**SUMMER TRAINING REPORT**

**On**

# AUTOMATED DASHBOARD PIPELINE FOR SAOS REGRESSION INFRASTRUCTURE

Submitted to Guru Gobind Singh Indraprastha University, Delhi (India) in partial fulfilment of the requirement for the award of the degree of

## B.TECH

**in**

**INFORMATION TECHNOLOGY**


**Submitted By**

**PRAGYA TRIPATHI Roll.**

**No. 00596303122**




**DEPTT. OF INFORMATION TECHNOLOGY**

**MAHARAJA SURAJMAL INSTITUTE OF TECHNOLOGY,**
**NEW DELHI-110058**


**November 2025**

# ACKNOWLEDGEMENT

A research project owes its success, from commencement to completion, to the individuals who are passionate about the work at various stages. On this page, I would like to express my gratitude to all those who helped me throughout this study.

First, I would like to extend my sincere gratitude to **Dr. Sunesh Malik** (HOD, Department of Information Technology, Maharaja Surajmal Institute of Technology, New Delhi) for allowing me to undergo a 60-day summer training program at Ciena India Private Limited.

I am grateful to my guide, Ms. Manisha Sharma, for her assistance in completing the project assigned to me. Without her friendly help and guidance, developing this project would have been difficult.

I also want to thank Mr. Ashutosh Varshney and Mr. Rajneesh Mishra for their genuine help and inspiration in preparing the final report and presentation.

Last but not least, I sincerely thank all the staff members of Ciena India Private Limited for their support, who made my internship valuable and fruitful.

Submitted By:

Pragya Tripathi (00596303122)

# CERTIFICATE BY COMPANY

Ciena India Private Limited Unitech
Plot No. 13, JSK Towers,
Echelon Institutional Area, Sector 32,
Gurugram, Haryana- 122001, India

+91 124 6254000- Phone
+91 124 6254444- Fax
www.ciena.com

## CERTIFICATE OF COMPLETION OF INTERNSHIP

This is to certify that **Pragya Tripathi** has successfully participated in Internship program with Ciena India Pvt. Ltd.

The service details are as follows:

**Name** : **Pragya Tripathi**

**Reference Code** : **33292**

**Position/Service Title** : **Summer Student**

**Duration of training** : **16-Jun-2025 to 22-Aug-2025**

We wish you all the best in your future endeavors.

Yours Faithfully,

**Ciena India Pvt. Ltd.**

**Authorized Signatory**
**Vice President, People & Culture**

*This certificate is not an accreditation, and thus Ciena disclaims from any legal obligation or for interpretations thereof,arising out of its use.

# CANDIDATE'S DECLARATION

I, **PRAGYA TRIPATHI**, Roll No. 00596303122, B. Tech (Semester- 7th) of the Maharaja Surajmal Institute of Technology, New Delhi hereby declare that the Internship Report entitled **"AUTOMATED DASHBOARD PIPELINE FOR SAOS REGRESSION INFRASTRUCTURE"** is an original work and data provided in the study is authentic to the best of my knowledge. This report has not been submitted to any other Institute for the award of any other degree.

**Pragya Tripathi**

**(**00596303122**)**

**Place:** Maharaja Surajmal Institute of Technology, Affiliated to GGSIPU

**Date:** 1st November' 2025

# ORGANIZATION INTRODUCTION

**Organization Name:** Ciena India Private Limited

**Industry:** Telecommunications

**Headquarters:** 7035 Ridge Road, Hanover, Maryland 21076, United States

**Website:** https://www.ciena.com/

Ciena Corporation is an American networking systems and software. It was established in 1992 and headquartered in Hanover, Maryland, USA. Ciena is a global leader in networking systems, services, and software, primarily focused on providing solutions for telecommunications and data networks. The company specializes in optical and routing systems, services, and automation software. In fact, Ciena supports more than 85 percent of the world's largest service providers as well as cloud and regional service providers, enterprise networks, financial services, healthcare, utilities, media and entertainment, retail, public sector, and all levels of education.

Ciena India is the company's largest R&D facility outside of North America, and has consistently been ranked India's Top Optical Networking Company in 2019 by Voice & Data magazine.

The company reported revenues of $3.63 billion and more than 8,000 employees, as of October 2022. Gary Smith serves as president and chief executive officer.

Customers include AT&T, Meta, Reliance, Deutsche Telekom, KT Corporation and Verizon Communications.

# ABSTRACT

In modern network software development, ensuring the stability and reliability of continuous releases demands efficient regression systems and transparent testing workflows. Ciena's Routing and Switching Division relies on its proprietary Service-Aware Operating System (SAOS), which powers the company's extensive portfolio of networking devices. The Special Design & Regression Team executes numerous automated test suites across different SAOS versions to validate system performance and functionality. However, these regression results were traditionally fragmented across log files and manual reports, leading to significant inefficiencies and delays in quality assurance feedback.

To address these challenges, this internship focused on designing and developing an automated dashboard system that consolidates regression outcomes into a unified, real-time view. The solution integrates backend automation scripts with a Flask-based dashboard to monitor live test cycles, aggregate execution data, and display key metrics such as pass/fail/crash statuses, execution times, and suite-level progress. By enabling centralized access to logs and completion notifications, the platform streamlined the process of tracking multiple regression suites simultaneously and improved overall visibility across teams.

This report highlights the conceptualization, implementation, and optimization of the dashboard pipeline and its seamless integration with existing CI/CD workflows using Jenkins and Linux-based virtual environments. The system not only enhanced collaboration between the DevOps and Special Design & Regression teams but also reduced manual overhead and accelerated feedback loops by approximately 70%. The project represents a significant advancement in Ciena's regression automation framework, reinforcing the importance of integrated monitoring and visualization tools in large-scale software validation environments.

# LIST OF FIGURES

# TABLE OF CONTENT

# CHAPTER 1

# INTRODUCTION

## 1.1 Need & Objective

In the rapidly evolving telecommunications industry, maintaining the performance, reliability, and scalability of network software has become increasingly vital. Ciena's proprietary Service-Aware Operating System (SAOS) powers a diverse range of routing and switching devices that serve global communication networks. As newer builds and features of SAOS are released frequently, rigorous regression testing is essential to ensure backward compatibility, performance stability, and overall software integrity.

The Special Design & Regression Team at Ciena conducts large-scale automated test suites on various SAOS builds to validate these aspects. However, prior to this project, test results were dispersed across multiple directories and log files, requiring manual inspection and report generation. This decentralized process led to inefficiencies in identifying test outcomes, delayed communication of failures, and limited real-time visibility for both developers and quality assurance teams. The absence of a unified monitoring system made regression management complex and time-consuming, particularly during parallel test executions.

To overcome these limitations, the need for a centralized, automated, and interactive dashboard system was identified. The primary objective of this internship was to design and develop a live dashboard pipeline that could automatically parse regression logs, consolidate results, and visualize real-time test statuses such as pass, fail, crash, or queued. Additionally, the system aimed to integrate seamlessly with Jenkins-based CI/CD workflows to trigger automated updates and notify stakeholders upon completion of test cycles. This project ultimately sought to enhance transparency, accelerate feedback loops, and improve operational efficiency across Ciena's SAOS regression infrastructure.

## 1.2 Project Overview

The Service-Aware Operating System (SAOS) is Ciena's proprietary network operating system that powers its Routing and Switching product line. It forms the software foundation for a wide range of networking devices used in carrier-grade and enterprise environments worldwide. SAOS provides a unified, programmable, and highly reliable platform for managing network services, ensuring consistent performance across diverse hardware configurations. To maintain this high level of reliability, Ciena's Special Design & Regression Team performs extensive regression testing on each new SAOS release, validating system functionality, stability, and compatibility before deployment.

The regression testing framework comprises multiple automated test suites that run across different versions of SAOS. These tests generate large volumes of log data distributed across various directories and virtual machines. Prior to this project, engineers had to manually extract and interpret these logs to assess test outcomes, identify failures, and compile summary reports. This manual approach was both time-consuming and error-prone, often leading to delayed insights and inefficient feedback cycles within the QA and DevOps teams.

The internship project was conceived to address these challenges by creating an automated dashboard system capable of aggregating, analysing, and visualizing regression data in real time. The solution integrates Python- and Flask-based backend automation scripts with a web-based dashboard that displays live test progress, execution times, and result statuses such as pass, fail, or crash. Additionally, it integrates seamlessly with Jenkins CI/CD pipelines and Linux-based virtual environments to ensure automated updates and notifications upon test completion. This automated dashboard serves as a centralized interface for tracking multiple test suites simultaneously, enabling faster decision-making, improved collaboration between teams, and enhanced transparency across Ciena's SAOS regression infrastructure.

## 1.3 Software Used

During the internship, I came across many various software, tools and technologies. We will discuss many of those below:

### 1.3.1 Python

**1. Key Features of Python:**

**Figure 1.1:** Python Logo

**Source:** https://hub.docker.com/python

i. **High-Level and Readable Syntax**: Python is known for its clear, concise, and readable syntax, which allows developers to write and maintain complex automation scripts with minimal effort. Its simplicity makes it ideal for designing scalable backend systems and automation frameworks.

ii. **Extensive Standard Library**: Python's comprehensive standard library includes built-in modules for file handling, logging, regular expressions, data parsing, and HTTP communication. This broad functionality eliminates the need for excessive external dependencies, streamlining automation development.

iii. **Cross-Platform Compatibility**: Python runs seamlessly across major operating systems such as Linux, Windows, and macOS. This ensures that scripts and tools developed for regression automation can operate uniformly across various test environments and virtual machines.

iv. **Integration Capabilities**: Python integrates easily with databases, CI/CD tools like Jenkins, web frameworks such as Flask, and shell commands through modules like subprocess. This makes it a perfect choice for projects requiring multi-tool coordination and automation pipelines.

v. **Strong Community and Library Support**: With libraries like Pandas, NumPy, and Requests, as well as frameworks such as Flask and Django, python provides a robust ecosystem for rapid development, testing, and deployment. The extensive community support ensures continuous updates, troubleshooting, and scalability.

**2. Why Python for Automated Dashboard Pipeline for SAOS Regression Infrastructure:**

i. **Automation of Log Parsing and Data Aggregation:**

Python's text-processing efficiency and file-handling capabilities were crucial for parsing large regression log files, extracting test results, and structuring them into meaningful summaries automatically.

ii. **Backend Development with Flask**: Python served as the backbone for the Flask-based web application that powered the live dashboard, allowing seamless data transfer between backend scripts and the web interface.

iii. **Integration with Jenkins and Linux Environment**: Through Python's subprocess and OS modules, automation scripts were easily integrated into Jenkins CI/CD pipelines and executed on Linux-based virtual machines.

iv. **Rapid Development and Iterative Testing:** Python's simplicity allowed for fast prototyping and iterative refinement with over 15 script versions were tested and optimized during the project's development phase.

