Major Qualifying Project Proposal

A Major Qualified Project Proposal

Submitted to the Faculty of

Worcester Polytechnic Institute

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

By

Maria Del Carmen Sacristan Benjet - Computer Science
Pradnya Mahurkar - Computer Science
Tyler Bouwens - Computer Science

Advisor
Professor Mark L. Claypool

Mentors
John Sadlier
James Austin
Nikhil Jukar

*This report represents the work of one or more WPI undergraduate students submitted to the faculty as evidence of completion of a degree requirement. WPI routinely publishes these reports on the web without editorial or peer review
Abstract

The production team at Juniper Networks developed the Juniper Support Automation Solution (JSAS) to track the status of network devices and troubleshoot issues that their clients may face. Their AWS CloudWatch service receives tens of thousands of log messages from network devices at client locations daily. To make the troubleshooting and maintenance process more efficient and easier to use, the production team had set up a dashboard to select relevant data and display it in the form of log table widgets. However, with the increasing scale of the JSAS system, the widgets in the existing dashboard are cluttered with information from multiple sources and make it difficult to utilize the tool. To address this problem, we developed an Infrastructure as Code (IaC) to build filtered dashboards by device serial numbers and built a new widget to display data from multiple sources parallely in one log table. The IaC tool was created using an AWS CloudFormation template and leveraged the nested stack features, allowing the user to build up to 1000 dashboards. The widget was developed using the AWS CloudWatch query language and was placed at the top of every dashboard, providing a summarized view of the current state of all Juniper devices, helping improve the developer’s user experience with the dashboard.
# Table of Contents

Abstract ................................................................................................................................. 1

1 Introduction ......................................................................................................................... 3

2 Background ......................................................................................................................... 6

2.1 IoT Greengrass .................................................................................................................... 6

2.2 AWS CloudWatch ............................................................................................................. 7

2.3 AWS S3 ............................................................................................................................ 10

2.4 AWS CloudFormation ....................................................................................................... 11

2.5 Summary .......................................................................................................................... 12

3 Methodology ....................................................................................................................... 14

4 Dashboards ........................................................................................................................ 16

4.1 Problem Statement ........................................................................................................... 16

4.2 Naïve Approach ................................................................................................................ 16

4.3 Singular Stack for all Dashboards Approach ................................................................... 18

4.4 Nested Stacks Approach .................................................................................................. 19

5 Organizing Data .................................................................................................................. 23

5.1 Problem Statement............................................................................................................ 23

5.2 Everything in One Place Widget ......................................................................................... 24

6 Testing ................................................................................................................................ 27

6.1 Unit Testing ........................................................................................................................ 27

6.2 Edge Case Testing ............................................................................................................. 27

6.3 Everything in One Place Widget ......................................................................................... 28

7 Future Work ........................................................................................................................ 29

7.1 Data Visualization and Insights ......................................................................................... 29

7.2 Improving the User Interface of the AWS Dashboards ........................................................ 30

7.3 Integrating In-Built Features in AWS Dashboards Widgets .............................................. 30

7.4 Transitioning to a Cloud-Based Computing System ......................................................... 31

8 Conclusion .......................................................................................................................... 32

9 References .......................................................................................................................... 33
1 Introduction

Juniper Networks provides networking infrastructure and cloud solutions for network management and network security. Their cloud software allows enterprise customers to gather network information that facilitates the maintenance, troubleshooting, and scalability of the network. Given that clients may have numerous devices with different firmware versions and varying security requirements, keeping track of all these devices and analyzing this data is a complex yet important task.

Juniper’s JSAS system, formerly called the Juniper Support Automation Solution (JSAS), enables the JSAS team to work with their client support team to troubleshoot the devices deployed at their client sites by collecting data using a Lightweight Collector (LWC). An LWC is a virtual machine that functions as a gateway to Amazon Web Services (AWS) Greengrass daemon and connects to Juniper devices at client locations. This gateway connection allows for the transfer of read-only logs from the client’s devices to Juniper’s AWS CloudWatch in a secure manner.

Over the years, the Beta version of the JSAS system has been deployed to an increasing number of client sites, fetching large volumes of log data to Juniper’s AWS console. To support the scaling of the JSAS system and enhance the efficiency of the developers an AWS CloudWatch Dashboard was set up with log table widgets to display log data. However, these widgets do not support filtering or sorting of data on an at-need basis. With more than twenty serial numbers in the system, there was a need to develop filtered dashboards to make the troubleshooting process easier for the client and the developers. However, the code supported the creation of only one dashboard, therefore, it needed to be modified to support the scaling of the number of dashboards.

