Automating Network Support Systems with the Juniper 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 6th, 2020

Submitted By:
Carla Duarte
Ben Emrick
Karitta Christina Zellerbach

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

Juniper Networks seeks to support enterprise networking customers by providing automated data collection and device monitoring services. The Lightweight Collector (LWC) project is designed to gather data from network devices on customer sites, process this data to cloud services, and present an interface for customer exploration. We designed a dashboard to aid in the diagnosis of underlying LWC issues, developed novel machine-learned techniques for semantic data type detection in a network domain, and built a system to gather LWC performance information through Amazon Web Services (AWS) Greengrass. Our work equipped the development team with a set of preliminary software components that increase visibility of the behaviors of the LWC and facilitate the collection of network data.
Contents

1 Introduction 1

2 Background 3
  2.1 Tools and Technologies ....................................................... 3
     2.1.1 Junos Devices and Commands ........................................ 3
     2.1.2 XML ............................................................................. 3
     2.1.3 NETCONF and SSHv2 ...................................................... 4
     2.1.4 Node.js ......................................................................... 5
     2.1.5 Virtual Machine .............................................................. 5
     2.1.6 AWS Lambda ................................................................. 5
     2.1.7 AWS IoT Greengrass ....................................................... 6
  2.2 Lightweight Collector ............................................................. 7
     2.2.1 Beta Functionality ......................................................... 7
     2.2.2 Production Ready System Requirements .............................. 9

3 Methodology and Implementation 11
  3.1 Producing Synthetic Data ....................................................... 11
  3.2 Configuring the Lightweight Collector Environment .................... 12
  3.3 Developing Log Data Streaming Pipeline .................................... 13
  3.4 Designing Diagnostics Tool .................................................... 13
  3.5 Investigating Semantic Data Type Detection ............................... 13

4 Streaming Log Data with AWS 14
  4.1 Background ........................................................................... 14
     4.1.1 Pipeline Scope ................................................................ 14
     4.1.2 Greengrass Logging Mechanisms ...................................... 15
     4.1.3 Data Streaming ............................................................... 17
     4.1.4 Amazon Kinesis .............................................................. 18
  4.2 Methodology ......................................................................... 18
     4.2.1 Amazon CloudWatch ...................................................... 19
     4.2.2 Streaming Pipeline Design .............................................. 21
     4.2.3 Log Data Extraction ....................................................... 24
  4.3 Evaluation ............................................................................ 27
4.3.1 Data Extraction Techniques ............................................. 27
4.3.2 CloudWatch vs Streaming Pipeline ............................... 28
4.4 Future Work ............................................................... 29

5 Diagnostics Tool .......................................................... 31
5.1 Background ............................................................... 31
5.1.1 Pipeline Scope .......................................................... 31
5.1.2 Tools ................................................................. 31
5.2 Methodology ............................................................. 33
5.2.1 Requirements Gathering ............................................. 33
5.2.2 Designing the Diagnostics Tool .................................. 34
5.2.3 Developing File Explorer ............................................. 37
5.2.4 Developing MQTT Client ............................................ 38
5.2.5 Developing Redis Client ............................................. 41
5.3 Evaluation ................................................................. 42

6 Semantic Data Type Detection ........................................... 44
6.1 Background ............................................................... 44
6.1.1 Pipeline Scope .......................................................... 44
6.1.2 Semantic Data Types .................................................. 45
6.1.3 Regular Expression ..................................................... 46
6.1.4 Deep Neural Networks ............................................... 46
6.1.5 Decision Trees and Random Forests ............................. 46
6.1.6 Feature Selection ...................................................... 48
6.1.7 Performance Evaluation .............................................. 48
6.1.8 Tools ................................................................. 49
6.2 Related Work ............................................................. 50
6.3 Methodology ............................................................. 51
6.3.1 Requirements Gathering ............................................. 51
6.3.2 Semantic Data Type Identification ............................... 52
6.3.3 Data Acquisition ...................................................... 53
6.3.4 Conditional Matching-Based Approach ......................... 54
6.3.5 Data Pre-Processing and Feature Extraction .................. 54
6.3.6 Sherlock Approach .................................................... 56
6.3.7 Decision Tree Approach ................................................. 60
6.3.8 Random Forest Approach .............................................. 62
6.4 Evaluation ........................................................................ 62
  6.4.1 Machine Learned vs. Conditional Logic ......................... 63
  6.4.2 Sherlock vs. Reduced Sherlock ......................................... 63
  6.4.3 Decision Tree vs. Random Forest ..................................... 66
  6.4.4 Multiple Classifier Comparison ....................................... 66
6.5 Summary ........................................................................... 67
6.6 Future Work ..................................................................... 67

7 Conclusion ........................................................................ 69

Appendix A Junos Command Outputs ...................................... 74

Appendix B AWS Greengrass Core Setup on the NVIDIA Jetson Xavier 75
Abbreviations

JSAS  Juniper Support Automation Solution
LWC  Lightweight Collector
AWS  Amazon Web Services
IoT  Internet of Things
GGC  Greengrass Core
MQTT  MQ Telemetry Transport
ZTP  Zero Touch Provisioning
CEC  Customer Engagement Center
JSS  Juniper Support System
XML  Extensible Markup Language
NLP  Natural Language Processing
VM  Virtual Machine
JSON  JavaScript Object Notation
SSH  Secure Shell
S3  Simple Storage Service
1 Introduction

Juniper Networks is an American company that provides network solutions to a large number of customers across the world. Many of Juniper’s customers have hundreds to thousands of Junos devices in a network. Thus, having an automated support system is imperative to the continual high-performance of these networks. The difficulty in automating support lies in the fact that Junos devices in the field do not automatically update their own software. Furthermore, due to business requirements and internal priorities, large service providers have over 6000 devices, would take a significant amount of time to manually update the software in each device throughout their entire network. Juniper’s current solution to performing system updates and other manual processes is to assign a resident engineer, or Juniper employee, to work at a customer site and maintain the health of the network.

The Juniper Support Automation Solution (JSAS) project seeks to build a Lightweight Collector (LWC) to help customers maintain expansive network environments. The role of the LWC is to remotely collect, parse and analyze data from various network equipment devices. These efforts are to help the JSAS team autonomously maintain the health of Juniper systems in the field.

The goal of our project was to assist in the exploration and development of different aspects of the LWC pipeline that would promote automation. We researched alternatives for computing power and system architectures that could enhance the LWC’s performance. We also addressed different areas of the LWC pipeline through three major components all related to monitoring the LWC system and improving data collection processes. These were configuring a data streaming pipeline, designing and developing an LWC monitoring tool, and investigating methods for semantic data type detection within the data collection pipeline.

We built a data streaming solution using AWS cloud technologies to automatically gain real-time insights about the performance of the LWC’s underlying computing resources. We then evaluated this data streaming solution in order to gauge its usefulness and the feasibility of future development and maintenance of the solution. This evaluation helped contextualize the capabilities and benefits of the data streaming pipeline in relation to the already existing AWS logging solution. Additionally, we designed a customized dashboard, the Diagnostics Tool, to aid in the diagnosis of LWC issues. We developed three main features within the Diagnostics Tool: the File Explorer, MQTT Client, and Redis Client. We were able to meet the overall requirements of the Diagnostics Tool established by the team. Finally, we explored and developed a variety of novel approaches for semantic data type detection and create both high-performing classification models and neural networks. Machine learned insights on the data provided valuable information that would help
further efforts to improve classification model performance.

This report outlines the tools and technologies relevant to the development of the LWC. Chapter 3 describes our general methodology regarding the preliminary steps to configure our environments, and we briefly describe the three major components of our project. Chapter 4, 5, and 6 give greater context for the roles of our three major components, explaining the methodology, results, and future work of each component in their respective chapter. Lastly, Chapter 7 summarizes the major components of this project in relation to the overall LWC.
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 will also outline the prominent tools and technologies involved in building the LWC and their relevance to the project.

2.1 Tools and Technologies

The following section describes some of the most pertinent tools and technologies relevant to our development of the LWC. It is necessary to understand not only what these tools and technologies are, but also how they relate to the production and deployment of the LWC.

2.1.1 Junos Devices and Commands

These commands work to gather network device data, and are run using Network Configuration Protocol (NETCONF) Secure Shell 2.0 (SSHv2) on the Junos OS command line interface. Commands like “show chassis hardware detail” will retrieve an XML return response containing hardware information such as item chassis, version, part number, serial number, and device description. Figure 1 shows a list of sample Junos commands that can be run on a device.

```
<jd_commands>
  <jd_command>show chassis alarms</jd_command>
  <jd_command>show ospf neighbor</jd_command>
  <jd_command>show isis adjacency</jd_command>
  <jd_command>show interfaces descriptions</jd_command>
  <jd_command>show bgp summary</jd_command>
  <jd_command>show system commit</jd_command>
  <jd_command>show chassis routing-engine</jd_command>
  <jd_command>show chassis hardware extensive</jd_command>
  <jd_command>show interfaces terse</jd_command>
  <jd_command>show version</jd_command>
  <jd_command>show system core-dumps</jd_command>
  <jd_command>show vrrp detail</jd_command>
  <jd_command>show chassis fpc</jd_command>
  <jd_command>show rsvp neighbor</jd_command>
</jd_commands>
```

Figure 1: Sample Junos Commands

2.1.2 XML

XML is an Extensible Markup Language that uses tags to define data [1]. The LWC pulls XML files from the network devices containing information relevant to the respective devices. This information
includes version numbers of various software, part numbers, serial numbers, performance information and more. While there are a number of solutions to handle transmitting data, such as JSON and CSV, the DEA team is leveraging XML files in order to maintain backwards compatibility with the existing network devices which use NETCONF, which is based on XML.

```xml
<physical-interface>
    <name>xe-0/0/37</name>
    <admin-status>up</admin-status>
    <oper-status>down</oper-status>
    <logical-interface>
        <name>xe-0/0/37.16386</name>
        <admin-status>up</admin-status>
        <oper-status>down</oper-status>
        <filter-information/>
    </logical-interface>
</physical-interface>
```

Figure 2: Sample Junos Output

### 2.1.3 NETCONF and SSHv2

The Network Configuration Protocol (NETCONF) is an Internet protocol used for interacting with network devices [2]. NETCONF is especially useful in managing device configuration data as it enables existing configurations to be retrieved and new data to be uploaded and manipulated. This protocol also provides robust tooling for controlling the operational state of specific devices. With NETCONF, network devices expose a simple Application Programming Interface (API) that can be invoked to communicate both full and fragmented configuration data components. This communication is performed over a Remote Procedure Call (RPC), which permits the client to send a network request and synchronously wait for a response from the server, which may exist on an independent system [3]. The data that is sent and received over NETCONF is fully encoded using Extensible Markup Language (XML). A significant benefit of NETCONF is that the management protocol closely mimics the native functionality of the device [4].

Secure Shell (SSH) is a network communication protocol that enables a secure and remote method for interface with devices over an unsecure network. Similar to NETCONF, SSH uses the client-server paradigm to connect a client application with an SSH server. A key benefit of SSH is its ability to securely issue commands on remote devices or entire systems. Rather than a standard username and password, SSH relies on public keys to establish a connection and authenticate hosts [5].

The LWC issues network-level commands using NETCONF within an SSH session. To initialize the session the client first establishes an SSH transport connection using the SSH transport protocol. The client and server can then exchange keys to be encrypted and used for authentication. Once authentication
completes, the SSH connection is established and the client opens an SSH session. The LWC can next invoke NETCONF which exists as an SSH subsystem in SSH version 2 (SSHv2). This allows messages to be exchanged between the client and server in XML format and later parsed throughout the software system [4].

2.1.4 Node.js

Node.js is the framework to handle the main functionalities of the LWC. By definition, Node.js is an event-driven non-blocking runtime environment for JavaScript that has become very popular on the server-side [6] due to the event-driven architecture of Node.js, making it capable of asynchronous I/O.

Node.js provides multiple ways to approach error handling. Programmers are able to throw errors, emit callbacks and trigger event emitters, and leverage patterns such as throw/catch statements and promises in order to handle errors. Along with these approaches, there are a number of error-handling architectures available to implement based on program needs. The LWC uses ES6 Promises [7] as their preferred approach to handling server-side errors. Through ES6 Promises, programmers can define asynchronous tasks with the promise that they will complete or eventually fail. Programmers can then chain together functions or error handling methods one after another which will consume the outputs or results of the preceding functions. This chain of events offers a straightforward approach to the error-handling process, avoiding layers of callbacks in potentially complex systems.

2.1.5 Virtual Machine

A virtual machine (VM) is an application that creates a virtual environment with the capability of running multiple operating systems within a single server [8].

2.1.6 AWS Lambda

AWS Lambda is “a compute service that lets you run code without provisioning or managing servers” [9]. The code written within a Lambda is only executed when needed, and the use of AWS Lambdas allow developers to integrate systems with other AWS tools. In the LWC, AWS Lambdas will be used to perform NETCONF operations and parse XML data.
2.1.7 AWS IoT Greengrass

AWS IoT Greengrass is software that extends cloud capabilities to local devices on the edge (see Figure 3). This diagram depicts the Greengrass Core’s communication channels with its direct counterparts. The enclosed rectangle represents a local network where a device running Greengrass interacts with multiple Internet of Things (IoT) devices. Greengrass can effectively manage the data from these devices through its ability to execute advanced AWS cloud process and integrate local computation with cloud services. The benefits of such a software include enabling devices to collect and analyze data closer to the source, react autonomously to local events, and communicate securely with each other on local networks. Local devices can also communicate securely with AWS IoT and export IoT data to the AWS Cloud. Additionally, the system supports the use of AWS Lambda functions and prebuilt connectors to create serverless applications that are deployed to Greengrass core and IoT devices for local execution [10].