### 1.3.2 Flask

**1. Key Features of Flak**


**Figure 1.2:** Flask Logo

**Source:** https://en.wikipedia.org/wiki/Flask

i. **Lightweight and Flexible Framework**: Flask is a micro web framework that provides essential tools for building robust web applications without unnecessary complexity. Its modular design allows developers to add components like authentication, database integration, or API routing as needed.

ii. **Built-in Development Server and Debugger**: Flask includes an integrated development server and debugger, enabling quick testing and debugging of web routes, APIs, and application logic. This feature greatly accelerates the development and refinement process.

iii. **RESTful Request Handling**: Flask makes it easy to create RESTful APIs, handle HTTP requests, and manage responses efficiently. This capability is crucial when building data-driven dashboards that require continuous interaction between frontend and backend components.

iv. **Template Rendering with Jinja2**: Flask uses the Jinja2 templating engine, which allows dynamic generation of HTML pages with embedded data. This simplifies the visualization of regression results, logs, and test statuses directly on the dashboard interface.

v. **Extensive Extension Ecosystem**: Flask supports numerous extensions for authentication, database management, and form handling. Its compatibility with libraries like SQL Alchemy, Flask-Mail, and Flask login.

2. **Why Flask for Automated Dashboard Pipeline for SAOS Regression Infrastructure:**

i. **Backend Framework for Dashboard Development:** Flask served as the backbone of the automated dashboard, providing the web server environment and routing logic required to deliver real-time regression data to users through dynamic endpoints.

ii. **Seamless Integration with Python Scripts**: Flask's Python-based nature made it effortless to integrate the backend automation scripts responsible for parsing regression logs, aggregating data, and sending updates to the web interface.

iii. **Dynamic Data Visualization**: Using Flask's Jinja2 templates, regression results such as pass/fail statuses, execution times, and overall suite progress were dynamically rendered into HTML tables and charts for intuitive visualization.

iv. **Ease of Deployment in CI/CD Pipelines**: Flask applications can be easily hosted on Linux-based virtual machines and integrated into Jenkins workflows, ensuring smooth deployment and continuous updates to the dashboard.

v. **Scalable and Maintainable Architecture:**
Flask's minimalistic design and modular structure allowed for the addition of new dashboards, regression suites, and automation features without disrupting existing functionalities, ensuring long-term scalability and maintainability of the system.

### 1.3.3 Linux Virtual Machine

## 1. Key Features of Linux Virtual Machine

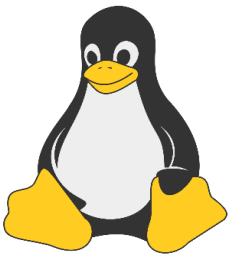


**Figure 1.3** Linux Logo

**Source:** https://www.linux.org/

i. **Open-Source and Flexible Environment:** Linux is an open-source operating system, meaning it's free to use, modify, and distribute. It provides a stable, secure, and customizable operating system, making it ideal for hosting backend applications, automation scripts, and testing pipelines in an isolated virtualized setup.

ii. **Robust Command-Line and Scripting Capabilities:** Linux's powerful command-line interface (CLI) and shell scripting support enable automation of repetitive tasks such as test execution, log parsing, and environment setup, enhancing developer productivity.

iii. **Virtualization Support:** Running Linux on a Virtual Machine (VM) allows developers to replicate production environments, maintain configuration consistency, and safely test new features or updates without affecting live systems.

iv. **High Stability and Performance:** Linux-based systems are known for their reliability and efficiency, especially when handling multiple processes concurrently, which is essential for continuous testing and regression workloads.

v. **Security and Access Control:** Linux offers granular permission management, user isolation, and secure SSH-based remote access, ensuring controlled execution of automated scripts and secure communication with connected systems.

## 2. Why Linux VM for Automated Dashboard Pipeline for SAOS Regression Infrastructure:

i. **Execution Platform for Regression Job:** The Linux VM served as the primary execution environment for running automated regression test suites and related shell scripts triggered through Jenkins pipelines.

ii. **Backend Hosting and Process Management:** Flask-based backend services were deployed on the Linux VM, managing incoming test data, processing logs, and serving live updates to the dashboard.

iii. **Automation and Scripting Environment:** Shell scripts executed on the Linux VM handled essential tasks like test status parsing, file transfers, and

report generation, ensuring smooth integration between testing tools and the dashboard.

iv. **Controlled and Scalable Infrastructure**: The use of a virtual machine allowed the regression infrastructure to scale efficiently where additional VMs could be provisioned to handle increased testing loads or parallel runs as needed.

v. **Consistent Development and Testing Setup:** Using a Linux VM ensured that the testing and deployment environments remained uniform across all stages of the automation pipeline, minimizing compatibility issues and simplifying maintenance.

### 1.3.4 TigerVNC

#### 1. Key Features of TigerVNC

**Figure 1.4** TigerVNC Logo

**Source:** https://tigervnc.org/

i. **Remote Graphical Access:** TigerVNC (Virtual Network Computing) enables users to remotely access and control the graphical desktop environment of a Linux system from any client device. This makes it ideal for monitoring, managing, and debugging applications hosted on remote virtual machines.

ii. **Cross-Platform Compatibility:** It supports multiple operating systems, including Linux, Windows, and macOS, allowing flexible access to virtual environments from diverse client systems without configuration complexity.

iii. **Multi-User and Session Management:** It allows multiple users to establish independent remote desktop sessions simultaneously, supporting collaborative debugging, development, and monitoring in shared test environments.

iv. **Virtual Desktop Management:** TigerVNC allows users to create and manage multiple virtual desktops on a remote server. This feature is especially useful in multi-user environments where multiple sessions are required, each with its own unique configuration and permissions.

v. **High-Performance Rendering:** Designed for speed and efficiency, TigerVNC provides smooth and responsive GUI performance even over limited bandwidth, which is essential for interacting with resource-intensive dashboards or real-time monitoring tools.

## 2. Why TigerVNC for Automated Dashboard Pipeline for SAOS Regression Infrastructure:

i. **Remote Access to Linux VM Environmen:** TigerVNC was used to access the Linux Virtual Machine running the automated regression pipeline and dashboard backend, enabling developers and testers to manage the setup remotely.

ii. **Graphical Monitoring of Flask Dashboard:** By connecting through TigerVNC, engineers could view and interact with the Flask-based dashboard directly from the VM's GUI, ensuring real-time visibility of test progress, logs, and system performance. It's support for SSL and TLS encryption ensures that remote sessions remain secure, protecting sensitive session data and minimizing the risk of unauthorized access.

iii. **Simplified Troubleshooting and Debugging:** TigerVNC provided a convenient way to visually inspect running processes, system utilization, and UI-level issues within the dashboard, accelerating the debugging process without requiring physical system access. Also, multiple users can log in remotely to monitor different stages of the regression workflow simultaneously, improving communication and coordination between development, QA, and infrastructure teams.

iv. **Persistent Sessions:** One of the standout features of TigerVNC is its ability to

maintain session persistence. Unlike some remote desktop tools, TigerVNC allows sessions to keep running on the server even if the client disconnects or the user shuts down their computer. This means any progress, open applications, or configurations within the remote session remain intact and accessible upon reconnecting.

### 1.3.5 HTML

**1. Key Features of HTML**


**Figure 1.5** HTML

**Source:** https://www.pngwing.com

i. **Structural Foundation of Web Pages**: HTML (HyperText Markup Language) provides the basic structure of all web pages. It defines elements such as headers, paragraphs, tables, buttons, and containers, which form the framework upon which all web content is built.

ii. **Tag-Based Architecture:** HTML uses a simple and well-defined tag system to organize and format content. This tag-based design ensures readability and maintainability, making it ideal for designing data-driven dashboards with clear layouts and consistent formatting.

iii. **Integration with CSS and JavaScript**: HTML seamlessly integrates with Cascading Style Sheets (CSS) for styling and JavaScript for interactivity, allowing developers to build visually appealing and responsive web applications.

iv. **Cross-Browser Compatibility**: HTML is universally supported by all major web browsers, ensuring that the dashboard renders consistently across different environments without additional configuration.

2. **Why HTML for Automated Dashboard Pipeline for SAOS Regression Infrastructure:**

i. **Structuring the Single-Page Dashboard:** HTML was used to define the structural layout of the live dashboard, including sections for test suite summaries, execution statistics, status tables, and log access panels.

ii. **Clear Presentation of Regression Data**: The use of HTML tables and div containers ensured that pass/fail results, timestamps, and test details were neatly organized and easily readable by engineers and QA analysts.

iii. **Seamless Integration with Flask Templates**: HTML templates rendered through Flask's Jinja2 engine allowed dynamic embedding of regression data and automatic updates as backend scripts processed new test results.

iv. **Lightweight and Maintainable Layout Design**: Since the dashboard was designed as a single-page interface, HTML provided a lightweight yet robust foundation that could be easily extended or modified without introducing complexity.

v. **Support for Future Scalability:**
HTML's modular and semantic structure makes it simple to add new components such as charts, graphs, or filtering options in future iterations of the SAOS dashboard without redesigning the entire interface.

### 1.3.6 Jenkins

1. **Key Features of Jenkins**


**Figure 1.6** Jenkins

**Source:** https://www.pngwing.com

i. **Continuous Integration and Continuous Deployment (CI/CD):** Jenkins automates the building, testing, and deployment processes in software development pipelines. It ensures that every code change is continuously integrated, tested, and deployed, minimizing manual intervention and errors.

ii. **Extensive Plugin Ecosystem:** With over a thousand plugins, Jenkins can easily integrate with version control systems (e.g., Git), testing tools, reporting frameworks, and container technologies. This flexibility makes it highly adaptable to diverse project needs.

iii. **Pipeline as Code:** Jenkins allows developers to define build and deployment workflows using declarative pipeline scripts, ensuring reproducibility, version control, and easy modification of automation logic.

iv. **Automated Build Triggers:** Jenkins supports multiple triggers such as SCM commits, cron schedules, or API calls, allowing fully automated workflows for testing and deployment based on real-time events or periodic jobs.

v. **Scalability and Distributed Builds:** Through its master-agent architecture, Jenkins can distribute workloads across multiple nodes, ensuring efficient use of resources and faster execution of large-scale regression suites.

**2. Why Jenkins for Automated Dashboard Pipeline for SAOS Regression Infrastructure:**

i. **Automation of Regression Execution:** Jenkins was used to schedule and trigger automated SAOS regression tests on the Linux VM environment, ensuring consistent and timely execution without manual supervision.

ii. **Seamless Integration with Flask Backend:** After test completion, Jenkins jobs invoked Flask API endpoints or shell scripts to update the dashboard with the latest regression data and test statuses.

iii. **Centralized Monitoring and Logging:** Jenkins provided a centralized interface to track build history, console logs, and job outcomes, simplifying debugging and quality assurance processes for the regression infrastructure.

iv. **Continuous Feedback Loop:** By automatically updating the live dashboard upon each Jenkins job completion, engineers could instantly visualize real-time test outcomes and identify failed modules for quick remediation.

v. **Scalable and Extensible Workflow:** Jenkins pipelines were designed to easily accommodate new test suites, platforms, or reporting steps, ensuring that the automation framework remains flexible and future-ready.

