Faculty of Engineering and Technology
Electrical and Computer Engineering Department

LINUX LABORATORY
ENCS3130

Report No.2
Project 2 - Shell Scripting Project
# gNMI-CLI Path Verification and Data Comparison

Prepared by:
Student Name: Raghad Murad Buzia `1212214`
Student Name: Layan burait `1211439`

Instructor: Aziz Qaroush

Teaching assistant: Loor Wael Sawalhi

Section: 1

Date: 24/12/2024

Abstract

Modern network management systems rely on telemetry data via gNMI and operational data obtained through CLI commands, creating a need for consistency validation between these data sources. This project establishes a comprehensive Python-based framework to extract, normalize, and compare gNMI and CLI outputs, ensuring reliable and accurate network state validation.

The methodology involves mapping gNMI paths to their corresponding CLI commands, retrieving data from both sources, and identifying discrepancies such as unit mismatches, format variations, and missing or null fields. The system is designed using three core classes: GNMI for extracting telemetry data from JSON files, CLI for simulating CLI command execution and retrieving outputs, and Comparator for normalizing and comparing the data. A dedicated ReportGenerator class generates detailed reports summarizing discrepancies and matches. Key features include unit conversion (e.g., 1G to 1,000,000,000 bytes), case normalization (e.g., LINK_UP vs. linkup), and precision adjustments (e.g., 31 vs. 31.0%).

By addressing challenges like data format diversity, unit inconsistencies, and missing fields, this project provides a robust and automated framework for validating network telemetry and operational data, ensuring consistency and enhancing the reliability of network management processes.

Table of Contents

List of Figures

List of Tables

1. Introduction

In modern network environments, maintaining consistency between telemetry data and operational data is essential for ensuring reliable network performance. Telemetry data is typically provided by protocols like GNMI in a structured JSON format, while operational data is retrieved via CLI commands in a human-readable format. Discrepancies between these two data sources can lead to misconfigurations or misinterpretations of the network state, potentially impacting the reliability of the entire system.

This project addresses the challenge of validating data consistency between GNMI and CLI by developing a Python-based framework that automates the processes of data extraction, normalization, and comparison. The system effectively identifies discrepancies, resolves mismatches in units and formats, and generates detailed reports to validate data accuracy and ensure network state reliability.

The methodology involves the following steps:

- **Mapping** GNMI **paths to CLI commands:** Establishing a clear mapping between GNMI telemetry paths and their corresponding CLI commands to enable seamless data retrieval.
- **Data retrieval:** Extracting GNMI outputs in JSON format and CLI outputs in plain text, leveraging dictionaries to simulate outputs for both sources.
- **Normalization and comparison:** Handling differences in units (e.g., 1G vs. 1,000,000,000 bytes), formats (e.g., LINK_UP vs. linkup), and precision (e.g., 31 vs. 31.0%).
- **Report generation:** Generating comprehensive reports to document discrepancies and matches between GNMI and CLI outputs, providing valuable insights for validating network data.

This project employs synthetic data, stored in dictionaries, to simulate gNMI and CLI outputs instead of directly interacting with actual network devices. By addressing challenges such as unit mismatches, missing fields, and diverse data formats, the project demonstrates a robust, modular, and scalable approach to network monitoring and telemetry validation, ensuring consistency and enhancing network reliability.

2. gNMI Data (gNMI Paths and their gNMI and CLI Output)

In this project, we used synthetic data instead of executing CLI and gNMI commands directly on a real device. Instead, we saved the paths and their respective outputs for both gNMI and CLI in dictionaries. This approach allows us to simulate the process without needing access to actual devices. Here is a representation of this data:

Table 1: gNMI Paths Output

gNMI Path	gNMI Output `.json`
/interfaces/interface[name=eth0]/state/counters	{ "in_octets": 1500000, "out_octets": 1400000, "in_errors": 10, "out_errors": 2 }
/system/memory/state	{ "total_memory": 4096000, "available_memory": 1024000 }
/interfaces/interface[name=eth1]/state/counters	{ "in_octets": 200000.00, "out_octets": "100K", "in_errors": 5 }
/system/cpu/state/usage	{ "cpu_usage": 65, "idle_percentage": 35 }
/routing/protocols/protocol[ospf]/ospf/state	{ "ospf_area": "0.0.0.0", "ospf_state": "up" }
/interfaces/interface[name=eth0]/state	{ "admin_status": "up", "oper_status": "up", "MACAddress": "00:1C:42:2B:60:5A", "mtu": 1500.00, "speed": "1K" }
/bgp/neighbors/neighbor[neighbor_address=10.0.0.1]/state	{ "peer_as": 65001, "connection_state": "Established", "received_prefix_count": 120,

	"sent_prefix_count": 95 }

/system/cpu/state	{
	"cpu_usage": 75,
	"user_usage": 45,
	"system_usage": 20,
	"idle_percentage": 25
	}

/ospf/areas/area[id=0.0.0.0]/state	{
	"area_id": "0.0.0.0",
	"active_interfaces": 4,
	"lsdb_entries": 200,
	"adjacencies": [
	{"neighbor_id": "1.1.1.1", "state": "full"},
	{"neighbor_id": "2.2.2.2", "state": "full"}
	]
	}

/system/disk/state	{
	"total_space": 1024000,
	"used_space": 500000,
	"available_space": 524000,
	"disk_health": "good"
	}

/interfaces/interface[name=eth0]/state/oper-status	LINK_UP
/interfaces/interface[name=eth0]/state/admin-status	ACTIVE
/interfaces/interface[name=eth0]/state/speed	400
/system/memory/state/used	361296 bytes
/system/cpu/state/utilization	31
/system/storage/state/used	43

Table 2: GNMI Paths CLI Commands Outputs

gNMI Path	CLI Command	CLI Output
/interfaces/interface[name=eth0]/state/counters	show interfaces eth0 counters	in_octets: 1500000 out_octets: 1400000 in_errors: 10 out_errors: 2

/system/memory/state	show memory	TotalMemory: 4096000 available_memory: 1000000
/interfaces/interface[name=eth1]/state/counters	show interfaces eth1 counters	in_octets: 200000; out_octets: 100000
/system/cpu/state/usage	show cpu	cpu_usage: 65
/routing/protocols/protocol[ospf]/ospf/state	show ospf status	ospf_area: "0.0.0.0"; ospf_state: "down"
/interfaces/interface[name=eth0]/state	show interfaces eth0 status	admin_status: up oper_status: up
	show interfaces eth0 mac-address	mac_address: 00:1C:42:2B:60:5A
	show interfaces eth0 mtu	mtu: 1500
	show interfaces eth0 speed	speed: 1000
/bgp/neighbors/neighbor[neighbor_address=10.0.0.1]/state	show bgp neighbors 10.0.0.1	peer_as: 65001 connection_state: Established
	show bgp neighbors 10.0.0.1 received-routes	received_prefix_count: 120
	show bgp neighbors 10.0.0.1 advertised-routes	sent_prefix_count: 95
/system/cpu/state	show cpu usage	cpu_usage: 75
	show cpu user	user_usage: 45
	show cpu system	system_usage: 20
	show cpu idle	idle_percentage: 25
/ospf/areas/area[id=0.0.0.0]/state	show ospf area 0.0.0.0	area_id: 0.0.0.0 active_interfaces: 4 lsdb_entries: 200
	show ospf neighbors	neighbor_id: 1.1.1.1, state: full neighbor_id: 2.2.2.2, state: full

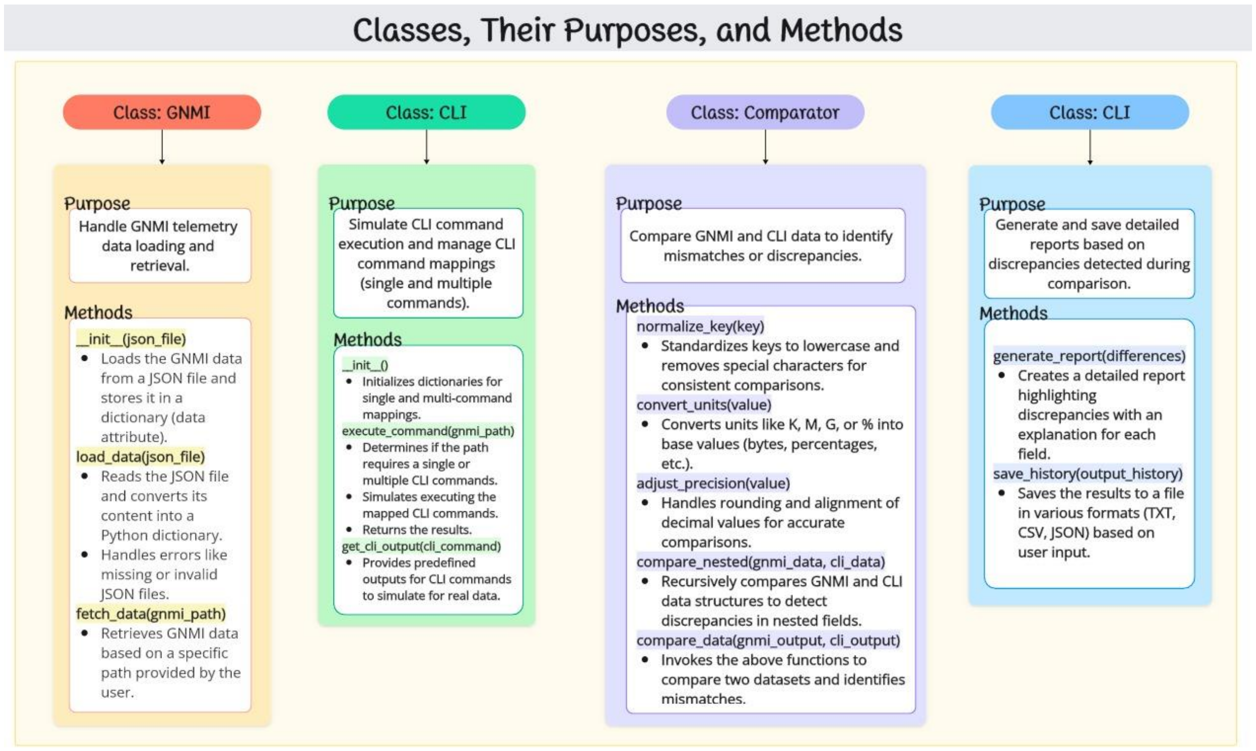
/interfaces/interface[name=eth0]/state/oper-status	show interfaces eth0 status	LinkUp
/interfaces/interface[name=eth0]/state/admin-status	: show interfaces eth0 admin-status	Active
/interfaces/interface[name=eth0]/state/speed	show interfaces eth0 speed	400G
/system/memory/state/used	show memory used	352.97 KB
/system/cpu/state/utilization	show cpu utilization	31.0%
/system/storage/state/used	show storage usage	43.00
/system/disk/state	show disk space	total_space: 1024000 used_space: 500000 available_space: 524000
	show disk health	disk_health: good

3. Methodology

3.1 Background

This section provides an explanation of the project's implementation specifics by analyzing each script, its role, inputs and outputs, and essential features.

The following mind map shown below illustrates the workflow and execution samples:



Classes, Their Purposes, and Methods

Class: GNMI

Purpose
Handle GNMI telemetry data loading and retrieval.

Methods
__init__(json_file)
- Loads the GNMI data from a JSON file and stores it in a dictionary (data attribute).

load_data(json_file)
- Reads the JSON file and converts its content into a Python dictionary.
- Handles errors like missing or invalid JSON files.

fetch_data(gnmi_path)
- Retrieves GNMI data based on a specific path provided by the user.

Class: CLI

Purpose
Simulate CLI command execution and manage CLI command mappings (single and multiple commands).

Methods
__init__()
- Initializes dictionaries for single and multi-command mappings.

execute_command(gnmi_path)
- Determines if the path requires a single or multiple CLI commands.
- Simulates executing the mapped CLI commands.
- Returns the results.

get_cli_output(cli_command)
- Provides predefined outputs for CLI commands to simulate for real data.

Class: Comparator

Purpose
Compare GNMI and CLI data to identify mismatches or discrepancies.

Methods
normalize_key(key)
- Standardizes keys to lowercase and removes special characters for consistent comparisons.

convert_units(value)
- Converts units like K, M, G, or % into base values (bytes, percentages, etc.).

adjust_precision(value)
- Handles rounding and alignment of decimal values for accurate comparisons.

compare_nested(gnmi_data, cli_data)
- Recursively compares GNMI and CLI data structures to detect discrepancies in nested fields.

compare_data(gnmi_output, cli_output)
- Invokes the above functions to compare two datasets and identifies mismatches.

Class: CLI

Purpose
Generate and save detailed reports based on discrepancies detected during comparison.

Methods
generate_report(differences)
- Creates a detailed report highlighting discrepancies with an explanation for each field.

save_history(output_history)
- Saves the results to a file in various formats (TXT, CSV, JSON) based on user input.