The AWS IoT Greengrass core software provides, but is not limited to, some of the following functionality:

- Deployment and local execution of connectors and Lambda functions.
- Processing of data streams locally with automatic exports to the AWS Cloud.
- MQ Telemetry Transport (MQTT) messaging over the local network between devices, connectors, and Lambda functions using managed subscriptions.
- MQTT messaging between AWS IoT and devices, connectors, and Lambda functions using managed subscriptions.
- Controlled access to local device and volume resources.
- Deployment of cloud-trained machine learning models for running local inference.
- Automatic IP address detection that enables devices to discover the Greengrass core device.

An AWS IoT Greengrass group is a collection of settings and components “used to define a scope of interaction” [10]. Groups are comprised of other software and physical resources, such as Lambdas, external devices in a network, subscriptions and an AWS IoT Greengrass core. A Greengrass core is the computing device that runs the AWS IoT Greengrass Core software. This software is central in the Greengrass environment because it extends AWS functionality onto the device. Within a Greengrass configuration there is a one-to-one relationship between the Greengrass group and the core. The core serves as the
central Greengrass communication point across a network and is responsible for managing local process and connections with cloud service.

![AWS IoT Greengrass Architecture](image)

Figure 3: AWS IoT Greengrass Architecture [10]

### 2.2 Lightweight Collector

The following section describes the preliminary functionality of the Lightweight Collector. This beta version was developed by the DEA team as a prototype to begin establishing requirements of the LWC. This section also describes the requirements for the production-ready LWC system.

#### 2.2.1 Beta Functionality

The existing automation solution is a beta version of the LWC. The beta LWC is installed on a virtual machine host, specifically CentOS 7, which is simulated by a VMWare Workstation running on a laptop or desktop computer. The beta host displays an IP address that leads developers to a front-end accessed through a browser connected to the network. Through this client, Juniper resident engineers can create and schedule jobs to run commands against the Junos devices in the network. The configuration of these jobs includes the following information:

- Network Username and Password
- Bastion Host (optional)
  - Username
  - Password
The attributes requested by each job in the LWC are stored in a JSON file. Once the form is completed, the jobs start to run the provided commands. These commands work to gather network device data, and they are run using NETCONF SSHv2 on the Junos OS CLI. Commands like “show chassis hardware detail” retrieve an XML return response containing hardware information such as item chassis, version, part number, serial number, and description. Figure 4 documents the structure of the XML response generated from the command where data values are associated with descriptor tags.

```xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/9.3R1/junos">
  <chassis-inventory>
    <chassis junos:style="inventory">
      <name>Chassis</name>
      <serial-number>00118</serial-number>
      <description>M20</description>
      <chassis-module>
        <name>Backplane</name>
        <version>REV 06</version>
        <part-number>710-001517</part-number>
        <serial-number>AB5911</serial-number>
      </chassis-module>
      <chassis-module>
        <name>Power Supply A</name>
      </chassis-module>
    </chassis>
  </chassis-inventory>
</rpc-reply>
```

Figure 4: Example of output from “show chassis hardware detail” command

Each piece of information is surrounded by an opening and closing tag. For instance, the chassis represented below has a serial number of 00118 and the description “M20”. From there, the XML data enters the Juniper Support System (JSS), which funnels data the AWS pipeline into S3 buckets where it is parsed and decoupled for easier interaction. Next, the data is written to an enterprise data storage platform, Snowflake, where specific queries and analysis can be executed. Figure 5 shows the architecture for the preliminary automation solution.
The intention of this system was to provide a mechanism for retrieving XML data from the Junos devices and sending it through the JSS pipeline to be cleaned, parsed, and stored. This data will be used to assist the Juniper customer support team, as well as Juniper’s customers, in monitoring the health of their networks. Naturally, because this was a beta version, the DEA team was able to identify major areas of improvement for the tool to enhance the automation and overall security of this pipeline.

### 2.2.2 Production Ready System Requirements

In an effort to move towards automated customer support, the goal of the LWC is to transition from the manual management of an on-site resident engineer to a one-time manual interaction. This single interaction will configure jobs to run on the LWC, allowing the pipeline to proceed automatically. Through discussions regarding a production ready system, the DEA team reconstructed the architecture of the LWC. This updated architecture is shown in Figure 6.
The updated architecture introduces a variety of changes from the beta architecture. The first change is that the web front-end previously hosted by the beta virtual machine will no longer exist. In order to configure jobs, customers will visit a single point of entry where they can also view the data received from their network. The second change is that the LWC will adopt an Amazon Web Services (AWS) Greengrass architecture in order to extend cloud capabilities to devices on the customer site. The final change is that the LWC will run AWS Lambda functions in order to mask, parse, and de-duplicate XML data prior to streaming it through a data pipeline to be stored and displayed to users.

Figure 6: Updated Lightweight Collector Architecture
3 Methodology and Implementation

The primary objective of our project was to assist in the exploration and development of the LWC pipeline, ultimately promoting automation. The following section describes the variety of components in our project which worked to address the flow, reduction, and use of data within the LWC. Section 3.1 describes the process of producing synthetic data. Section 3.2 describes how we configured the LWC environment. Section 3.3 describes the development of a log data streaming pipeline. Section 3.4 describes the design and development of a tool used for diagnostics of the LWC. Section 3.5 describes the investigation into semantic data type detection.

3.1 Producing Synthetic Data

In order to gain an understanding of the data we would be working with, the first part of the project involved implementing commands on the beta LWC. More specifically, our goal was to monitor the response of the beta LWC when given misspelled or invalid inputs. These inputs included commands, IP addresses, username and password data, and other input fields from the front-end LWC component. One example of an invalid input in the beta LWC can be seen in Figure 7, where we ran `show cssis alars` instead of `show chassis alarms`.

![Figure 7: Invalid input error in the beta LWC web front-end](image)

The output in Figure 7 was in the web front-end. In this error message, we could clearly see that the command `show cssis alars` was marked as a `bad_command`. In the XML output, however, we found the messaging to be less clear. As seen in Figure 8, there is a missing `<output>` tag which indicates that the XML structure has been broken; there is no formal indication of what went wrong. This is an example of a bad...
response that we identified in the system, and we concluded that the error was not handled properly.

```xml
<data>
  <command-header>
    <run-time>2020-01-22T23:06:32Z</run-time>
    <command>show cssis alarms</command>
    <database-version>show_cssis_alarms-junos-1.0</database-version>
  </command-header>
  <output/>
</data>
```

Figure 8: JData Output

Once we understood the behavior of the XML data, we used this insight to fuel other components of our project. More specifically, the insights from our synthetic data production directly influenced our investigation of semantic data type detection which we describe in Chapter 6.

### 3.2 Configuring the Lightweight Collector Environment

The next component of our project was to configure our environments for LWC development. The beta LWC system uses the Juniper NFX150; however, our goal was to explore the use of different computers and tools in order to support the LWC processes. To do this, the DEA selected two NVIDIA devices for us to configure: the NVIDIA Jetson Nano and the NVIDIA Jetson Xavier. Additionally, we were tasked with installing AWS Greengrass on these devices. Ultimately, our goal was to test the viability of supporting the LWC on either of the NVIDIA devices using AWS Greengrass.

Our first step in this phase was to configure the environments of the NVIDIA devices. In order to configure these devices, we had to install the NVIDIA Jetpack SDK through a Linux host machine. This required us to use a Linux VM for the installation, as none of us had Linux machines. Once the SDK was installed on the devices, we moved onto the next configuration piece.

The next step in this phase was to register the devices as Greengrass core devices. To do so, we followed the configuration steps in Appendix B. Once the NVIDIA devices were registered as part of the Greengrass system, we then began to implement the final three components of our project.
3.3 Developing Log Data Streaming Pipeline

Once AWS Greengrass was configured on the NVIDIA Xavier, the next step in our project was to develop a log data streaming pipeline. This streaming pipeline would facilitate the preservation of Greengrass logs, ultimately allowing developers to monitor health of the device. For an in-depth description of this portion of the project, the methodology used for development, and the results of the streaming pipeline, see Chapter 4.

3.4 Designing Diagnostics Tool

The next component of our project was to design and develop a Diagnostics Tool. This tool’s role was to give developers insight on processes running on the LWC. For more context regarding this component’s role in the LWC project and the methods used to design and develop the Diagnostics Tool, see Chapter 5.

3.5 Investigating Semantic Data Type Detection

The final component of our project was to investigate semantic data type detection. This is an important step in achieving efficient data masking in order to reduce the amount of data that is sent through the LWC pipeline. For more information regarding this component’s role in the LWC project, the methodology followed, and results of the investigation, see Chapter 6.
4 Streaming Log Data with AWS

As Juniper relies on external hardware, such as the NVIDIA Jetson Xavier to host the LWC and other software components, there is an increasing need to better understand the behaviors of this technology. Juniper is interested in learning the Xavier’s capacity to manage various use-defined processes, computing performance and its default error handling mechanisms. Within the framework of the intended JSAS architecture, the AWS Greengrass core device will serve as the underlying computing mechanism responsible for powering the various operations of the LWC on the client network. Therefore, it is essential to develop a system that automatically delivers insights about the Greengrass core performance. Furthermore, this solution must transmit the data efficiently without misrepresenting the original data source. A log streaming service will equip Juniper professionals with the knowledge to understand Greengrass’s capabilities under certain circumstance and appropriately assess potential issues. The goal of this aspect of the project is to discover a reliable and timely method for remotely retrieving log data from the Greengrass core device while minimizing resource overhead and system complexity. The immediate use cases for this tool is for internal developers to view incoming data in real time and manually assess the Greengrass core’s performance through an organized collection of system and user level log messages.

4.1 Background

The following section describes the scope of the log data streaming pipeline within the overall LWC project, as well as the tools and technologies used in the development of this component of the project.

4.1.1 Pipeline Scope

The data streaming pipeline is responsible for delivering local log data from the Greengrass core device to the AWS Cloud. The pipeline does not interact with the standard operations of the the LWC. Instead, running entirely outside and in parallel with, the LWC and all other software components within the JSAS architecture. This system includes end-to-end integration of AWS products, which greatly facilitates the transfer of data in a self-contained structure. The pipeline is driven by a single, user-defined AWS Lambda function running in on the Greengrass core device. Depending on the user-selected runtime configuration, the Lambda either executes inside of a default container in an isolated namespace or with no containerization at all. Ultimately, the Lambda function is responsible for interfacing with the host machine and the AWS Cloud in order to transport log data.
4.1.2 Greengrass Logging Mechanisms

Installation of the AWS IoT Greengrass core software creates a file structure on the host computer. This process makes the `/greengrass` folder in the root directory of the host file system. Sub-directories and files within the top level folder maintain the information relevant to all of the Greengrass built-in software and the outputs produced by its processes. Figure 9 displays a high-level structure of the Greengrass file system where the root directory is `/greengrass/ggc/var/log/`. The leftmost positioned items in the tree denote high level directories. Subsequent indented items represent items contained within the parent folder.
Figure 9: Unix Tree Command Output

A Greengrass Group is capable of recording logs for system and user Lambda functions on the core
computer. System logs pertain to pre-installed components that ship with the Greengrass software such as the stream manager, connections manager and certificate manager. User logs are generated for each user-defined Lambda function running on the core device. These logs contain the standard output for code executing inside of the Lambda runtime environment. All of the logs generated by Greengrass are written to the /greengrass/ggc/var/log directory. Within this top-level log directory a crash.log file is created in order to give visibility into the operations of the Greengrass instance as a whole. The crash.log file contains output triggered by any sudden termination of the Greengrass core instance [11]. Depending on the Lambda function type, logs are displayed to the /greengrass/ggc/var/log/system and /greengrass/ggc/var/log/user directories, respectively. At the system level there is a runtime.log that is responsible for compiling activity across the various Greengrass system processes and displaying a chronological sequence of events. This file offers a broadened mode of surveying overall system performance. Accessing any log data on the core device requires root permission on the host system. By default, all log data produced by Greengrass exists exclusively on the local host computer and cannot be extracted without further configuration [11].

Through the AWS Console, developers can specify logging preferences for the Greengrass Group. Both user and system logs can be enabled or disabled. Furthermore, the user can determine the log level that should be sent for each log type. Each log message has an associated log level that signifies the severity of the message [12]. The highest available log level setting that Greengrass permits is Debug. We selected this option in order to receive Debug and all lower level log types such as Info, Warn, Error, and Fatal. As the log level decrease, the severity of the issue and cause for alarm increases. Thus, messages preceded by the lower level descriptors, such as Fatal and Error, suggest significant malfunction and require immediate corrective action. We enabled user and system logs to be written to the local Greengrass core device with the lowest level in order to gain maximum visibility into the internal functionality of system and user software components.

4.1.3 Data Streaming

Transmitting large amounts of data from a source to a target destination is a critical challenge encountered by various applications and software systems. There are multiple techniques to address this situation. One approach includes batch processing, which is a method that will delay delivery while it aggregates a larger data store to then transfers all at once [13]. In contrast, streaming is a procedure that continuously delivers data in small, record sized portions at a high frequency interval. Streaming is a data delivery technique that introduces a constant flow of micro batches of data, usually on the order of Kilobytes [14]. Thus, the source data repository is processed on a granular, usually record-by-record basis. Streaming offers a
a more flexible approach than batching because it allows for thorough processing of dynamic data that is generated on a continual and unspecified basis.