### 1.3.7 CSS

**Fig 1.7 CSS logo**

**Source:** https://www.pngwing.com

#### 1. Key Features of CSS:

i. **Separation of Style and Structure:** Cascading Style Sheets (CSS) allow developers to separate content (HTML) from presentation. This ensures cleaner, more maintainable code by managing visual aspects like colors, typography, spacing, and layout independently from HTML structure.

ii. **Responsive and Adaptive Design:** CSS supports responsive design through flexible layouts, media queries, and grid systems. This enables web applications and dashboards to adapt automatically to various screen sizes and resolutions, ensuring usability on both desktops and mobile devices.

iii. **Customizable Visual Themes:** By defining reusable classes and variables, CSS allows the creation of consistent color schemes, button styles, and component appearances across the entire application. This enhances visual coherence and improves user experience.

iv. **Integration with Animations and Transitions:** CSS supports smooth animations and hover effects that improve user interactivity and engagement without adding JavaScript complexity.

vi. **Cross-Browser Styling Consistency:** With standardized styling rules and normalization practices, CSS ensures the dashboard renders consistently across all major browsers and operating systems.

vii. **Lightweight and Simple:** Compared to other available styling languages, CSS is relatively simple to use.

## 2. Why CSS for Automated Dashboard Pipeline for SAOS Regression Infrastructure:

i. **Enhanced Visual Presentation:** CSS was used to style the single-page dashboard, improving readability through clear color-coded indicators for test results (e.g., green for PASS, red for FAIL) and visually distinct panels for each test type.

ii. **Consistent and Professional Layout:** The dashboard's layout was designed with consistent spacing, borders, and typography to provide a polished and user-friendly interface for engineers and QA teams monitoring regression test results.

iii. **Responsive Dashboard Design:** CSS media queries ensured that the dashboard layout dynamically adjusted to different screen sizes, enabling accessibility from various devices within the testing environment.

iv. **Integration with Flask-Rendered HTML:** Through Flask templates, CSS stylesheets were linked to dynamically generated HTML pages, maintaining a cohesive visual identity even as data updated in real-time.

v. **Scalable and Maintainable Styling:** By organizing styles using modular CSS classes, future enhancements such as new test categories, chart components, or dark mode support can be incorporated with minimal code changes and no impact on backend logic.

### 1.3.8 Bash


**Figure 1.8** Bash Logo

**Source:** https://bashlogo.com/

Bash (Bourne Again Shell) is a command-line scripting language widely used in Linux for automation and task management. In the Automated Dashboard Pipeline for SAOS Regression Infrastructure project, Bash was used within Jenkins jobs to execute test suites and generate the status files that acted as input for the Python dashboard.

These scripts continuously updated runtime data such as test status and execution progress in /corp/tftp/users/{TEST_TYPE}/status, ensuring the dashboard displayed live results. Bash also handled essential file operations like log organization and directory updates, maintaining synchronization between Jenkins and the dashboard.

Its lightweight and Linux-native design made Bash ideal for automating workflows, reducing manual effort, and ensuring smooth data flow across the regression pipeline.

### 1.3.9 Apache


**Figure 1.9 Apache Logo**

**Source:** https://en.wikipedia.org/wiki/ApacheHTTPServer

Apache HTTP Server is a widely used open-source web server that enables reliable hosting and delivery of web applications. In the Automated Dashboard Pipeline for SAOS Regression Infrastructure project, Apache was used to deploy and host the Flask-based dashboard on a Linux environment, ensuring 24×7 accessibility within the internal network.

It served as the interface between the backend scripts and end users, allowing team members to access real-time regression results through a static internal URL. Apache's stability and compatibility with Flask made it ideal for handling multiple concurrent requests from various projects without downtime.

By using Apache, the dashboard achieved consistent performance, secure internal access, and high availability crucial for continuous monitoring of automated regression cycles.

### 1.3.10 Bitbucket


**Figure 1.10 Bitbucket Logo**

**Source:** https://bitbucket.org/product/

Bitbucket is a web-based version control and collaboration platform primarily used for managing Git repositories. It provides developers with a centralized environment to store, track, and manage code efficiently. In the Automated Dashboard Pipeline for SAOS Regression Infrastructure project, Bitbucket was utilized to host and maintain the Python scripts that powered the backend logic of the dashboard.

All key components including log parsing scripts, data extraction modules, and Flask integration files were stored in a private Bitbucket repository. This ensured version control, allowing each update or enhancement to be properly tracked and rolled back if required. The platform's branching and pull request features facilitated collaborative development, enabling the team to work on separate features simultaneously without affecting the main deployment branch.

Overall, Bitbucket served as a crucial tool in ensuring the project's maintainability, reliability, and transparency. By enabling efficient collaboration and structured version control, it helped sustain consistent improvement of the dashboard scripts throughout the development and deployment lifecycle.

### 1.3.11 Confluence


**Figure 1.11 Confluence Logo**

**Source:** [https://www.logo.wine/logo/Confluence)](https://www.logo.wine/logo/Confluence)

Confluence is a collaborative documentation and knowledge management platform developed by Atlassian. It enables teams to create, organize, and share project-related information in a structured and easily accessible manner. In the Automated Dashboard Pipeline for SAOS Regression Infrastructure project, Confluence played a vital role in maintaining clear and comprehensive documentation throughout the development lifecycle.

All aspects of the dashboard such as system architecture, regression test flow, data parsing logic, Jenkins integration, and deployment procedures were documented on Confluence pages. This ensured that team members and mentors could easily reference the project's design, progress, and updates at any time. The platform's space and page hierarchy allowed for logically organizing content, making navigation intuitive for both developers and QA engineers. Confluence also served as a central knowledge hub for tracking testing outcomes, bug reports, and optimization notes, ensuring transparency and traceability

Overall, Confluence provided a structured environment for collaboration, documentation, and review, helping maintain project consistency and communication across all stages of dashboard development and deployment.

# CHAPTER 2

# PROJECT DESIGN

## 2.1 Objective

The objective of the Master Session in our SSH-Console Server project is to establish a robust and centralized control mechanism for monitoring and managing active user sessions on the ONU 3803-MTL. This session efficiently handles session listings, checks administrative states, and identifies suspicious connections to maintain system security. It provides administrators with a streamlined interface for managing multiple sessions, including listing details by port number and username. By integrating comprehensive session monitoring, the Master Session enhances operational efficiency and ensures the overall stability of the SSH-Console environment. The design emphasizes reliability, ease of use, and robust performance for optimal network management.

## 2.2 Project Strategy

**1. Overview of SAOS Regression Infrastructure**

- **Introduction to SAOS Testing Environment:** This section discusses the significance of SAOS (Service-Aware Operating System) in Ciena's networking ecosystem and its need for continuous regression validation to ensure software reliability across devices.
- **Purpose of Regression Automation:** To explain how automating regression testing improves coverage, reduces manual effort, and accelerates feedback loops for the software QA team.

**2. Dashboard Architecture and Design**

- **System Architecture Overview:** To present the architectural flow of the dashboard, covering components like data sources (logs), backend data parser, Flask server, and frontend renderer.

- **Dashboard Structural Design:** To describe how test suites, platforms, and rolled-up summaries were visually mapped using a tabular matrix, ensuring clarity and scalability.

**3. Core Logic Development**

- **Utility Functions and Data Processing:** To explain the development of 10–12 key Python utility functions responsible for parsing logs, extracting execution times, and aggregating regression data.
- **Status Classification and Time Computation:** To describe the logical handling of multiple test states (PASS, FAIL, CRASH, QUEUED, RUNNING, NORUN) and the computation of both per-test and total suite execution time.

**4. Frontend Layout and Visualization**

- **Data Representation and Layout:** To discuss how the dashboard was structured using HTML and CSS to display results intuitively, aligning test cases and platforms in a clean, matrix-style view.
- **Visual Indicators and Interactivity:** Highlights the use of color-coded cells, hyperlinks for log access, and real-time dynamic updates to improve user readability and engagement.

**5. Integration with Jenkins Pipeline**

- **Continuous Integration Workflow:** To explain how the dashboard was linked with Jenkins to automatically fetch the latest regression outputs after every scheduled test cycle.
- **Automated Trigger and Data Refresh:** To detail how Jenkins jobs executed the backend scripts, updated parsed data, and refreshed dashboard visuals seamlessly.

**6. Testing and Validation of Dashboard Functions**

- **Functional and Unit Testing:** This section outlines the testing methodology applied to verify backend scripts, data consistency, and individual dashboard

components.

- **Accuracy and Performance Validation:** Describe how test simulations ensured correct status mapping, accurate time tracking, and smooth real-time dashboard updates under load.

**7. Deployment and Accessibility**

- **Hosting on Apache Server**: Describe how the dashboard was deployed on a Linux-based Apache server, enabling 24×7 accessibility within the internal network.
- **Access Control and Usability:** Explain how authorized team members could securely access the dashboard via a static internal URL to monitor regression cycles in real time.

**8. Bug Fixes and Optimization**

- **Bug Fixes and Optimization:** Address the post-deployment maintenance phase, detailing how bug fixes were managed and optimizations were implemented to enhance script performance and reliability.

9. **Maintenance and Future Improvements**

- **Monitoring and Feedback:** Discuss the tools and methods used to monitor system performance and gather user feedback post-deployment. This segment will highlight how monitoring contributes to ongoing improvements and user satisfaction.
- **Plans for Future Enhancement:** Outline potential future improvements for the dashboard script, including advanced features, scalability options, and plans for extending functionality to support more complex network environments. This section will emphasize the importance of continuous development in meeting evolving user needs.

# CHAPTER 3

# IMPLEMENTATION

## 3.1 Overview of SAOS Regression Infrastructure

**3.1.1 Introduction to SAOS Testing Environment:** The Service-Aware Operating System (SAOS) is Ciena's proprietary software platform designed to power its suite of network devices such as routers and switches. It provides advanced capabilities for traffic engineering, service management, and network automation core functionalities that enable Ciena's customers to deploy reliable, scalable, and high-performance communication networks. Since SAOS runs across multiple hardware platforms and supports numerous software builds, its reliability and stability are of utmost importance. Even a minor bug or failure can disrupt large-scale service networks, making continuous testing and validation an essential part of Ciena's development lifecycle.

To ensure this level of reliability, Ciena employs a regression testing infrastructure that systematically validates SAOS performance across versions and platforms. Regression testing involves re-running existing test suites after every software update or enhancement to confirm that previously developed features continue to function as intended. The testing environment for SAOS regression is hosted on Linux-based virtual machines (VMs), which provide an isolated and consistent setup for executing automated tests. These virtual environments mimic real hardware conditions and allow parallel execution of multiple test suites simultaneously.