Figure 1: Structure and Workflow of the Data Comparison Process Project

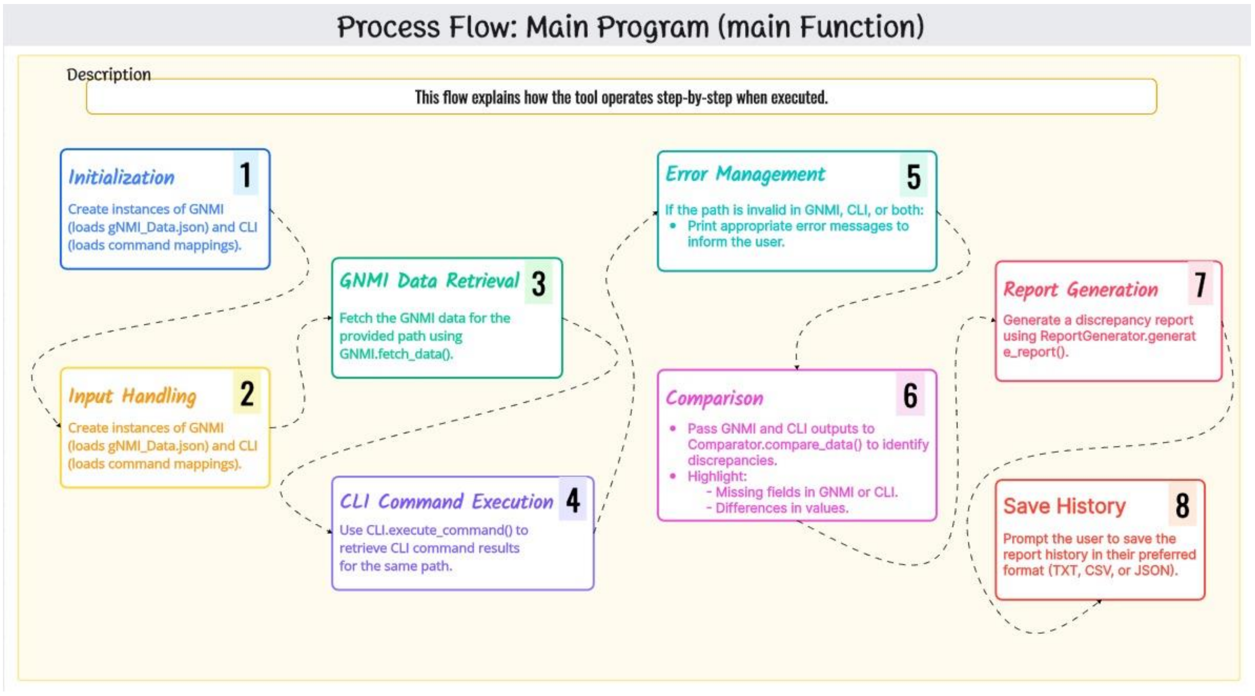
The background of the project revolves around GNMI (GRPC Network Management Interface) and CLI (Command Line Interface) comparisons, focusing on path verification and data consistency in network management systems. These technologies are commonly used in network configuration, monitoring, and management.

gNMI is an advanced network management protocol developed by the IETF (Internet Engineering Task Force) to facilitate communication between network devices and management systems. It allows for more flexible and efficient configuration and monitoring by using gRPC (Google Remote Procedure Call) as the underlying transport protocol. This enables real-time streaming of telemetry data, device configurations, and other network-related information.

CLI, on the other hand, has been a staple in network management for decades. It allows users to interact with devices through textual commands to configure, monitor, and troubleshoot network systems. CLI outputs typically provide the status, configuration, and operational data of network devices, but the format of the data can vary from vendor to vendor.

In the context of this project, the goal is to compare the data retrieved from both GNMI and CLI sources, ensuring consistency and accuracy between the two. Network administrators and engineers often need to verify that the data provided by both interfaces align, especially when transitioning from legacy CLI-based systems to more modern, flexible GNMI systems. This comparison can also help identify discrepancies or issues in network device configurations or telemetry data, which may be critical in maintaining network performance and security.

The project utilizes Python for automation and scripting, employing JSON for GNMI data parsing and CSV for storing the comparison results. The overall goal is to create a system that automates the path verification process, normalizes the data from both sources, and generates meaningful reports that highlight any differences. This will allow for easier analysis and troubleshooting in network management environments.


Process Flow: Main Program (main Function)

Description

This flow explains how the tool operates step-by-step when executed.

Initialization 1
Create instances of GNMI (loads gNMI_Data.json) and CLI (loads command mappings).

Input Handling 2
Create instances of GNMI (loads gNMI_Data.json) and CLI (loads command mappings).

GNMI Data Retrieval 3
Fetch the GNMI data for the provided path using GNMI.fetch_data().

CLI Command Execution 4
Use CLI.execute_command() to retrieve CLI command results for the same path.

Error Management 5
If the path is invalid in GNMI, CLI, or both:
- Print appropriate error messages to inform the user.

Comparison 6
- Pass GNMI and CLI outputs to Comparator.compare_data() to identify discrepancies.
- Highlight:
  - Missing fields in GNMI or CLI.
  - Differences in values.

Report Generation 7
Generate a discrepancy report using ReportGenerator.generate_report().

Save History 8
Prompt the user to save the report history in their preferred format (TXT, CSV, or JSON).

Figure 2: process flow

4. Structure

The structure of the code is organized into several classes and methods, each responsible for handling specific tasks related to GNMI and CLI comparison. Here's a breakdown of the key components:

4.1 Imports:

The code imports the json and csv libraries:

json: This library is used for parsing, writing, and reading JSON files, which are utilized for handling the GNMI data.

csv: This library is used for handling CSV files, typically for storing the comparison results.

4.2 Classes:

GNMI Class: This class handles GNMI data:

Constructor (__init__): Initializes by loading the GNMI data from a JSON file.

load_data(): Reads the JSON file containing GNMI data, handling potential errors such as missing files or invalid formats.

fetch_data(): Retrieves data based on the specified GNMI path.

CLI Class: This class simulates the execution of CLI commands based on gNMI paths:

execute_command(): Simulates the execution of CLI commands, mapping them to corresponding GNMI paths.

get_cli_output(): Returns predefined CLI outputs based on different commands like system stats, interfaces, etc.

Comparator Class: This class compares gNMI data with CLI command outputs:

normalize_keys(): Removes special characters from keys to ensure consistent comparison.

convert_units(): Converts various units (e.g., bytes to megabytes) to make the data comparable.

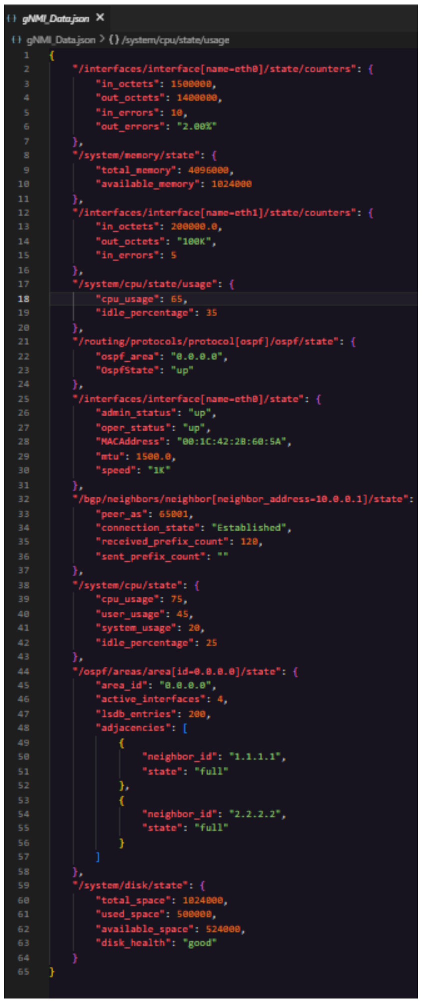
adjust_precision(): Adjusts numerical precision for a more accurate comparison.

compare_data(): Main comparison method, comparing nested data structures between gNMI and CLI outputs, highlighting any differences.

Report Generator Class: This class is responsible for generating a report:

generate_report(): Creates a detailed report showing discrepancies found during the comparison process.

save_history(): Prompts the user to save the history of the comparison in formats such as text, CSV, or JSON.



```json
{
    "/interfaces/interface[name=eth0]/state/counters": {
        "in_octets": 1500000,
        "out_octets": 1400000,
        "in_errors": 10,
        "out_errors": "2.00%"
    },
    "/system/memory/state": {
        "total_memory": 4096000,
        "available_memory": 1024000
    },
    "/interfaces/interface[name=eth1]/state/counters": {
        "in_octets": 200000.0,
        "out_octets": "100K",
        "in_errors": 5
    },
    "/system/cpu/state/usage": {
        "cpu_usage": 65,
        "idle_percentage": 35
    },
    "/routing/protocols/protocol[ospf]/ospf/state": {
        "ospf_area": "0.0.0.0",
        "OspfState": "up"
    },
    "/interfaces/interface[name=eth0]/state": {
        "admin_status": "up",
        "oper_status": "up",
        "MACAddress": "00:1C:42:2B:60:5A",
        "mtu": 1500.0,
        "speed": "1K"
    },
    "/bgp/neighbors/neighbor[neighbor_address=10.0.0.1]/state":
        "peer_as": 65001,
        "connection_state": "Established",
        "received_prefix_count": 120,
        "sent_prefix_count": ""
    },
    "/system/cpu/state": {
        "cpu_usage": 75,
        "user_usage": 45,
        "system_usage": 20,
        "idle_percentage": 25
    },
    "/ospf/areas/area[id=0.0.0.0]/state": {
        "area_id": "0.0.0.0",
        "active_interfaces": 4,
        "lsdb_entries": 200,
        "adjacencies": [
            {
                "neighbor_id": "1.1.1.1",
                "state": "full"
            },
            {
                "neighbor_id": "2.2.2.2",
                "state": "full"
            }
        ]
    },
    "/system/disk/state": {
        "total_space": 1024000,
        "used_space": 500000,
        "available_space": 524000,
        "disk_health": "good"
    }
}
```

Figure 3: GNMI.json file

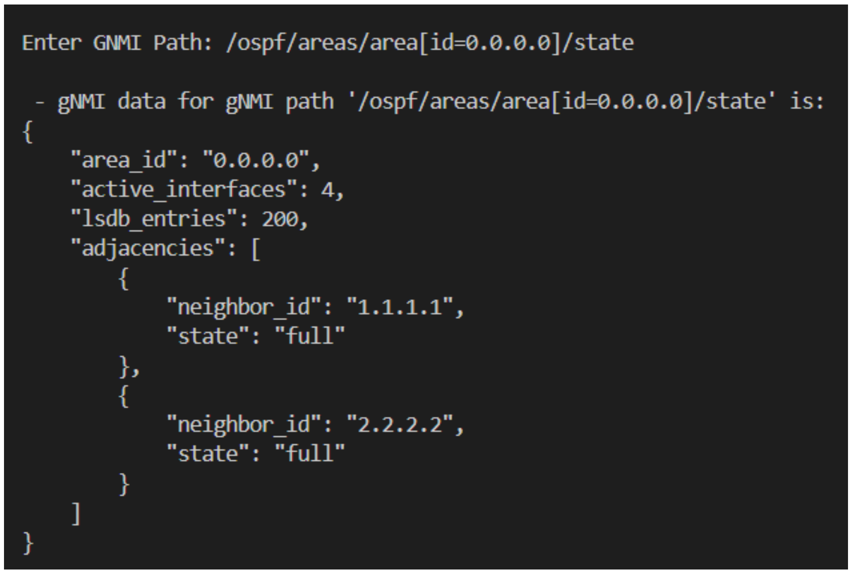
4.3 Flow of Execution:

The GNMI data is loaded via the GNMI class, and the corresponding CLI command outputs are simulated using the CLI class.

The Comparator class compares the two data sets, and any discrepancies are identified and reported.

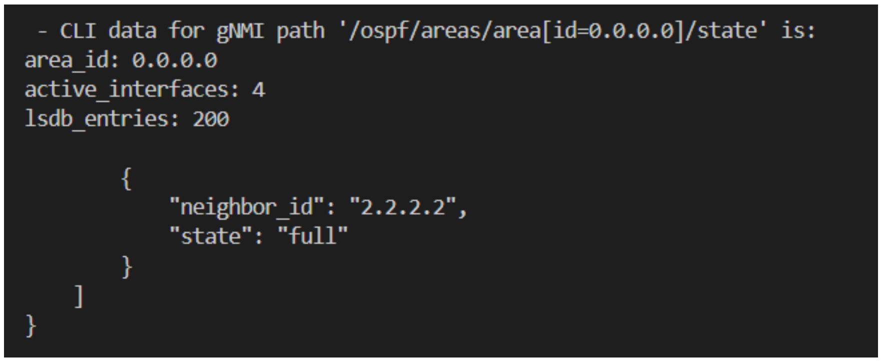
The user can choose to save the history of the test results, which are formatted and stored using the Report Generator class.