Our approach to solving the clutter, scattering, and high volume of log data in tables required the parsing and processing of said data. The JSAS log groups had standard formats but were
received as large strings of information. Parsing these logs from their standard chunks was the first, and most critical step in the process of adding, modifying, and understanding dashboard widgets and log tables. These named chunks could then be filtered or viewed, which served as a debugging tool for ensuring our dashboard tables were receiving the right data, and that it was formatted as expected. It also allowed us to easily compare the contents of varying log streams to one another, without the need to check the high volume of irrelevant information in the logs. Understanding how to parse the logs, and what each named chunk represented also allowed us to create operations on new and existing parsed chunks, allowing for more fine-tuned results, and filtering of table content.

The pre-existing dashboard included log table widgets that displayed statistics by parsing logs from different log streams. There was at least one log table widget dedicated to each log stream. Scrolling through more than nine log table widgets while troubleshooting was time-consuming for the developers. The developers needed a centralized widget that would join data from multiple log streams and display it in one place, which would enable them to troubleshoot in a time-efficient manner.

Our other problem to be solved, the limits and lack of scalability within the JSAS-Dashboard creation infrastructure as code (IaC) tool, was solved by expanding and refactoring the functionality of the existing IaC tool. Dashboard templates with filters for specific NFXSNs (Serial Number) were added, meaning the one IaC tool now could create not only the default, unfiltered dashboard, but also individual dashboards where all widgets and log tables are filtered to only retrieve results associated with the specified NFXSN. The process to generate, package, and upload all these dashboards was also automated, ensuring scalability for any quantity of NFXSNs to be utilized. Most importantly, the process was worked through to be time-efficient, such that all dashboards needed are created concurrently, as opposed to simply running the entire creation IaC tool once per NFXSN, which would be remarkably cost-inefficient.
To give a broader and deeper understanding of this project, and the work we did through the past 7 weeks, we will start by expanding on the background knowledge we needed, and technologies used. Next, we will touch on the methodology used to solve the problems brought to us and go into further details about our process and solutions after that. Next, we will discuss our testing methodologies, and the potential future projects that could arise from the work we have accomplished, before finally giving a conclusion to our 7 weeks of work.
2 Background

To gain a functional understanding of Juniper’s JSAS system, we gained exposure to its supporting technologies and software. This enabled us to better understand the Juniper team’s needs and develop solutions for them. The technologies we studied were: IoT Greengrass, AWS CloudWatch, AWS S3, and AWS CloudFormation.

2.1 IoT Greengrass

AWS Greengrass is a cloud service that allows the connection of multiple devices to a core IoT (Internet of Things) device to communicate with the cloud [2]. This connection allows remote access to the devices, which can be used to obtain data from the devices, as well as build and deploy new software to the devices [2].

As seen in Figure 1, devices connect to a device running the Greengrass daemon which serves as a gateway to the AWS cloud services. Juniper uses Greengrass to connect customers’ devices to their Lightweight Collector System, which communicates with Juniper’s AWS console using MQTT messages. Juniper's Greengrass device (the Lightweight Collector System) aggregates the data from the devices and passes it to the Greengrass cloud service to collect and analyze. The Greengrass IoT core serves as a gateway so that the Greengrass cloud can analyze data,
run maintenance, and deploy updates to all their customers' devices remotely, without the security risk of connecting directly to the devices. The logs from all these devices get displayed via dashboards in AWS CloudWatch.

2.2 AWS CloudWatch

AWS CloudWatch is an analytics and visualization tool for use alongside many of the other AWS products and technologies [1]. CloudWatch collects data from other resources in the form of logs and event metrics, often based on user definitions or instructions of what to collect and when. Once collected, this data can be visualized through several automated tools provided within CloudWatch [1] and can be customized or specialized through user-created dashboards.

![Figure 2: JSAS dev dashboard design](image)

Dashboards provide a read-only snapshot of the status of the network devices. They contain widgets that can display certain log streams in table or graph form. Widgets can filter, sort, parse, concatenate, and perform simple computations. These widgets use their own AWS CloudWatch
query language. An AWS CloudWatch query must be a string of commands separated by a “|”. Tables 1 and 2 below consist of lists of some in-built functions and commands in the CloudWatch query language that were primarily used while developing the widgets in the Juniper dashboard.