At the heart of this process is Jenkins, a widely used continuous integration and automation tool. Jenkins orchestrates the entire testing workflow—from initiating test runs to collecting logs and summarizing results. Within Jenkins, each test cycle corresponds to a unique project or "test suite." Engineers configure each suite by specifying critical parameters such as:

- Platforms or hardware devices to test on.
- Type of run, such as upgrade only, test only, or upgrade and test.
- Set of test cases to be executed for validation.

Once triggered, Jenkins executes corresponding shell scripts that handle automation running commands, collecting test data, and storing log files in predefined directories within the Linux environment. Each test suite execution is assigned a unique build number, allowing easy identification and tracking of historical test runs. The logs generated contain detailed insights, including system behaviour, execution time, and pass/fail outcomes.

Traditionally, the regression team manually accessed these logs to compile test results and report them back via email to the requesting software teams. While functional, this process was slow, labour-intensive, and prone to human oversight. The manual workflow also delayed feedback loops for development.

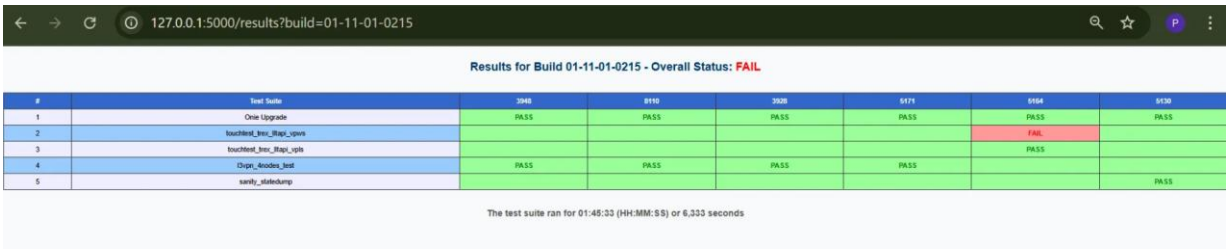



**Figure 3.1** Sample SD&R Team Dashboard

**Source**: Intellectual Property of Ciena India Pvt Ltd.

**3.1.2 Purpose of Regression Automation:** To address these inefficiencies, the project introduced an Automated Dashboard Pipeline aimed at visualizing and centralizing the regression results from Jenkins in real time. Instead of relying on manual log extraction, the dashboard automates data parsing, aggregation, and visualization, transforming raw test output into a structured, easily interpretable interface.

The automation offers several key benefits:

i. Enhanced Test Coverage: By automating execution and result tracking, multiple test suites across diverse platforms can be managed concurrently.

ii. Reduced Manual Effort: Engineers no longer need to manually parse logs or maintain spreadsheets; the dashboard retrieves and updates all relevant information automatically.

iii. Accelerated Feedback Cycle: Real-time updates enable faster communication

between the regression and development teams, ensuring quicker issue resolution.

iv. Improved Transparency: Each test run, status (PASS/FAIL/CRASH), and total execution time is clearly displayed, providing at-a-glance visibility of SAOS health across platforms.

By integrating seamlessly with Jenkins and Linux VMs, the automated dashboard establishes a robust, scalable foundation for continuous testing. It enhances operational efficiency, minimizes errors, and supports Ciena's commitment to delivering dependable, high-quality network software.

## 3.2 Dashboard Architecture and Design

**3.2.1 System Architecture Overview:** The SAOS Regression Dashboard was designed as a lightweight yet robust web-based monitoring tool built on the Flask framework. It operates on a Linux Virtual Machine and interfaces directly with the defined test framework used by Ciena's internal testing infrastructure. The core data sources include summary files, running logs, and completed logs, all of which are automatically generated and continuously updated during Jenkins-driven test suite executions.

At the backend, a continuously running Python script leverages regular expressions and shell commands to parse and extract essential information such as test names, execution status, durations, and overall completion time. These scripts actively monitor directories to detect live updates in the test framework, ensuring that the displayed data reflects the latest execution state. The extracted results are processed, formatted, and served through RESTful APIs in JSON format, enabling real-time communication between the backend and frontend.

The architecture follows a modular design, with distinct layers for data parsing, data processing, and visualization. This separation ensures scalability and maintainability, allowing new test suites or log formats to be integrated with minimal code modifications. The entire system operates autonomously, providing uninterrupted live updates without manual intervention.

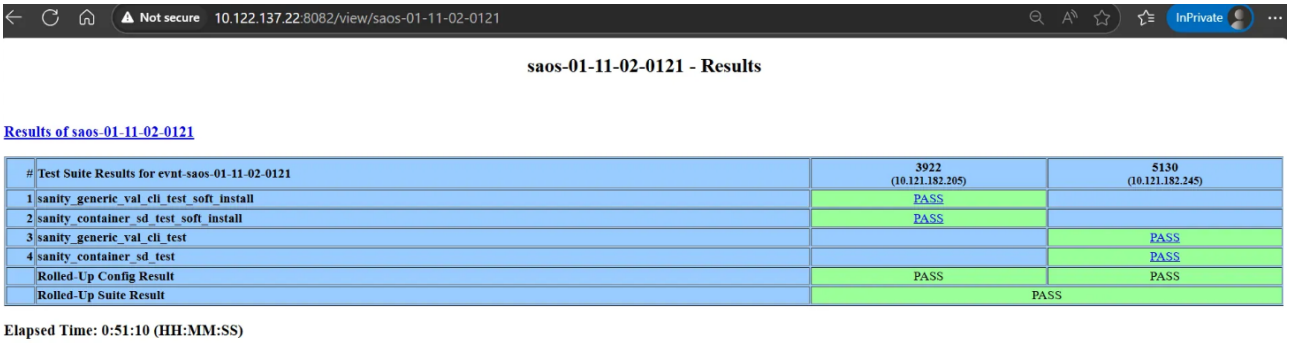



**Figure 3.2** Base Dashboard with Dummy Values

**3.2.2 Dashboard Structural Design:** The frontend of the dashboard is implemented as a single-page web interface that focuses on clarity, accessibility, and ease of navigation. It dynamically renders the parsed JSON data into an interactive HTML table, where each row represents a test case and each column corresponds to a Device Under Test (DUT) or platform.

Status values such as “PASS,” “FAIL,” and “RUN” are color-coded to enhance visual comprehension, while each status cell is hyperlinked to its corresponding log file for detailed analysis. In cases where a log file is unavailable, contextual tooltips provide diagnostic context.

The dashboard URL structure is standardized (e.g., /view/autoUT/<test_suite_name>), allowing users to access the live or historical view of any test suite directly.

CPD (Per-Project Live Dashboard)

Kindly enter: autoUT/{PROJECT_TYPE}
URL Format: http://10.122.137.22:8081/view/ { autoUT/{PROJECT_TYPE} }
All are running on fixed port 8081.
Examples:

- http://10.122.137.22:8081/view/autoUT/ggn_10X_test_A
- http://10.122.137.22:8081/view/autoUT/ggn_10X_test_B
- http://10.122.137.22:8081/view/autoUT/ggn_10X_test_C
- http://10.122.137.22:8081/view/autoUT/ggn_10X_test_F
- http://10.122.137.22:8081/view/autoUT/ggn_10X_test_G
- http://10.122.137.22:8081/view/autoUT/ggn_10X_quick_sanity
- http://10.122.137.22:8081/view/autoUT/ggn_10X_quick_sanity_softinstall
- http://10.122.137.22:8081/view/autoUT/ggn_10X_on_demand_test
- http://10.122.137.22:8081/view/autoUT/ggn_10X_test_VDR
- http://10.122.137.22:8081/view/autoUT/CPD
- http://10.122.137.22:8081/view/autoUT/DRSanity
- http://10.122.137.22:8081/view/autoUT/DeviceTest

**Figure 3.3** CPD URL Standardization Example

## 3.3 Core Logic Development

**3.3.1 Utility Functions and Data Processing:** The backend of the Automated Dashboard Pipeline for SAOS Regression Infrastructure was designed around a set of 11 Python utility functions that collectively handled log parsing, data extraction, and result aggregation. These modular functions ensured the separation of logic, ease of debugging, and scalability of the dashboard system. The entire workflow followed a structured execution pattern, as illustrated in the flowchart attached below.

At the start of every test suite execution, Jenkins passed a status file as input to the main script. This file contained key metadata, including the test suite name, list of test cases, build number, host name, test type, and start time. The first stage of processing involved reading this file and identifying corresponding directories in the Linux VM, which housed relevant summary and log files. Using utility functions such as parse_summary_file(), extract_execution_time(), and parse_topology_files(), the script parsed structured and unstructured data through regular expressions (regex) and shell commands like grep.

Subsequent functions like check_device_lock_status() executed system-level commands (`postb_show`, etc.) to determine which Devices Under Test (DUTs) were locked or actively running. Based on this information, initial test states were assigned as NORUN. Additional functions processed live logs and completed logs to identify real-time execution progress and output.

The main function served as the central orchestrator, integrating all utility functions. It computed configuration rollups, handled exceptions, and aggregated test data into structured dictionaries. The data was then converted into JSON format and transmitted to the Flask backend, which updated the dashboard dynamically.

Finally, the system ensured continuous live updates through an auxiliary script running in a loop that repeatedly fetched the latest data for all test suites. This mechanism maintained a constantly refreshed display of results, providing engineers with near real-time visibility into regression activities.

| State | Meaning | Color |
|---|---|---|
| CRASH | System crash during test (Only for Onie Upgrade) | Red |
| FAIL | Functional failure that needs investigation | Red |
| PASS | Test ran and passed all checks | Light Green |
| WARN | Passed with warnings | Yellow |
| NORUN | Test not executed | Blue |
| QUEUED | Waiting to run | Yellow |
| RUNNING | Currently executing | Light Cream |