This structure ensures that the code is modular and each class has a clear responsibility, making it easy to extend or modify for future needs. The primary workflow focuses on fetching GNMI data, simulating CLI outputs & comparing the results.



```
Enter GNMI Path: /ospf/areas/area[id=0.0.0.0]/state

 - gNMI data for gNMI path '/ospf/areas/area[id=0.0.0.0]/state' is:
{
    "area_id": "0.0.0.0",
    "active_interfaces": 4,
    "lsdb_entries": 200,
    "adjacencies": [
        {
            "neighbor_id": "1.1.1.1",
            "state": "full"
        },
        {
            "neighbor_id": "2.2.2.2",
            "state": "full"
        }
    ]
}
```

Figure 4: Executing & enter GNMI path then the GNMI data of GNMI path as shown above



```
 - CLI data for gNMI path '/ospf/areas/area[id=0.0.0.0]/state' is:
area_id: 0.0.0.0
active_interfaces: 4
lsdb_entries: 200

        {
            "neighbor_id": "2.2.2.2",
            "state": "full"
        }
    ]
}
```

Figure 5: CLI data for GNMI path

4.4 Extraction CLI Output for a specific GNMI Path

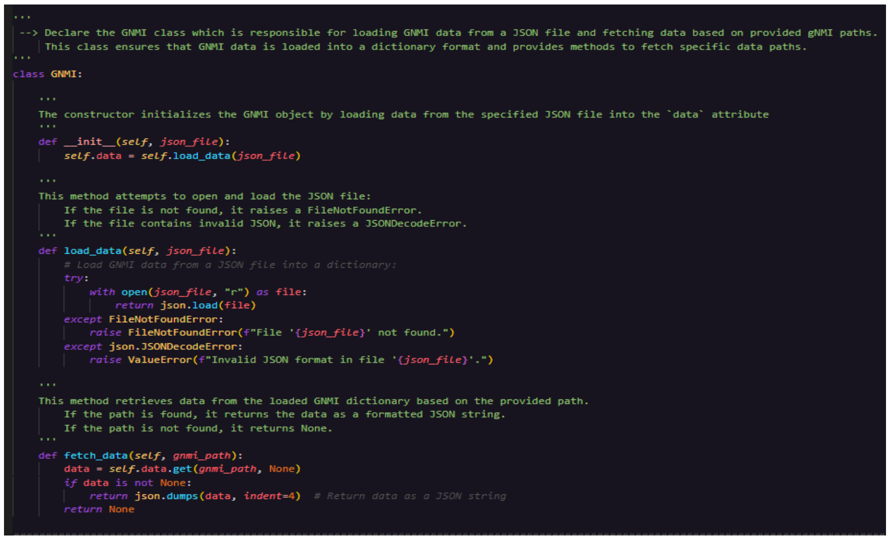
In this section, the process of extracting CLI output for a specific GNMI path is explained. This step is essential to ensure that the data retrieved from the CLI matches the desired GNMI path, enabling effective comparison and validation between the two sources.

The first task in this extraction process involves selecting a specific path in the GNMI interface. A GNMI path is a structured URI that identifies a specific resource or data point within a network device. This path can be used to query network configurations, telemetry, and state information from the device using gNMI's RPC (Remote Procedure Call) methods, such as Get, Set, and Subscribe.

Once the gNMI path is determined, the equivalent CLI command must be identified. Network devices often provide CLI commands that correspond to the data available through gNMI. However, the output from the CLI can vary significantly in format, depending on the vendor and device. Typically, CLI output is presented in a text-based format, which may need to be parsed or converted into a machine-readable format for comparison with gNMI data.

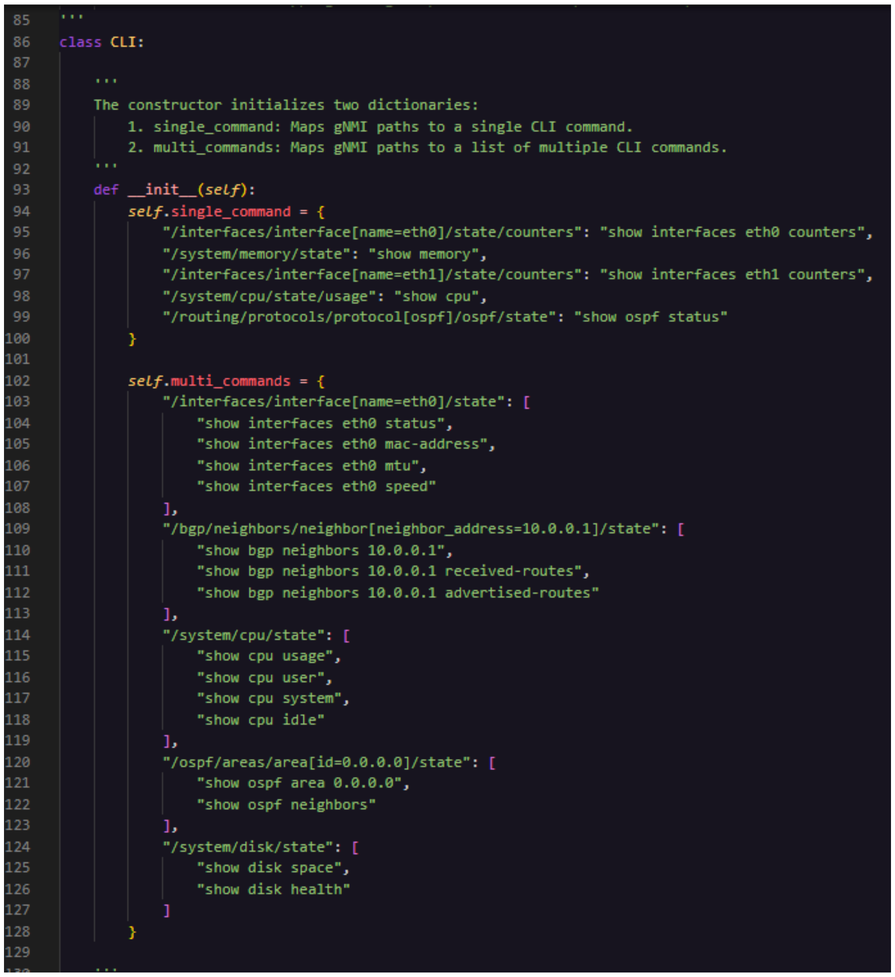
To begin the extraction of CLI output, the following steps are typically followed:

1. **Identify the Specific gNMI Path**: This involves selecting the data point or resource you wish to query in the network device using gNMI. The path might be something like /interfaces/interface[name=eth0]/state/counters, which refers to the interface statistics of a network interface.
2. **Determine the Corresponding CLI Command**: After identifying the gNMI path, the next step is to determine the CLI command that retrieves similar data. For example, a corresponding CLI command might be show interfaces eth0 statistics on a Cisco device or show interface ethernet 0/0 counters on a Juniper device.
3. **Extract the CLI Output**: Once the CLI command is known, the next step is to execute the command on the network device and capture the output. This output is typically in a human-readable format, often presented as plain text.
4. **Normalize the CLI Output**: Since CLI output can vary greatly in structure and formatting between different vendors, it is important to normalize this output into a consistent format for easier comparison. This may involve parsing the output into a structured format, such as JSON or CSV.
5. **Store the CLI Output**: The normalized CLI output is then stored in a file or database for later comparison with GNMI data. This allows for easier retrieval and processing when performing data validation and comparison.
6. **Prepare for Comparison**: With both the GNMI data and CLI output extracted and normalized, the final step is to prepare both sets of data for comparison. This ensures that any discrepancies between the two can be identified, which is critical for verifying the consistency and accuracy of network device configurations.



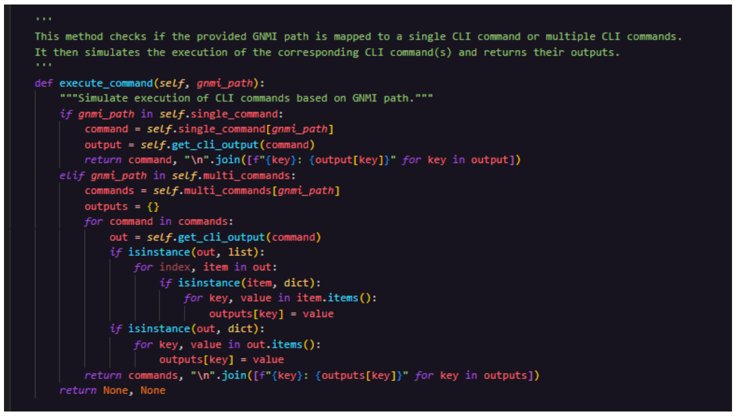
```python
'''
 --> Declare the GNMI class which is responsible for loading GNMI data from a JSON file and fetching data based on provided gNMI paths.
     This class ensures that GNMI data is loaded into a dictionary format and provides methods to fetch specific data paths.
'''
class GNMI:

    '''
    The constructor initializes the GNMI object by loading data from the specified JSON file into the `data` attribute
    '''
    def __init__(self, json_file):
        self.data = self.load_data(json_file)

    '''
    This method attempts to open and load the JSON file:
        If the file is not found, it raises a FileNotFoundError.
        If the file contains invalid JSON, it raises a JSONDecodeError.
    '''
    def load_data(self, json_file):
        # Load GNMI data from a JSON file into a dictionary:
        try:
            with open(json_file, "r") as file:
                return json.load(file)
        except FileNotFoundError:
            raise FileNotFoundError(f"File '{json_file}' not found.")
        except json.JSONDecodeError:
            raise ValueError(f"Invalid JSON format in file '{json_file}'.")

    '''
    This method retrieves data from the loaded GNMI dictionary based on the provided path.
        If the path is found, it returns the data as a formatted JSON string.
        If the path is not found, it returns None.
    '''
    def fetch_data(self, gnmi_path):
        data = self.data.get(gnmi_path, None)
        if data is not None:
            return json.dumps(data, indent=4)  # Return data as a JSON string
        return None
```

Figure 6 GNMI class which is responsible for loading GNMI data from a JSON file and fetching data based on provided gNMI paths



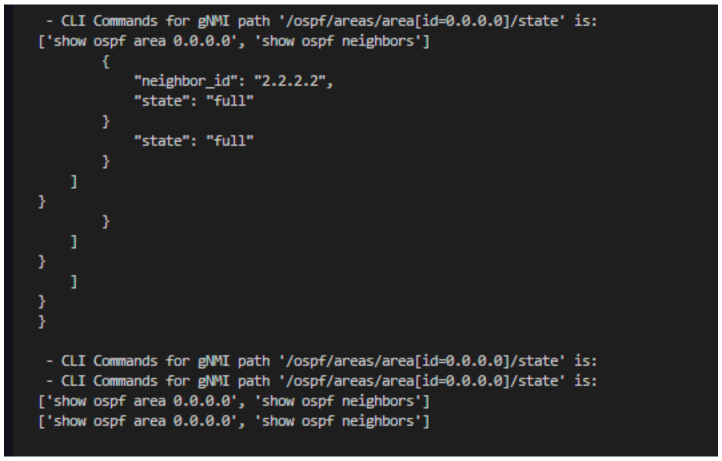
```python
'''
class CLI:

    '''
    The constructor initializes two dictionaries:
        1. single_command: Maps gNMI paths to a single CLI command.
        2. multi_commands: Maps gNMI paths to a list of multiple CLI commands.
    '''
    def __init__(self):
        self.single_command = {
            "/interfaces/interface[name=eth0]/state/counters": "show interfaces eth0 counters",
            "/system/memory/state": "show memory",
            "/interfaces/interface[name=eth1]/state/counters": "show interfaces eth1 counters",
            "/system/cpu/state/usage": "show cpu",
            "/routing/protocols/protocol[ospf]/ospf/state": "show ospf status"
        }

        self.multi_commands = {
            "/interfaces/interface[name=eth0]/state": [
                "show interfaces eth0 status",
                "show interfaces eth0 mac-address",
                "show interfaces eth0 mtu",
                "show interfaces eth0 speed"
            ],
            "/bgp/neighbors/neighbor[neighbor_address=10.0.0.1]/state": [
                "show bgp neighbors 10.0.0.1",
                "show bgp neighbors 10.0.0.1 received-routes",
                "show bgp neighbors 10.0.0.1 advertised-routes"
            ],
            "/system/cpu/state": [
                "show cpu usage",
                "show cpu user",
                "show cpu system",
                "show cpu idle"
            ],
            "/ospf/areas/area[id=0.0.0.0]/state": [
                "show ospf area 0.0.0.0",
                "show ospf neighbors"
            ],
            "/system/disk/state": [
                "show disk space",
                "show disk health"
            ]
        }

