EasyChair Preprint

№ 9858

NEOS: Non-Intrusive Edge Observability Stack Based on Zero Trust Security Model for Ubiquitous Computing

Abhijit Kumar, Ahmed Tauseef, Konica Saini and Jay Kumar

March 10, 2023

NEOS: Non-intrusive Edge Observability stack based on Zero Trust security model for Ubiquitous Computing

Abhijit Kumar
Bengaluru, India
Email: abhijit.kumar@intel.com

Tauseef Ahmed
Bengaluru, India
Email: tauseef.ahmed@intel.com

Konica Saini
Bengaluru, India
Email: konica.saini@intel.com

Jay Kumar
Bengaluru, India
Email: jay.kumar@intel.com

***Abstract*—This paper introduces non-intrusive Edge Observability Stack(NEOS) that simplifies the process of collecting, analyzing, and visualizing telemetry data. It reduces the amount of code instrumentation needed to collect telemetry data up to 80% and offers extensive system and application observability. This is done by offering a set of user-friendly abstractions and easy-to-use APIs, which minimizes the effort needed for manual instrumentation of application code. NEOS leverages popular open-source tools such as Grafana, Prometheus, Jaeger, and Loki, for the visualization of telemetry data. Furthermore, NEOS security is based on the zero-trust model, which means that we assume that no user or system can be trusted by default. The security of every connection is done by employing mutual Transport Layer Security (mTLS) to prevent unauthorized access and safeguard sensitive data. Experiments were conducted to assess the efficiency of the stack by comparing the time and effort needed to instrument code with and without the stack. The outcomes showed a considerable reduction in instrumentation code and enhanced telemetry data. NEOS can be used by product managers, engineering and operation team for system and application health monitoring, real-time business insights, and debugging system.**

Keywords—Observability, Edge Observability Stack, monitoring, zero-trust security model, one-stop solution.

I. INTRODUCTION

Distributed edge systems need comprehensive observability to fulfill the requirement for high system availability and performance. However, the growing complexity of these systems has made observability an increasingly challenging discipline. In the absence of a comprehensive observability stack, observing distributed systems can be a time-consuming and error-prone process. Developers and system administrators will have to manually instrument code to collect telemetry data, which can be challenging and time-consuming. Furthermore, without adequate visualization tools, interpreting the collected data and finding issues can be an arduous task.

The modern-day challenges of observability include dealing with the complex and distributed nature of modern applications, which are often built using microservices, containers, and cloud platforms. This makes it difficult to trace requests through multiple services and find the root cause of issues. Additionally, managing the large volume of data generated by these systems can be a challenge, and without proper tools, interpreting the data can be overwhelming.

To address these challenges, we present the Edge Observability Stack, a complete observability stack that significantly reduces the amount of code instrumentation needed to collect telemetry data while also offering comprehensive system and application and system observability with a zero trust security model. Our Stack provides a set of easy-to-use APIs and user-friendly abstractions that eliminate the need for manual instrumentation of application code, making it an effective tool for simplifying the collection of telemetry data.

NEOS is designed to be platform-agnostic, working seamlessly on both Linux and Windows operating systems. To reduce onboarding efforts, we have developed containerized as well as native host implementations. This ensures that our stack can be integrated seamlessly into a wide variety of software environments, and it includes popular open-source tools such as Grafana [1], Prometheus [2], Jaeger [3], and Loki [4], all of which are integrated within the solution, eliminating the need for manual installation and configuration making it all-in-one solution for comprehensive system and application observability.

Our Stack is based on Zero Trust Security model as it ensures to authorize and authenticate every communicating micro-services using SSL certificates and keys. Mutual Transport Layer Security (mTLS) [5] is implemented for secure communication between different components of Edge Observability Stack. This provides end-to-end encryption and ensures that observability data is protected from unauthorized access.

NEOS provides real-time insights and data aggregation. For example, Grafana provides real-time data visualization, allowing developers and system administrators to monitor the performance of their systems in real-time making it powerful tool for optimizing system performance and identifying potential issues before they can impact users.