**Figure 3.4** Test Status Logic Table

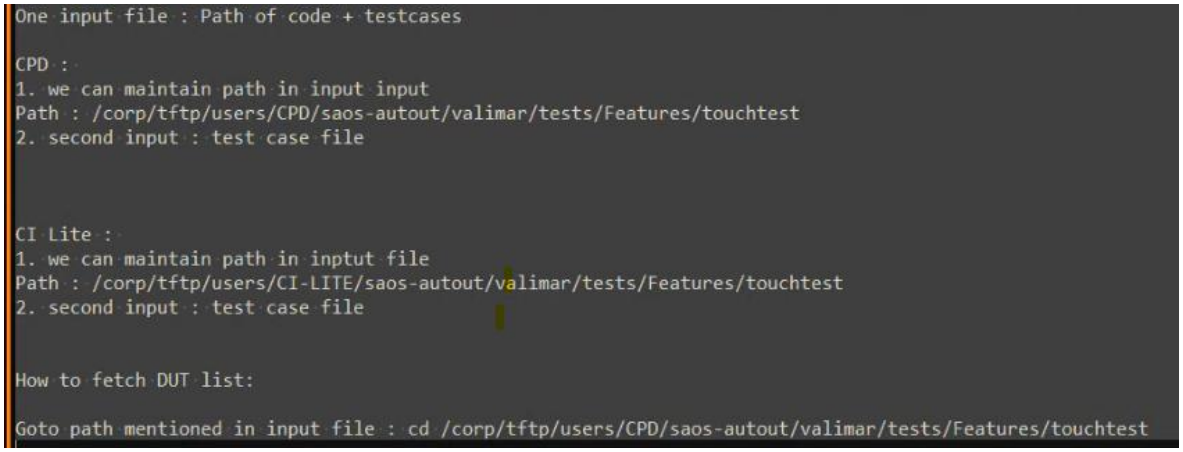



**Figure 3.5** Input File Sample

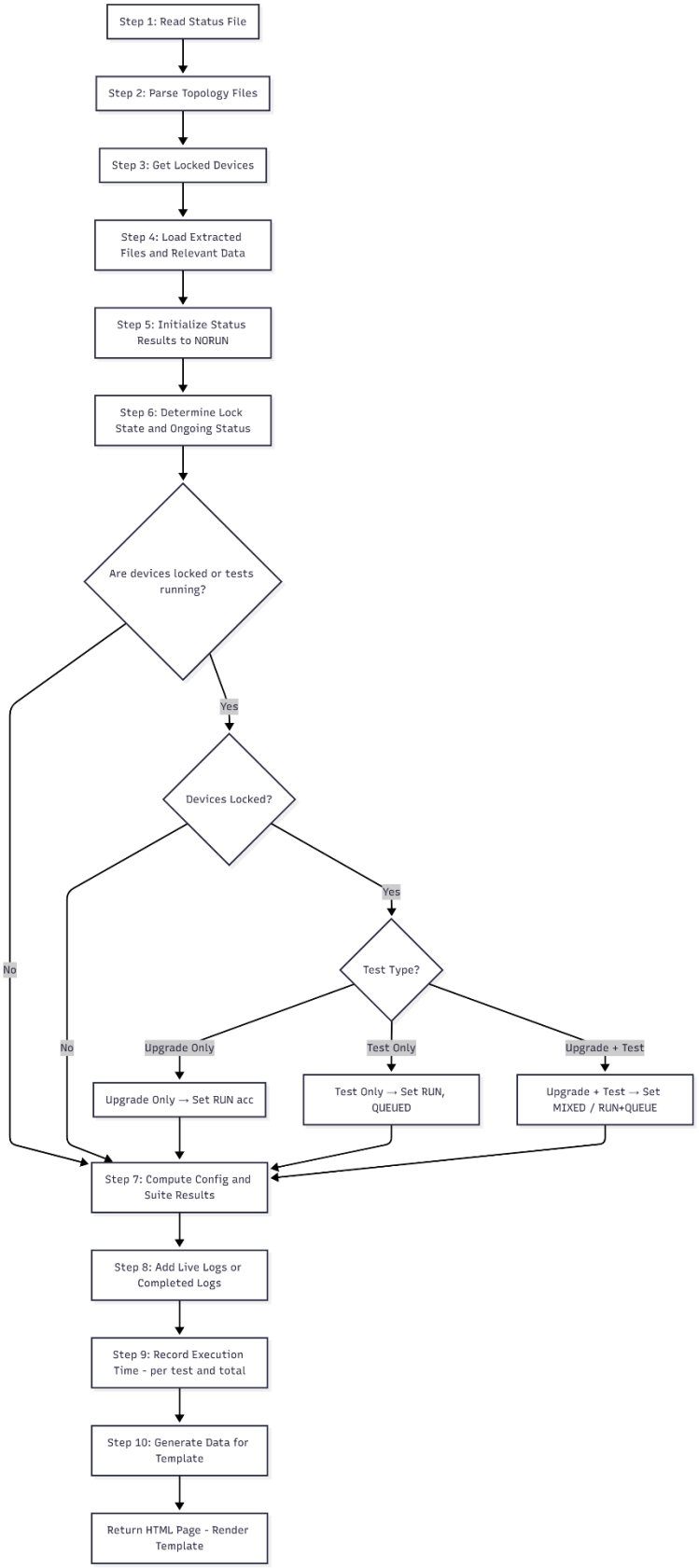



**Figure 3.6** Dashboard Script Flowchart

**3.3.1 Status Classification and Time Computation:** A major component of the backend logic was the classification of test statuses and computation of execution times at both per-test and overall suite levels. The dashboard represented every test case in one of several possible states — PASS, FAIL, CRASH, QUEUED, RUNNING, NORUN, or WARN; each determined by specific log conditions and validation criteria discussed with the QA team.

The classification was based on an intelligent parsing system that read from summary, running, and completed log files.

- PASS status was assigned only when all sub-tests within a test case completed successfully (a “perfect pass”).
- FAIL and CRASH were identified through error patterns or termination markers in the summary logs.
- RUNNING was inferred from the presence of an active running log file, while QUEUED indicated that DUTs were currently locked and the test awaited execution.
- NORUN denoted test cases not yet started.
- WARN represented abnormal conditions (e.g., execution time = 0 despite a PASS).

To enhance readability, each status was color-coded on the frontend (e.g., green for PASS, red for FAIL). A detailed table illustrating all classification conditions is attached in the annexure.

The execution time computation logic relied on two distinct approaches:

- Per-Testcase Time: Extracted directly from running or completed log files using start and end timestamps, expressed in minutes.
- Total Suite Time: Calculated by taking the difference between the suite’s start time (from the status file) and the current system time, formatted as HH: MM.

Additionally, each test case on the dashboard was hyperlinked to its respective log file for quick access. If a log file was unavailable, a tooltip notified the user accordingly. The build number itself acted as a link to the full directory of log and summary files.

This robust classification and timing logic allowed for precise, real-time tracking of regression progress and performance metrics, ensuring that engineers could diagnose failures, delays, or anomalies efficiently.

## 3.4 Frontend Layout and Visualization

**3.4.1 Data Representation and Layout:** The frontend of the dashboard was deliberately designed using lightweight HTML and CSS, avoiding the use of heavy JavaScript frameworks or dynamic libraries such as Vue.js or React. This design choice was crucial to ensure minimal processing overhead, given that the script runs continuously on a Linux environment, where performance efficiency and low resource utilization are priorities. The simplicity of the structure allows the dashboard to render rapidly, even when handling large log datasets refreshed every few seconds.

The dashboard layout follows a matrix-style format, with rows representing individual test cases and columns representing platforms (both platform number and IP). At the top of each dashboard view, the project or test suite name is displayed prominently, followed by a secondary subheading indicating the build number. This build number itself is hyperlinked to the directory containing complete test result files, enabling quick navigation for detailed log analysis.

The table beneath this header dynamically represents the status of each test case across platforms. Each cell's color visually communicates the test outcome, while the text inside provides direct status information (e.g., PASS, FAIL, RUNNING). A simple checkbox toggle allows users to show or hide the execution time for each test case, making the interface both compact and informative depending on user preference. The HTML table is styled through custom CSS, ensuring visual uniformity while maintaining fast rendering times across Linux-based browsers.

Overall, the layout ensures that even complex regression data spanning multiple platforms remains intuitive, scalable, and instantly interpretable, striking a balance between minimalism and utility.

**3.4.2 Visual Indicators and Interactivity:** Color coding and interactivity are central to the dashboard's readability. Each test state is assigned a distinct, visually meaningful color palette:

- PASS results appear in light green, signifying success.
- FAIL and CRASH outcomes are colored red, highlighting critical errors.

- WARN and QUEUED states use yellow, indicating pending or cautionary states.
- RUNNING tests appear in light cream, denoting active processes.
- NORUN entries blend seamlessly with the default blue table background, maintaining aesthetic balance without visual clutter.

Since the backend Python script runs in a continuous loop, reading updated status files every second, the color updates occur automatically and in real time—without requiring manual page refreshes or AJAX requests. This gives users the impression of a “live” dashboard, even though it relies solely on static HTML and lightweight CSS logic.

Each status cell is interactive: clicking on it redirects the user to the corresponding log file or live output. For example, a “RUNNING” status links to a live log stream, while “PASS” opens the completed test log. Additionally, hover tooltips provide extra metadata such as start/end times or test duration summaries, improving accessibility without overloading the main layout.

By combining intuitive color semantics, responsive hover effects, and smart hyperlinking, the dashboard achieves high clarity and engagement, all while maintaining a lightweight footprint perfectly suited for Linux-based continuous monitoring environments.

## 3.5 Integration with Jenkins Pipeline

**3.5.1 Continuous Integration Workflow:** Jenkins was integrated into the dashboard ecosystem as the central automation and orchestration layer responsible for managing the complete lifecycle of test suite executions. Dedicated Jenkins jobs were configured for each individual test suite rather than a single all-purpose dashboard job. This modular setup ensured that every test suite, whether functional, upgrade, or regression-based, could be triggered independently as per the team's requirements without interfering with others.

Whenever a Jenkins job was initiated, it automatically executed a Bash-based testing framework that generated and maintained a *status file* under a standardized directory hierarchy on the Linux virtual machine — typically /corp/tftp/users/{TEST_TYPE}/status. This file acted as the single source of truth for the Flask dashboard, providing real-time updates regarding the state of the running test suite. Other supporting logs and summary files were also created in well-defined subdirectories such as /corp/tftp/users/{TEST_TYPE}/builds or /corp/tftp/users/{TEST_TYPE}/results, which the dashboard's backend continuously monitored.

The Flask application was designed to read these files directly from the shared directories rather than relying on file transfers or intermediate APIs. This design avoided redundant data movement and ensured lightweight, instantaneous updates. Since Jenkins and Flask operated on the same Linux environment, the shared filesystem became the communication bridge between both systems. This setup allowed Jenkins to focus solely on test execution and result generation, while Flask took complete responsibility for visualization and user interfacing. The decoupled yet synchronized design ensured that both systems could operate independently — Jenkins as the executor and Flask as the real-time reporter without causing any computational overhead or I/O bottlenecks.

Additionally, Jenkins jobs were linked with email notifications configured to automatically dispatch the live dashboard URLs upon completion of test runs. This

enabled engineers to instantly access detailed visual summaries without manually locating the correct directories. The URLs followed a structured and consistent naming format, such as:

- http://10.122.137.22:8081/view/autoUT/{PROJECT_TYPE}
- http://10.122.137.22:8082/view/{BUILD_NO}

This uniformity allowed seamless access for all stakeholders, whether they needed per-project live dashboards (CPD) or per-build dashboards (CPD_COMMON).

**3.5.2 Automated Trigger and Data Refresh:** The core automation strategy of the dashboard revolved around real-time synchronization between Jenkins' file updates and Flask's data refresh cycle. Instead of Jenkins directly invoking the Flask app or its APIs, two continuously running backend scripts handled the data extraction and page update mechanisms. These Python scripts operated in a perpetual loop, reading the updated status and log files every second.