```

Figure 7: CLI Command in class CLI MULTI COMMAND & SINGLE ONE



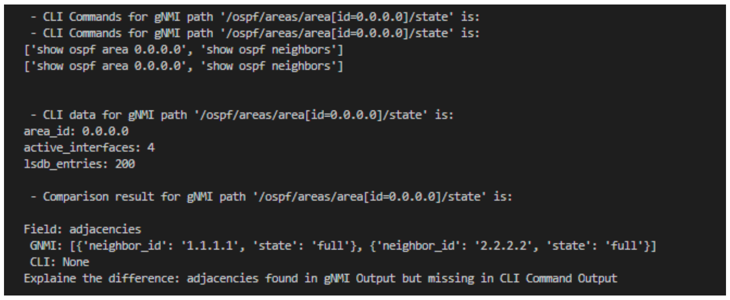
```python
'''
This method checks if the provided GNMI path is mapped to a single CLI command or multiple CLI commands.
It then simulates the execution of the corresponding CLI command(s) and returns their outputs.
'''
def execute_command(self, gnmi_path):
    """Simulate execution of CLI commands based on GNMI path."""
    if gnmi_path in self.single_command:
        command = self.single_command[gnmi_path]
        output = self.get_cli_output(command)
        return command, "\n".join([f"{key}: {output[key]}" for key in output])
    elif gnmi_path in self.multi_commands:
        commands = self.multi_commands[gnmi_path]
        outputs = {}
        for command in commands:
            out = self.get_cli_output(command)
            if isinstance(out, list):
                for index, item in out:
                    if isinstance(item, dict):
                        for key, value in item.items():
                            outputs[key] = value
            if isinstance(out, dict):
                for key, value in out.items():
                    outputs[key] = value
        return commands, "\n".join([f"{key}: {outputs[key]}" for key in outputs])
    return None, None
```

Figure 8method checks if the provided GNMI path is mapped to a single CLI command or multiple CLI commands



```
  - CLI Commands for gNMI path '/ospf/areas/area[id=0.0.0.0]/state' is:
 ['show ospf area 0.0.0.0', 'show ospf neighbors']
         {
             "neighbor_id": "2.2.2.2",
             "state": "full"
         }
             "state": "full"
         }
     ]
 }
         }
     ]
 }
     ]
 }
 }

  - CLI Commands for gNMI path '/ospf/areas/area[id=0.0.0.0]/state' is:
  - CLI Commands for gNMI path '/ospf/areas/area[id=0.0.0.0]/state' is:
 ['show ospf area 0.0.0.0', 'show ospf neighbors']
 ['show ospf area 0.0.0.0', 'show ospf neighbors']
```

Figure 9: Executing CLI Command



```
 - CLI Commands for gNMI path '/ospf/areas/area[id=0.0.0.0]/state' is:
 - CLI Commands for gNMI path '/ospf/areas/area[id=0.0.0.0]/state' is:
['show ospf area 0.0.0.0', 'show ospf neighbors']
['show ospf area 0.0.0.0', 'show ospf neighbors']


 - CLI data for gNMI path '/ospf/areas/area[id=0.0.0.0]/state' is:
area_id: 0.0.0.0
active_interfaces: 4
lsdb_entries: 200

 - Comparison result for gNMI path '/ospf/areas/area[id=0.0.0.0]/state' is:

Field: adjacencies
 GNMI: [{'neighbor_id': '1.1.1.1', 'state': 'full'}, {'neighbor_id': '2.2.2.2', 'state': 'full'}]
 CLI: None
Explaine the difference: adjacencies found in gNMI Output but missing in CLI Command Output
```

Figure 10: CLI_Command_Script.sh and CLI_Output

4.5 Data Comparison

Data comparison is a critical step in validating the consistency and accuracy of network device configurations, especially when comparing outputs from different sources, such as gNMI and CLI. This section outlines the process of comparing the extracted data to identify discrepancies, inconsistencies, or errors between the two sources.

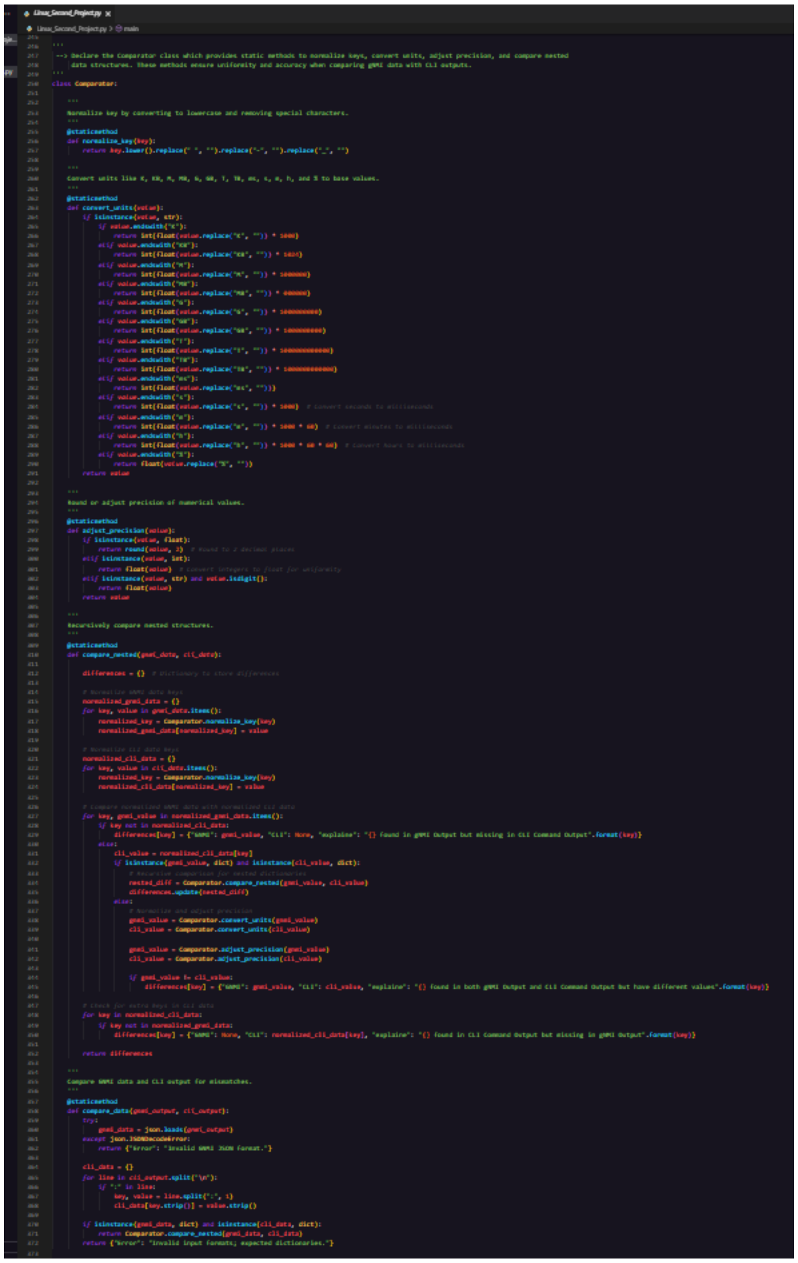
The comparison process involves several key steps to ensure that the data from both gNMI and CLI are aligned and can be compared accurately:

1. **Data Preprocessing and Normalization**: Before starting the comparison, the extracted data from both gNMI and CLI must be preprocessed and normalized into a consistent format. This might involve converting data from one format to another (e.g., from CLI's plain text to JSON or CSV) and standardizing units of measurement, timestamps, or other variables to ensure comparability.
2. **Data Alignment**: The next step is to align the data from both sources. This involves ensuring that the data from the CLI output corresponds correctly to the data from the gNMI output. For example, if comparing interface statistics, the data for each interface from both sources should be matched based on interface identifiers (such as eth0 or ge-0/0/0). The alignment process may also involve sorting or filtering the data to make sure that the comparisons are made on corresponding data points.

4.6 Automating gNMI and CLI Output Comparison

3. Once the data is aligned, the comparison itself can take place. This typically involves checking the equivalence of key values, such as counters, status flags, or configuration settings, across both data sources. Depending on the type of data being compared, this step can be done using:

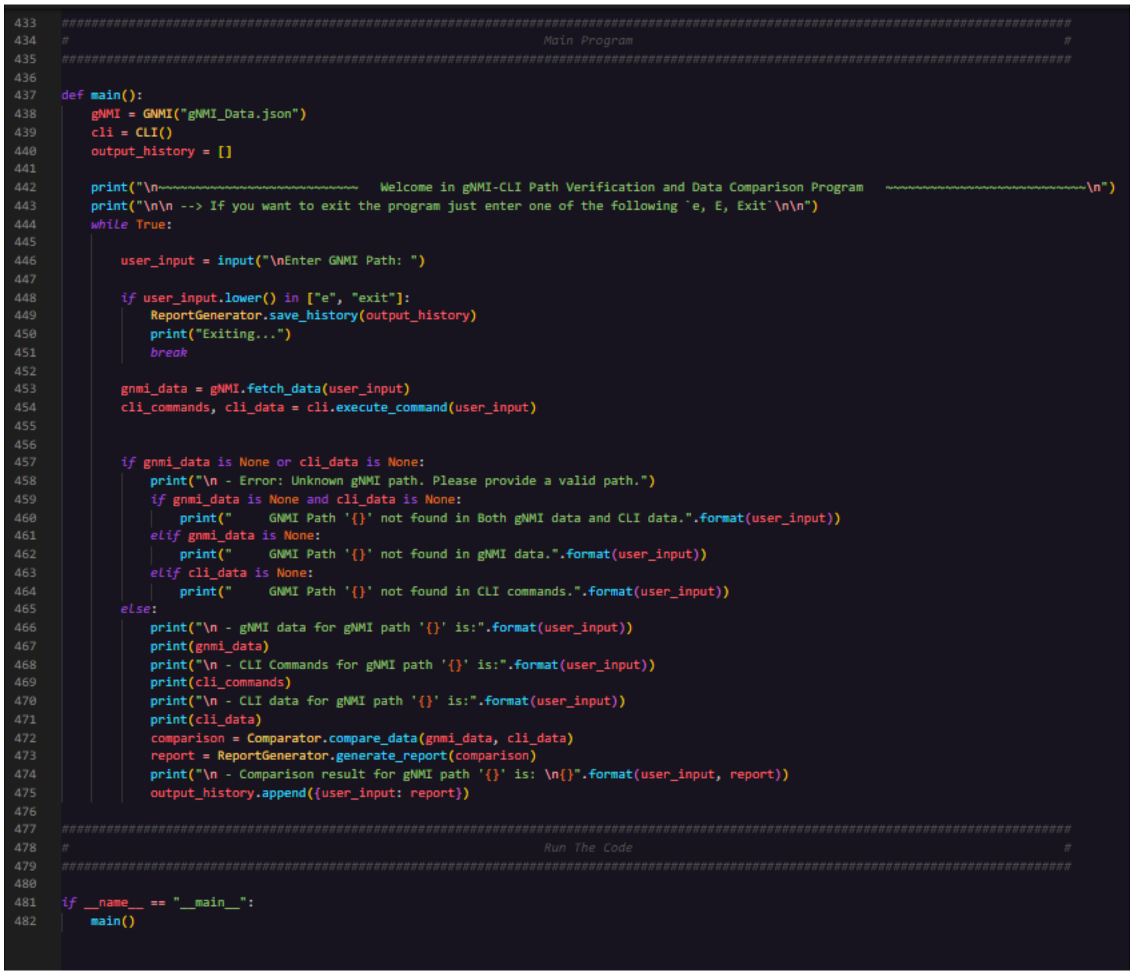
    - **Exact Value Matching**: Where both values from the gNMI and CLI output are compared for exact equivalency.
    - **Range Matching**: Where the values might have slight variations, and a defined tolerance is allowed, especially when comparing performance metrics such as throughput or latency.
    - **Conditional Matching**: Where data from one source may be dependent on certain conditions, and matching is done based on those conditions (e.g., comparing only when certain parameters are true or meet specific criteria).


