSystem/runtime log group first. The filter pattern is the same as the log error-message received, which is ‘Worker consumed all allocated memory! Memory Usage (KB)’. Based on that metric filter, we created a CloudWatch Alarm, which we set to run on a five-minute interval. The alarm is triggered whenever a warning, indicating that the allocated memory was consumed, occurs.

![Runtime Warning (No Memory)](image)

Figure 37: The graph of the Alarm

Next, we created a subscription filter on the /aws/greengrass/GreengrassSystem/runtime log group to connect with an AWS Lambda function. This subscription filter detects the log events in the runtime log that fits the pattern, encodes the log messages, and sends the encoded information to the target Lambda function. This function decodes the log messages sent from the subscription filter and sends a relevant notification message to a Microsoft Teams channel via webhook. This allows a notification directly to the JSAS team whenever a runtime warning matches the filter pattern. Figure 38 below shows an example of a "Allocated Memory Consumed" notification message sent to Microsoft Teams.

![Figure 38: Example of decoded log message received in the Microsoft Teams channel](image)
5.2.3 Invoke handler error

Since the LWC is based on AWS services, there are a lot of logs related to services provided by Lambda Functions. All logs related to Lambda functions that come from different regions are summarized into six log groups starting with alpha00 (Figure 39). A commonly occurring error is called “Failed to invoke handler”. It happens when a type error occurs in the Lambda function, more specifically, when it cannot read property ‘then’ of undefined. It is important to identify whenever this error occurs, as it does not appear consistently and the cause of the error is still unknown. The more often the JSAS team can observe the error occurring, the more likely it is that they will be able to identify the cause of the error and subsequently fix it.

Figure 39: Alpha00 log groups

In order to better record occurrences of the error, we created a webhook to a Microsoft Teams channel and sent notifications from AWS. To separate the failure of invoke handler error from other notifications, an Incoming Webhook connector is added to this channel, named as “Failed to invoke handler error”. All notifications related to the error are sent by this Webhook Connector.

In the AWS Cloud side of this component, a subscription filter is configured for each “Alpha00” log group mentioned above. The filter pattern used to identify the error is “Failed to invoke handler”. This filter is created as a Lambda subscription filter. Hence, it triggers the Lambda function that sends the relevant notification to the appropriate Microsoft Teams channel.

5.3 Evaluation and Results

When developing and maintaining Juniper’s services, debugging based on received logs is important work but sometimes can be difficult. Manually monitoring logs from hundreds or thousands of devices all day is not realistic. To assist the JSAS team in anomaly detection, we developed automated methods to notify them when relevant errors occur. By sending the relevant notifications to the Microsoft Teams channel, indicating the issues discussed in this chapter, the components we implemented empower the JSAS team to track the errors and work towards resolving them in a timely manner.
As the Customer Support Automation initiative grows, the need for automated error detection will only increase. As more and more instances are deployed, the quantity of log messages will become too great to debug manually. Therefore, the functionality that we developed will become increasingly important.

### 5.4 Future Work

It is likely that the solutions we developed will need to be modified in order to handle the increased scale of the Lightweight Collector in the future. For some services, AWS places limits on the total usage or number of actions per unit time. Currently, these quotas are more than enough for our solutions to work, but we anticipate that in the future, increased load will lead to these quotas being exceeded. For instance, there is a default limit of 10 concurrent CloudWatch Insights queries per AWS account. According to AWS documentation, the quotas for most services can be raised by requesting a higher limit from AWS. We encourage Juniper to proactively work to increase these limits as the LWC infrastructure expands so that the anomaly detection solutions we developed continue to work as expected.

Another area that potentially warrants future development would be some hierarchical system of organizing warning and error messages. Our work concerned sending notifications to a Microsoft Teams channel only for error messages, but in the future, the JSAS team might consider creating a separate channel dedicated to warning messages. While typically non-critical, warning messages can be just as important to investigate and fix as error messages.
6. Designing Scalable Solutions

Scalability is one of the most important components when it comes to creating enterprise solutions. That is why there was a particular emphasis from the JSAS team on the scalability of the components we’re creating. The chapter describes how we contributed to the goal of creating scalable components as part of our project.

6.1 Background

Juniper Networks currently has 30,000 enterprise customers, and the number may grow in the future. Despite the fact that at the time of writing, the support automation solution provided by the JSAS team is being tested with only a handful of beta customers, the goal in the future is to extend the service to all of the Juniper customers. As the number of customers joining the support automation initiative grows, so does the size and complexity of the JSAS systems and AWS infrastructure. Thus, we focus on creating scalable solutions.