In the following sections, we will describe the design and implementation of the Edge Observability Stack, evaluate its effectiveness in reducing the amount of code instrumentation

needed, and show its capabilities for conducting comprehensive observability of distributed systems.

II. PROPOSED METHODOLOGY

In recent years, application observability has become a critical aspect of modern software development. It involves collecting and analyzing data about the performance and behavior of software applications to ensure that they are functioning as intended. This data is then used to diagnose and resolve issues, improve performance, and optimize resource utilization. To achieve this, it is necessary to instrument the application code with telemetry to provide visibility into the application's behavior and performance.

NEOS provides an application instrumentation library that implements an easy-to-use wrapper around the Opentelemetry metrics, traces, and logs APIs. This library is designed to be lightweight and does not add significant overhead to the application's runtime. It can integrate seamlessly with the application code and should not require significant changes to the codebase. Using our APIs, users can easily create Opentelemetry [6] instances required to send application telemetry data. This wrapper significantly reduces the amount of code instrumentation required to collect telemetry data, making it simple and efficient to integrate into existing applications. More details about it will be further discussed in the implementation section.

Instrumented applications emits observability signals(i.e trace, metrics, log), which are transmitted to a centralized location where it can be analyzed and visualized. This is typically achieved using an Opentelemetry Collector [7], which collects the telemetry data from various sources and sends it to a destination such as a logging service or a metrics aggregator. NEOS comes with pre-configured backends such as Prometheus [2], Jaeger [3], and Loki [4] that are used to visualize and store the collected data. Prometheus [2] is primarily used for monitoring systems and application metrics, while Jaeger is used to trace the performance of the system and troubleshoot latency issues. Loki [4] is used to store and query logs, providing a centralized location for application logs.

NEOS utilizes Grafana [1] as visualization tools for observability data. Our solution includes pre-configured Grafana charts and dashboards that we have created for visualizing a wide range of metrics and logs, such as system telemetry like CPU usage, GPU frequency, memory used, disk used to name a few. These dashboards are already included in our solution, allowing developers and system administrators to quickly gain insights into the health of their systems without having to create their own dashboards.

Our observability stack is designed to simplify the deployment process across different platforms. For Linux deployment, we offer two versions: one for secure connections and another for non-secure connections. Our stack leverages Docker [8] containerization to package all the necessary components, including the Opentelemetry collector [7], Telegraf [9] for system telemetry, Intel xpumanager [10] for GPU telemetry, Prometheus [2], Jaeger [3], and Grafana [1], into a single docker-compose.yaml file. The configuration files for all the tools are stored in a folder, which is volume-mounted in Docker-compose [11]. To ensure secure connections, we provide a bash script that generates SSL [12] certificates.

For Windows deployment, our stack provides batch scripts that run natively in the environment. The scripts include download-observability.cmd to download all the necessary tools and to configure all the tool configuration files, start-observability.cmd to start all the tools, and stop-observability.cmd to stop all the tools. Here too, to ensure secure connections, we have provided script that generates SSL certificates. Like the Linux version, we also offer two versions for Windows: one with secure connections and another for non-secure connections.

In addition to the ease of deployment of our stack in Linux and windows, we have ensured that all configurations of the tools are optimized to work well with one another. This is crucial for providing a seamless and efficient user experience, especially in complex distribution systems. For instance, if a sample contains all three components - metrics, traces, and logs - our stack enables users to jump from one trace to its corresponding log, simplifying the process of troubleshooting issues.