Linux_Second_Project.py

Linux_Second_Project.py > main

```python
'''
 --> Declare the Comparator class which provides static methods to normalize keys, convert units, adjust precision, and compare nested
     data structures. These methods ensure uniformity and accuracy when comparing gNMI data with CLI outputs.
'''
class Comparator:

    '''
    Normalize key by converting to lowercase and removing special characters.
    '''
    @staticmethod
    def normalize_key(key):
        return key.lower().replace(" ", "").replace("-", "").replace("_", "")

    '''
    Convert units like K, KB, M, MB, G, GB, T, TB, ms, s, m, h, and % to base values.
    '''
    @staticmethod
    def convert_units(value):
        if isinstance(value, str):
            if value.endswith("K"):
                return int(float(value.replace("K", "")) * 1000)
            elif value.endswith("KB"):
                return int(float(value.replace("KB", "")) * 1024)
            elif value.endswith("M"):
                return int(float(value.replace("M", "")) * 1000000)
            elif value.endswith("MB"):
                return int(float(value.replace("MB", "")) * 1048576)
            elif value.endswith("G"):
                return int(float(value.replace("G", "")) * 1000000000)
            elif value.endswith("GB"):
                return int(float(value.replace("GB", "")) * 1073741824)
            elif value.endswith("T"):
                return int(float(value.replace("T", "")) * 1000000000000)
            elif value.endswith("TB"):
                return int(float(value.replace("TB", "")) * 1099511627776)
            elif value.endswith("ms"):
                return int(float(value.replace("ms", "")))
            elif value.endswith("s"):
                return int(float(value.replace("s", "")) * 1000)  # Convert seconds to milliseconds
            elif value.endswith("m"):
                return int(float(value.replace("m", "")) * 1000 * 60)  # Convert minutes to milliseconds
            elif value.endswith("h"):
                return int(float(value.replace("h", "")) * 1000 * 60 * 60)  # Convert hours to milliseconds
            elif value.endswith("%"):
                return float(value.replace("%", ""))
        return value

    '''
    Round or adjust precision of numerical values.
    '''
    @staticmethod
    def adjust_precision(value):
        if isinstance(value, float):
            return round(value, 2)  # Round to 2 decimal places
        elif isinstance(value, int):
            return float(value)  # Convert integers to float for uniformity
        elif isinstance(value, str) and value.isdigit():
            return float(value)
        return value

    '''
    Recursively compare nested structures.
    '''
    @staticmethod
    def compare_nested(gnmi_data, cli_data):

        differences = {}  # Dictionary to store differences

        # Normalize GNMI data keys
        normalized_gnmi_data = {}
        for key, value in gnmi_data.items():
            normalized_key = Comparator.normalize_key(key)
            normalized_gnmi_data[normalized_key] = value

        # Normalize CLI data keys
        normalized_cli_data = {}
        for key, value in cli_data.items():
            normalized_key = Comparator.normalize_key(key)
            normalized_cli_data[normalized_key] = value

        # Compare normalized GNMI data with normalized CLI data
        for key, gnmi_value in normalized_gnmi_data.items():
            if key not in normalized_cli_data:
                differences[key] = {"GNMI": gnmi_value, "CLI": None, "explaine": "{} found in gNMI Output but missing in CLI Command Output".format(key)}
            else:
                cli_value = normalized_cli_data[key]
                if isinstance(gnmi_value, dict) and isinstance(cli_value, dict):
                    # Recursive comparison for nested dictionaries
                    nested_diff = Comparator.compare_nested(gnmi_value, cli_value)
                    differences.update(nested_diff)
                else:
                    # Normalize and adjust precision
                    gnmi_value = Comparator.convert_units(gnmi_value)
                    cli_value = Comparator.convert_units(cli_value)

                    gnmi_value = Comparator.adjust_precision(gnmi_value)
                    cli_value = Comparator.adjust_precision(cli_value)

                    if gnmi_value != cli_value:
                        differences[key] = {"GNMI": gnmi_value, "CLI": cli_value, "explaine": "{} found in both gNMI Output and CLI Command Output but have different values".format(key)}

        # Check for extra keys in CLI data
        for key in normalized_cli_data:
            if key not in normalized_gnmi_data:
                differences[key] = {"GNMI": None, "CLI": normalized_cli_data[key], "explaine": "{} found in CLI Command Output but missing in gNMI Output".format(key)}

        return differences

    '''
    Compare GNMI data and CLI output for mismatches.
    '''
    @staticmethod
    def compare_data(gnmi_output, cli_output):
        try:
            gnmi_data = json.loads(gnmi_output)
        except json.JSONDecodeError:
            return {"Error": "Invalid GNMI JSON format."}

        cli_data = {}
        for line in cli_output.split("\n"):
            if ":" in line:
                key, value = line.split(":", 1)
                cli_data[key.strip()] = value.strip()

        if isinstance(gnmi_data, dict) and isinstance(cli_data, dict):
            return Comparator.compare_nested(gnmi_data, cli_data)
        return {"Error": "Invalid input formats; expected dictionaries."}
```

Figure 11:Declare the Comparator class which provides static methods to normalize keys, convert units, adjust precision, and compare nested



```python
##################################################################################################################
#                                                  Main Program                                                  #
##################################################################################################################

def main():
    gNMI = GNMI("gNMI_Data.json")
    cli = CLI()
    output_history = []

    print("\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~   Welcome in gNMI-CLI Path Verification and Data Comparison Program   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n")
    print("\n\n --> If you want to exit the program just enter one of the following `e, E, Exit`\n\n")
    while True:

        user_input = input("\nEnter GNMI Path: ")

        if user_input.lower() in ["e", "exit"]:
            ReportGenerator.save_history(output_history)
            print("Exiting...")
            break

        gnmi_data = gNMI.fetch_data(user_input)
        cli_commands, cli_data = cli.execute_command(user_input)


        if gnmi_data is None or cli_data is None:
            print("\n - Error: Unknown gNMI path. Please provide a valid path.")
            if gnmi_data is None and cli_data is None:
                print("     GNMI Path '{}' not found in Both gNMI data and CLI data.".format(user_input))
            elif gnmi_data is None:
                print("     GNMI Path '{}' not found in gNMI data.".format(user_input))
            elif cli_data is None:
                print("     GNMI Path '{}' not found in CLI commands.".format(user_input))
        else:
            print("\n - gNMI data for gNMI path '{}' is:".format(user_input))
            print(gnmi_data)
            print("\n - CLI Commands for gNMI path '{}' is:".format(user_input))
            print(cli_commands)
            print("\n - CLI data for gNMI path '{}' is:".format(user_input))
            print(cli_data)
            comparison = Comparator.compare_data(gnmi_data, cli_data)
            report = ReportGenerator.generate_report(comparison)
            print("\n - Comparison result for gNMI path '{}' is: \n{}".format(user_input, report))
            output_history.append({user_input: report})

##################################################################################################################
#                                                  Run The Code                                                  #
##################################################################################################################

if __name__ == "__main__":
    main()
```

Figure 12: Main_Script.sh

Handling Discrepancies: In cases where discrepancies or mismatches are found, it is important to identify the cause of the difference. This might involve further investigation into the data collection process, configuration mismatches, or potential issues with the network device itself. It may also require checking for known vendor-specific differences between CLI and GNMI outputs or issues related to timing (e.g., network configuration changes made during data retrieval).

After performing the comparison, the results should be documented in a structured manner. This includes highlighting any discrepancies, along with additional context such as the data points affected, the degree of difference, and any potential impacts on network operations. Reporting tools might be used to generate visualizations or summaries, making it easier to present the findings to network engineers or stakeholders.

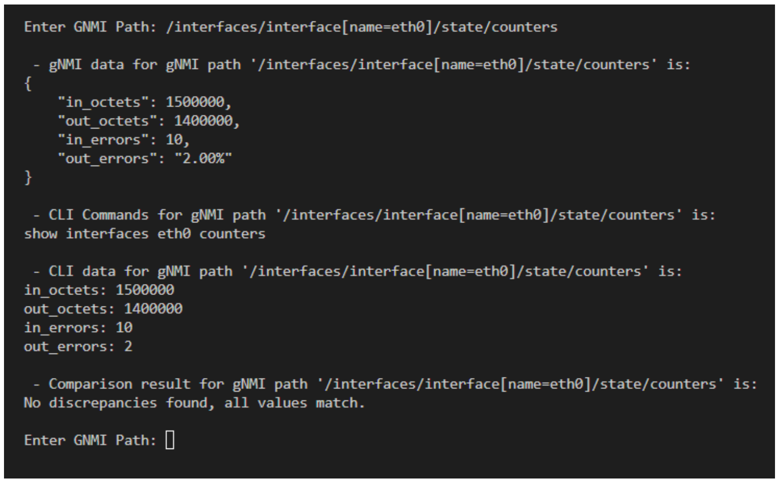
Data comparison is not always a one-time task. The comparison process should be iterative, refining the methods and logic based on feedback, additional data points, or updated network configurations. Continuous improvement can help automate the comparison process for future validation efforts.

By following these steps, the data comparison process ensures that the outputs from gNMI and CLI are consistent and reliable, allowing network engineers to trust the accuracy of the network data being compared. This process is crucial for validating network configurations and ensuring that the network operates as intended.

5. Results

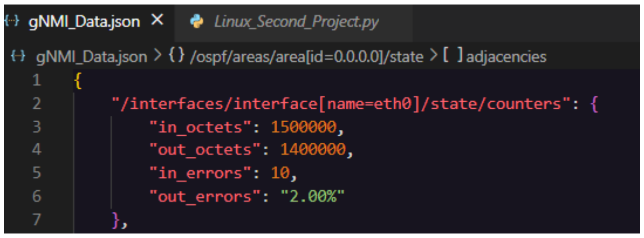
The scripts were tested on all provided paths. Below are examples of test results:

Result of Path 1: /interfaces/interface[name=eth0]/state/counters



```
Enter GNMI Path: /interfaces/interface[name=eth0]/state/counters

 - gNMI data for gNMI path '/interfaces/interface[name=eth0]/state/counters' is:
{
    "in_octets": 1500000,
    "out_octets": 1400000,
    "in_errors": 10,
    "out_errors": "2.00%"
}

 - CLI Commands for gNMI path '/interfaces/interface[name=eth0]/state/counters' is:
show interfaces eth0 counters

 - CLI data for gNMI path '/interfaces/interface[name=eth0]/state/counters' is:
in_octets: 1500000
out_octets: 1400000
in_errors: 10
out_errors: 2

 - Comparison result for gNMI path '/interfaces/interface[name=eth0]/state/counters' is:
No discrepancies found, all values match.

Enter GNMI Path: 
```

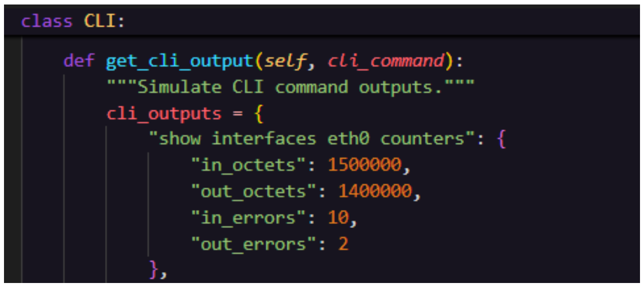
Figure 13: Result for Path 1 `a` - Main execute enter GNMI path



gNMI_Data.json
Linux_Second_Project.py
gNMI_Data.json > {} /ospf/areas/area[id=0.0.0.0]/state > [ ] adjacencies

```
{
    "/interfaces/interface[name=eth0]/state/counters": {
        "in_octets": 1500000,
        "out_octets": 1400000,
        "in_errors": 10,
        "out_errors": "2.00%"
    },
```

Figure 14: Result for Path 1 `b` - GNMI_Output.json

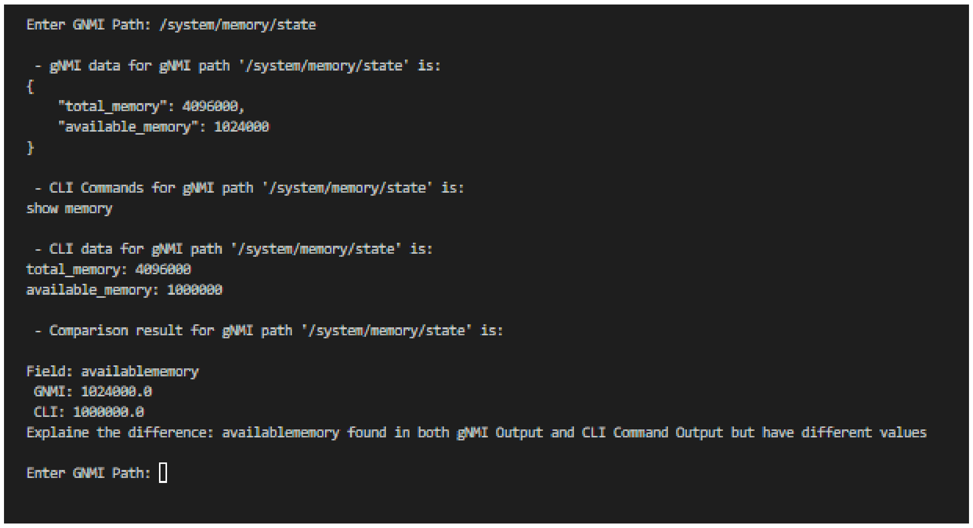


```
class CLI:

    def get_cli_output(self, cli_command):
        """Simulate CLI command outputs."""
        cli_outputs = {
            "show interfaces eth0 counters": {
                "in_octets": 1500000,
                "out_octets": 1400000,
                "in_errors": 10,
                "out_errors": 2
            },
```

Figure 15:CLI OUT

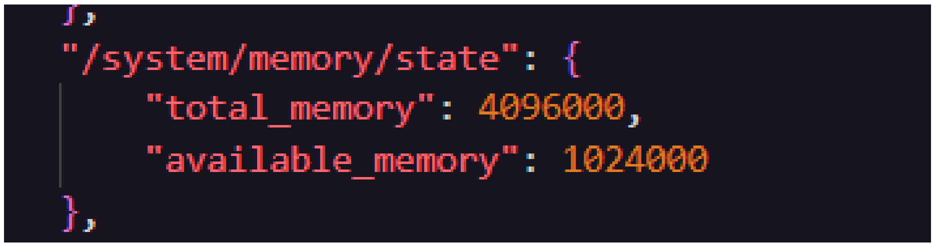
As shown above in both all value as same then the result was all value match.