A streaming solution can monitor a data source and quickly transmit incoming data to external computing resources. This highly responsive system is well-suited for an application that assess a continuously growing log file, such as that on the Greengrass core device. With a streaming pipeline there is minimal delay between writes to a monitored log file and delivery into consumer endpoints that can better manage the information. A data streaming pipeline is the preferred mechanism for processing continuously expanding log files because it offers faster visibility into the underlying behaviors of the Greengrass software. This aligns with the central mission of Juniper’s technical support efforts to respond promptly to emerging network issues.

4.1.4 Amazon Kinesis

Amazon Kinesis Data Streams is a software services offered by AWS that allows for applications to transport large amounts of data in real time [14]. Kinesis serves as an intermediary stream that supports a high throughput volume from data producing clients and reliably delivers the data to external consumers. Kinesis does not batch incoming data that is constantly pushed to a managed data store. Instead, it provides data streaming. The delay between the time a record is put into the stream and the time it can be retrieved by a consumer, referred to as put-to-get delay, is typically less than 1 second [14]. Furthermore, Kinesis offers variability in determining the scale of a stream so that resource size can be adjusted based on application need. The service also supports a variety of configuration options for specifying producers and consumers of the data transfer. Producers are any entity that writes data to a Kinesis Stream, whereas consumers read data that is exported by the stream. The feature set of Kinesis Data Streams aligns well with the central requirements for building a highly scalable log delivery pipeline.

4.2 Methodology

The main objective of this endeavor is to identify a useful and accurate system for extracting log data from the Greengrass core device and making it remotely accessible. This system is not responsible for manipulating or analyzing the data in order to automatically develop actionable insights about the Greengrass core’s health. Instead, the preliminary functionality of this system must merely transfer data fast and guarantee that it directly mimics the original source. Therefore, data can not be miscounted or altered during the process.
4.2.1 Amazon CloudWatch

Aside from write logs exclusively to the local file system, Greengrass can also connect to Amazon CloudWatch to automatically route logs to the AWS Cloud. Amazon CloudWatch is a monitoring and observability service that integrates with a myriad of AWS products, including Greengrass [13]. This product provides a mechanism for collecting and processing large amounts of data and includes exposes a wide variety of features through a dashboard and data visualization tool in the AWS Console. Since CloudWatch can be configured for the Greengrass core device and is the officially supported AWS logging solution, a crucial component of this project included exploring its functionality and determining its value.

Similarly to the procedure for setting up user and system Lambda functions to be written to the local host file system, these properties can be configured for logging to CloudWatch. The settings for local and CloudWatch options are managed in in the AWS Console. Within this web interface the setting for enabling CloudWatch must simply switch on (off by default). The user can also specify the log level that should be reported by the system. Lastly, to finalize the process for setting up CloudWatch on the Greengrass core device, an Identity and Access Management (IAM) Policy must be attached the group role on the Greengrass Group. This policy permits Greengrass to begin writing local log data into the CloudWatch engine.

Once properly configured, CloudWatch will automatically monitor all of the system and user log files that are available on the local device. The /greengrass/ggc/var/log/crash.log file is the only log data that continues to be written to on the local machine, but is not tracked by CloudWatch. In the CloudWatch console a Log Group is created for the Greengrass core device. Figure 10 shows this Log Groups dashboard with the naming conventions /aws/greengrass/GreengrassSystem/greengrass-system-component-name and /aws/greengrass/Lambda/aws-region/account-id/lambda-function-name which contain the system components and Lambda function logs, respectively.
Further, once a log stream is selected, the CloudWatch user interface displays a scrollable table window with built-in pagination. This table displays the log messages exactly how they appear on the local device in the same chronological order. The table supports further customization by permitting the user to choose desired columns, alter the order of records, dynamically search the existing data, and change the time interval for which data is displayed. Also, the Logs Insights page allows users to execute queries against the data set of a selected log file. Figure 11 displays a query interface that outputs log messages with text that match the string “Error”. This tactic proves to be an effective way to filter through large amounts of data to achieve a result set that fits within a user specified criteria. As the DEA team better understands the log messages associated with troublesome Greengrass behavior, this refinement strategy will become critically important in locating instances of alarm activity.

Figure 10: Amazon CloudWatch Log Group

<table>
<thead>
<tr>
<th>Log group</th>
</tr>
</thead>
<tbody>
<tr>
<td>/aws/greengrass/GreengrassSystem/GGCloudSpooler</td>
</tr>
<tr>
<td>/aws/greengrass/GreengrassSystem/GGConnManager</td>
</tr>
<tr>
<td>/aws/greengrass/GreengrassSystem/GGDeviceCertificateManager</td>
</tr>
<tr>
<td>/aws/greengrass/GreengrassSystem/GGIPEndPointer</td>
</tr>
<tr>
<td>/aws/greengrass/GreengrassSystem/GGSecretManager</td>
</tr>
<tr>
<td>/aws/greengrass/GreengrassSystem/GGShadowService</td>
</tr>
<tr>
<td>/aws/greengrass/GreengrassSystem/GGShadowSyncManager</td>
</tr>
<tr>
<td>/aws/greengrass/GreengrassSystem/GGStreamManager</td>
</tr>
<tr>
<td>/aws/greengrass/GreengrassSystem/GGTES</td>
</tr>
<tr>
<td>/aws/greengrass/GreengrassSystem/runtime</td>
</tr>
<tr>
<td>/aws/greengrass/Lambda/us-west-2/608370444947/TransferStream</td>
</tr>
<tr>
<td>/aws/kinesisfirehose/GGLLogData</td>
</tr>
</tbody>
</table>
4.2.2 Streaming Pipeline Design

As part of the objective to automatically transport data from the Greengrass core device to a remotely accessible endpoint, we built a data streaming pipeline. The goal of developing this software component is to explore the native capabilities of Greengrass and assess its integration ability with other AWS products. Through designing and implementing a streaming system we aimed to determine the feasibility and practicality of using Greengrass and the counterparts AWS services in a production scale environment.

Architecture Figure 12 shows a diagram of the data streaming pipeline. This architecture consists software that runs both locally on the Greengrass core device (leftmost box of Figure 12) and in the AWS Cloud (rightmost box of Figure 12). The seamless connection between these different runtime scopes significantly reduced the complexity and difficulty during the development process.
**Long-Lived Lambda** A long-lived Lambda function contained the core business logic that drove task execution. The long-lived state allows the function to run indefinitely without termination on the Greengrass core device. This is essential for the reliability of the streaming pipeline because the Greengrass system processes continuously output data to their respective log files. Therefore, the Lambda function execution lifetime must resemble the data input behavior of the Greengrass system. All of the code within the Lambda functions is written in Javascript using the Node.js runtime environment. See Section 4.2.3 for more details regarding the user-defined code that runs in the data streaming pipeline.

**Stream Manager** Stream Manager is an Amazon software client that runs inside of the AWS IoT Greengrass Core SDK for Node.js. Stream manager is the initial data receptor on the Greengrass device that can store, process, and export information [15]. Stream manager provides set up parameters for the code to determine the consumer entity. This is possible through the `StreamManagerClient` object exposed by the SDK. This object is used by the Lambda function to create and interact with streams in stream manager. Figure 13 demonstrates the process of creating a message stream with an export definition to an user-owned Kinesis Stream. Once the export definition is configured, the data intake feed to the `StreamManagerClient` will then automatically route all traffic into the specified Kinesis Stream in the AWS Cloud.
const c = new StreamManagerClient();
c.onConnected(async () => {
  try {
    const exports = new ExportDefinition()
      .withKinesis([new KinesisConfig()
        .withIdentifier('KinesisExport${STREAM_NAME}"
        .withKinesisStreamName(KINESIS_STREAM_NAME)]]);

    await c.createMessageStream(
      new MessageStreamDefinition()
        .withName(STREAM_NAME)
        .withStrategyOnFull(StrategyOnFull.OverwriteOldestData)
        .withExportDefinition(exports),
    );

    Figure 13: StreamManagerClient Usage

    **Amazon Kinesis** Amazon Kinesis serves as the cornerstone of the pipeline in that it connects the local computing resources with the AWS Cloud. This communication greatly facilitates the flow of data off of the core device without having to alter its structure for further transport. Kinesis is easily configured via the AWS Console by adding an IAM policy to the Greengrass group role. This policy clarifies the authorized actions involving the Kinesis Stream. This grants all processes within the Greengrass Group the necessary permission to interact with the Kinesis Stream.

    **Amazon Kinesis Data Firehose** Amazon Kinesis Data Firehose is an AWS offering that is part of the Kinesis Stream platform. Firehose can manage Kinesis Streams so that flowing data can be properly processed and extracted by eventual destinations. The foundation of Firehose is the delivery stream that continues the sequence of real time data delivery [16]. The data streaming pipeline uses a Kinesis Firehose delivery stream as an intermediary layer for buffering incoming data from the Kinesis Stream and sending it to a persistent storage location. While Kinesis Firehose supports data transformation and other analysis techniques, this pipeline does not incorporate any advanced processing. Instead, the Firehose delivery stream is merely responsible for establishing its source input as the Kinesis Stream and its destination as a user owned Amazon S3 Bucket.
**Amazon S3 Bucket**  Amazon S3 is a cloud based data storage platform [17]. A user owner S3 bucket serves as the single endpoint for all data that flows through the streaming pipeline. Amazon S3 buckets are resources that are persistent in nature, meaning data loss will not occur on termination of the pipeline. Therefore, this solution increases reliability in that all parties seeking access to the log data can trust that it is retained and not corrupted by inadvertent system operations. For the preliminary implementation of a logging solution, Juniper is mostly concerned about a trustworthy single point of access for all of the log data. Further, the data must be readable and organized so that it is manually parsable. The usage of S3 as a final destination for the satisfies the criteria that the log data must exist in a permanent and persistent data store in the AWS Cloud.

The configuration of this series of AWS technologies comprises the entirety of the data streaming pipeline. These services coordinate with each other to safely maintain a flow of data at a rate that resembles the frequency of new data added to local log files. Throughout this procedure data is not duplicated in any aspect of the process or permanently written to tangential resources. The configuration of the pipeline ensures that the sequence of streaming services act only as temporary storage. This minimizes resource overhead and the level of management required to clean up trails of data. Lastly, the most critical feature of the data streaming pipeline is that it automatically and transports records of data through every layer of the system. Once the delivery process is initiated by appending a message to the `StreamManagerClient` in the Lambda function on the Greengrass core, data can flow uninterrupted to its final destination for and become available for remote user access.

### 4.2.3 Log Data Extraction

The long-lived Lambda function in the pipeline is solely responsible for accessing the file system on the host computer and extracting local data. This section discusses two viable approaches for retrieving and entering data into the remainder of the pipeline.

**SSH**  The default Lambda function deployment on the Greengrass core initializes a unique runtime environment inside of a Greengrass container. This is the recommended mode of execution because the container provides a barrier between the functions and the host. This increases security for all entities involved and prevents incidental changes on the core device. By running a Lambda function inside of a container the code maintains access to all features of Greengrass and all child process derived from the Lambda exists in the same container [18]. Therefore, Lambda code that is deployed to Greengrass cannot directory interface any other Lambda code also running on the system. Further, the default configuration for a Lambda deployment
uses the user ID and group ID created upon installation of the Greengrass software. This account runs the Lambda function with restricted access the the device’s resources. The initial Lambda solution involving an SSH session was run with a standard Greengrass container and the default user ID and group ID values.

Since this solution runs in a Greengrass container within its own runtime environment, the code cannot directly access several areas of the host file system. As mentioned in Section 4.1.2, all of the desired log data is stored in the /greengrass/ggc/var/log/, which is protected by root privilege. Since the Lambda function runs with the default user and group ID, access to the files and sub-directories in this folder are restricted. To circumvent this issue, the Lambda code uses the ssh2 node module [19] to SSH onto the host machine. Then, through this secure connection the function establishes an interactive shell session on the device. Now, the function can carefully parse the shell stream and issue a series of commands to gain root privilege and navigate into the /greengrass/ggc/var/log directory. However, even with the appropriate permissions, the log files cannot be read directly. Instead the the tail -f command is written to the shell stream in order to output the last ten lines of the target log file into an external file that exists elsewhere on the host file system. This command line utility will set the output file to automatically watch for changes on the source log [20]. Therefore, new data will be appended to the output file in real time, just as it is on the original file. This strategy includes a level of indirection to bypass the inability to read the file at its original location, but preserves a way to monitor changes to the log in real time. Figure 14 displays the logic for achieving this task.

```javascript
let conn = new Client();

    conn.on('ready', function() {
        conn.exec('sudo -s', { pty: true }, function (err, stream) {
            if (err) throw err;

            stream.on('data', function(data) {
                if (data.indexOf(':') >= data.length - 2) {
                    stream.write(process.env.PASSWORD + "\n");
                } else if (((data.indexOf('#') >= data.length - 2) && !flag) {
                    stream.write('tail -f ${logFile} | tee ${outputFile}\n');
                }
            });
        });
    });
```

Figure 14: Interactive Shell Session via SSH

The foundation of building a data streaming pipeline is correctly taking action on the data from the lowest level read location. In this scenario, log messages are dynamically appended to an exists file. Therefore, the Lambda function must be responsive and trigger writes the the StreamManagerClient in real-time as new
data is added. This is possible by routing the data through a stream object in Node.js. In Node.js a readable stream is an object used to store data in an internal buffer and then place it into a queue for future consumption [21]. When a readable stream is created from a data file, the buffer becomes occupied with specific size of data present in the target file. Then, once the data is requested by another resource, it exits the buffer and is popped off of internal queue. The `readable.on('data')` event handler is responsible for executing its body once there is a data buffer in the queue that is ready to be transported. This process repeats until there is no longer data in source file. The implementation of `createReadStream()` in Node.js will close the stream once all of the data in the file is read. This is problematic in the situation where the readable stream can process the entire file before new data is written to it. Then, the stream will close and the entire pipeline will seize if there is a halt in log file changes. The first line in Figure 15 solves this challenge by creating a readable stream from a target file on the host file system using the `tailing-stream` node module [tailing]. The `tailing-stream` Node.js module that allows for the instantiation of a stream object that does not stops reading once it gets to the last byte that existed at the time the stream was originally opened. This is ideal for application that needs to continuously read from a file that is being actively written to [tailing].

```javascript
const readable = TailingReadableStream.createReadStream(logPath, {timeout: 0});

readable.on('data', async (buffer) => {
  try {
    console.log('Appending new record to Kinesis Stream');
    await c.appendMessage(STREAM_NAME, buffer);
  } catch (e) {
    console.log("Caught error writing to stream");
    throw e;
  }
});
```

Figure 15: Create an Indefinitely Open Readable Stream

**No Container Mode** This Lambda function solutions heavily differs from the previous approach in terms of its runtime configuration. This solution runs without a Greengrass container in No Container mode. Thus, it does not operate with any separation from the other processes running on the host device. Further, this function does not run with the default user and group ID. Instead, 0/greengrass/configs/config.json file is edited within the Greengrass software to allow Lambda functions to run with root privilege. Therefore, the user and group ID are set to 0 in the configuration options for this lambda deployment. Now, this function
can operate with sudo privilege and achieve direct access to the local file system. This removes the need to SSH onto the host device and gain further permissions through the command line. This privilege enables the function to create the read stream directly from the log file at its original path in the file system. There is no longer the need to indirectly reference a file that is monitoring the desired target. Ultimately, adjusting these configuration parameters result in a much more straightforward log extraction procedure.