Table 1:

CloudWatch Query Commands [7]

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter</td>
<td>Filters the result of a query given a set of conditions provided.</td>
</tr>
<tr>
<td>fields</td>
<td>Retrieves the specified fields from the log events. You can use functions to modify these fields, such as ispresent().</td>
</tr>
<tr>
<td>parse</td>
<td>Extracts Strings from a field to create multiple fields. Used to extract values from a JSON object field.</td>
</tr>
<tr>
<td>concat</td>
<td>Can be used to join multiple log groups by concatenating fields that exist in all the log streams to be joined.</td>
</tr>
<tr>
<td>stats</td>
<td>Used to support the statistical functions such as earliest(), latest(), count(), sum().</td>
</tr>
<tr>
<td>display</td>
<td>Denotes fields to be displayed.</td>
</tr>
<tr>
<td>sort</td>
<td>Is used to establish the order in which the rows will be displayed. The order can be asc or dec.</td>
</tr>
</tbody>
</table>
Table 2

CloudWatch Query Functions [7]

<table>
<thead>
<tr>
<th>Functions</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ispresent(fieldName:</td>
<td>boolean</td>
<td>Returns true if the field exists</td>
</tr>
<tr>
<td>LogField)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sum(fieldName:</td>
<td>number</td>
<td>Sums the values in the specified field</td>
</tr>
<tr>
<td>NumericalLogField)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>count(fieldName:</td>
<td>number</td>
<td>Adds all the records in the field</td>
</tr>
<tr>
<td>LogField)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>earliest(fieldName:</td>
<td>LogField</td>
<td>Returns the value from fieldName which has the log event with the earliest</td>
</tr>
<tr>
<td>LogField)</td>
<td></td>
<td>timestamp in the queried logs</td>
</tr>
<tr>
<td>latest(fieldName:</td>
<td>LogField</td>
<td>Returns the value from fieldName which has the log event with the latest</td>
</tr>
<tr>
<td>LogField)</td>
<td></td>
<td>timestamp in the queried logs</td>
</tr>
</tbody>
</table>

Figure 3: JSAS dev dashboard
The dashboard above in Figure 3 as a collection of widgets that compile, analyze and display information from multiple log streams for the JSAS team to get an overview of the status of the system and track message requests and their responses. Being able to track the request and response messages allows the Juniper team to debug and locate connectivity issues. The widgets allow users to manually select a duration for which to display the data. Dashboards and widgets can be built manually through the AWS web page or programmatically via a CloudFormation template.

2.3 AWS S3

AWS Simple Storage Service (S3) is a service that allows for simple cloud storage, with interaction via a web-based graphical interface [1]. The service is configured to host storage for any type of object, allowing for versatility of use. There are access points to S3 from several other AWS resources, allowing for running Lambda functions, data querying, batch operations, and access restriction, to create cohesion in the overall cloud infrastructure. S3 is organized into buckets that can be individually managed or added to, allowing for separated storage and preferences between them. S3 also has dedicated APIs with the ability to use it as a host for public-facing information, increasing the use cases and accessibility to data within.

In our project, S3 was used for the storing of temporary files generated in the Dashboard creation process. This step was critical to the functioning of the IaC tool because of file size limits imposed on files specified and uploaded via the command line. We were able to use the S3 commands provided in the AWS CLI [15] to create a bucket, upload the generated package files to it, then delete the bucket after the generation of the package dashboards, to create a scalable solution to several roadblocks in the dashboard creation process. The swift creation, uploading, and deletion of bucket contents also meant that costs incurred from the service remain low to non-
existent, which is helpful for the financial side of the company, but also ensures scalability does not come at a cost significant enough to lose benefit.

2.4 AWS CloudFormation

AWS CloudFormation is a resource that allows the user to manage their resources across the world and enables them to model their infrastructure [1]. By creating templates using any necessary AWS resources and defining their respective properties, the user can automate the provisioning and configuration of resources. These templates can also be reused across multiple regions if the resources have been configured in them. This tool allows users to model, provision, and control third-party AWS resources by handling infrastructure as code.