Result of Path 2: /system/memory/state



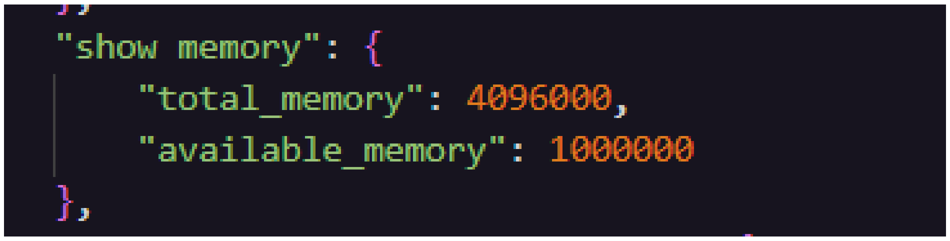
```
Enter GNMI Path: /system/memory/state

 - gNMI data for gNMI path '/system/memory/state' is:
{
    "total_memory": 4096000,
    "available_memory": 1024000
}

 - CLI Commands for gNMI path '/system/memory/state' is:
show memory

 - CLI data for gNMI path '/system/memory/state' is:
total_memory: 4096000
available_memory: 1000000

 - Comparison result for gNMI path '/system/memory/state' is:

Field: availablememory
 GNMI: 1024000.0
 CLI: 1000000.0
Explaine the difference: availablememory found in both gNMI Output and CLI Command Output but have different values

Enter GNMI Path: 
```

Figure 16: Result for Path 2 `a` -Main execute then data od GNMI & CLI THEN THE RESULT OF COMPARISION IS HAVE DIFFERENT VALUE



```
},
"/system/memory/state": {
    "total_memory": 4096000,
    "available_memory": 1024000
},
```

Figure 17:GNMI out of memory path



```
},
"show memory": {
    "total_memory": 4096000,
    "available_memory": 1000000
},
```

Figure 18: OUT OF CLI COMMAND OF ABOVE PATH

Result of Path 3: /interfaces/interface[name=eth1]/state/counters

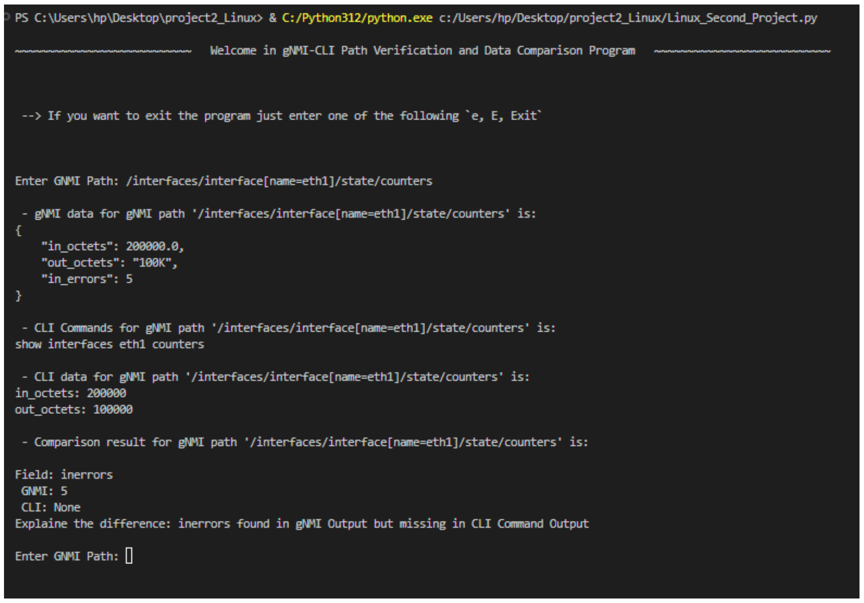

```
PS C:\Users\hp\Desktop\project2_Linux> & C:/Python312/python.exe c:/Users/hp/Desktop/project2_Linux/Linux_Second_Project.py

~~~~~~~~~~~~~~~~~~~~~~~~~~~   Welcome in gNMI-CLI Path Verification and Data Comparison Program   ~~~~~~~~~~~~~~~~~~~~~~~~~~~



 --> If you want to exit the program just enter one of the following `e, E, Exit`



Enter GNMI Path: /interfaces/interface[name=eth1]/state/counters

 - gNMI data for gNMI path '/interfaces/interface[name=eth1]/state/counters' is:
{
    "in_octets": 200000.0,
    "out_octets": "100K",
    "in_errors": 5
}

 - CLI Commands for gNMI path '/interfaces/interface[name=eth1]/state/counters' is:
show interfaces eth1 counters

 - CLI data for gNMI path '/interfaces/interface[name=eth1]/state/counters' is:
in_octets: 200000
out_octets: 100000

 - Comparison result for gNMI path '/interfaces/interface[name=eth1]/state/counters' is:

Field: inerrors
 GNMI: 5
 CLI: None
Explaine the difference: inerrors found in gNMI Output but missing in CLI Command Output

Enter GNMI Path:
```

Figure 19: Result for Path 3 `a` - Main execute & THE OUT OF BOTH GNMI & CLI THERE ARE MISSING IN CLI COMMAND OUTPUT

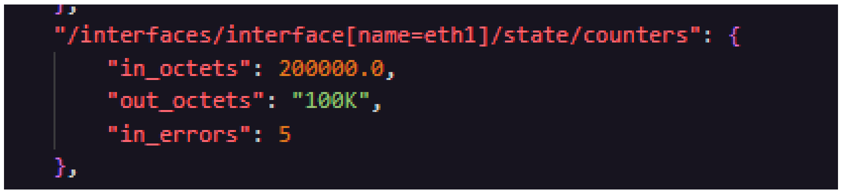

```
    "/interfaces/interface[name=eth1]/state/counters": {
        "in_octets": 200000.0,
        "out_octets": "100K",
        "in_errors": 5
    },
```

Figure 20: Result for Path 3 `b` - GNMI.JSION

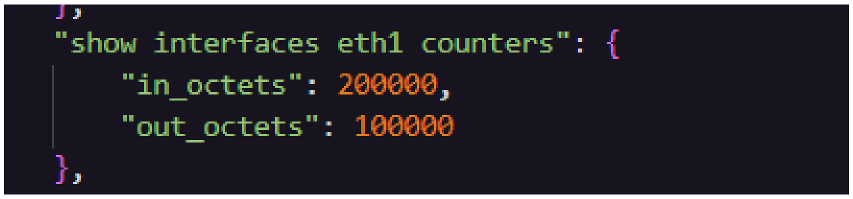
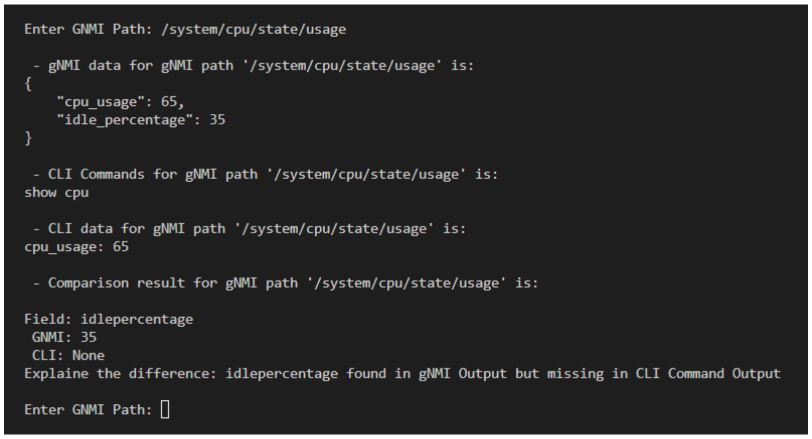

```
    "show interfaces eth1 counters": {
        "in_octets": 200000,
        "out_octets": 100000
    },
```

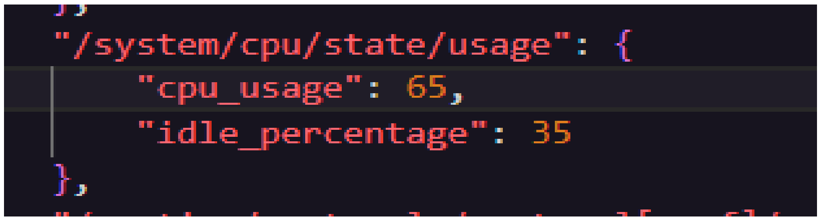
Figure 21: Result for Path 3 `c` - CLI_OUT COMMAND IN CLI CLASS THAT IN SINGLE COMMAND

Result of Path 4: /system/cpu/state/usage



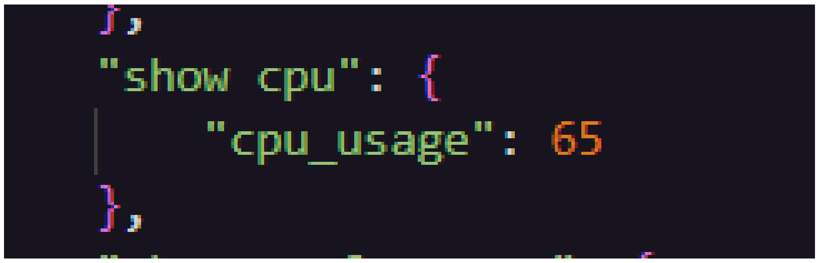
```
Enter GNMI Path: /system/cpu/state/usage

 - gNMI data for gNMI path '/system/cpu/state/usage' is:
{
    "cpu_usage": 65,
    "idle_percentage": 35
}

 - CLI Commands for gNMI path '/system/cpu/state/usage' is:
show cpu

 - CLI data for gNMI path '/system/cpu/state/usage' is:
cpu_usage: 65

 - Comparison result for gNMI path '/system/cpu/state/usage' is:

Field: idlepercentage
 GNMI: 35
 CLI: None
Explaine the difference: idlepercentage found in gNMI Output but missing in CLI Command Output

Enter GNMI Path: 
```

Figure 22: Result for Path 4 `a` - Main execute thar are missing in CLI command output



```
"/system/cpu/state/usage": {
    "cpu_usage": 65,
    "idle_percentage": 35
},
```

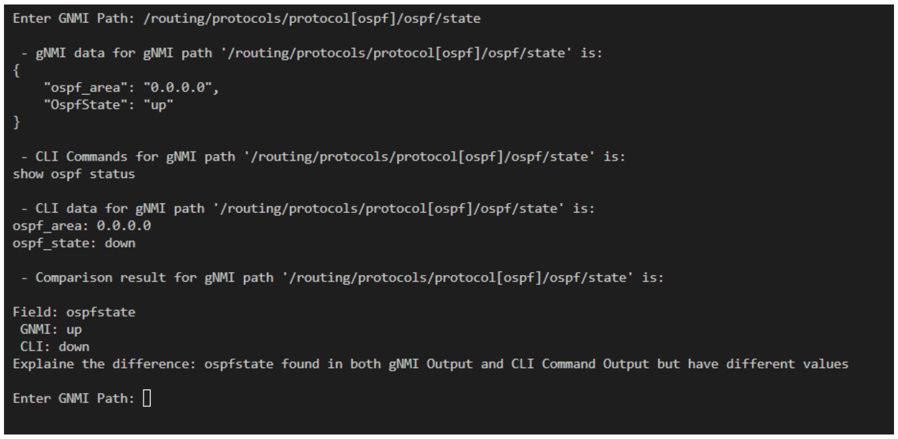
Figure 23: Result for Path 4 `b` - GNMI_Output.json



```
"show cpu": {
    "cpu_usage": 65
},
```

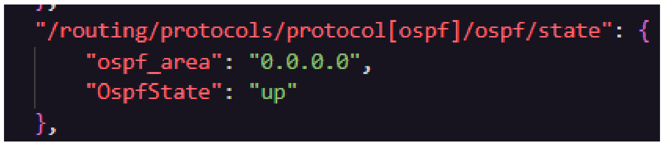
Figure 24: Result for Path 4 `c` - CLI_Output in class of CLI

Result of Path 5: /routing/protocols/protocol[ospf]/ospf/state



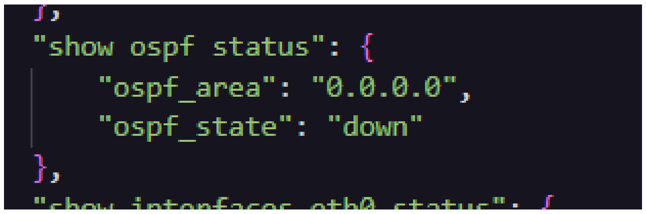
```
Enter GNMI Path: /routing/protocols/protocol[ospf]/ospf/state

 - gNMI data for gNMI path '/routing/protocols/protocol[ospf]/ospf/state' is:
{
    "ospf_area": "0.0.0.0",
    "OspfState": "up"
}

 - CLI Commands for gNMI path '/routing/protocols/protocol[ospf]/ospf/state' is:
show ospf status

 - CLI data for gNMI path '/routing/protocols/protocol[ospf]/ospf/state' is:
ospf_area: 0.0.0.0
ospf_state: down

 - Comparison result for gNMI path '/routing/protocols/protocol[ospf]/ospf/state' is:

Field: ospfstate
 GNMI: up
 CLI: down
Explaine the difference: ospfstate found in both gNMI Output and CLI Command Output but have different values