6.2 Methodology and Implementation

Creating and configuring software components are not the only parts of the software engineering lifecycle. Provisioning, maintaining and keeping the deployed components up-to-date are equally important challenges that need to be overcome.

6.2.1 Provisioning Implemented Components with Cloudformation Template

Once software components are deployed to AWS, updating or provisioning all their relevant resources and settings manually is a time-consuming and error-prone process, as detailed resource settings need to be considered. For these scenarios, the AWS Cloudformation Template tool provides an efficient way to organize, maintain and provision the created software components by describing the relevant AWS infrastructure as code in a Cloudformation template file. By deploying the template, AWS automatically creates an application stack with all the related AWS resources as a single unit. Cloudformation configures and connects the resources according to the description contained within the template. Since the resources, which are created as part of the stack, are managed as a single unit, destroying the stack automatically removes all the respective resources. Additionally, when updating AWS resources in the template file and redeploying it, only the updated resources are recreated.

Thus, to make sure our software components are easy to maintain and provision, we created a Cloudformation Template describing the AWS resources that are used and how they are set up in the respective software components. Among the resources used are: Lambda functions, IAM Roles, Permissions, Subscription Filters, State Machine, and resources related to API Gateway. Figure 40 below
is a snippet of how some of the resources for ResponseVsTargetCount Lambda function are declared in the YAML template file. The Lambda compares the number of responses received with the number of scheduled devices for a particular job and if there is a mismatch, sends a relevant notification to MS teams.

![Figure 40: Example of how some of the resources are specified for ResponseVsTargetCountNotification Lambda in a Cloudformation template](image)

After deploying the Template, AWS creates a stack with all of the declared resources. Next, we verified that the behavior of our solutions remained unchanged after deployment. Afterwards, to make sure the JSAS team members would easily be able to get up to speed with the solutions we created, we documented how each component works and how each component is provisioned using the Cloudformation Template.

### 6.2.2 Version Control and Stack Checking

Version control software assists in organizing and compartmentalizing different components of software projects and is especially helpful when multiple developers are working on the same (or related) components. Breaking up a project into discrete pieces based on functionality and separating each piece into its own Git repository makes it easier to keep track of changes as they are made, and even revert to previous versions if necessary. Using a service such as Github or GitLab to host Git repositories on a web server allows developers to view code changes that others make, as well as share their own changes with others, and allows each developer to easily keep software up to date as changes are made. Version
control is helpful for organizing and keeping code up to date in a single central location; however, the code also needs to be deployed periodically as changes are made so that the live application reflects these changes.

One problem that arises when developing and deploying a cloud infrastructure on a wide scale is that versions of components (or entire applications) may not be uniformly deployed across multiple client AWS accounts. Furthermore, the JSAS team uses multiple “stages” for development work, testing, and the live application, resulting in even more versions of code to keep synchronized. It is entirely possible for a developer to make and deploy changes to one piece of an application that results in an error or incompatibility with another part of the application. As different versions of components are developed and deployed at different times by different people, it becomes increasingly likely that at some point, versions will mismatch and something will break as a result. For this reason, our team developed a tool that inspects application stacks deployed to AWS with CloudFormation templates and compares them against the corresponding code repositories deployed to GitLab.

By default, applications deployed with CloudFormation templates do not contain any Git version information. In order to keep track of Git version information, the JSAS team uses tags on the deployed stack. These tags indicate both the branch that this stack is deployed from as well as the commit ID, which specifies what particular version this code is. An example of this is shown in Figure 41 below:

![Table of tags](image)

**Figure 41: An example of tags applied to a CloudWatch stack**

Using tags to keep track of version control information, code deployed to the AWS cloud with CloudWatch can be tracked and compared to the corresponding repository on GitLab.

We designed our solution to meet the following requirements for the CloudFormation stack checking functionality:

- Be able to check information about any CloudFormation stack in an AWS account
- Support checking for a commit ID match if a stack was deployed from the given Git branch
• If a deployed stack is on the same branch as the given branch and does not match the given commit ID, send a notification to a channel in Microsoft Teams
• Automatically check deployed CloudFormation stacks once a day

With these requirements in mind, we use an AWS Lambda function to do the actual checking, which is then activated periodically by an EventBridge rule using a CRON expression. However, due to the constraint that GitLab was only accessible via the Juniper Networks VPN, which AWS does not have access to, we altered our design to include an AWS API Gateway endpoint so that the Lambda function could instead be invoked via HTTP request, which can be instead be triggered periodically by Jenkins, a CI/CD tool that does have access to the necessary Git version control information in GitLab.