Figure 4: AWS CloudFormation: How it Works [1]

Figure 4 shows a sample workflow of using AWS CloudFormation the user can build and upload a template to the Amazon S3 bucket. Next, they can use AWS CloudFormation to create a stack on the template code, which automatically provisions stacks, and resources based on the template.
During the project, we leveraged AWS CloudFormation to create AWS CloudWatch Dashboards programmatically. As shown in Figure 5 above, the CloudFormation template defined parameters like the name of the dashboard, type of environment, and region; the nested stacks (stacks created as part of other stacks to support scalability of the solution); and the output type of an AWS CloudWatch Dashboard including its value and description. The template was developed using YAML.

2.5 Summary

Throughout the duration of the project, we utilized all the technologies listed above in one way or another. IoT Greengrass was the backbone of the log system and understanding how it works and how messages flow in and out of Greengrass devices allowed us to work better with log messages. CloudWatch was used to create the widgets and queries that populated the dashboards we created, and learning the commands, syntax, and formatting allowed us to write the queries necessary efficiently and correctly. These queries were funneled into CloudFormation templates, which made up the bulk of our use of CloudFormation tools. The rest was the command
line interfacing that handled the packaging and deployment of those templates. Finally, S3 was used to store those templates, and several other important files, which expanded our capabilities and reduced restrictions of command line inputs.
3 Methodology

In our collaboration with Juniper Networks, we were initially given a broad goal of what they wanted. Their goal was to have filtered dashboards that can be built and deployed programmatically in a scalable fashion. As well as including a widget in these dashboards with multiple log streams where they could get an overview of all the log information all in one place. The overall goal was to make the dashboards less cluttered and more readable. Especially since they plan to grow the JSAS projects and sifting through logs will only get more complicated.

We had daily stand-up meetings with the JSAS team, where we would share the approaches that we were exploring and present our work in iterations; they would provide feedback and add new goals to build on for the next iteration. With each solution iteration, we built upon the previous design and fixed any pressing issues or concerns. The stand-up meetings were also great opportunities to discuss any roadblocks or course-correct our solutions. Apart from the stand-up meetings with Juniper, our internal team meetings were key for collaborative problem-solving. We took an “all-hands” approach where all of us actively participated in the innovation process through brainstorming and devising approaches and testing ideas.

Figure 6 depicts our general development process. Our process for developing each iteration of our solution started by learning the relevant technologies and familiarizing ourselves with the inner structure and logic of the infrastructure already in use by Juniper. Subsequently, we assessed the current state of the dashboard. Understanding needs and assessing possibilities and roadblocks for scalability. An approach to developing the IaC solutions was created. These solutions were broken down into their most basic components. Once the basic components were tested and proven to work, they were built upon and assembled. Once the completed product was built, the drawbacks and scalability potential were again analyzed, and another development cycle would begin to address the new concerns. Finally, when there were no efficiency concerns and indefinite scalability was secured, detailed documentation was provided.
Figure 6: Team Iterative Solution Cycle
4 Dashboards

Our first and foremost task during this project was to work on the revitalization and advancement of a dashboard creation Infrastructure as Code (IaC) tool. This tool worked to create dashboards filled with filtered tables to specific serial numbers, to make the viewing of specific company-related data easier and more convenient. This process to be scaled for customer growth and reliable, efficient, and convenient for current use. We ended up taking several approaches, each an improvement on the last: The Naive Approach, Singular Stack Approach, and Nested Stack Approach.

4.1 Problem Statement

The JSAS dashboard contains over ten thousand log messages on more than twenty customer devices. With the current volume of log messages displayed by the JSAS dashboard, searching for one specific log message requires scrolling through the clutter. With expanded use and clients in the future, this task will become more difficult. Juniper needs a way to view the dashboard information for only specific client devices without it getting hidden by messages from other client devices. The desired result was to create an IaC that would create individual dashboards for each client device with widgets filtered by their serial number.

4.2 Naive Approach

For a start, we referred to the original dashboard creation IaC which would create or update one stack for the “JSAS-dashboard-dev” dashboard. The first approach towards building filtered dashboards was a brute force approach, meaning that the same code was run again and again to create or update a stack for every filtered dashboard. An array of serial numbers was stored in the params.env file. This array was imported into the deploy script with the other environment variables. The script would iterate through the array of serial numbers to create a stack and a respective dashboard.
This approach took approximately one minute per dashboard creation, resulting in a total of twenty-one minutes for the creation of all the filtered dashboards. This enabled the programmatic creation of filtered dashboards, which increased the readability of the log data received by Juniper’s AWS CloudWatch daily. This approach did not cover the creation or updating of the main unfiltered dashboard; the Juniper team still had to manually deploy the original IaC for this task. There was a need to decrease the time and space used by this approach and reduce any manual intervention in the build of the main and filtered dashboards.