Enter GNMI Path: 
```

Figure 25: Result for Path 5 `a` - Main execute thar are different value as shown



```
"/routing/protocols/protocol[ospf]/ospf/state": {
    "ospf_area": "0.0.0.0",
    "OspfState": "up"
},
```

Figure 26: Result for Path 5 `b` - out of GNMI.json

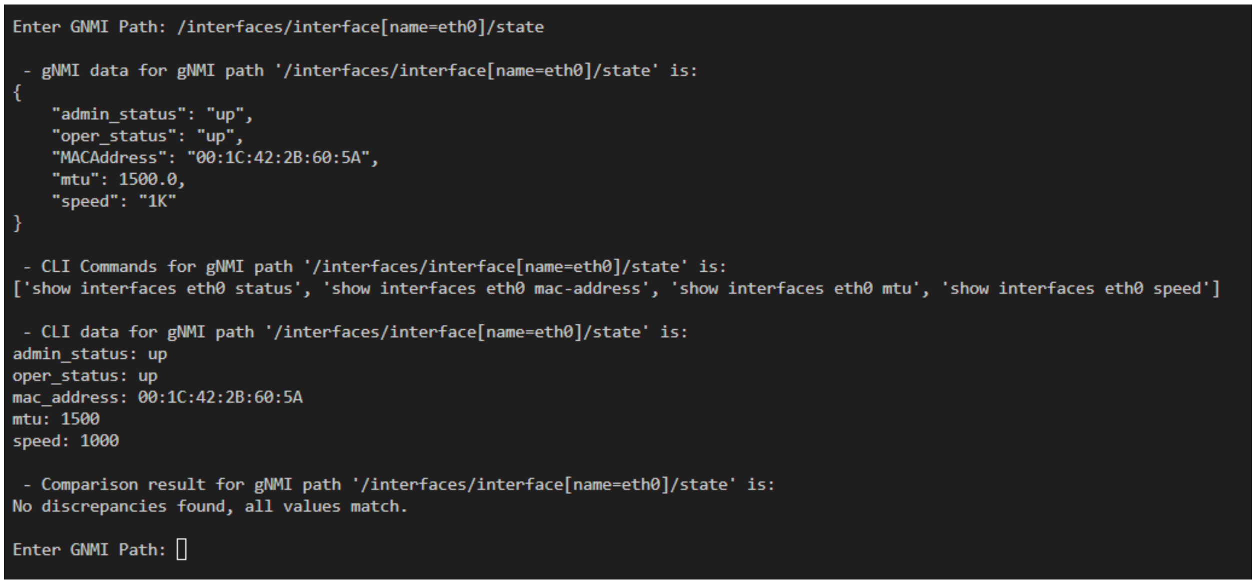


```
"show ospf status": {
    "ospf_area": "0.0.0.0",
    "ospf_state": "down"
},
```

Figure 27: Result for Path 5 `c` - CLI output command

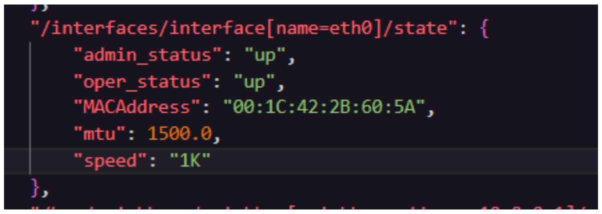
OSPF state in GNMI is up ,but in CLI is down so the value is different as shown above when executed.

Result of Path 6: /interfaces/interface[name=eth0]/state



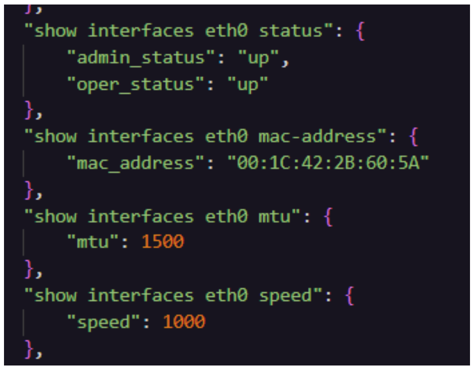
```
Enter GNMI Path: /interfaces/interface[name=eth0]/state

 - gNMI data for gNMI path '/interfaces/interface[name=eth0]/state' is:
{
    "admin_status": "up",
    "oper_status": "up",
    "MACAddress": "00:1C:42:2B:60:5A",
    "mtu": 1500.0,
    "speed": "1K"
}

 - CLI Commands for gNMI path '/interfaces/interface[name=eth0]/state' is:
['show interfaces eth0 status', 'show interfaces eth0 mac-address', 'show interfaces eth0 mtu', 'show interfaces eth0 speed']

 - CLI data for gNMI path '/interfaces/interface[name=eth0]/state' is:
admin_status: up
oper_status: up
mac_address: 00:1C:42:2B:60:5A
mtu: 1500
speed: 1000

 - Comparison result for gNMI path '/interfaces/interface[name=eth0]/state' is:
No discrepancies found, all values match.

Enter GNMI Path: 
```

Figure 28: Result for Path 6 `a` - Main execute the result is all value is match



```
  "/interfaces/interface[name=eth0]/state": {
      "admin_status": "up",
      "oper_status": "up",
      "MACAddress": "00:1C:42:2B:60:5A",
      "mtu": 1500.0,
      "speed": "1K"
  },
```

Figure 29: Result for Path 6 `b` - GNMI_Output.json

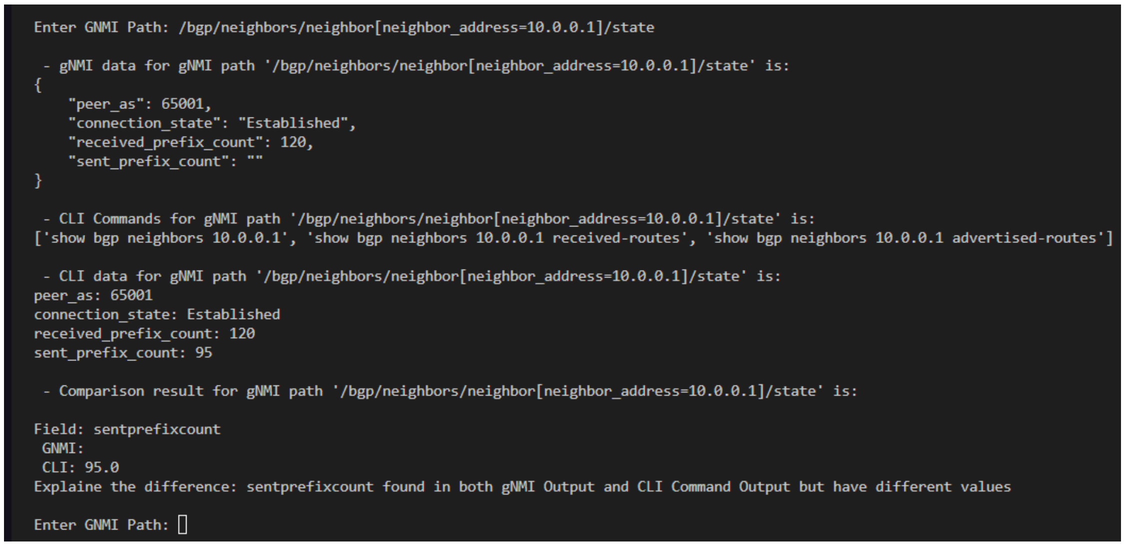


```
  "show interfaces eth0 status": {
      "admin_status": "up",
      "oper_status": "up"
  },
  "show interfaces eth0 mac-address": {
      "mac_address": "00:1C:42:2B:60:5A"
  },
  "show interfaces eth0 mtu": {
      "mtu": 1500
  },
  "show interfaces eth0 speed": {
      "speed": 1000
  },
```

Figure 30: Result for Path 6 `c` - CLI_Output MULTY COMMAND

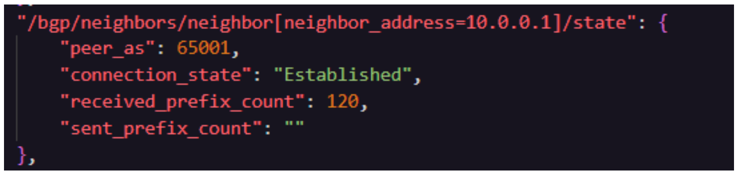
As shown above the result is match ,because all value above as same in both CLI & GNMI

Result of Path 7: /bgp/neighbors/neighbor[neighbor_address=10.0.0.1]/state



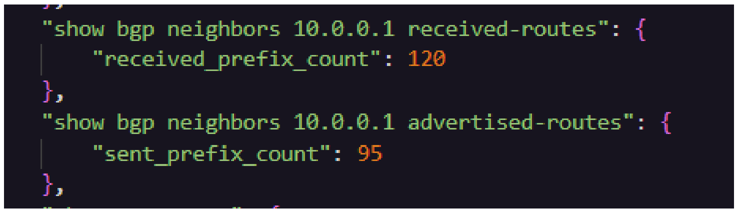
```
Enter GNMI Path: /bgp/neighbors/neighbor[neighbor_address=10.0.0.1]/state

 - gNMI data for gNMI path '/bgp/neighbors/neighbor[neighbor_address=10.0.0.1]/state' is:
{
    "peer_as": 65001,
    "connection_state": "Established",
    "received_prefix_count": 120,
    "sent_prefix_count": ""
}

 - CLI Commands for gNMI path '/bgp/neighbors/neighbor[neighbor_address=10.0.0.1]/state' is:
['show bgp neighbors 10.0.0.1', 'show bgp neighbors 10.0.0.1 received-routes', 'show bgp neighbors 10.0.0.1 advertised-routes']

 - CLI data for gNMI path '/bgp/neighbors/neighbor[neighbor_address=10.0.0.1]/state' is:
peer_as: 65001
connection_state: Established
received_prefix_count: 120
sent_prefix_count: 95

 - Comparison result for gNMI path '/bgp/neighbors/neighbor[neighbor_address=10.0.0.1]/state' is:

Field: sentprefixcount
 GNMI:
 CLI: 95.0
Explaine the difference: sentprefixcount found in both gNMI Output and CLI Command Output but have different values

Enter GNMI Path:
```

Figure 31: Result for Path 7 `a` - Main_Script.sh execute have different value



```
"/bgp/neighbors/neighbor[neighbor_address=10.0.0.1]/state": {
    "peer_as": 65001,
    "connection_state": "Established",
    "received_prefix_count": 120,
    "sent_prefix_count": ""
},
```

Figure 32: Result for Path 7 `b` - GNMI_Output.json

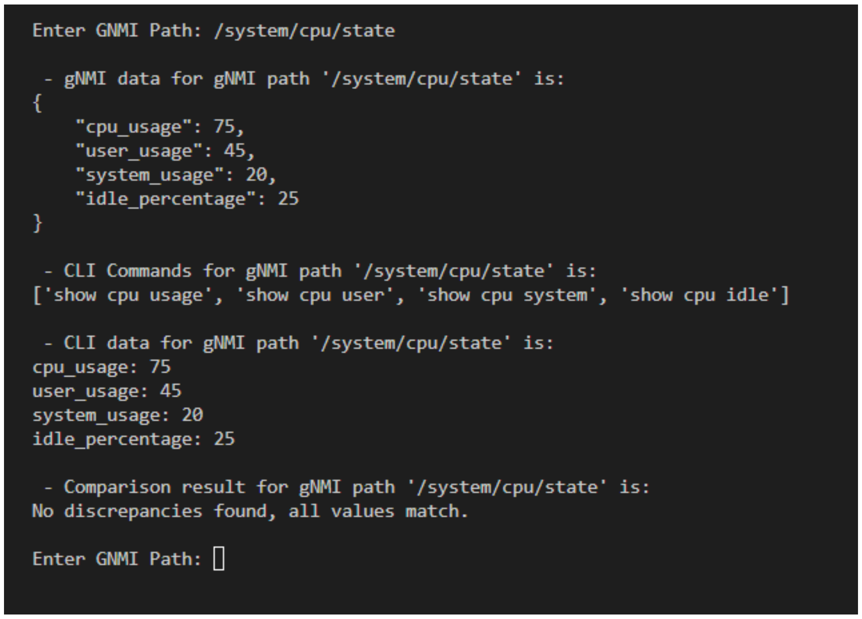


```
  "show bgp neighbors 10.0.0.1 received-routes": {
      "received_prefix_count": 120
  },
  "show bgp neighbors 10.0.0.1 advertised-routes": {
      "sent_prefix_count": 95
  },
```

Figure 33: Result for Path 7 `c` - CLI_Output.sh

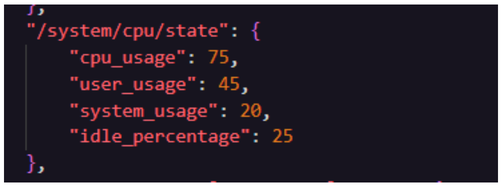
The first value in both is 120 ,but the second value was different in GNMI was null compare with CLI out = 95 ,So that the result was have different value as shown above.

Result of Path 8: /system/cpu/state



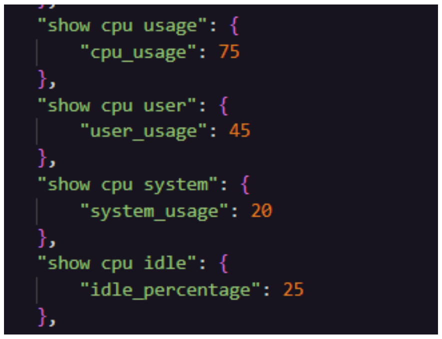
```
Enter GNMI Path: /system/cpu/state

 - gNMI data for gNMI path '/system/cpu/state' is:
{
    "cpu_usage": 75,
    "user_usage": 45,
    "system_usage": 20,
    "idle_percentage": 25
}

 - CLI Commands for gNMI path '/system/cpu/state' is:
['show cpu usage', 'show cpu user', 'show cpu system', 'show cpu idle']

 - CLI data for gNMI path '/system/cpu/state' is:
cpu_usage: 75
user_usage: 45
system_usage: 20
idle_percentage: 25

 - Comparison result for gNMI path '/system/cpu/state' is:
No discrepancies found, all values match.