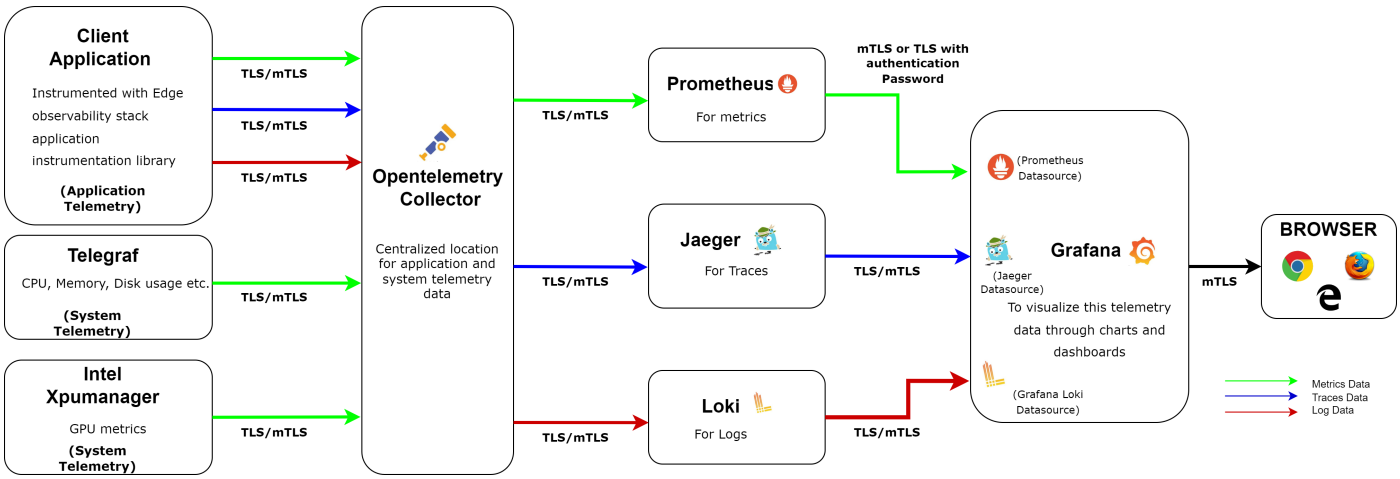

Figure 1: Edge Observability Stack Components

Moreover, the stack is highly customizable, allowing developers and system administrators to create their dashboards and alerts to monitor specific metrics and logs.

Figure 1 shows the components and flow chart of our observability solution depicting each connection between different components.All these components in detail are discussed in further sections.

NEOS is a powerful tool for developers and system administrators seeking to improve observability and troubleshoot issues in complex systems while maintaining a secure and non-intrusive environment.

III. IMPLEMENTATION AND DETAILS OF COMPONENTS

Here is a description of the system based on the sequence diagram in Figure 2. The system is comprised of a client application that is being instrumented using the Application Instrumentation Library for performance monitoring and data collection. The Application Instrumentation library is designed with a pure abstract class and base class of MetricsImpl, TracesImpl, LogsImpl classes which have MetricsImpl, TracesImpl, and LogsImpl APIs responsible for sending application data like metrics, traces and logs to the OpenTelemetry collector [7]. Telegraf [9] is utilized to collect system telemetry data such as CPU, memory, and disk usage, as well as GPU data, which is fed in from Xpumanager [10]. Once the data is collected, the collector is responsible for separating out metrics, traces and logs based on the configuration outlined in its configuration file. The data is then sent to Prometheus, Jaeger, and Loki tools via respective exporters from opentelemetry. Prometheus [2], Jaeger [3], and Loki [4] systems are used to store and manipulate collected. Finally, Grafana [1] is used to visualize this metrics via Prometheus [13], Jaeger [14] and Loki [15] datasources or by creating charts and dashboards of these collected metrics. In summary, the system's primary goal is to collect, analyze, and visualize application as well as system telemetry data, storing it in various systems for ease of use and analysis.

Each component of the system is explained in greater detail in the following sections, providing more in-depth information about how they work together to accomplish the goal of collecting, analyzing, and visualizing application and system telemetry data.