4.3 Evaluation

Through building a customized data streaming solution and configuring AWS CloudWatch for Green-grass, we were able to identify the key advantages of each approach. Furthermore, within the streaming pipeline we evaluated two unique Lambda function implementations for extracting log data from the system. These explorations offered significant insight into the usefulness and feasibility of developing a maintaining an in-house streaming solution. Also, this project aims to identify the techniques that best leverage the Greengrass environment to craft a performant and expandable software component.

4.3.1 Data Extraction Techniques

Ultimately, through the development process we determined that the non-containerized Lambda function is the recommended version for safely retrieving and processing local log data. This approach avoids the unnecessary complexities of managing an interactive shell session via SSH. The development of this method presented multiple hardships and introduced room for error. Firstly, this solution relies on establishing a successful connection on the host device through SSH. Depending on the state of the network and connectivity status of the device, there is an increased potential for the Lambda function to be unable to communicate at all with the target machine. Furthermore, this solution contains a known issue in that the standard output of the data will be skewed when watching a Lambda function’s own log file. This is because the `tail -f` command used to track changes to the target file will write the log messages to `STDOUT`. In this scenario, the `STDOUT` location is the log file being watched. This causes the log data to continuously nest timestamps and duplicate the generated data in a format that is difficult to analyze by humans. Therefore, the several shortcoming of the SSH Lambda function solution render it poorly equipped to satisfy the intended functionality and reliability of the streaming pipeline. The non-containerized version provides the most straightforward and robust foundation for future development.
4.3.2 CloudWatch vs Streaming Pipeline

This evaluation seeks to compare the CloudWatch log monitoring solution against the custom streaming serving. Since CloudWatch is the officially supported AWS monitoring tool, the user experience for enabling it in the Greengrass environment is simple. This lightweight process limits the resource management from an engineering perspective. Furthermore, CloudWatch ships with an expansive feature set that can be utilized to manually and automatically evaluate the data. The primary features of immediate interest to Juniper are persistent cloud storage, message filtering, rate of transfer and coverage of the local source data. CloudWatch delivers a practical solution to each area of desired functionality with limited shortcomings. This solution satisfies the requirements of remote accessibility because CloudWatch aggregates log data in the AWS Cloud and presents the information in the AWS web console. Therefore, Juniper professionals can view the data in an organized manner through a client-side user interface on a remote machine. This interface possess a Log Insights dashboard (see Figure 11) that can execute and user-defined queries against the selected data set and store them for future use. The viewer also contains a top-level search bar to filter the records by input string. The internal design of CloudWatch does contain several logging limitation. Firstly, it batches log events locally before sending them to CloudWatch rather than continuously streaming the data at the same rate it enters the log file. Also, if the Greengrass device lacks internet connectivity, then CloudWatch will fail to receive logs and any data stored in local memory will be lost [13]. This presents a potential gap in gathering valuable troubleshooting information. Lastly, CloudWatch is set to automatically record data from nearly all of the supported system and user process on Greengrass. The only inconsistency between the collection of local data and data in the cloud is that CloudWatch cannot monitor the /greengrass/ggc/var/log/crash.log file. This inability may become a significant deficiency if sudden termination events interfere with the Greengrass environment. Juniper support staff will have a narrowed collection of data to draw conclusions from and use to diagnose the root of the crash.

Currently, the streaming pipeline built throughout this project exists in a rudimentary form with a limited feature set. The pipeline is configured to support automatic streaming of a single local log file into a persistent data store in the AWS Cloud. This system does not cover streaming multiple files, which remarkably reduces the collection of data available to engineers. Furthermore, it does not include a dynamic filtering procedure to selectively choose a subset of overall data. While the S3 bucket does organize the incoming logs into easily viewable files sorted by creation date, the interface lacks many of the features and analytics tools that CloudWatch possesses. Despite its infancy and the remaining work to achieve feature parody with CloudWatch, the streaming pipeline does offer several advantageous prospects. This system harnesses the ability to implement a filtering mechanism at the Lambda function level. Therefore, intelligent
and business domain specific logic can be added during the initial process that reads data from the local log instance. A robust decision system at this level will prevent excess data from entering the remainder of the pipeline and consuming resource and increasing compute time. This is a significant opportunity for the JSAS team to define the criteria for meaningful log data and employ a solution that facilitates the scalability of the entire pipeline. Also, CloudWatch fails to monitor the system crash.log. Contrarily, both Lambda function approaches produced in this project can track the Greengrass crash.log file in real-time and stream its contents to a remote storage site. This may prove valuable in the diagnostic efforts of the support organization. Furthermore, this pipeline only streams log data and relies entirely on a real-time flow state, whereas CloudWatch must batch data locally before its is transported off of the Greengrass core device. While the rate limits for each solution have not been thoroughly tested, execution in a production environment with a greater magnitude of data may reveal significant differences in the speed of delivery.

The observations gathered through the development and usage of these tools suggest that CloudWatch is a more suitable immediate solution. It monitors a wider range of Greengrass logs and contains a valuable feature set through the CloudWatch Management Console. However, the streaming pipeline’s user-defined nature allows for much more flexibility and growth that can be influence directly by Juniper’s support goals. By merely adopting the CloudWatch solution, Juniper is restricted to abiding by the functionality defined by AWS. The development of a data streaming service resulted in meaningful discoveries of the Greengrass platform and its capabilities within the scope of AWS. The streaming pipeline serves as a foundation for understanding the utilization of operational data in a networking domain.

4.4 Future Work

The efforts of the project produced a minimally viable data streaming pipeline for AWS Greengrass. We explored the functionalities of Greengrass as an environment for deploying and executing user-defined code. This included evaluating Greengrass’s integration capabilities with other AWS products, especially Cloud managed services. Overall, the configuration of the pipeline was useful in determining the suitability for Greengrass within the JSAS architecture. Coupling this exploration, we identified numerous areas for future work in order to expand the functionality of the system and continue to add value to the DEA team.

Filtering Employing a filtering mechanism to restrict the amount of data the enters the streaming pipeline would be beneficial. This can be build at the most granular computing level where the log data is initially read from its original file. This will ensure that unnecessary log messages are excluded from further processing. This should be implemented once Juniper has identified the types of log messages that best signify alarm or
corrective action.

**Multiple Log Files** The Lambda function currently implemented only watches a single local log file. Instead, the streaming pipeline should be dynamic enough to support reading multiple files and sending the data to external consumers.

**Multiple Greengrass Cores** This solution was devised within the scope of a single Greengrass core device running on a network. A production level solution must account for various cores across multiple networks. A worthwhile effort would be to restructure the pipeline so that it can process a distributed fleet of cores.

**Unique Record Identification** Creating a unique identification mechanisms to attach to each log stream record would couple the efforts to scale the pipeline to incorporate multiple data sources. If data from various sources is all routed through a central Kinesis stream and into a single S3 bucket, then a unique ID is necessary to associate each log message with the core device it was generated on.

**Recovery Index** Currently, the Lambda function returns to reading the entire log file from the beginning on every Greengrass software restart. This causes significant data duplication. In order to reduced this repetition and excess compute time the system should maintain a progress index into the read state of the log file so that a known offset can be used for continued reading.
5 Diagnostics Tool

Another component of our project was to design and develop the Diagnostics Tool. This Diagnostics Tool allows developers to monitor certain processes running on the LWC. In the following section, we provide context for the tool’s role in the overall LWC project, as well as describe our methodology for development.

5.1 Background

Prior to discussing the development of the Diagnostics Tool, it is necessary to understand the role of the tool within the scope of the LWC project. The following section provides context for the tool within the LWC pipeline, as well as what tools and technologies were selected to aid in its development.

5.1.1 Pipeline Scope

The Diagnostics Tool will be deployed by AWS Greengrass, and will work to deliver metrics regarding processes running on the host device, i.e. the LWC. The Juniper Escalation Team and other developers intend on using this tool only when an unforeseen issue arises with the LWC that they cannot easily troubleshoot from afar. This tool does not directly fit into the pipeline in terms of data streaming and retrieval on the LWC, but rather, serves as a supplemental feature to diagnose LWC problems or errors.

5.1.2 Tools

Due to the full-stack nature of the Diagnostics Tool development, the following section is split in two in order to highlight which tools were primarily used for front-end development and which were used for back-end development.

Front-end Components  During development of the Diagnostics Tool front-end, we intentionally stuck to basic HTML, CSS, and JavaScript. While our focus in developing this tool was to enhance user experience through user interface (UI) design, the future maintainers of the Diagnostics Tool will not be front-end developers. That being said, we did not incorporate any popular front-end frameworks (such as React, Angular, or Vue) in order to avoid an unnecessary learning curve for future developers. Front-end frameworks can be valuable in terms of structure and organization, but in considering the scope of the Diagnostics Tool, we did not deem them necessary at this time. Further, we wanted to leave room for future developers to incorporate frameworks or other front-end development practices as needed. We did, however, leverage a
few libraries to supplement our development process. The following sections work to describe these libraries.

**jQuery UI**  jQuery UI is an open source jQuery library that provides a number of UI interactions [22]. More specifically, through jQuery UI, developers can make components in their UI draggable, droppable, resizable, selectable, and sortable. jQuery UI also provides styling options; however, we did not make use of them as we were using Bootstrap to enhance our styling.

**JSTree**  JSTree is an open source jQuery library that builds interactive tree displays [23]. It ingests data as JSON structures or HTML unordered lists. We selected this tree-structure library primarily because it has a number of out of the box features that enhance the display. For example, JSTree has plugins that support features such as fuzzy searching, checkboxes, icons, and drag and drop.

**Bootstrap**  Bootstrap is a free, open source CSS framework that facilitates uniform and responsive styling [24]. Bootstrap provides an off-the-shelf solution to styling native HTML components such as headers, buttons, and links. It goes even further by providing styling for components, which are defined through classes. For example, rather than dealing with the difficulties of the HTML, CSS, and JavaScript required to display a navigation bar within a website, Bootstrap contains a Navbar class which has predefined configurations for the spacing and styling of a navigation bar (see Figure 16).

![Bootstrap Navbar example code](image)

**Back-end Components**  The Diagnostics Tool back-end is written in Node.js (see Section 2.3.1) primarily to mirror the existing AWS Lambda functions written by the DEA team. The following section describes
the additional tools involved in the development of the Diagnostics Tool back-end.

**Redis**  The DEA team had selected Redis as the database that stores job configuration details [26]. Redis is an open source, in-memory database that stores key-value pairs. It can support a number of data types, such as strings, sets, and lists. Redis can also store JSON objects as strings when properly configured with a module called ReJSON [27].

**MQTT**  MQTT stands for MQ Telemetry Transport, and is a lightweight messaging protocol used by connected devices [28]. It works by defining topics which devices in a network can publish and subscribe to. It is frequently used by Internet of Things (IoT) devices, and is also used by Amazon Web Services (AWS) for communication [29]. Because the jobs run through AWS Lambda functions, we adopted MQTT as the communication protocol.

### 5.2 Methodology

The primary objective of the Diagnostics Tool is to provide the Juniper escalation team and developers with a way to diagnose issues on the remote LWC device. In order to accomplish this, we needed to design a real-time interface to monitor a variety of processes running on the device.

#### 5.2.1 Requirements Gathering

Our initial discussions regarding the Diagnostics Tool clarified some general use cases, allowing us to narrow down the role of this tool in the scope of this evolving project. Firstly, we established that rather than functioning as a proactive monitoring tool and alerting developers of issues, the Diagnostics Tool exists as a way to take a closer look at different aspects of the remote device if an issue were to arise. There was no expectation of the Diagnostics Tool to fix a problem, only to expose what may be causing the problem.