Enter GNMI Path: 
```

Figure 34: Result for Path 8 `a` - Main_Script.sh execute 1



```
"/system/cpu/state": {
    "cpu_usage": 75,
    "user_usage": 45,
    "system_usage": 20,
    "idle_percentage": 25
},
```

Figure 35: Result for Path 8 `b` - GNMI_Output.json

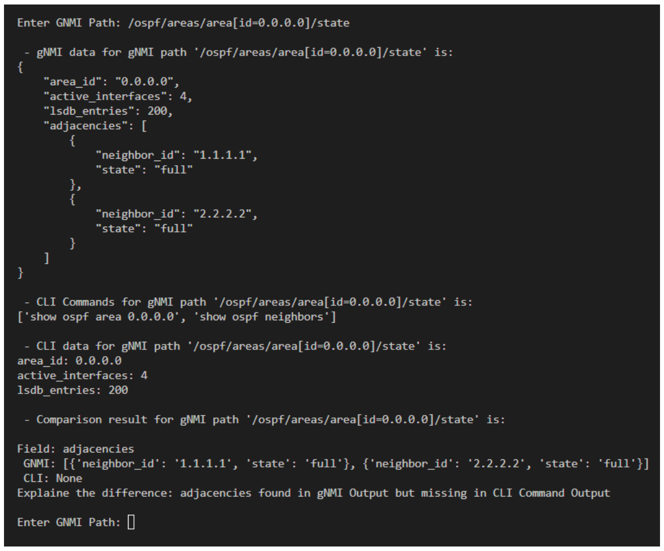


```
"show cpu usage": {
    "cpu_usage": 75
},
"show cpu user": {
    "user_usage": 45
},
"show cpu system": {
    "system_usage": 20
},
"show cpu idle": {
    "idle_percentage": 25
},
```

Figure 36: Result for Path 8 `c` - CLI_Output in CLI def get_cli_output FUNCTION IN CLI CLASS

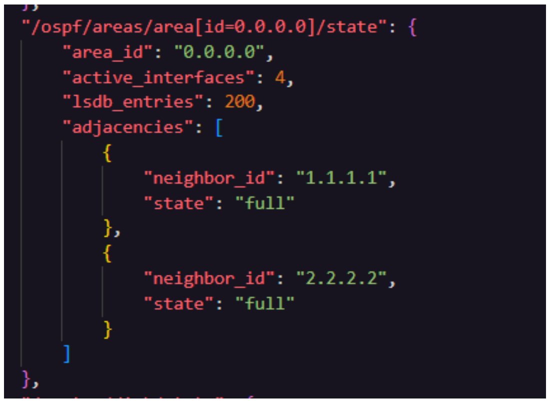
As shown above all value in both GNMI & CLI as same then the result when executed was all vale match.

Result of Path 9: /ospf/areas/area[id=0.0.0.0]/state



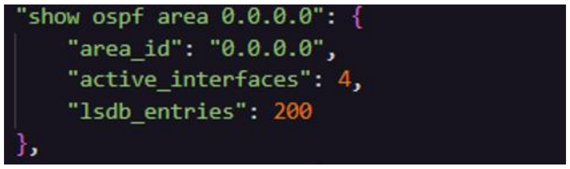
```
Enter GNMI Path: /ospf/areas/area[id=0.0.0.0]/state

 - gNMI data for gNMI path '/ospf/areas/area[id=0.0.0.0]/state' is:
{
    "area_id": "0.0.0.0",
    "active_interfaces": 4,
    "lsdb_entries": 200,
    "adjacencies": [
        {
            "neighbor_id": "1.1.1.1",
            "state": "full"
        },
        {
            "neighbor_id": "2.2.2.2",
            "state": "full"
        }
    ]
}

 - CLI Commands for gNMI path '/ospf/areas/area[id=0.0.0.0]/state' is:
['show ospf area 0.0.0.0', 'show ospf neighbors']

 - CLI data for gNMI path '/ospf/areas/area[id=0.0.0.0]/state' is:
area_id: 0.0.0.0
active_interfaces: 4
lsdb_entries: 200

 - Comparison result for gNMI path '/ospf/areas/area[id=0.0.0.0]/state' is:

Field: adjacencies
 GNMI: [{'neighbor_id': '1.1.1.1', 'state': 'full'}, {'neighbor_id': '2.2.2.2', 'state': 'full'}]
 CLI: None
Explaine the difference: adjacencies found in gNMI Output but missing in CLI Command Output

Enter GNMI Path: 
```

Figure 37: Result for Path 9 `a` - Main_Script.sh execute



```
"/ospf/areas/area[id=0.0.0.0]/state": {
    "area_id": "0.0.0.0",
    "active_interfaces": 4,
    "lsdb_entries": 200,
    "adjacencies": [
        {
            "neighbor_id": "1.1.1.1",
            "state": "full"
        },
        {
            "neighbor_id": "2.2.2.2",
            "state": "full"
        }
    ]
},
```

Figure 38: Result for Path 9 `b` - GNMI_Output.json

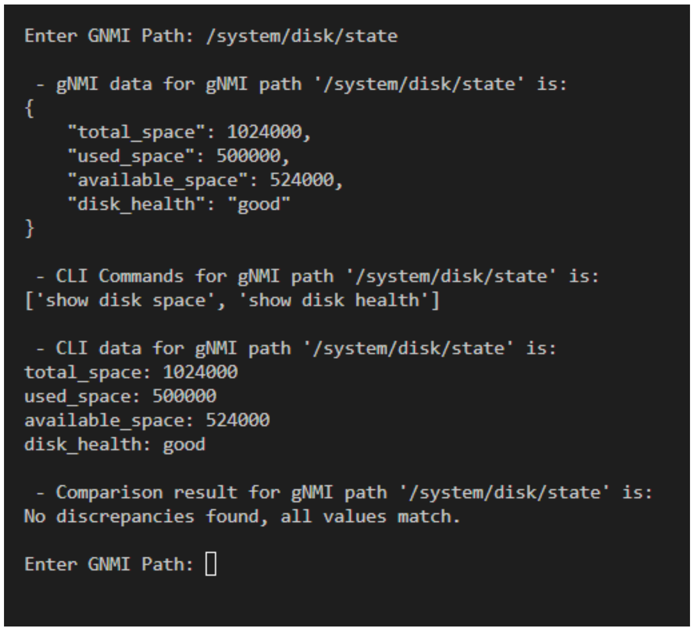


```
"show ospf area 0.0.0.0": {
    "area_id": "0.0.0.0",
    "active_interfaces": 4,
    "lsdb_entries": 200
},
```

Figure 39: Result for Path 9 `c` - CLI_Output of def get_cli_output FUNCTION IN CLI CLASS

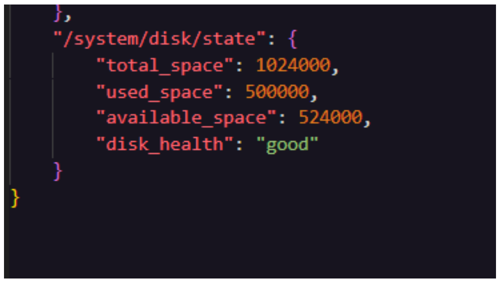
As shown above the vale in both not match ,because adjacencies found in GNMI but there are missing value of CLI

Result of Path 10: /system/disk/state



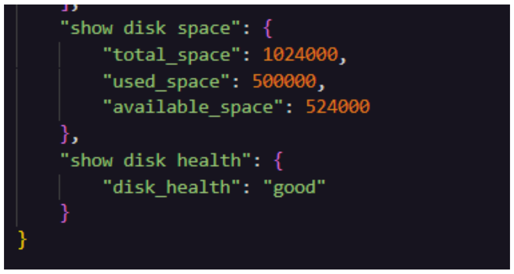
```
Enter GNMI Path: /system/disk/state

 - gNMI data for gNMI path '/system/disk/state' is:
{
    "total_space": 1024000,
    "used_space": 500000,
    "available_space": 524000,
    "disk_health": "good"
}

 - CLI Commands for gNMI path '/system/disk/state' is:
['show disk space', 'show disk health']

 - CLI data for gNMI path '/system/disk/state' is:
total_space: 1024000
used_space: 500000
available_space: 524000
disk_health: good

 - Comparison result for gNMI path '/system/disk/state' is:
No discrepancies found, all values match.

Enter GNMI Path: 
```

Figure 40: Result for Path 10 `a` - Main execute enter path



```
    },
    "/system/disk/state": {
        "total_space": 1024000,
        "used_space": 500000,
        "available_space": 524000,
        "disk_health": "good"
    }
}
```

Figure 41: Result for Path 10 `b` - GNMI_Output.json



```
    "show disk space": {
        "total_space": 1024000,
        "used_space": 500000,
        "available_space": 524000
    },
    "show disk health": {
        "disk_health": "good"
    }
}
```

Figure 42: Result for Path 10 `c` - CLI_Output

As shown above the result was all value match ,because all value same in both GNMI & CLI.

6. Discussion

6.1 Code Analysis

The modular design of the project ensures clarity and ease of implementation. Each script is tailored to perform a specific function:

- GNMI_Script.sh retrieves telemetry data in JSON format, simulating real-world gNMI outputs.
- CLI_Command_Script.sh maps gNMI paths to corresponding CLI commands and retrieves operational data.
- compare_script.sh normalizes and compares the outputs, addressing discrepancies in units, formats, and precision.

This structure minimizes redundancy and allows for easy debugging and future scalability. For example, new gNMI paths and CLI commands can be added seamlessly without significant changes to the existing scripts.

6.2 Results Evaluation

The project demonstrates consistent validation across most tested paths, highlighting the importance of addressing unit and format discrepancies. Key achievements include:

- Unit Conversion: Successfully handled cases like 1G (gNMI) being converted to 1000 Mbps (CLI).
- Case Normalization: Resolved mismatches like LINK_UP (gNMI) vs. linkup (CLI).
- Precision Adjustments: Matched values with differing decimal formats, e.g., 31 (gNMI) vs. 31.0 (CLI).

6.3 Challenges Encountered

- Handling Missing or Extra Field: Some gNMI paths included fields that were missing in CLI outputs, requiring the system to identify and report these discrepancies clearly.
- Unit Discrepancies: Differences in units, such as KB vs. bytes, posed challenges in comparison but were resolved using dynamic unit conversion logic.
- MAC Address Handling: Preserving non-numeric data like MAC addresses required special handling to prevent unwanted conversions.

6.Conclusion

This project underscores the critical importance of ensuring consistency between telemetry and operational data in modern network environments. By utilizing gNMI for structured telemetry outputs and CLI commands for operational insights, the developed system successfully bridges the gap between these two data sources, addressing discrepancies that can otherwise lead to misconfigurations or unreliable network interpretations.

The Python-based modular framework integrates data extraction, normalization, and comparison into a cohesive process. Key accomplishments include resolving unit mismatches (e.g., 1G vs. 1,000,000,000 bytes), normalizing case differences (e.g., LINK_UP vs. linkup), and addressing precision variances (e.g., 31 vs. 31.0%). These achievements highlight the system's ability to identify discrepancies and validate data consistency across a variety of network paths.

While this project employs synthetic data for testing and simulation, its design is highly adaptable to real-world scenarios. Potential applications include continuous network monitoring, proactive anomaly detection, and seamless integration with multi-protocol environments. The insights gained from this implementation provide a solid foundation for future enhancements, such as expanding support for additional data formats, improving scalability, and introducing real-time processing capabilities.

In conclusion, this project delivers a reliable and automated solution for network telemetry validation and operational data monitoring. Its robust design and practical applications offer a significant step toward enhancing the accuracy, efficiency, and reliability of modern network management systems.

References

[1]: ENCS3130 Lab Manual

Appendix

[1]: Link of Code on GitHub:

layanbuirat/ENCS3130-Linux-Laboratory-Shell-Scripting-Project-gNMI-CLI-Path-Verification-and-Data-Comparison-