A. Application Instrumentation Library

The Application Instrumentation Client Library is designed to provide an easy-to-use interface for developers to instrument their code with metrics, traces, and logs. Design of this library address the need of an application to configure instrumentation of component at a subcomponent level which is a common requirement in instrumentation use cases. To address this, the Edge Instrumentation library adopts a two-part design, consisting of a static and dynamic library. The application subcomponent links with the static library, which includes their custom instrumentation configuration. The dynamic library, which is loaded by the static library, handles the common communication channel with the instrumentation collector [7]. This approach allows for per-component configuration, while minimizing the number of communication channels on a per-process basis.

To ensure the compatibility of the instrumentation library with other components, the dynamic library is loaded with custom-loaded properties that encapsulate the instrumentation library symbols from the rest of the application component. This is particularly important since the instrumentation library uses a protobuf [16] component for serializing and deserializing data, which is commonly used by inferencing frameworks like tensorflow [17]. Exposing the symbols of the instrumentation library can cause problems if there is a mismatch in the version of protobuf [16] used by these components. The dynamic library approach with custom loader flags effectively solves this problem by isolating the instrumentation-specific symbol from the global symbol space.

This library comprises Metrics, Traces and Logs classes which have APIs such MericsImpl, TraceImpl and LogsImpl API providing specific functionality for their respective instrumentation type. For example, the Metrics class provides methods for recording metrics, while the Traces class provides methods for recording traces. Another important class in the Application Instrumentation Library is the Client implementation class. This class is responsible for initializing the library and creating instances of the concrete classes that are inherited from base class. Application Instrumentation Library also provides several macros for logging messages of different severity levels, such as _INFO, _DEBUG, _WARN, _TRACE, _ERROR, and _FATAL. These macros are designed to be easy to use and provide a consistent format for logging messages.

Overall, the Application Instrumentation Library is a powerful and flexible library for instrumenting code with metrics, traces, and logs. It provides a clean and easy-to-use interface, while also allowing for customization and extensibility through its use of abstract classes and inheritance.

B. Opentelemetry Collector

Opentelemetry Collector is a vendor-agnostic, scalable, and customizable telemetry data collector that can receive, process, and export telemetry data from multiple sources to multiple destinations. The Opentelemetry Collector [7] architecture consists of four main components: receivers, processors, exporters, and extensions. The receiver component is responsible for ingesting telemetry data from various sources, including agents, libraries, and third-party services. Receivers can support various data protocols and formats, such as Opentelemetry Protocol (OTLP), Prometheus etc. Here, we have used OTLP receiver which have support for http and grpc protocol. The processor component is responsible for filtering, transforming, and enriching telemetry data before forwarding it to exporters. Processors can be used to extract additional metadata, redact sensitive data, and perform sampling and aggregation on the incoming data stream. Here, we have used a batch processor. The exporter component is responsible for sending the processed telemetry data to external systems, such as observability platforms, logging systems, and tracing analysis tools. Exporters support various data formats, such as JSON, Protocol Buffers, and Prometheus [2] exposition formats. Here we are using Prometheus exporter for metrics, Jaeger exporter for traces and Loki exporter for logs including

logging exporter enabled in all three to see logs on console as well.

The Opentelemetry Collector configuration is defined in a YAML-based configuration file. This file specifies the different components that make up the Opentelemetry Collector [7], their properties, and their relationships.