Further, we established three primary components of the tool. The Diagnostics Tool had to be capable of:

1. accessing local files on the LWC device,
2. subscribing and publishing to AWS topics through MQTT, and
3. displaying job data stored in a Redis instance.
While we considered leveraging third-party monitoring services to support these different aspects, building this tool internally provides Juniper with a single point of entry which they could fully customize for their diagnostic needs.

Based on these requirements in mind, this tool runs locally on the device, i.e. a long-lived Lambda running on the Greengrass Core device will be serving the Diagnostics Tool front-end in order to have local access to files.

Lastly, we had no constraints in terms of development tools and UI design because this would be an internal tool. Our intentions were to choose development tools that had little to no learning curve and to design an intuitive and customizable UI.

5.2.2 Designing the Diagnostics Tool

Our primary role in building the Diagnostics Tool was to design the UI with our target audience, the Juniper escalation team and developers, in mind. In order for users of the tool to quickly and accurately diagnose problems within the LWC, the Diagnostics Tool UI had to be easy to understand. We also had to develop a back-end that would efficiently gather data to facilitate the diagnostic process. The following section details our methodology in developing the tool.

Architecture Figure 17 shows a diagram of the current architecture of the Diagnostics Tool. The Diagnostics Tool consists of a Node.js server running from an AWS long-lived Lambda function hosted by the AWS Greengrass Core device. The server hosts the Diagnostics Tool front-end, written in HTML and JavaScript. The server is tasked with making SSH calls on the LWC to retrieve local files and directories, as well as interacting with a locally run instance of a Redis database through a Node module called Node-Redis. Client-side MQTT subscriptions interface with other AWS Lambda functions running on the LWC.
UI Mockups  Prior to designing each component within the tool, we designed a general layout for the UI. We wanted the Diagnostics Tool to feel like a customizable dashboard. Figure 18 shows the proposed dashboard. The left-hand side would be home to a navigation panel. Within the navigation panel, users could click a component of their choosing which would open up a card in the main body of the dashboard. The card could then be closed to make space for larger components as needed.
We envisioned each of the components as resizable and draggable cards. Rather than displaying a single component at a time, the cards would allow multiple components to be displayed at once. We could not specify each component card’s actual design until we gathered requirements for that specific component’s role in the tool. That being said, the ability to drag and resize cards would provide users with the necessary flexibility to diagnose LWC issues.

Through our own preliminary use of the Diagnostics Tool, it quickly became clear that users did not need the ability to customize the dashboard to such a high extent. We found that dragging and resizing functionalities were not necessary for each card. Each time we opened a component from the navigation pane, the component’s card had to be dragged to a certain position and then resized to fit its contents. This added an unnecessary barrier to accessing the data within the cards, burdening users with the need to organize their dashboard prior to benefiting from the tool. As a result, we decided to instantiate each component card with the same width as seen in Figure 19, spreading across the entire dashboard. Users would then be allowed to vertically drag and sort these cards as needed.

![Figure 19: Refactored layout with component cards at full-width](image)

With a general dashboard layout and functionality in place, we were able to move onto gathering requirements and designing the cards for the first three components in the Diagnostics Tool.
5.2.3 Developing File Explorer

The first component we developed was the File Explorer. This component would allow users to SSH into the LWC and interact with local files. The following section outlines the exact requirements and design decisions encountered through development of the File Explorer component card.

File Explorer Requirements In discussing the functionality of the File Explorer, we established requirements regarding internal composition and design. The first requirement of the component was that the card should contain a visual display of the hierarchy of directories and files at a given path. This path would initially be predefined, but editable by users. The hierarchy display would include the name, size, and date last modified of each file. The second requirement was that the card should include a search functionality to locate files or directories by name. The last requirement of the File Explorer component was that the card should support both downloading and deleting files through SSH. Further, users should be able to select multiple files at once to perform either of those actions.

File Hierarchy Display To aid in the file hierarchy display of this component, we used an open source library called JSTree. JSTree functions by ingesting a JSON object of nodes and then building an HTML tree display, which is equipped with a number of plugins that can be used to enhance the display. One JSTree plugin that we chose to incorporate was the Types plugin. This plugin allows you to define different types and assign styles to them. We defined types for files and folders and associated different icons to them in order to distinguish between the different nodes in the tree. Figure 20 shows these icons in use.

![Figure 20: File hierarchy display using JSTree and Types Plugin](image)

JSTree displays nodes in a way that allows users to open and close directories to expose any underlying children. This user experience felt familiar to other file systems within popular operating systems (such as the Mac OS file display shown in Figure 21), suggesting it would be a comfortable experience for our users to adopt a tree-structure display.
Search Component  To satisfy our second requirement for the File Explorer, we incorporated a search bar within the component card to locate files and directories by name. Fortunately, JSTree provides a plugin to facilitate this exact need. By incorporating the Search plugin, we were able to provide a search solution that could traverse the nodes and display any matching nodes within their parent-child structure (see Figure 22).

![File Explorer component card demonstrating search functionality](image)

File Interaction  The final requirement of the File Explorer component was to support downloading and deleting files. Our first initiative in developing this feature was to use the JSTree Checkbox plugin to support the selection of multiple files at once. Next, we added two simple buttons to the component card to support downloading and deleting (see Figure 22). Upon clicking either of these buttons, the component collects the paths of the selected nodes and passes them through an HTTP POST request to our server. In the back-end, the component executes an SSH command into the device to perform either operation, then traverses the files at the root path and returns an updated node structure of the files, which is used to update the hierarchy display.

5.2.4 Developing MQTT Client

The second component we developed was the MQTT Client. This component will have the ability to publish and subscribe to different topics within a message bus. The following section outlines the design decisions regarding the MQTT Client component card.

MQTT Client Requirements  The requirements of the MQTT Client were established through a discussion of different use cases. The first requirement was to allow users to both subscribe and publish to
multiple topics. This allows users to monitor MQTT communications originating from AWS Lambda func-
tions. The second requirement was that the card should support viewing multiple topics in a single feed. This requirement manifested through preliminary use of the MQTT Client on the AWS console platform. The AWS MQTT Test Client allows users to subscribe to multiple topics at once; however, topics’ incoming message streams are isolated, forcing users to switch tabs to view messages from different topics. In order to ensure that the Diagnostics Tool MQTT Client needed to improve this functionality, we established that the MQTT Client should support viewing multiple topics in a single message feed. The last requirement encompassed some basic features such as unsubscribing from topics, clearing the message log, and exporting all logs from a session to a local text file.

**Subscribe and Publish Implementation**  As seen in Figures 23 and 25, we designed a simple interface to support subscribing and publishing to MQTT topics. We separated these two functionalities into tabs in order to establish a visual difference between both operations. When the Diagnostics Tool loads, it establishes an MQTT connection using AWS credentials. We leveraged a library called Eclipse Paho to facilitate establishing connections, subscribing, publishing, and unsubscribing [30].

![Figure 23: MQTT Client subscribe feature](image)

Once the connection is established, users can input the name of a topic to subscribe to (see Figure 23). In an effort to make the system more robust, we incorporated form validation as seen in Figure 24. This restricts users from subscribing to the same topic multiple times. Upon subscribing to a topic, the client listens for messages to arrive and populates a message container.

![Figure 24: Example of MQTT Client form validation](image)

Users can also submit the name of a topic and the message they would like to publish (see Figure 25). Again, the Eclipse Paho library facilitates publishing messages to topics within the established connection.
While testing this component, we discovered some limitations in establishing an MQTT connection from the Diagnostics Tool’s server. When we first implemented the MQTT connection on the server-side, it became clear that incoming messages could not be pushed to the client without a trigger of sorts (ex. a refresh button within the MQTT Client UI). Because we wanted messages to continuously and automatically populate the message container, we attempted to incorporate WebSockets in our solution. WebSockets, however, did not integrate well with the Greengrass infrastructure. We thus migrated the Eclipse Paho MQTT library to the client-side in order to listen to incoming messages and automatically populate the message container display.

**Topic Subscription Display**  When a user enters a topic to subscribe to, the topic will populate a list of current subscriptions in the lower half of the MQTT Client card. This subscription panel is shown in Figure 26.

In order to support multiple topics at once, we incorporated checkboxes for users to select which topics they would like to display. The selected topics populate the message container on the right-hand side of the subscription panel in chronological order. To further improve the multi-topic display, we incorporated a color picker to differentiate between different topics. All topics are defaulted to display in white text, however, users can use the color picker as seen in Figure 27 to assign colors to different topics.
We incorporated a button in the client card that allows users to export all of the messages received in that session to a text file on their computer. We also incorporated a button that clears the message container of any messages received during the session. To support unsubscribing from topics, we included a red button next to each topic in the subscription panel to facilitate the unsubscribe feature.

5.2.5 Developing Redis Client

The third component we developed was the Redis Client. This component interfaces with an instance of a Redis database used to store LWC job data. The following section outlines the Redis Client component card design process.

Redis Client Requirements The role Redis has in the overall LWC project was continuously evolving as we attempted to narrow down requirements for the Redis Client within the Diagnostics Tool. This limited our ability to establish in-depth use cases, however, we were able to outline general features that future developers could iterate on and improve. The first requirement of the client was to support the retrieval of key-value pairs from the Redis instance running on the LWC. The second requirement was to support storing or changing key-value pairs within the client.

Retrieving Key-Value Pairs To support retrieving key-value pairs, we designed a simple interface, as seen in Figure 28, which allows users to input the name of the key they were interested in. On input of a key, the Redis Client makes a request to the respective Diagnostics Tool endpoint. The server-side Redis interaction checks the type of the value being requested prior to retrieving it from the Redis instance running on the LWC. The type of value being queried is crucial to successfully retrieving the value because the Redis
Node module has different retrieval calls for different types. For example, if the value is of type string, the retrieval call is `redisClient.get('key')`. If the value is of type set, the retrieval call is `redisClient.smembers('key')`. Lastly if the value is a JSON object, the type is ReJSON-RL. Redis does not have a native retrieval call for this type. The correct way to retrieve this value would be to use a ReJSON Node module and to call `rejsonInstance.get('key')`. Once the value has been correctly retrieved, the Redis Client displays the value in the UI.

![Redis Client key-value pair retrieval UI](image)

**Figure 28: Redis Client key-value pair retrieval UI**

**Storing Key-Value Pairs** To support storing key-value pairs, we used a similar interface as the retrieval UI. This is shown in Figure 29. Currently, the key-value storage UI only supports storing values of type string.

![Redis Client key-value pair storage UI](image)

**Figure 29: Redis Client key-value pair storage UI**

### 5.3 Evaluation

Through the DEA team’s preliminary use of the Diagnostics Tool, we received positive feedback regarding its design and functionality. Figure 30 shows the final design of the Diagnostics Tool.
Once more specific use cases are defined for the Diagnostics Tool, each component can be iterated on to improve their usability. Future work for the tool may include:

- implementing sorting within the File Explorer to allow users to sort by file size or date last modified,
- adjusting the key-value storage UI within the Redis Client to support storing values of different types such as set and ReJSON-RL, and
- creating a key-value retrieval UI within the Redis Client that automatically displays keys, rather than prompting users to input key strings.

We were able to meet the overall requirements of the Diagnostics Tool established by the team. The Diagnostics Tool provides a closer look into various aspects of the LWC through its three main components: the File Explorer, MQTT Client, and Redis Client. We also made design and development decisions influenced by the requirement that the Diagnostics Tool will ultimately be hosted by a long-lived Lambda running on the Greengrass core device.
6 Semantic Data Type Detection

Scale is often an issue for companies that deal with large amounts of data, and Juniper is no exception. The LWC system will need to handle ingesting data from hundreds of devices which have thousands of XML values. As devices are added to customer networks, the amount of data being collected increases exponentially. Therefore, it is necessary to consider data reduction to filter out unnecessary data wherever possible. In order to implement data reduction as close to the edge as possible, the LWC will have two core functionalities: data masking and de-duplication. Through discussion with the DEA team, we established that an area of interest regarding data masking involved classifying the semantic types of data that flow through the LWC pipeline.

In the following section, we examine the scope of this investigation into semantic data types, providing justification for a machine learning approach within a project of this scale. Further, we describe the different data detection techniques we used and compare the results of each one. Finally, we summarize the findings of this investigation and outline next steps for incorporating these insights into the LWC masking and de-duplication pipeline.

6.1 Background

Prior to discussing the methods and results of the investigation into semantic data type detection, this section provides context for the necessity of data type detection as well as the relevant tools and technologies leveraged throughout the process.

6.1.1 Pipeline Scope

As the LWC collects data, there are two main components that work to perform data reduction. These two components are masking and de-duplication as seen in Figure 31. The semantic data type detection, if successful, would take place in the masking portion of this data reduction. This is because the main goal of the data masking phase is to extract data that is not useful to stream through the pipeline.
Because the LWC will have to interact with thousands of devices, it is crucial to be proactive in identifying areas in which efficiency can be applied. Therefore, we would like to detect the semantic type of each XML tag and update the metadata to represent the result in the masking phase. After the masking phase, the XML data would be passed to the de-duplication phase. In the de-duplication phase, a probabilistic data filter is used to filter out Junos command output that has already been seen. This helps reduce the size of the file to stream through the data pipeline, as well as eliminates redundant processing. The de-duplication phase is implemented as an inverted bloom filter. This is done in order to allow false negatives (data we did want, but excluded from the data stream), and exclude false positives (data that we did not want, but included in the data stream). The system works by streaming the contents of a Junos XML file and adding values between CDATA tags into the filter. If the value already exists, or maybe exists in case of a false negative, then the program replaces the value with a unique string that will provide context to the back-end servers analyzing the files. These two components, masking and de-duplication, work to reduce the amount of data being streamed through the collection pipeline.