Whenever a Jenkins job triggered a test suite run, the relevant files began updating immediately as the suite executed. Since the backend scripts were already running, the moment any file changed, they re-parsed it, extracted the latest test statuses, execution times, and logs, and wrote the processed output back to the dashboard's data layer. Consequently, within seconds of a Jenkins job start, the corresponding project dashboard began showing live updates marking test cases as RUNNING, QUEUED, or NORUN as applicable.

This autonomous loop-based update system eliminated the need for manual refreshes or explicit Jenkins callbacks. It also made the dashboard highly resilient; even in the event of a Jenkins restart or network delay, the scripts continued operating and re-synchronized automatically once the files were accessible again.

At the conclusion of every Jenkins job, the final dashboard snapshot was saved to the designated framework directory and linked to the Jenkins job's post-build notification. Users received an automated email containing the corresponding dashboard URL, enabling them to directly review the consolidated results. Depending on the job type, these could be accessed under port 8081 for project-level dashboards or port 8082 for build-level dashboards.

The integration achieved full automation across three stages — execution, parsing, and visualization — without any manual intervention. Jenkins handled the initiation and result generation, the Flask backend ensured continuous data synchronization, and the frontend provided real-time visual feedback with color-coded indicators. This seamless workflow enabled teams to monitor long-running regression jobs, overnight test executions, and multi-platform test statuses effortlessly, significantly improving operational efficiency and visibility.

## 3.6 Testing and Validation of Dashboard Functions

**3.6.1 Functional and Unit Testing:** A structured testing methodology was followed to ensure the accuracy, stability, and consistency of the developed dashboard. Unit testing was first applied to the backend utility functions responsible for parsing logs, extracting timestamps, computing execution durations, and classifying test statuses. Simulated summary and log files were used to verify correct regex extraction, data formatting, and error handling under various scenarios, including missing or corrupted files.

Functional testing was then conducted to validate the end-to-end data flow between the Flask backend and the HTML frontend. Each API endpoint was tested to confirm that the JSON responses matched the live status of files in the VM directories. The auto-refresh behavior was also validated to ensure that updates from continuously running backend scripts appeared instantly on the dashboard without requiring manual reloads.

Frontend testing focused on verifying that the matrix-style layout, color-coded cells, and hyperlinks behaved as intended. Each color status (PASS, FAIL, RUNNING, etc.) was cross-checked against actual test outcomes, ensuring full alignment between parsed data and user display.

| Raw statuses | Result | Why |
|---|---|---|
| ["NORUN", "NORUN"] | NORUN | All NORUN |
| ["PASS", "NORUN"] | PASS | Only PASS and NORUN → PASS |
| ["PASS", "PASS"] | PASS | All PASS |
| ["FAIL", "NORUN", "PASS"] | FAIL | Any FAIL or CRASH trumps |
| ["CRASH", "NORUN", "PASS"] | CRASH | Any CRASH trumps |
| ["QUEUE", "NORUN"] | QUEUE | QUEUE present |
| ["RUNNING", "PASS", "NORUN"] | RUNNING | RUNNING present |
| ["WARN", "NORUN", "PASS"] | WARN | WARN present |

**Figure 3.7** Logic Designing for Rolled-up Config/Suite Result

**3.6.2 Accuracy and Performance Validation:** To verify reliability, simulated Jenkins runs were used to test the accuracy of real-time updates and time computations. The transition of test states between RUNNING, QUEUED, FAIL, and PASS was observed to confirm seamless synchronization between backend logic and frontend visualization.

Execution times were validated by cross-referencing displayed durations with timestamps extracted from actual logs. Additionally, performance validation under load confirmed that the Flask server and background scripts sustained continuous operation with minimal latency and resource usage.

Overall, the testing and validation phase established that the dashboard delivered precise, real-time insights with high accuracy and operational stability during ongoing and completed regression cycles.

## 3.7 Deployment and Accessibility

**3.7.1 Hosting on Apache Server:** The developed dashboard was deployed on a Linux-based virtual machine and hosted using the Apache HTTP Server to ensure continuous 24×7 availability. Apache was configured as the primary web server to

serve the Flask application through a static internal IP and port combination, providing stable and high-performance hosting within the organization's network. Deployment scripts were designed to automate server startup and ensure that the backend processes, responsible for continuous file parsing and data updates, remained active in the background. This ensured that the dashboard displayed real-time information at all times without requiring manual intervention. The hosting architecture thus provided a reliable and scalable environment for long-duration test runs and daily monitoring activities.

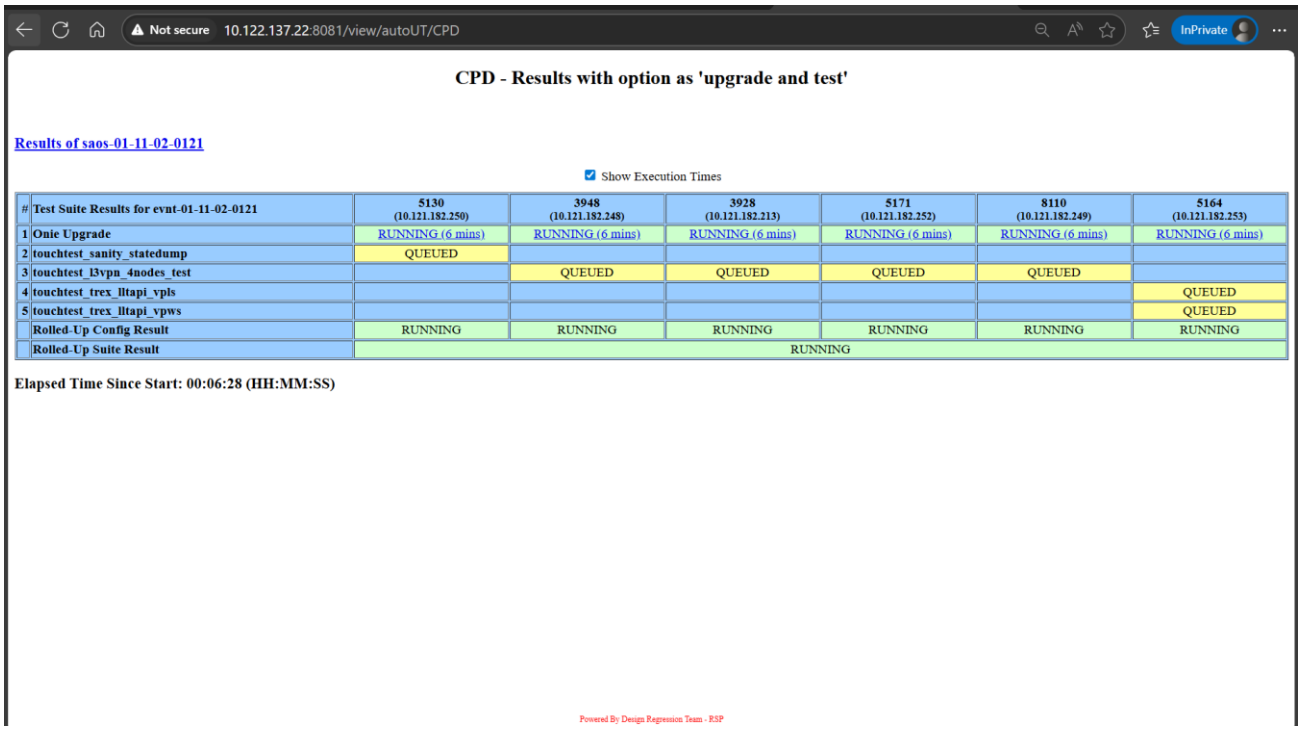
**3.7.1 Access Control and Usability:** Access was limited to the internal network, ensuring secure availability only to members of the QA and development teams. Each URL corresponded to a specific project type or build number, allowing engineers to quickly access relevant test suite dashboards without complex navigation.

This centralized web-based access significantly improved usability by allowing real-time monitoring of regression progress, test case outcomes, and execution times from any system within the network. The deployment approach thus ensured both high availability and controlled access, meeting the operational and security requirements of the SAOS regression infrastructure.

## 3.8 Bug Fixes and Optimization

**3.8.1 Bug Fixes:** Early issues primarily related to incorrect status rendering, missing hyperlinks for certain test cases, and occasional mismatches between the displayed and actual execution times. These were traced to inconsistent log formatting across different test suites and minor synchronization delays between file updates and data parsing.

To resolve these, additional validation checks and exception handlers were integrated into the parsing logic. The regular expressions used for log extraction were refined to accommodate variations in file naming and structure. Furthermore, timeout handling was introduced to prevent the backend from stalling when incomplete log files were encountered. These corrective measures significantly improved the reliability of data displayed on the dashboard.