**Figure 7-8: One Dashboard per Stacks**
4.3 Singular Stack for all Dashboards Approach

The brute-force approach, though effective, was too inefficient to be considered a final solution. Since it required an entire extra run for every new NFXSN that may be added, it was not a scalable approach. On top of this, it became much harder to manage stacks, both in the AWS web UI, but also deleting the stacks programmatically still required an entire run of the delete stack script for every NFXSN in the list. Therefore, we had to re-evaluate our options for an alternate approach.

After some research, we discovered that AWS CloudFormation templates can contain and deploy multiple resources at once. Therefore, we could have all twenty-two NFXSN filtered dashboards and the default dashboard within one template, and deployed all at once, rather than waiting through an entire loop of the IaC tool for each NFXSN. To accomplish this, we made modifications to an existing text replacement script utilized by the tool. The script now took the default YAML representation of the dashboard resource and inserted a new copy into the template file for each NFXSN. We also made modifications to read the files on which the find and replace was happening, as opposed to the initial functionality which was to read the files in as strings from command line arguments in the shell script to avoid running into the string limit in the command line. Now, the script handled the creation of a template containing all the required dashboards and could be deployed all at once.

However, completing said step caused a new issue to arise: The template, when containing the source code for all twenty-three dashboards, was over the allowable size limit of 1000000 bytes (1MB) for templates passed in through the command line, as seen in figure 9. Our options were to either find a way to abstract the dashboard code and allow it to only be defined once or expand our file size capabilities. After some research and attempts at both, we were able to utilize S3 to hold a copy of the template file, increasing our potential size from a handful of kilobytes to an entire megabyte in size. Once this was established to be a proper fix, we created an S3 bucket
and added functionality to the shell script which handled uploading, deploying, and deleting the template within the S3 bucket.

![Shell Script Output]

*Figure 9: 1MB limit error using the S3 bucket when doubling the number of NFXSN* 

The resulting tool now handled the creation of all twenty-three dashboards at once. This saved time compared to the first approach. Deleting or managing the stack containing all the dashboards was easier, and the stack list was less cluttered.

### 4.4 Nested Stacks Approach

Although much more efficient than the initial brute force approach, the singular stack approach had some flaws which became evident and needed to be addressed. Though utilizing S3 to hold the template file surmounted the barrier of command-line file size, there was still a one-megabyte limit on the file. With all twenty-three dashboards' source code, the generated template was just under seven hundred megabytes, meaning the addition of a handful more NFXSN filtered dashboards would reach the limit. Thus, due to scalability concerns, we decided to investigate a solution that would give Juniper the potential for a higher volume of dashboards in the future. On top of the scalability concern, the IaC tool needed to be self-sufficient and completely automated, and not reliant on an already existing S3 bucket to function properly.

After a bit of research, it was discovered that AWS stacks can be nested within each other, which would break up the files into multiple stacks while retaining the benefit of being deployed all at
once. This potentially meant that we could break any number of NFXSN dashboards across as many templates as needed, each of which would be deployed in a nested stack by the IaC tool. To accomplish this, we created a new overhauled script to handle the generation of the dashboard template files. This new script took in the necessary information as before, as well as a user-defined value representing how many dashboards to contain within one stack. Next, it generated the templates which would be placed into child stacks, each containing the proper number of NFXSN filtered dashboards. Finally, it generated a root stack template, containing the proper references to each of the child stacks created in the previous step. Once done, all the templates were written to files within a temporary directory to be handled by the shell script.

Once the template files were generated, the shell script created a brand new S3 bucket, using the AWS S3 command-line tools. Each new bucket generated was given a standard name, but with a cryptographically secure 6-character, randomized hex string concatenated at the end, to create a globally unique S3 bucket name. Once the bucket was created, the script uploaded the contents of the temporary directory to it, and passed the URL reference to the package, validate, and deploy commands.