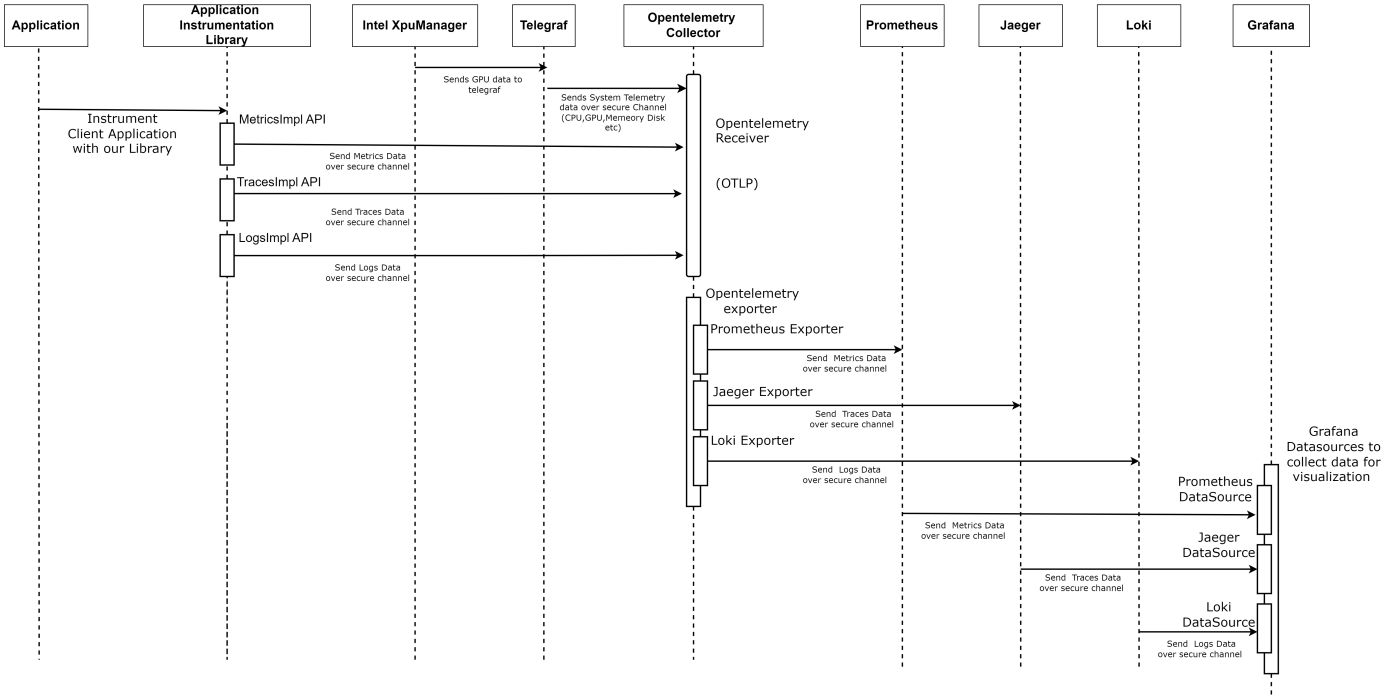

Figure 2: Sequence Diagram of Edge Observability Stack

For Application telemetry data, in our Application Instrumentation Library, we provide three instrumentation classes for Metrics, Traces, and Logs. These classes allow applications to send telemetry data to an Opentelemetry collector. For example, in our Logs instrumentation class, we initialize Opentelemetry instances depending on the type of exporter specified by the user with the endpoint specified in the instrumentation library configuration class. This exporter type is passed to the library as an environment variable. In this Logs instrumentation class, we initialize opentelemetry all instances such as exporter, processor, and provider necessary to send application telemetry logs to opentelemetry collector. Similar processes we are following in metrics and traces instrumentation classes for application telemetry data.

For System telemetry data we have used telegraf. Telegraf [9] is an open-source agent that collects, processes, and sends system data to various outputs. Telegraf uses a plugin-driven architecture, which allows it to gather data from a wide range of sources and formats. To collect system data, Telegraf comes with a set of built-in plugins, which can be configured in the telegraf.conf configuration file. These plugins include system metrics such as CPU, memory, disk usage, network traffic, and more. Telegraf can also be extended with third-party plugins, which can provide additional functionality, such as monitoring specific applications or devices.

Once Telegraf [9] has collected the system data, it can send it to various outputs using its output plugins. Here, we have set output plugin as opentelemetry [6] so that all collected system telemetry data can be sent to opentelemetry collector [7].