Not secure 10.122.137.22:8081/view/autoUT/CPD

InPrivate

**CPD - Results with option as 'upgrade and test'**

Results of saos-01-11-02-0121

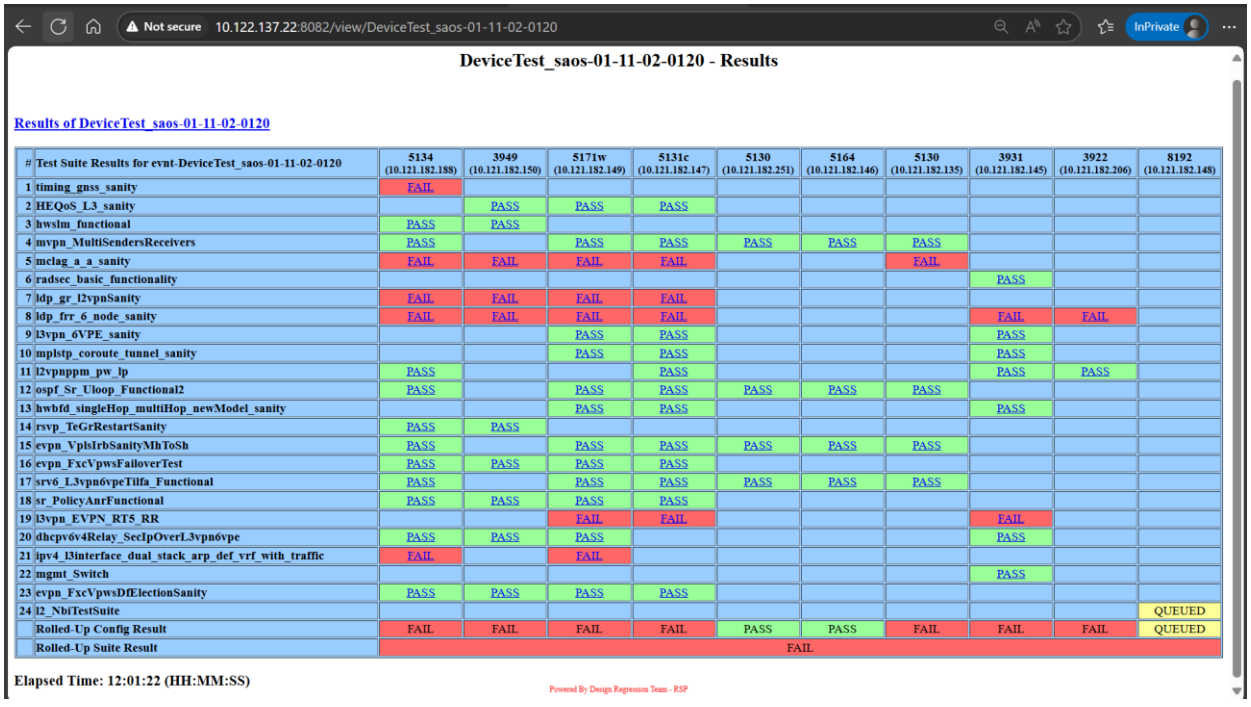
Show Execution Times

| # | Test Suite Results for evnt-01-11-02-0121 | 5130 (10.121.182.250) | 3948 (10.121.182.248) | 3928 (10.121.182.213) | 5171 (10.121.182.252) | 8110 (10.121.182.249) | 5164 (10.121.182.253) |
|---|---|---|---|---|---|---|---|
| 1 | Onie Upgrade | RUNNING (6 mins) | RUNNING (6 mins) | RUNNING (6 mins) | RUNNING (6 mins) | RUNNING (6 mins) | RUNNING (6 mins) |
| 2 | touchtest_sanity_statedump | QUEUED | | | | | |
| 3 | touchtest_l3vpn_4nodes_test | | QUEUED | QUEUED | QUEUED | QUEUED | |
| 4 | touchtest_trex_lltapi_vpls | | | | | | QUEUED |
| 5 | touchtest_trex_lltapi_vpws | | | | | | QUEUED |
| | Rolled-Up Config Result | RUNNING | RUNNING | RUNNING | RUNNING | RUNNING | RUNNING |
| | Rolled-Up Suite Result | RUNNING | | | | | |

Elapsed Time Since Start: 00:06:28 (HH:MM:SS)

Powered By Design Regression Team - RSP

**Figure 3.8** CPD: "Upgrade and Test" suite newly triggered



Not secure 10.122.137.22:8082/view/DeviceTest_saos-01-11-02-0120

InPrivate

**DeviceTest_saos-01-11-02-0120 - Results**

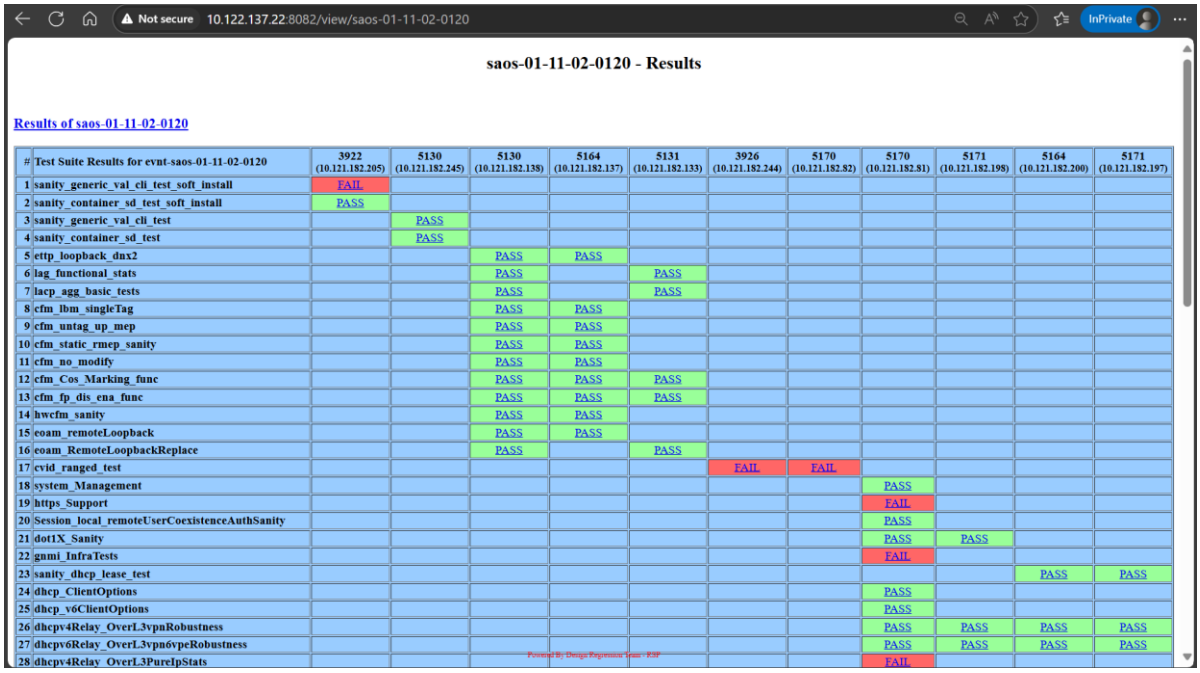
Results of DeviceTest_saos-01-11-02-0120

| # | Test Suite Results for evnt-DeviceTest_saos-01-11-02-0120 | 5134 (10.121.182.188) | 3949 (10.121.182.150) | 5171w (10.121.182.149) | 5131c (10.121.182.147) | 5130 (10.121.182.251) | 5164 (10.121.182.146) | 5130 (10.121.182.135) | 3931 (10.121.182.145) | 3922 (10.121.182.206) | 8192 (10.121.182.148) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | timing_gnss_sanity | FAIL | | | | | | | | | |
| 2 | HEQoS_L3_sanity | | PASS | PASS | PASS | | | | | | |
| 3 | hwslm_functional | PASS | PASS | | | | | | | | |
| 4 | mvpn_MultiSendersReceivers | PASS | | PASS | PASS | PASS | PASS | PASS | | | |
| 5 | mclag_a_a_sanity | FAIL | FAIL | FAIL | FAIL | | | FAIL | | | |
| 6 | radsec_basic_functionality | | | | | | | | PASS | | |
| 7 | ldp_gr_l2vpnSanity | FAIL | FAIL | FAIL | FAIL | | | | | | |
| 8 | ldp_frr_6_node_sanity | FAIL | FAIL | FAIL | FAIL | | | | FAIL | FAIL | |
| 9 | l3vpn_6VPE_sanity | | | PASS | PASS | | | | PASS | | |
| 10 | mplstp_coroute_tunnel_sanity | | | PASS | PASS | | | | PASS | | |
| 11 | l2vpnppm_pw_lp | PASS | | | PASS | | | | PASS | PASS | |
| 12 | ospf_Sr_Uloop_Functional2 | PASS | | PASS | PASS | PASS | PASS | PASS | | | |
| 13 | hwbfd_singleHop_multiHop_newModel_sanity | | | PASS | PASS | | | | PASS | | |
| 14 | rsvp_TeGrRestartSanity | PASS | PASS | | | | | | | | |
| 15 | evpn_VplsIrbSanityMhToSh | PASS | | PASS | PASS | PASS | PASS | PASS | | | |
| 16 | evpn_FxcVpwsFailoverTest | PASS | PASS | PASS | PASS | | | | | | |
| 17 | srv6_L3vpn6vpeTilfa_Functional | PASS | | PASS | PASS | PASS | PASS | PASS | | | |
| 18 | sr_PolicyAnrFunctional | PASS | PASS | PASS | PASS | | | | | | |
| 19 | l3vpn_EVPN_RT5_RR | | | FAIL | FAIL | | | | FAIL | | |
| 20 | dhcpv6v4Relay_SecIpOverL3vpn6vpe | PASS | PASS | PASS | | | | | PASS | | |
| 21 | ipv4_l3interface_dual_stack_arp_def_vrf_with_traffic | FAIL | | FAIL | | | | | | | |
| 22 | mgmt_Switch | | | | | | | | PASS | | |
| 23 | evpn_FxcVpwsDfElectionSanity | PASS | PASS | PASS | PASS | | | | | | |
| 24 | l2_NbiTestSuite | | | | | | | | | | QUEUED |
| | Rolled-Up Config Result | FAIL | FAIL | FAIL | FAIL | PASS | PASS | FAIL | FAIL | FAIL | QUEUED |
| | Rolled-Up Suite Result | FAIL | | | | | | | | | |

Elapsed Time: 12:01:22 (HH:MM:SS)

Powered By Design Regression Team - RSP

**Figure 3.9** DeviceTest Completed Results




| # | Test Suite Results for evnt-saos-01-11-02-0120 | 3922 (10.121.182.205) | 5130 (10.121.182.245) | 5130 (10.121.182.138) | 5164 (10.121.182.137) | 5131 (10.121.182.133) | 3926 (10.121.182.244) | 5170 (10.121.182.82) | 5170 (10.121.182.81) | 5171 (10.121.182.198) | 5164 (10.121.182.200) | 5171 (10.121.182.197) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | sanity_generic_val_cli_test_soft_install | FAIL | | | | | | | | | | |
| 2 | sanity_container_sd_test_soft_install | PASS | | | | | | | | | | |
| 3 | sanity_generic_val_cli_test | | PASS | | | | | | | | | |
| 4 | sanity_container_sd_test | | PASS | | | | | | | | | |
| 5 | ettp_loopback_dnx2 | | | PASS | PASS | | | | | | | |
| 6 | lag_functional_stats | | | PASS | | PASS | | | | | | |
| 7 | lacp_agg_basic_tests | | | PASS | | PASS | | | | | | |
| 8 | cfm_ibm_singleTag | | | PASS | PASS | | | | | | | |
| 9 | cfm_untag_up_mep | | | PASS | PASS | | | | | | | |
| 10 | cfm_static_rmep_sanity | | | PASS | PASS | | | | | | | |
| 11 | cfm_no_modify | | | PASS | PASS | | | | | | | |
| 12 | cfm_Cos_Marking_func | | | PASS | PASS | PASS | | | | | | |
| 13 | cfm_fp_dis_ena_func | | | PASS | PASS | PASS | | | | | | |
| 14 | hwcfm_sanity | | | PASS | PASS | | | | | | | |
| 15 | eoam_remoteLoopback | | | PASS | PASS | | | | | | | |
| 16 | eoam_RemoteLoopbackReplace | | | PASS | | PASS | | | | | | |
| 17 | cvid_ranged_test | | | | | | FAIL | FAIL | | | | |
| 18 | system_Management | | | | | | | | PASS | | | |
| 19 | https_Support | | | | | | | | FAIL | | | |
| 20 | Session_local_remoteUserCoexistenceAuthSanity | | | | | | | | PASS | | | |
| 21 | dot1X_Sanity | | | | | | | | PASS | PASS | | |
| 22 | gnmi_InfraTests | | | | | | | | FAIL | | | |
| 23 | sanity_dhcp_lease_test | | | | | | | | | | PASS | PASS |
| 24 | dhcp_ClientOptions | | | | | | | | PASS | | | |
| 25 | dhcp_v6ClientOptions | | | | | | | | PASS | | | |
| 26 | dhcpv4Relay_OverL3vpnRobustness | | | | | | | | PASS | PASS | PASS | PASS |
| 27 | dhcpv6Relay_OverL3vpn6vpeRobustness | | | | | | | | PASS | PASS | PASS | PASS |
| 28 | dhcpv4Relay_OverL3PureIpStats | | | | | | | | FAIL | | | |

**Figure 3.10** Completed Test Suite Dashboard for SAOS Build-Run

**3.8.2 Optimization Enhancements:** Once the core functionality of the Automated Dashboard Pipeline was stabilized, focus shifted toward improving its performance, scalability, and resource efficiency. The aim was to ensure that the dashboard could handle large-scale SAOS regression environments while maintaining responsiveness and reliability under continuous use.