6.1.2 Semantic Data Types

There are two main data types that appear when dealing with data: basic atomic data types, such as strings, booleans, and integers, as well as semantic data types, which help to describe the kind of information the data represents by establishing correspondences between columns and real-world concepts. While the detection of atomic types is straightforward and supported by most systems, semantic data types are much harder to classify due to their richer nature, currently leaving them without a supported system for labeling. The data from Junos devices includes a lot of network-domain data with semantic types such as “ip address”, “OS version”, “serial number”, and “operational status”. 
6.1.3 Regular Expression

A regular expression is a means for specifying string patterns concisely, usually through a sequence of characters. The specification may be used by a specialized engine for extracting the strings matching the expression from a data stream. Regular expressions are a long-established technique for a wide variety of application domains, including text processing, and continue to be a routinely used tool due to their expressiveness and flexibility [31]. Our initial approach utilizes matching-based computations through regular expressions.

6.1.4 Deep Neural Networks

Deep learning is a class of machine learning algorithms that uses multiple layers to progressively extract higher level features from the raw input [32]. For example, in image processing, lower layers may identify edges, while higher layers may identify the concepts relevant to a human such as digits, letters or faces. Figure 32 shows a basic architecture for a Neural Network, which is comprised of input, hidden, and output layers. Deep neural networks are included in one of our approaches.

![Figure 32: Neural Network](image)

6.1.5 Decision Trees and Random Forests

A decision tree is a flowchart-like tree structure, where each internal node represents a test on an attribute, each branch represents an outcome of the test and each class label is represented by a leaf node (or terminal node) [33]. Figure 33 shows an example of a basic Decision Tree used to classify the dietary...
needs of users based on tests such as eating meat or dairy. For instance, the root node poses the question “Eat Meat?”, which is then able to make a distinction between Vegetarians and Non-Vegetarians in the data. The root node is typically the decision with the largest effect on the end classification. The further down the decision tree, the finer-grained questions that are asked serve to label a variety of classes. Decision Trees are often used in classification tasks due to their ability to be easily understood and interpreted, and thus are known as a white box classifier. However, they are often relatively inaccurate, as a small change in the input data can lead to a substantial change in the structure of the decision tree.

![Decision Tree](image)

**Figure 33: Decision Tree to classify Dietary Needs**

Algorithms for constructing decision trees usually work top-down, by choosing a variable at each step that best splits the set of items. Different algorithms use different metrics for measuring “best”. However, these all generally measure the homogeneity of the target variable within the subsets. Gini impurity, used most often by the CART (classification and regression tree) algorithm for classification trees, is a measure of how often a randomly chosen element from the set would be incorrectly labeled if it was randomly labeled according to the distribution of labels in the subset [34]. The formula for the Gini impurity can be seen in Equation 1, where C is the total number of classes, and \( p_i \) is the distribution of the class in the node.

\[
Gini = 1 - \sum_{i=1}^{C} (p_i^2)
\]  

Random forests are an example of an ensemble learner built on decision trees, which are also commonly used for classification, regression and other tasks. By using multiple learning algorithms, ensemble methods are able to obtain better predictive performance than could be obtained from any of the individual learning
algorithms alone [34]. Random forests in particular operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. Random decision forests correct for decision trees’ habit of over-fitting to their training set [35].

**Overfitting** Overfitting occurs when the gap between the training error and test error is too large [36]. This means that the model has learned to follow the errors, noise, and random fluctuations too closely [37].

**Underfitting** Underfitting occurs when the model is not able to obtain a sufficiently low error value on the training set [36]. This is because it cannot capture the underlying trend of the data and will decrease the accuracy of the model.

Our approach utilizes both decision trees and random forest models.

### 6.1.6 Feature Selection

Decision trees tend to overfit on data with a large number of features. Getting the right ratio of samples to number of features is important, since a tree with few samples in high dimensional space is likely to overfit. One way of minimising overfitting is through feature selection, which is typically used to improve estimators’ accuracy scores or to boost their performance on very high-dimensional datasets. Feature selection algorithms work by selecting a subset of relevant features for use in model construction. Feature selection can also be used to shorten training time, simplify models, and avoid the “curse of dimensionality” [37].

For decision trees and random forests, feature selection can be done using the Gini importance, also known as the “mean decrease impurity”. For decision trees, Gini importance is calculated as the (normalized) total reduction of the criterion brought by that feature. For random forests, it is the total decrease in node impurity weighted by the probability of reaching that node. The probability is approximated by the proportion of samples reaching that node, where the impurity is then averaged over all trees of the ensemble [38].

### 6.1.7 Performance Evaluation

**Precision** Precision (also called positive predictive value) is the fraction of relevant instances among the retrieved instances. The precision is intuitively the ability of the classifier to avoid labeling a negative sample positive [39]. Precision can be represented in Equation 2, where \( tp \) is the number of true positives and \( fp \)
the number of false positives.

\[
\text{precision} = \frac{tp}{(tp + fp)} \tag{2}
\]

**Recall** Recall (also known as sensitivity) is the fraction of the total amount of relevant instances that were actually retrieved [39]. The recall is intuitively the ability of the classifier to find all the positive samples and is expressed in Equation 3, where tp is the number of true positives and fn the number of false negatives.

\[
\text{recall} = \frac{tp}{(tp + fn)} \tag{3}
\]

**F₁ Score** The F₁ score can be interpreted as a weighted average of the precision and recall, where an F1 score reaches its best value at 1 and worst score at 0. The relative contribution of precision and recall to the F1 score are equal [39]. The formula for the F1 score is shown in Equation 4.

\[
F_1 = 2 \frac{(\text{precision} \times \text{recall})}{(\text{precision} + \text{recall})} \tag{4}
\]

### 6.1.8 Tools

These are the main and most relevant tools that were used in the development of this project.

**Jupyter Notebook** The Jupyter Notebook is an open-source web application that allows for creation and distribution of documents that contain live code, equations, visualizations and narrative text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, and machine learning [40].

**Pandas** Pandas provides a flexible platform for handling data in a data frame. It contains many open-source data analysis tools written in Python, such as the methods to check missing data, merge data frames, and reshape data structures [41].

**NumPy** NumPy builds on (and is a successor to) the successful Numeric array object in Python. Its goal is to create the corner-stone for a useful environment for scientific computing [42].
**Keras**  Keras is an open-source neural-network library written in Python. Keras contains several implementations of essential building blocks such as layers, objectives, activation functions, and optimizers, allowing for rapid development of neural networks [43].

**Scikit-learn**  Scikit-learn is a Python module integrating a wide range of state-of-the-art machine learning algorithms for medium-scale supervised and unsupervised problems. This package focuses on bringing machine learning to non-specialists using a general-purpose high-level language. Emphasis is put on ease of use, performance, documentation, and API consistency [44].

**Matplotlib**  Matplotlib is a 2D graphics package used for Python for application development, interactive scripting, and publication-quality image generation across user interfaces and operating systems [45].

**Fabric**  Fabric is a high level Python library designed to execute shell commands remotely over SSH, yielding useful Python objects in return [46].

### 6.2 Related Work

Although semantic type detection is still a novel field, there have been attempts to provide accurate methods of classification. After a survey of relevant literature, two possible approaches were identified. The first, Sherlock [47], a multi-input deep neural network for detecting semantic types and the second, AutoType [48], a system that can synthesize type-detection logic for rich data types by leveraging code from open-source repositories like GitHub.

Correctly detecting the semantic type of data can be helpful for a myriad of tasks in the data science lifecycle, like cleaning and pre-processing. According to the researchers on Sherlock, existing data preparation and analysis systems utilize dictionary lookups and regular expression matching to perform semantic labeling. However, these are approaches are often unable to accurately detect dirty data, and thus can only detect a limited number of types [47]. The neural network in Sherlock was trained on 686,785 data columns retrieved from the VizNet corpus by matching 78 semantic types from DBpedia to column headers. They characterize each matched column with 1,588 features describing the statistical properties, character distributions, word embeddings, and paragraph vectors of column values. Sherlock was able to achieve a support-weighted F1 score of 0.89, where the score is the average of the F1 score of each class weighted by support, or the number of true instances for each class. It can also be interpreted as an accuracy of 89%, which exceeds that of other machine learning baselines, dictionary and regular expression benchmarks, and the consensus
of crowd-sourced annotations.

Pham, Minh et al investigate a domain-independent approach [49] by using similarity metrics as features to compare against labeled domain data and learning a matching function to infer the correct semantic labels for data. They claim that because their approach depends on the learned similarity metrics but not the data itself, it is domain-independent and only needs to be trained once to work effectively across multiple domains. Some of the similarity metrics used are attribute name similarity, value similarity, distribution similarity, histogram similarity, and mixtures of numeric and textual data. After an evaluation of two classifiers, logistic regression and random forests, it was concluded that a logistic regression model was the best choice due to a smaller training and labeling time, as the accuracy results were comparable among both classifiers.

6.3 Methodology

The primary objective of the investigation into semantic data type detection is to enhance the data reduction component of the LWC data collection. To complete this investigation, we implemented a number of classification models and neural networks in order to compare the efficacy of each approach.

6.3.1 Requirements Gathering

In discussing the value of this investigation, we established a few general expectations and requirements. The first expectation was that, although we leaned towards a machine learning approach in our preliminary discussions, we would not confine this investigation to only exploring machine learned solutions.

A requirement of the investigation is that it would attempt to classify data types as one of the following categories:

- State values (Enumeration)
- Counters (Monotonically increasing/decreasing)
- Time/date stamps
- Fluctuating values (Memory, CPU, resource consumption rates)
- Other

There are a variety of other data types that could be included, however, we decided it was important to start small. Values categorized as “other” would theoretically be let through the pipeline, while others may
be subject to being masked out. Additionally, we established that this investigation would not encompass decisions regarding which values to mask. For example, timestamps that represent the time when a command was run would not be necessary to push through the data streaming pipeline. On the other hand, a timestamp generated by an alarm is a piece of data would be of interest to customers. Thus, it is not accurate to label all timestamps as irrelevant. In order to complete the masking pipeline, we would need to quantify how positive we are that a value with a certain data type is not important. While masking data is an important process within the LWC project, this investigation would focus more on methods for semantic type labeling, leaving masking decisions as an area for future development.

The last general expectation of this investigation was that an approach would be deemed acceptable if it were fast, scalable, and highly accurate. These characteristics were important in regards to integrating data type detection into the data reduction pipeline.

### 6.3.2 Semantic Data Type Identification

While our production of synthetic data provided a foundation of knowledge regarding Junos data, a critical analysis of the contents and structure of the XML data was required. Through manual execution of Junos commands on Juniper devices in the test lab, commonalities between output files were studied and identified. From the initial investigation of the data, several semantic types were identified, as can be seen in Table 1.

<table>
<thead>
<tr>
<th>ENUM</th>
<th>COUNTER</th>
<th>TIME</th>
<th>RATE</th>
<th>OTHER</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>up</td>
<td>time</td>
<td>cpu</td>
<td>ip</td>
</tr>
<tr>
<td>version</td>
<td>elapsed</td>
<td>size</td>
<td>name</td>
<td></td>
</tr>
<tr>
<td>status</td>
<td></td>
<td></td>
<td></td>
<td>serial</td>
</tr>
</tbody>
</table>

Table 1: Semantic Data Type Classes

Firstly, we needed to address the limits of a single command output. For instance, a tag that contains a counter value will only have one value, a single data point. In order to effectively detect sequences and patterns, multiple values are needed. The same principle applies for both rates and enumerations. In order to get multiple data points, we ran each command an n-number of times (preserving the order) and then detected the semantic type based on each sequence of data points.

Throughout this process, we also had to consider the inherently nested structure of the XML file. For instance, should empty tags be ignored? Should our approaches be expected to handle a nested XML structure, or can they expect a standard tabular input structure? There is a flattening process that occurs
in the back-end of the LWC pipeline, however, this is expected to occur after data masking. In order to focus on developing the models, an intermediate solution throughout the investigation was to assume that the data ingested by the model would be in a tabular structure.

### 6.3.3 Data Acquisition

To continue further with the exploration, large amounts of data were needed. A Jupyter Notebook Python script was implemented to automatically gather data from Junos devices.

The script was initially created to avoid the monotonous process of tunnelling through Bastion hosts to run Junos commands. Use of the terminal requires Junos command data to be saved in the Bastion host as an intermediate storage through FTP, which the host device would then be able to access. Due to the need to tunnel through the Bastion host, the Junos PyEZ library could not be used. Instead, the Fabric library was used. The script (seen in Figure 34) was able to run commands from an array of Junos hosts in a loop with specified time intervals in-between commands. Looking at lines 1 and 2, we can see that the connection to the Junos device, conn2 had be made through the Bastion, conn1, gateway. Lines 4 to 8 show calls for user input, which are the command to run, the number of times to run the command, and the delay between commands. Data returned from the commands would be stored in an XML file with the timestamp and loop index in the filename, which is accomplished in line 19.

```python
1 conn1 = Connection("user@host", connect_kwargs={"password": "*******"})
2 conn2 = Connection(host="user@host", connect_kwargs={"password": "*******"}, gateway=conn1)
3 4 command = input("Command:")
5 num_times = input("Number of times to run the command:")
6 num_times = int(num_times)
7 delay = input("Delay(s):")
8 delay = int(delay)
9 10 for i in range(num_times):
11    result = conn2.run(command + " | display xml", hide=True)
12    output = "{0.stdout}"
13    command_output = output.format(result)
14    15    root = ElementTree.fromstring(command_output)
16    et = ElementTree.ElementTree(root)
17    ElementTree.dump(root)
18    timestr = time.strftime("%Y%m%d-%H%M%S")
19    et.write("output" + "-" + str(i) + "-" + timestr)
20    sleep(delay)
```

Figure 34: Model Output Classes
6.3.4 Conditional Matching-Based Approach

The first approach was a conditional matching-based approach, which utilized inbuilt Node.js packages and regular expressions. A divide-and-conquer structure was used, beginning with atomic type detection in order to split values into numeric, alphanumeric, and alphabetic bins. In order to be in a strictly numeric or alphabetic bin, all values in the sequence had to be in that bin.