Furthermore, for GPU telemetry, to enable the collection and transmission of GPU telemetry data such as GPU frequency, engine utilization, and other related metrics to the collector, we have created a custom input plugin within Telegraf. This input plugin interacts with Intel XPUManager [10] to collect data and utilizes an in-built accumulator to process the collected data into a specified format before transmitting it to the Opentelemetry collector [7]. This approach ensures that the collected GPU telemetry data is processed and transmitted in a consistent and standardized format, thereby improving the overall efficiency and effectiveness of the data collection and transmission process.

C. Prometheus

Prometheus [2] is a highly scalable and efficient tool for collecting and storing time-series data, making it a powerful resource for system administrators. It boasts a flexible pull-based model for gathering metrics data, allowing it to scrape targets such as applications, servers, and services. This approach makes it possible for users to monitor a vast range of systems and applications with ease. One of the key advantages of Prometheus is its query language, which enables users to filter and aggregate metrics data in real-time. This capability is invaluable for identifying trends, troubleshooting issues, and optimizing performance. To configure Prometheus, we have defined targets and endpoints in a YAML configuration file called prometheus.yml. Targets refer to the endpoints from which Prometheus collects metrics data, and each target has a unique name (here otel_prom) and set of labels that can be used to filter and aggregate collected data. In our case,

the target is opentelemetry, and the endpoint is the same as defined in the opentelemetry collector Prometheus exporter. This configuration enables opentelemetry to export metrics data into Prometheus. Prometheus UI, we have secured and enabled user authentication using web.yml configuration file which comprises of SSL certificate [12] (root, server certificate and key) path data, username and password stored in hashed for authentication. The data collected from each target is stored in a time-series database, which is essential for efficient analysis and observation.

D. Jaeger

Traces are sequences of events that occur during the execution of an application, such as a request being processed, or a database query being executed. Jaeger [3] is an open-source distributed tracing system that is widely used to monitor and troubleshoot applications. With Jaeger, developers and system administrators can gain insights into the performance of their applications by collecting, storing, and analyzing traces. Jaeger provides several useful features for working with traces. One of the key features is its ability to trace requests across multiple services and systems. This is particularly useful for distributed applications that are composed of multiple microservices. Another useful feature of Jaeger is its ability to integrate with other monitoring and observability tools, such as Prometheus [2] and Grafana [1]. Here, from opentelemetry collector [7] we send traces at jaeger pre-confined protocol port (http or grpc) and via secure channel. Jaeger provides flags such as "–collector.grpc.tls.enabled" to enable grpc protocol to receive data "–collector.grpc.tls.cert" and "–collector.grpc.tls.key" to enable security over its UI and data transmission channels.