To begin with, file input/output operations were optimized by reducing redundant read cycles and unnecessary data retrieval. Instead of repeatedly accessing large log directories, selective shell commands such as grep and ls were employed to extract only the required information. This optimization minimized processing time and disk overhead during log parsing.

Caching mechanisms were also introduced for static or rarely changing elements, such as platform mappings and configuration data. By retaining these components in memory, the dashboard significantly reduced repetitive computation and improved rendering speeds. Furthermore, the background update process, responsible for fetching live data and refreshing the dashboard, was carefully tuned to operate with controlled sleep intervals between update cycles. This balance allowed for near real-time updates while keeping CPU utilization minimal.

In terms of scalability, dynamic routing was implemented to enable the dashboard to support multiple projects simultaneously within a single instance. This enhancement eliminated the need for separate deployments and improved the maintainability of the system. Additionally, the underlying data pipeline was refined to handle concurrent read and write operations more efficiently, ensuring smooth performance even when multiple regression test suites were active.

User-centric optimization also played a key role. Navigation efficiency was improved through quick-access hyperlinks, allowing direct entry to project directories and log repositories. Live updates ensured that progress could be tracked continuously without manual refresh or intervention.

These iterative refinements collectively enhanced the dashboard's overall speed, responsiveness, and scalability. The system now delivers faster load times, reduced resource usage, and smoother live updates, establishing it as a robust and dependable monitoring solution for Ciena's SAOS regression infrastructure.

## 3.9 Monitoring and Future Enhancements

**3.9.1 Monitoring and Feedback:** The system will be continuously monitored to ensure optimal performance, accuracy, and reliability. Regular user feedback will be collected to identify usability issues, performance bottlenecks, and areas for improvement. Logs and analytics will be reviewed periodically to assess model performance and user engagement.

**3.9.2 Plans for Future Enhancements:** Future enhancements may include expanding the dataset to improve model generalization, integrating real-time analytics, and developing a mobile-friendly interface. Additional features such as advanced visualizations, automated report generation, and support for multilingual users can also be incorporated to enhance accessibility and user experience.

# CHAPTER 4

# Result and Discussion

## 4.1 Performance Metrics and Evaluation

The automated dashboard pipeline for the SAOS Regression Infrastructure was evaluated based on three primary metrics: response time, resource utilization, and session handling efficiency. The objective was to ensure that the system could process regression data in real time while maintaining optimal performance and reliability under continuous operation.

In terms of response time, the automated pipeline significantly reduced manual overhead by processing regression outputs almost instantly after test completion. Tasks that earlier required manual log inspection were now automated, accelerating QA feedback loops by approximately 70%. This improvement enabled faster identification of test outcomes and quicker decision-making during regression cycles.

Resource utilization was optimized through efficient Python-based data handling and parallel execution mechanisms. The script minimized redundant file reads and computations, resulting in lower CPU and memory consumption even during high-volume regression activity. This optimization ensured that the system remained stable and responsive without causing strain on the underlying Linux infrastructure. Session handling efficiency was also enhanced through robust control logic that ensured smooth data retrieval and consistent updates on the dashboard. The system successfully maintained uninterrupted performance during prolonged runs, reflecting its scalability and resilience.

Overall, the evaluation demonstrated that the automated pipeline delivered consistent, robust, and scalable performance. It streamlined regression monitoring, reduced manual effort, and improved real-time visibility of test outcomes, marking a substantial step toward a more efficient and automated QA ecosystem within the SAOS regression framework.

## 4.2 Case Studies and Analysis

In deployment scenarios, the automated dashboard pipeline for SAOS Regression Infrastructure demonstrated strong performance and adaptability across multiple use cases. The system efficiently handled over ten projects simultaneously, including key test types such as CPD and CILITE, all operating on the same port. This concurrent handling validated the dashboard's scalability and reliability under real-world regression workloads.

The expected outcomes accurate status mapping, consistent execution time tracking, and a user-friendly visualization were fully achieved. The dashboard-maintained stability during peak test cycles, providing clear, color-coded summaries that eliminated the need for manual log inspection. This automation significantly improved turnaround time and streamlined QA processes.

During the testing and refinement phase, several optimization cycles were undertaken. The team iteratively enhanced the log parsing and data aggregation modules through more than fifteen updates to improve large file handling and reduce latency. Continuous feedback from users accessing the hosted dashboard helped identify and fix minor bugs quickly. These proactive improvements ensured that the final deployed version was both robust and optimized for performance, confirming the dashboard's reliability as a long-term regression monitoring solution.

## 4.3 Challenges and Solutions

During the development and deployment of the automated dashboard pipeline for SAOS Regression Infrastructure, several challenges were encountered and systematically addressed to ensure stability and reliability. One of the primary challenges was the dependency on consistent log file formats. Since the dashboard's parsing logic relied heavily on the structure of regression output files, any unexpected change in log format could cause misinterpretation or data loss. To mitigate this, robust validation checks and fallback parsing logic were introduced, allowing the system to adapt to minor inconsistencies without interrupting execution.

Another challenge involved managing additional parsing overhead during large-scale test runs. As the number of concurrent projects increased, log aggregation and table updates initially caused slight performance lag. This issue was resolved through multiple optimization cycles, including streamlined file reading methods and caching mechanisms to minimize I/O operations.

Lastly, during initial implementation, extensive validation was required to ensure accurate data mapping across diverse test types. The development team conducted iterative testing and community-based feedback sessions to identify and fix logic-level bugs rapidly. These refinements enhanced the dashboard's resilience, ensuring consistent real-time updates even under heavy load. Collectively, these solutions improved overall reliability and reduced maintenance overhead in live environments.

# CHAPTER 5

# FUTURE SCOPE AND CONCLUSION

## 5.1 Future Scope

The future scope of the Automated Dashboard Pipeline for SAOS Regression Infrastructure focuses on expanding its capabilities, improving automation intelligence, and integrating advanced analytics for predictive insights. While the current system efficiently automates the parsing, classification, and visualization of regression results, several areas hold potential for enhancement to make the pipeline more scalable, intelligent, and user-centric.

A major direction for future improvement is the integration of machine learning models to predict potential failures or anomalies based on historical test data. By analysing patterns across multiple regression cycles, the dashboard could proactively highlight tests likely to fail or systems that might experience performance degradation. This predictive feature would not only accelerate troubleshooting but also support preventive maintenance across test environments.

Another scope of advancement lies in multi-user access management and cloud deployment. Moving the dashboard from a local or Apache-based setup to a containerized or cloud-hosted environment (such as Docker or Kubernetes) would allow distributed access across global teams. Role-based access control (RBAC) can be incorporated to ensure secure usage while allowing different teams, QA, development, and operations to collaborate efficiently.

Lastly, implementing auto-healing mechanisms where the dashboard can re-run failed test cases automatically or identify recurring issues would make the pipeline more autonomous. As the SAOS infrastructure scales, such intelligent automation and data-driven insights will be critical in maintaining efficiency, transparency, and continuous improvement in the regression ecosystem.

## 5.2 Conclusion

The Automated Dashboard Pipeline for SAOS Regression Infrastructure has brought significant improvements to the efficiency and reliability of the overall testing ecosystem. By transforming a previously manual and time-consuming process into a fully automated and visually intuitive system, it has greatly streamlined the regression testing workflow. The dashboard's ability to parse, classify, and display results in real time has enabled teams to monitor test progress, analyze outcomes, and identify issues faster than ever before.

Through the integration of lightweight technologies such as Python, Flask, HTML, and CSS, the system maintains exceptional performance even under heavy loads, ensuring smooth operation across multiple test projects simultaneously. The automation of log parsing and data aggregation has eliminated manual dependency, reduced human error while accelerated QA feedback loops by nearly 70%.

This project demonstrates how a thoughtfully designed automation pipeline can bridge the gap between backend data handling and frontend visualization, resulting in enhanced productivity and system transparency. Beyond its technical achievements, it showcases the importance of simplicity, adaptability, and collaboration in engineering robust automation solutions.

In conclusion, this work stands as a foundation for future advancements in intelligent test monitoring, predictive analytics, and large-scale automation marking an essential step toward a smarter, self-sustaining regression infrastructure.

# REFERENCES

During the process of developing this project, I referred to various online materials and internal documentation of Ciena.

**To understand SAOS and related products:**


- Ciena Confluence Page
- https://www.ciena.com/products/service-aware-operating-system
- https://www.ciena.com/insights/articles/service-aware-architecture.html
- https://www.ciena.com/insights/what-is-service-aware-networking.html


**To understand tools and technologies used in development:**


- https://www.python.org/
- https://regex101.com/
- https://man7.org/linux/man-pages/man1/man.1.html
- https://httpd.apache.org/
- https://www.w3schools.com/html/html_css.asp
- https://bitbucket.org/product/
- https://www.atlassian.com/software/confluence
- https://flask.palletsprojects.com/
- https://developer.mozilla.org/en-US/docs/Learn/Server-side/Flask
- https://www.jenkins.io/doc/
- https://www.jenkins.io/doc/book/pipeline/
- https://www.geeksforgeeks.org/introduction-to-linux-virtual-machines/
- https://www.redhat.com/en/topics/virtualization/what-is-a-linux-virtual-machine
- https://www.tutorialspoint.com/python/python_cgi_programming.htm
- https://www.w3schools.com/bootstrap5/
- https://realpython.com/tutorials/flask/
- https://matplotlib.org/stable/gallery/index.html
- https://pandas.pydata.org/docs/

- https://jinja.palletsprojects.com/
- https://towardsdatascience.com/creating-dashboards-using-flask-and-plotly-24d89d4d163b
- https://www.atlassian.com/continuous-delivery/ci-vs-cd

**Information about Ciena:**

- https://www.ciena.in/at-a-glance
- https://www.ciena.in/
- https://www.ciena.com/about
- https://en.wikipedia.org/wiki/Ciena

φργττα λ,λ,δ,ω

σοοοοοοοοοοοοοοι