Creating an API Gateway endpoint provides the additional benefit of decoupling the Lambda function from any particular AWS trigger or service, which makes invoking it simple. If the needs of the development team change in the future, the function can just as easily be triggered by a tool other than Jenkins, or even manually, via the API endpoint. Figure 42 below provides an illustration of the workflow for comparing GitLab commit IDs.
Due to the limitation that GitLab is only accessible via the Juniper VPN, we were unable to fully implement the workflow; the components we developed are highlighted by the red box in Figure 42. However, because the Lambda function is triggered via an API endpoint, connecting the workflow to Jenkins (or any other tool) in the future is simple. Any application can make a request to the API endpoint to trigger the stack check, using the syntax shown in Figure 43 below to specify the names of stacks to check and the expected Git branch and commit ID for each.
When a request is made to the API Gateway endpoint, it activates an AWS Lambda function that inspects the tags of the deployed CloudFormation stacks with the provided project names and compares them to the given commit IDs and branch names. If a stack was deployed from the same branch as the branch specified in the request, and a mismatch between the commit IDs is found, the function sends a notification to a Microsoft Teams channel with the name of any stacks found to not match the expected commit ID. An example of such a message is in Figure 44 below:

![Figure 43: The syntax used to check CloudFormation stacks for a particular Git version.](image)

**Figure 44:** A notification sent to Microsoft Teams that the version of a deployed CloudFormation stack differs from the version provided.

With this tool, numerous applications hosted in GitLab and deployed as CloudFormation stacks can be periodically checked for being up to date, and if they are not, the JSAS team is immediately notified. This helps alleviate issues arising from incompatible versions or outdated code, and assists in tracking versions of different LWC deployments for different customers to ensure everything works as expected.

### 6.3 Evaluation and Results

To achieve the scalability of the software solutions we created, we declared all the relevant resources and described how they are set up in a Cloudformation Template file. Thus, our work can be managed and deployed to the production stage as a single unit. Additionally, we wrote detailed documentation of all of the components that we created, and together with the source code and the Cloudformation Template, we
added the final deliverables to the version control system. Furthermore, we created a tool which aims to ensure version consistency of each deployed CloudFormation stack.

### 6.4 Future Work

One area for potential future work in terms of the scalability aspect of this project would be to fully set up the stack checking function to trigger automatically, using a CRON job or similar. As discussed above, due to limitations with GitLab only being accessible via Juniper VPN, we were not able to automatically schedule our stack version checking functionality. However, the JSAS team is currently in the process of moving GitLab to a new hosting location, which should be accessible from AWS. Therefore, it should soon be possible to automatically check the stack versions at a set interval.
7. Conclusion

As the LWC project grows and more clients sign up for the customer support automation initiative, the tasks of monitoring data collection processes, detecting problematic behavior in a timely manner and scaling out the infrastructure become increasingly important. Thus, these were the areas we focused on as part of our project.

By adding a number of new widgets to the existing AWS Cloudwatch Dashboard of the JSAS team, we improved how the data collection processes are being monitored. Additionally, when developing the widgets, we discovered a method of running a single query on multiple log groups containing differently formatted log events. This knowledge can be further used by the JSAS team when developing new widgets or updating existing ones.

To assist the JSAS team in anomaly detection, we developed automated methods to notify the JSAS developers whenever abnormal behavior occurs. We focused on detecting two types of issues. The first type is related to LightWeight Collectors’ local Lambda function execution flaws. The second type is related to issues caused when the number of scheduled target devices is different from the number of responses received for a particular data collection job. By sending relevant notifications to the Microsoft Teams channel, the components we implemented empower the JSAS team to track the issues and work towards resolving them in a timely manner. Additionally, the methodology we used for developing the anomaly detection solutions can be further extended by the JSAS team to monitor new issues that arise as the LWC project grows.

In order to ensure our work remains useful in the future, we provisioned the created components by describing all their relevant resources in AWS Cloudformation Template. Additionally, we developed a solution which aims to ensure that the source code of each deployed Cloudformation stack remains up-to-date with the source code of the respective GitLab project. Thus, our contributions help ensure the continuous scalability of the LWC project.

In short, we improved data collection processes monitoring, created automated methods to detect problematic behavior, and created a solution which aims to keep all deployed stacks up-to-date. Furthermore, to account for scalability, we provisioned each component we developed using CloudFormation templates.
8. References