Overall, this approach accomplished the final goal we were working towards scalability. Since NFXSNs could now be broken up across multiple templates and stacks as seen in Figure 10, the 1MB file size could be avoided by simply increasing the number of nested stack templates used. The root stack was now the only concern, as it could not be broken up further under the current format, but with each reference to a nested stack only being approximately 100 bytes, an entire nine-thousand more nested stacks could be added before reaching the 1MB limit. With the current definition of twenty-five dashboards per stack, that left the maximum possible serial number filtered dashboards at 225,000 which, given the current count of twenty-two dashboards, was considered enough for this to be a scalable and viable solution.
Figure 10: Diagram explaining the nested stack creation process

The abstraction of S3 creation, and the new template generation script being more comprehensive in its scope, meant that the process was now reliably automatic, and put all modifiable constraints together in a convenient place for an end-user. As seen in figure 11, the lack of clutter within the AWS GUI meant that stacks and dashboards could be accessed easily, and in the event of changes or stack issues, could be modified or deleted with ease as well. As seen in Figure 12, using nested stacks also did not affect the way the dashboards were listed in the GUI, nor where they are stored for updates or modifications, meaning the list is displayed altogether just as before. The tool checked the boxes of ease of use, scalability, and time efficiency, and all the concerns that came up across the development cycle were fully addressed.
Figure 11: Nested Stack as Viewed in CloudFormation

Figure 12: All NFXSN Dashboards generated by the IaC tool
5 Organizing Data

Our second project was to organize the large volumes of log data received from the Juniper devices at client sites. While the nested dashboards enabled filtering of data, this data needed to be sorted by relevance to support troubleshooting. In this chapter, we share the need and approach taken to implement this in the form of a widget: the Everything in One Place widget.

5.1 Problem Statement

There was a need to condense information from different log streams into one line to provide a complete overview of the status. In the absence of this, the JSAS team would need to follow the trace of a client device across many different widgets, which is time-consuming and only provides a fragmented view. The tool to address this need was the dubbed Everything in One Place Widget which condensed the information from multiple log streams. Since its creation, it had become outdated as newer changes to the JSAS resulted in new issues arising. As seen in Figure 13, the user would see incorrect actualTargetCount values, the expectedTargetCount column was empty, and additionally, necessary log streams from which these values were being pulled were not getting properly joined in the query.

![Older version of the Everything in One Place widget](image)

*Figure 13: Older version of the Everything in One Place widget*
There was also a need for the widget to differentiate between cases where a message did not have a response logged because it was lost, signaling a problem, versus when it simply had not been sent yet.

5.2 Everything in One Place Widget

To satisfy the need of joining multiple log groups to view multiple statistics in parallel and facilitate faster debugging, the dashboards had an Everything in One Place widget which combined data from the following four log groups: jsas-control-config-update-dev, jsas-control-scheduled-request-dev, jsas-control-response-handler-dev, and ig04-l-snspublish-dev.

![Figure 14: Logs received by a widget over a 12-hour period](image)

As seen in Figure 14, the four log groups combined were fetching more than ten thousand logs to the widget in a twelve-hour time frame, which made the troubleshooting process tedious. It was important to sort the logs by the events’ completion status (completed or in-progress). To
implement this, another log group called `jsas-control-sfdc-sender-dev` was imported into the widget. This log group only kept a record of the completed events and their start and end times. By joining the data from this log group with the other log groups, the log table widget was sorted by the start time of the events, displaying the completed events at the top followed by events in progress.

Lastly, following the best practices of software development, we added comments to the query. Additionally, we cleaned up the code by removing unnecessary lines of code and changing variable names to follow a standard and relevant naming convention, making the log group of the query variables easier to identify.