After that, regular expressions were used to infer semantic details. For instance, potential timestamp values were matched against MM:DD:YYYY, MM/DD/YYYY, and other similar formats. Another level of validation was through the XML tag name, the word “time” in the name, for example.

6.3.5 Data Pre-Processing and Feature Extraction

In order to prepare data for a machine-learned approach, data pre-processing and feature extraction were required. Figure 35 shows the overall data acquisition process including pre-processing and feature extraction. In order to create training and testing data set, column names were mapped to one of the twelve identified semantic types, in order to act as class labels (seen in Figure 36). The example in Figure 35 represents CPU data, where cpu-user is the percentage of CPU time being used by user processes, cpu-system is the percentage of CPU time being used by system processes, system-interrupt is the percentage of CPU time being used by interrupts, and cpu-idle is the percentage of CPU time that is idle.

![Figure 35: Data Gathering and Preprocessing](image)

<table>
<thead>
<tr>
<th>Time</th>
<th>Junos XML</th>
<th>1. Data Collection</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpu-user</td>
<td>...</td>
<td>cpu-user</td>
</tr>
<tr>
<td>cpu-system</td>
<td>...</td>
<td>cpu-system</td>
</tr>
<tr>
<td>cpu-interrupt</td>
<td>...</td>
<td>cpu-interrupt</td>
</tr>
<tr>
<td>cpu-idle</td>
<td>...</td>
<td>cpu-idle</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RATE-CPU</th>
<th>RATE-CPU</th>
<th>RATE-CPU</th>
<th>RATE-CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpu-user</td>
<td>cpu-system</td>
<td>cpu-interrupt</td>
<td>cpu-idle</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>0</td>
<td>92</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>0</td>
<td>92</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>0</td>
<td>91</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>0</td>
<td>91</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>0</td>
<td>90</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n_{(1)}-agg-mean</th>
<th>n_{(1)}-agg-max</th>
<th>fraucion</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 35: Data Gathering and Preprocessing
Next, feature extraction was used to gain information from the data set. Two main feature categories were used: character-level distributions and global statistics.

**Character-Level Distributions** Motivated by the success in Sherlock [47], a similar character-level distribution feature extraction process was used. These are features describing the distribution of characters in a column. Specifically, the count of all 96 ASCII-printable characters (i.e., digits, letters, and punctuation characters, excluding whitespaces) within each value of a column are computed. These counts are then aggregated with 10 statistical functions (i.e., any, all, mean, variance, min, max, median, sum, kurtosis, skewness), which results in 960 features. Example features include “whether all values contain a ‘:’ character” and the “median number of ‘-’ characters.” For instance, Table 2 shows various distributions of the character ‘n’, where \( n_{[n]}\text{-agg-max} \) refers to the aggregate maximum number of n’s in a sequence of command output values. Therefore, a command that has been run in succession with values \(["up", "down", "up", "down", "up", "down"]\) would have a \( n_{[n]}\text{-agg-max} \) value of 1, such as in the first row in Table 2. Continuing to look at this example, we can observe that the \( n_{[n]}\text{-agg-all} \) column, which represents if all the characters are ‘n’, has a value of 0 for the first row. This is because none of the values “up” and “down”, have ‘n’ as all their values.

**Global Statistics** The global statistics describes high-level statistical characteristics of columns. For example, the “column entropy” feature describes how uniformly values are distributed. These features help differentiate between types that contain more repeated values, such as the various enumerations, from types that contain many unique values, such as name. Other types, like CPU rates and file sizes, may consist of many numerical characters, which are captured by the “mean of the number of numerical characters in
Table 2: Character-level Distribution Feature Subset

<table>
<thead>
<tr>
<th>n_n-agg-all</th>
<th>n_n-agg-any</th>
<th>n_n-agg-kurtosis</th>
<th>n_n-agg-max</th>
<th>n_n-agg-mean</th>
<th>n_n-agg-median</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>1.0</td>
<td>-2.000000</td>
<td>1.0</td>
<td>0.500</td>
<td>0.5</td>
</tr>
<tr>
<td>0.0</td>
<td>1.0</td>
<td>-1.733333</td>
<td>1.0</td>
<td>0.375</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>1.0</td>
<td>3.142857</td>
<td>1.0</td>
<td>0.125</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>3.142857</td>
<td>1.0</td>
<td>0.125</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>-3.000000</td>
<td>0.0</td>
<td>0.000</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>-3.000000</td>
<td>0.0</td>
<td>0.000</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>-3.000000</td>
<td>0.0</td>
<td>0.000</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>-3.000000</td>
<td>0.0</td>
<td>0.000</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>-3.000000</td>
<td>0.0</td>
<td>0.000</td>
<td>0.0</td>
</tr>
</tbody>
</table>

values.” Examples of data can be found in Table 3, which are also in the same format as the Character-level distribution, where each row represents the output of a single XML tag that has been run in succession. The column avg\_word\_cells represents the average number of words in throughout a sequence of values, where ["up", "down", "up", "down", "up", "down", "up", "down"] would produce the value of 1, as each entry only contains a single word. Inversely, the column avg\_num\_cells would contain a value of 0, as there are no numerical cells.

Table 3: Global Statistic Feature Subset

<table>
<thead>
<tr>
<th>avg_num_cells</th>
<th>avg_spec_cells</th>
<th>avg_text_cells</th>
<th>avg_word_cells</th>
<th>col_entropy</th>
<th>frac_num_cells</th>
<th>frac_text_cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>3.00</td>
<td>1.0</td>
<td>1.000000</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>2.75</td>
<td>1.0</td>
<td>0.954434</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>2.25</td>
<td>1.0</td>
<td>0.543564</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>2.25</td>
<td>1.0</td>
<td>0.543564</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>2.00</td>
<td>1.0</td>
<td>-0.000000</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>2.00</td>
<td>1.0</td>
<td>-0.000000</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>2.00</td>
<td>1.0</td>
<td>-0.000000</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>2.00</td>
<td>1.0</td>
<td>-0.000000</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

6.3.6 Sherlock Approach

Sherlock is informed by existing commercial and open source systems for data preparation and analysis, as well as prior research work on ontology-based, feature-based, probabilistic, and synthesized approaches
to semantic type detection. Furthermore, the python code for Sherlock was made available online, and thus was used as the preliminary machine-learned approach. Figure 37 shows the underlying Sherlock model architecture. Each of the boxes in the diagram represents a layer in the model.

While training the neural network, a keras History object is automatically updated with the loss and accuracy metrics of the model. These values are saved for each epoch, which represents the number of passes through the entire training dataset. Looking at the plots in Figure 38, we can examine the loss and accuracy curves to make observations about the model. From the loss curve (a), we can observe that there is a good fit, as the training loss is decreasing to a point of stability. However, the large gap between the training and testing curves indicates that the training data may be unrepresentative, meaning that there is not enough training data as compared to the testing data. The accuracy plot in (b) also suggests that there may be some overfitting, due to the large gap between the training and testing curves. Overfitting is undesirable as it means the model may be too specialized and unable to perform well on new data.

While insight gained from the Sherlock approach was useful, after careful evaluation of the values produced during feature extraction of both the paragraph vectors and word embeddings, it was apparent that these values were producing little information about the data. For instance, due to the lack of data
including words such as “the”, “who”, and “which”, similarity vector values were not informative to the model, and excluded the need for word embedding feature extraction. Similarly, the lack of words in each data point negated the need to represent values as vectors of paragraphs. Thus, these sub-networks were removed a new neural network was constructed, as can be seen in Figure 39. Compared to the architecture in Figure 37, we can observe that the layers connected to input_3 and input_4 have been removed.

Figure 38: Sherlock Accuracy and Loss on the training and validation datasets over training epochs
The network was compiled using categorical accuracy function, categorical cross-entropy loss function, and an Adam optimizer. Both the loss and accuracy plots can be seen in Figure 40. The accuracy plot (b) still shows evidence of over-fitting, but also indicates that there may be too few training epochs, due to the fact that the testing curve is not leveling off to a constant value. This is also supported in plot (a), as neither training nor testing loss curves appear to be levelling off either. These results suggest that the network should be trained for longer, in order to give it more time to learn.
6.3.7 Decision Tree Approach

In an attempt to simplify the model presented in Sherlock, we trained a decision tree classifier using only character-level distributions and global statistics as features. A decision tree was chosen due to its ability to generate insights on the data itself. The DecisionTreeClassifier from the sklearn python package was used for the actual implementation, with the random state set to zero and all other parameters left at default values.

A 75/25 split was used to separate the data into random training and testing subsets. Stratified sampling was also used to preserve the percentage of samples for each class.

Figure 41 represents a decision tree generated where the maximum depth was set to four. Each internal node has a decision rule that splits the data; this example in particular uses the Gini index splitting criterion, which means that the tree is created in such a way that the degree or probability of a particular variable being wrongly classified when it is randomly chosen is minimized. The darker colored leaf nodes represent classification where the Gini impurity score is zero, meaning that the node is pure, as all of its records belong to the same class. As the maximum depth was set to four, the tree was not able to effectively predict all 11 classes, and in this example, was only able to classify five different types. This statement is also reinforced by plot (a) in Figure 42, as we observe that the top 6 values in the on the plot, which represent each of the 11 classes, have an accuracy score (computed as F1 score) of 0. Looking at plot (b), we see the feature importance scores, which represent how useful or valuable each feature was in the construction of the decision tree. It is computed as the (normalized) total reduction of the criterion brought by that feature rendering it unitless. By cross-referencing figures 41 and 42 (b), we can observe why length-agg-min was the root of the decision tree, as it was the feature with the highest Gini importance (> 0.3), meaning that it held the most
weight in determining classification.

Figure 41: Decision Tree (maximum depth = 4)

![Decision Tree Diagram]

(a) Class Accuracy  (b) Feature Importance

Figure 42: Accuracy and Feature Importance for the Decision Tree (maximum depth = 4)
6.3.8 Random Forest Approach

In order to improve the accuracy of the Decision Tree and control over-fitting, a Random Forest classifier was used. Specifically, the `sklearn.ensemble.RandomForestClassifier` package. The classifier was trained on a random state of 0, with all other values left at default.

![Random Forest Feature Importance](image)

**Figure 43: Random Forest Feature Importance**

As another measure against overfitting, feature reduction was performed on the random forest model. The `SelectFromModel` function was used, which is a meta-transformer that can be used along with any estimator that has a `feature_importances` attribute after fitting. The features are considered unimportant and removed if the corresponding feature importance values are below a threshold parameter. As tree-based estimators can be used to compute feature importance, the random forest model itself was used to generate feature importance, which in turn, was used to discard irrelevant features. The model was then re-trained on the pruned data set. In total, the number of features was reduced from 960 to 103, and the F1 score increased from 0.64 to 0.71, meaning that the model’s accuracy increased by 7%.

6.4 Evaluation

We evaluate the performance of the various models that have been described in the previous section using both quantitative and qualitative metrics, including accuracy, precision, recall, time, and scalability. The following section works to compare the results of the various approaches throughout the investigation. All experiments were run in a Jupyter Notebook environment on a macOS computer with a 2.6 GHz Intel Core i7 CPU and 16 Gigabytes of memory.
6.4.1 Machine Learned vs. Conditional Logic

The first main evaluation is between the machine-learned and conditional matching-based approach. This is the only evaluation in which there are no comparable quantitative metrics; therefore, we look at the evaluation from a holistic view. First, it can be argued that a machine-learned approach could simply produce the conditional logic which would have been manually implemented for a matching-based approach. Thus, we can assume that a machine-learned approach would require less user input and time. This also ties in with concerns over scalability. In theory, any new data added to Junos commands would be able to be quickly learned by the model using little human feedback, whereas a matching-based approach would more likely require a human to manually make changes to a conditional logic code-base.

There is certainly a compromise between the two approaches that may be essential for future development. Currently, the machine learned model is unable to tell whether information is important. Therefore, it may be beneficial to add a conditional logic phase after the model prediction in order to perform the final labelling of important and unimportant. On this same thread, it may also be useful to cross-validate the outcome of the machine-learned model with any insights from the XML tag name. A conditional logic approach also has the advantage of quick solutions to small and easy logic checks. Overall, it can be concluded that neither approach is perfect by itself, and that the best solution may be to utilize the benefits of each approach to create a tiered system.

6.4.2 Sherlock vs. Reduced Sherlock

Next, we can compare the original Sherlock model against the reduced Sherlock model. There are two main metrics we can use: training/testing time and accuracy. In order to obtain a comparison of both training and testing time, each model was trained and tested 10 times, with the average times computed for each condition. The training set was comprised of 174 rows, while the validation set had 87, and the testing set 44, respectively.

The results of the evaluation can be in Figure 44, where we can observe that the original Sherlock model takes over half a second for training. However, there is a negligible difference in testing time.