Figures 15 (i), (ii), and (iii) are screenshots of the widget: a wide table with multiple columns of statistics put together by merging the five log streams. These statistics are grouped by the event’s JSAS unique ID, group name, and NFXSN (serial number).

```
<table>
<thead>
<tr>
<th>#</th>
<th>:start_time</th>
<th>:end_time</th>
<th>:time_scheduled</th>
<th>:last_response_or_s3_upload_time_elapsed_so_far</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2021-12-07T01:01:52.219</td>
<td>2021-12-07T03:01:58.425</td>
<td>2021-12-07T01:01:52.219</td>
<td>2021-12-07T01:01:58.425</td>
</tr>
<tr>
<td>2</td>
<td>2021-12-07T01:01:30.434</td>
<td>2021-12-07T03:01:58.426</td>
<td>2021-12-07T01:01:30.434</td>
<td>2021-12-07T01:01:58.426</td>
</tr>
<tr>
<td>3</td>
<td>2021-12-07T01:01:30.434</td>
<td>2021-12-07T03:01:58.434</td>
<td>2021-12-07T01:01:30.434</td>
<td>2021-12-07T01:01:58.434</td>
</tr>
<tr>
<td>4</td>
<td>2021-12-07T01:01:30.434</td>
<td>2021-12-07T03:01:58.434</td>
<td>2021-12-07T01:01:30.434</td>
<td>2021-12-07T01:01:58.434</td>
</tr>
<tr>
<td>5</td>
<td>2021-12-07T01:01:30.434</td>
<td>2021-12-07T03:01:58.434</td>
<td>2021-12-07T01:01:30.434</td>
<td>2021-12-07T01:01:58.434</td>
</tr>
<tr>
<td>6</td>
<td>2021-12-07T01:01:30.434</td>
<td>2021-12-07T03:01:58.434</td>
<td>2021-12-07T01:01:30.434</td>
<td>2021-12-07T01:01:58.434</td>
</tr>
<tr>
<td>7</td>
<td>2021-12-07T01:01:30.434</td>
<td>2021-12-07T03:01:58.434</td>
<td>2021-12-07T01:01:30.434</td>
<td>2021-12-07T01:01:58.434</td>
</tr>
<tr>
<td>8</td>
<td>2021-12-07T01:01:30.434</td>
<td>2021-12-07T03:01:58.434</td>
<td>2021-12-07T01:01:30.434</td>
<td>2021-12-07T01:01:58.434</td>
</tr>
<tr>
<td>9</td>
<td>2021-12-07T01:01:30.434</td>
<td>2021-12-07T03:01:58.434</td>
<td>2021-12-07T01:01:30.434</td>
<td>2021-12-07T01:01:58.434</td>
</tr>
</tbody>
</table>
```

*Figure 15 (i): Everything in One Place Widget*
This final widget was a comprehensive store of information that was otherwise scattered across multiple tables, and modifications ensured that it was displaying all that information correctly. The addition of values to display unaccounted data also added more vital information at-a-glance, which otherwise would only be inferred through the inspection of multiple tables in relation to one another.
6 Testing

Throughout each project iteration, as well as within projects, we concluded by creating tests to ensure our work was up to standard for use by the JSAS team and functioning as intended.

6.1 Unit Testing

The first kind of testing for our project was the structured and rigid unit testing we created for the programmatic side of the tool. By writing tests, we verified the outputs of our template generation scripts (at all stages) worked as specified and allowed us to iteratively fix formatting issues and bugs, we ran into in the middle of development, rather than at the end. The linting step within the overall infrastructure as code tool also helped with this, as it gave us feedback on formatting and syntactical issues before deployment. On top of this, user-made diff-checker scripts, and checks for correct substitutions, built up a perfect suite of tests to ensure our script work was proper.

6.2 Edge Case Testing

The next kind of testing was less structured, and usually more part of the solution or problem identification phase. For example, we were able to confirm the effects of large dashboard file sizes, both in S3 and on the command line, by testing inputs and scaling our existing resources to potential future expansions. These tests helped us identify problems to bring to the team, to make it clear the features we had attempted for future use by the team. Once the nested stack approach was reached, we did stress and edge-case tests to ensure our solutions could handle large quantities of serial numbers and a wide range of intervals a user could specify. This type of testing also allowed us in practice to measure the scalable volume pragmatically, rather than just theoretically.
6.3 Everything in One Place Widget

The final type of testing was the testing specific to the Everything in One Widget. Testing the information here was done manually, as the primary face of direct interaction with the widget was within the AWS web GUI. This meant that tests were done often by making edits and comparing their results to calculated or otherwise discovered expected results. A specific example includes checking for logs which were confirmed to have been from completed tasks, and ensuring they were labelled as “done”. By checking the fragmented log data, we were able to calculate manually the number of logs from ‘done’ tasks, and we could use that to compare to the values that were being produced by the widget. We were also able to trace logs to their origins, to ensure that the widget did not accidentally corrupt or modify data in any way. Both practices helped us identify errors and gave us possible solutions as to why or how those values were incorrect, and what could be causing them. Finally, in doing tests to check that the success and failure count added up to the total amount of tasks, we were then able to use the difference as a new value in the table to get insight into where unaccounted values may be. This testing then had the result of identifying a useful value for troubleshooting to the widget.
7 Future Work

During the MQP, we successfully enabled the filtering of dashboards by their NFXSNs and joined multiple log streams to combine statistics and sort the huge volumes of data by relevance. By getting hands-on experience of leveraging dashboards to find data relevant to troubleshooting, we identified the following additional three areas of improvement in the dashboards: Data Visualization and Insights, Improving the User Interface of the AWS Dashboards, Integrating In-Built Features in AWS Dashboards, and Transitioning to a Cloud-Based Computing System.