In order to evaluate testing accuracy, both models were re-trained 10 times, with the average of the 10 predictions used as the final score. This can be seen in Figure 45, where the standard deviation of each set of trials is represented through error bars. There is a significant difference in accuracy scores, with the reduced model 0.21 higher than the original Sherlock model, for an F1 score of 0.93.
We can also look at individual class accuracy. While the reduced Sherlock network was able to predict all of the classes, the original model failed to classify \texttt{RATE-SIZE}, \texttt{RATE-CPU}, and \texttt{COUNTER-UP}. This is most likely due to the fact that the two extra sub-networks in the Sherlock architecture are primarily used to predict textual data, while all three of the aforementioned classes are numerical.
Figure 46: Individual class accuracy for Sherlock and Reduced Sherlock

One last metric we can look at is feature extraction time. Feature extraction is a vital part of the process, as in order to create training and testing datasets, feature extraction is required. To evaluate these times, feature extraction was performed 10 times, with the average of the 10 times and standard deviation used in Figure 47. We can observe that the Sherlock model takes almost double the amount of time for the same amount of input data (n = 174). This can be explained due to the fact that the original Sherlock model contains two extra feature categories which require their own feature extraction process.

Figure 47: Sherlock vs. Reduced Sherlock Feature Extraction Time Comparison (n = 10)

Overall, it appears that the reduced Sherlock model is a better candidate than the original model, due to its shorter total time and significantly better performance.
6.4.3 Decision Tree vs. Random Forest

The next evaluation involves the decision tree and random forest models (before feature selection). Each model was trained and tested using a 25/75 split on 174 rows of data. Looking at Table 4, we can observe that the random forest’s F1 score is 0.06 higher than the decision tree, indicating a 6% increase in accuracy. When applied correctly, it can be assumed that a random forest classifier will always match or outperform a single decision tree. The only real disadvantage to using a random forest is the loss of generated data insights and easy visualization that a decision tree provides. The models also have both high precision and recall, an ideal combination, meaning that they are able to accurately predict a large amount of data.

<table>
<thead>
<tr>
<th>Table 4: Performance Metric Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precision</td>
</tr>
<tr>
<td>Decision Tree</td>
</tr>
<tr>
<td>Random Forest</td>
</tr>
</tbody>
</table>

6.4.4 Multiple Classifier Comparison

Lastly, we can look at the comparison between 5 different classification methods. All methods were implemented using the `sklearn` package with default model parameters, except for models that supported the `random_state` parameter, which were set to 0. Figure 48 displays the accuracy of each model, which have also been trained on the exact same data set (n = 174).

Looking at the figure, we can observe that logistic regression (LR) model had best performance, closely followed by the random forest (RF) and naïve Bayes (NB) models, and then decision tree (DT). Unsurprisingly, the support vector classifier (SVC) did not perform well at all, as it is typically not suited for this type of task. Overall, the top four models performed comparatively, reinforcing the benefits of using a random forest model while also making an argument for both logistic regression and naïve Bayes classifiers as candidates for semantic data type detection.
6.5 Summary

With the need for machine-learned applications directed towards data processing and cleaning tasks ever-growing, we present a myriad of novel techniques that the JSAS team can use as they continue work on the Lightweight Collector. Throughout this project, we were able to explore and develop a variety of approaches for semantic data type detection and create both high-performing classification models and neural networks.

Specifically, we were able to suggest a machine-learned approach over a conditional-based matching approach. Following this, we tailored and trained a neural network based on an existing architecture to outperform classification tasks for the Junos data domain. We were then able to create several classification models to provide insight on the data itself, which would help further efforts to improve classification model performance.

6.6 Future Work

Although several classification models and neural networks were implemented, there is still much more work to be done on semantic data type detection and data masking.

XML Tag An important facet of the Junos data was the actual XML tag name, though only the conditional matching-based approach used this feature. However, it seems like a neural network model may benefit from further validation against the tag name.
**Data Masking**  While the focus of this project was data type detection, the logic for data masking needs to be implemented. This could either be implemented as a conditional logic check after semantic labeling, or as another output in a multi-output neural network.

**Data**  The need for more cleaned, pre-classified data is a constant challenge, and one that would serve this project well. The system we created to obtain data relies on a substantial amount of manual input, due to the nested structure of the raw XML data from Junos devices. Talking to subject matter experts on the structure of Junos data outputs may be beneficial. Furthermore, improving the data acquisition script to implement a flattening functionality would help reduce the amount of time it takes to create both training and testing data sets.

**XML Structure**  Another consideration may be to incorporate the nested structure of the XML data into the architecture of the model itself. This would eliminate the need to flatten and transpose the data into a tabular format, and would allow the model to be used even further on the edge.

**Model Accuracy and Scalability**  In line with the need for more data, the accuracy of the chosen model may need to be improved before it can be used in a production-ready system. Currently, the highest accuracy achieved was 95%. However, intended for a system that processes magnitudes of data, a model that allows even 5% of data to be classified incorrectly can quickly add up. Furthermore, extensive tests on scalability may be required to ensure that the model will be able to handle vasts amounts of data.

**Deployment**  Most of the code was written in a Jupyter Notebook Python environment. However, in able to deploy the system into the LWC pipeline, the model code will need to be re-written in an AWS Lambda function. Furthermore, the model will need to be able to read from streams of XML data, as that will be the final data source.
7 Conclusion

The overall goal of the LWC project was to automate the collection of network device data in order to empower customers to make more informed decisions regarding their networks. We were able to assist in this mission by contributing to several parts of the LWC pipeline. Our work can be summarized as an exploration and development of the flow, reduction, and use of data in the LWC.

Through building a customized data streaming solution and configuring AWS CloudWatch for Greengrass, we were able to identify the key advantages of each approach. Furthermore, within the streaming pipeline we evaluated two unique Lambda function implementations for extracting log data from the system. These explorations offered significant insight into the usefulness and feasibility of developing a maintaining an in-house streaming solution. Also, this project aims to identify the techniques that best leverage the Greengrass environment to craft a effective and expandable software component.

We were able to meet the overall requirements of the Diagnostics Tool established by the team. The Diagnostics Tool provides a closer look into various aspects of the LWC through its three main components, the File Explorer, MQTT Client, and Redis Client. We also made design and development decisions influenced by the requirement that the Diagnostics Tool would ultimately be hosted by a long-lived lambda running on the Greengrass core device.

Motivated by the limited amount of work on data semantic data type detection, we were able to explore and develop a variety of novel approaches for semantic labeling. By tailoring existing work to fit our domain, we were able create high-accuracy neural networks. Furthermore, machine learned insights on the data provided valuable information that would help further efforts to improve classification model performance.

In short, we gained near real-time insights about Greengrass performance, designed a dashboard to aid in the diagnosis of LWC issues, and developed a novel technique for semantic data type detection in a network domain. Each of these components addressed different aspects of the overall LWC pipeline. Together, we were able to determine that both the NVIDIA Xavier and AWS Greengrass are viable tools to support the LWC system. Our hope is that Juniper and the DEA team will be able use any insights gained to develop a production-ready system.
References


[21] Node.js v13.10.1 Documentation. URL: https://nodejs.org/api/stream.html#:~:text=There%20are%20four%20fundamental%20stream,read%20(for%20example,%20fs..


[31] A. Bartoli et al. “Inference of Regular Expressions for Text Extraction from Examples”. In: IEEE Transactions on Knowledge and Data Engineering 28.5 (May 2016), pp. 1217–1230. ISSN: 2326-3865. DOI: 10.1109/TKDE.2016.2515587.


Appendix A  Junos Command Outputs

show chassis routing-engine

<rpc-reply xmlns:junos="http://xml.juniper.net/junos/15.1X49/junos">
  <route-engine-information xmlns="http://xml.juniper.net/junos/15.1X49/junos-chassis">
    <route-engine>
      <status>OK</status>
      <temperature junos:celsius="28">28 degrees C / 82 degrees F</temperature>
      <cpu-temperature junos:celsius="28">28 degrees C / 82 degrees F</cpu-temperature>
      <memory-system-total>4096</memory-system-total>
      <memory-system-total-used>1556</memory-system-total-used>
      <memory-system-total-avail>2540</memory-system-total-avail>
      <memory-system-total-util>38</memory-system-total-util>
      <memory-control-plane>2624</memory-control-plane>
      <memory-control-plane-used>840</memory-control-plane-used>
      <memory-control-plane-avail>1784</memory-control-plane-avail>
      <memory-control-plane-util>32</memory-control-plane-util>
      <memory-data-plane>1472</memory-data-plane>
      <memory-data-plane-used>707</memory-data-plane-used>
      <memory-data-plane-avail>765</memory-data-plane-avail>
      <memory-data-plane-util>48</memory-data-plane-util>
      <cpu-user>7</cpu-user>
      <cpu-background>0</cpu-background>
      <cpu-system>3</cpu-system>
      <cpu-interrupt>0</cpu-interrupt>
      <cpu-idle>89</cpu-idle>
      <model>RE-SRXSME-SRX550M</model>
      <serial-number>ACPT6388</serial-number>
      <start-time junos:seconds="1541090928">2018-11-01 09:48:48 PDT</start-time>
      <up-time junos:seconds="41154591">476 days, 7 hours, 49 minutes, 51 seconds</up-time>
      <last-reboot-reason>0x1:power cycle/failure</last-reboot-reason>
      <load-average-one>0.13</load-average-one>
      <load-average-five>0.03</load-average-five>
      <load-average-fifteen>0.01</load-average-fifteen>
    </route-engine>
  </route-engine-information>
  <cli>
    <banner></banner>
  </cli>
</rpc-reply>
Appendix B  AWS Greengrass Core Setup on the NVIDIA Jetson Xavier

This procedure is abstracted from the AWS Greengrass Developer User Guide [10]. Begin setup from the Setting Up Other Devices tab in Module 1. This will provide instructions for initiating the NVIDIA Jetson Xavier as the AWS IoT Greengrass core unit.

If you have not already already flashed the NVIDIA Jetson Xavier with the JetPack SDK, do so before configuring AWS IoT Greengrass.

1. Run the following commands on the Xavier to create user ggc_user and group ggc_group.

   $ sudo adduser --system ggc_user
   $ sudo addgroup --system ggc_group

show version

<rpc-reply xmlns:junos="http://xml.juniper.net/junos/15.1X49/junos">
  <software-information>
    <host-name>jtac-SRX550M-r008</host-name>
    <product-model>srx550m</product-model>
    <product-name>srx550m</product-name>
    <jsr/>
    <junos-version>15.1X49-D140.2</junos-version>
    <package-information>
      <name>junos</name>
      <comment>JUNOS Software Release [15.1X49-D140.2]</comment>
    </package-information>
    <package-information>
      <name>jais</name>
      <comment>AI-Scripts [7.0120191029_1620_barmstrong]</comment>
    </package-information>
  </software-information>
  <cli>
    <banner/>
  </cli>
</rpc-reply>
Dependency Installation

Perform the following steps from the command line on the Xavier.

1. Refresh and upgrade the local package index.

   $ sudo apt update
   $ sudo apt -y upgrade

2. Install the Java 8 runtime on the core device.

   $ sudo apt install openjdk-8-jdk

3. Create a symbolic link in order to map the installed java binary to the proper naming convention recognized by Greengrass.

   $ ln -s /usr/bin/java /usr/bin/java8

4. Verify that Java 8 installed successfully.

   $ java -version

You should see an output similar to:

   > openjdk version "1.8.0_232"
   > OpenJDK Runtime Environment (build 1.8.0_232-Ubuntu-3ubuntu1)
   > OpenJDK 64-Bit Server VM (build 1.8.0_232Ubuntu-3ubuntu1, mixed mode)

5. Install Node.js 12 on the core device through adding the Node.js APT Repository

   $ sudo apt -y install curl dirmngr apt-transport-https lsb-release ca-certificates
   $ curl -sL https://deb.nodesource.com/setup_12.x | sudo -E bash -
   $ sudo apt -y install nodejs

6. Verify that Node.js installed successfully.

   $ node --version

You should see an output similar to:

   > v12.14.1

7. Create a symbolic link in order to map the installed nodejs binary to the proper naming convention recognized by Greengrass.

   $ ln -s /usr/bin/node /usr/bin/nodejs12.x
8. Change to /home/usr directory on the Xavier. Finally, download and run the Greengrass dependency checker. Unzip and execute the script.

```
$ mkdir greengrass-dependency-checker-GGCv1.10.x
$ cd greengrass-dependency-checker-GGCv1.10.x
$ wget https://github.com/aws-samples/aws-greengrass-samples/raw/master/greengrass-
  dependency-checker-GGCv1.10.x.zip
$ unzip greengrass-dependency-checker-GGCv1.10.x.zip
$ cd greengrass-dependency-checker-GGCv1.10.x
$ sudo ./check_ggc_dependencies | more
```

9. Check the output of the script to confirm that all of the required dependencies are properly installed.

**Installing the AWS IoT Greengrass Core Software**

1. Complete Module 2 of the AWS Greengrass Developer Guide. This module can be performed directly on the Xavier. This will require an internet connection on the Xavier in order to access the AWS Console. This will allow you to download your core’s security resources, configuration file and the AWS IoT Greengrass core software directly on the core device, instead of copying them over from an external machine.

   NOTE: You can run `\$ ps aux | grep greengrass.*daemon*` to check if AWS IoT Greengrass is running on the core device.

**Lambda Functions and Core Deployments**

1. Complete Module 3 of the AWS Greengrass Developer Guide. This module can be completed on the Xavier or a personal computer because it only requires action in the AWS Console.

   These steps demonstrate how to create a Python Lambda function using the AWS IoT Greengrass Core SDK for Python. If you wish to instead create a Node.js Lambda function download the AWS IoT Greengrass Core SDK for Node.js and continue with the remaining instructions. Adapt the suggested naming conventions to reflect the newly selected language.