7.1 Data Visualization and Insights

A possible solution to the Juniper team’s need of making the log data more readable is visualizing it by leveraging one of the data insights tools shown in Figure 16. Data related to the client devices is stored in AWS CloudWatch and Juniper’s Snowflake database. Visualizing data from both these databases may help the team analyze the data and make more informed decisions faster. This may reduce user scrolling and searching through chunks of text to perform their analysis.

Figure 16: Data Visualization Tools [6][10][13][15]
While the dashboards are a useful way to identify relevant logs for troubleshooting, an abstract layer for the detailed logs in the form of a data visualization tool may be useful for the Juniper team. This abstract layer will provide an enhanced user experience for them as well.

7.2 Improving the User Interface of the AWS Dashboards

The log table widget in the AWS dashboards displays logs in the form of plain text in white and gray alternate columns. In the past, the Juniper team had expressed their need for color-coding the log messages, for example, success logs to show in green, warning logs to show in yellow, and error logs to show in red. Such color coding would help catch the debugger’s attention and make the troubleshooting process faster. With our new dashboard solution, the CloudWatch Dashboard page is filled with multiple dashboards spread out over two pages. As the JSAS system scales further, the number of dashboards is going to increase as well, making it hard to scroll through and find the required filtered dashboard. To make this process easier, the JSAS team wanted the filtered dashboard to be nested under the main “JSAS-Dashboard-Dev” dashboard in the form of a drop-down of sorts.

7.3 Integrating In-Built Features in AWS Dashboards Widgets

Currently, the log table widgets in AWS display the logs received by Juniper devices from client locations in a read-only format. These log tables use a CloudWatch query in the backend to compute statistics and display the log data. This query language is unique to AWS. As developers for the dashboards, we had no exposure to the language and had to learn it from scratch before we could create the Everything in One Place Widget. As an alternative to filtering and sorting the log data using a query, it would be helpful to have a UI feature in the form of a drop-down or button to filter and sort the data just like the functionality offered by Microsoft Excel for its tables. It will make the data filtering and sorting process quicker and more convenient. Additionally, having search bars in each widget can allow users to look for specific keywords rather than having to read through all the entries in the log table. If integrated in the future, this will eliminate the
need for having filtered dashboards. It will also allow users other than software developers to make changes and find entries in the log tables.

7.4 Transitioning to a Cloud-Based Computing System

The current infrastructure at Juniper for testing network devices is not to scale with the increasing number of clients and is dependent on physical devices. A possible solution is to implement a system of Virtual Machine testing environments at a scalable level through Amazon EC2. EC2 could, in theory, offer a reliable and already interconnected platform to host these devices to test. It can be as complex or as simple as needed since they would have fewer constraints when compared to physical devices or VMs running on-site.
8 Conclusion

The Juniper Support Automation Solution (JSAS) collects log messages in a Lightweight Collector (LWC) on customers' devices. This allows the JSAS team to remotely monitor, maintain and detect any system errors in the network. These log messages are filtered, sorted, analyzed, and viewed through Amazon Web Services (AWS) dashboards. It can be hard to sift through the over ten thousand log messages from over twenty client devices displayed in the dashboards. Being able to separate the log messages by client device serial number (NFXSN) would greatly alleviate the clutter of logs.

To fulfil this need as Juniper prepares for the growth of the JSAS system, we built an Infrastructure as Code (IaC) to programmatically create a series of dashboards filtered by NFXSN. The templates created by the IaC are stored in an S3 bucket and are deployed through nested stacks. The use of nested stacks in our approach accommodated the increase in the number of dashboards and their size. Having filtered dashboards will reduce the number of log messages in each dashboard to the most relevant, making data more accessible.

Additionally, inside the dashboards we fixed and improved the Everything in One Place widget to show a comprehensive review of the concatenated message log data. This provides information that might otherwise be hard to discern by searching through different widgets and log streams in isolation.

Unit testing throughout the process and edge case testing with the final solution provided for integration into the Juniper infrastructure. The IaC tool and the Everything in One Place widget are now in use by the JSAS team. In aggregate, our work helped make dashboards less cluttered and the information more accessible and organized, keeping our IaC solution scalable and preparing the JSAS system for growth.
9 References