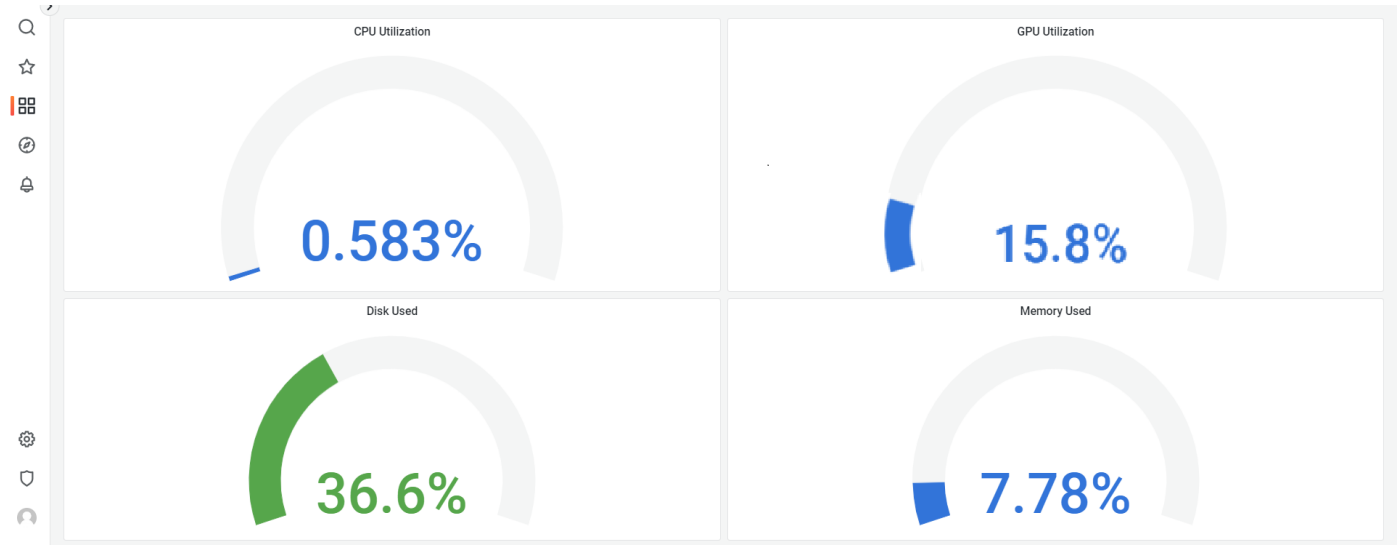

Figure 3: System Metrics Gauge View in Grafana

E. Loki

Loki [4] is a horizontally scalable, universally available, multi-tenant log aggregation system inspired by Prometheus [2]. It is designed to handle high-volume, high-throughput logging data, and provides a powerful query language for filtering and aggregating log data in real-time. One of the key benefits of Loki is that it uses a streaming architecture, which means that log data is processed and indexed in real-time, without the need for batch processing or indexing. Loki is built on top of the same infrastructure as Prometheus, which means that it shares many of the same benefits, such as a powerful query language and a scalable, distributed architecture. Loki also integrates seamlessly with other components of the Prometheus ecosystem, such as Grafana [1], which makes it easy to build powerful, real-time log dashboards. Here, we are mounting Loki endpoint for collecting logs in opentelemetry [7] config.yaml. Loki does not have mTLS [5] supported so we are using Nginx [18] to redirect http request to https to enable secure connection as our solution follows zero-trust security model.

F. Grafana

We have chosen Grafana [1] as our visualization tool and it has inbuilt support to add Prometheus [13], Jaeger [14], and Loki [15] as data sources. In these data sources we have mentioned all details of tools such as server details, end-points, ssl certificate [12], service names etc. required to build connection with these tools. We have designed customized dashboards with dedicated charts for CPU and GPU utilization, memory, and disk usage, which can be further modified based on the user's requirements. These datasources and dashboards are mounted to Grafana via docker [8] volumes in Linux environment and copy-paste these files at location in windows via script. To retrieve metrics data, we are using PromQL [19] queries, for instance, the CPU utilization is calculated using simple query '100 - max(otel_prom_cpu_idle_percentage)'. Here, otel_prom is the job name that we have assigned in Prometheus to distinguish the data and 'cpu_idle_percentage' is the system telemetry data obtained by Telegraf [9].

This system telemetry we have visualized in Grafana as Gauge meter as shown in Figure 3. For logs, we have used LogQL [20] queries to separate the severity, text, timestamp, and other details, which are stored in separate columns. This

approach makes it easier to troubleshoot issues. For traces, we are visualizing distributed traces according to the services and microservices invoked. Figure 4 shows the distributed trace view of sample_traces application depicting how much time each request took (server, client-server, server-request). Figure 5 shows the detailed view of request showing information like its start time, time taken by this request etc. as visualized in Grafana.

Additionally, we have implemented an inter-jumping feature that allows users to move seamlessly from logs to traces at a particular instance. Grafana does not have mTLS [5] support thus, enabling it via nginx. Nginx [18] is used as a redirecting tool in our edge observability stack to redirect http request to https via its configuration file. Grafana [1] tool we can open on any browser.

By integrating all these features into our system, we have provided an intuitive, secure, and efficient platform for monitoring and managing large-scale applications.

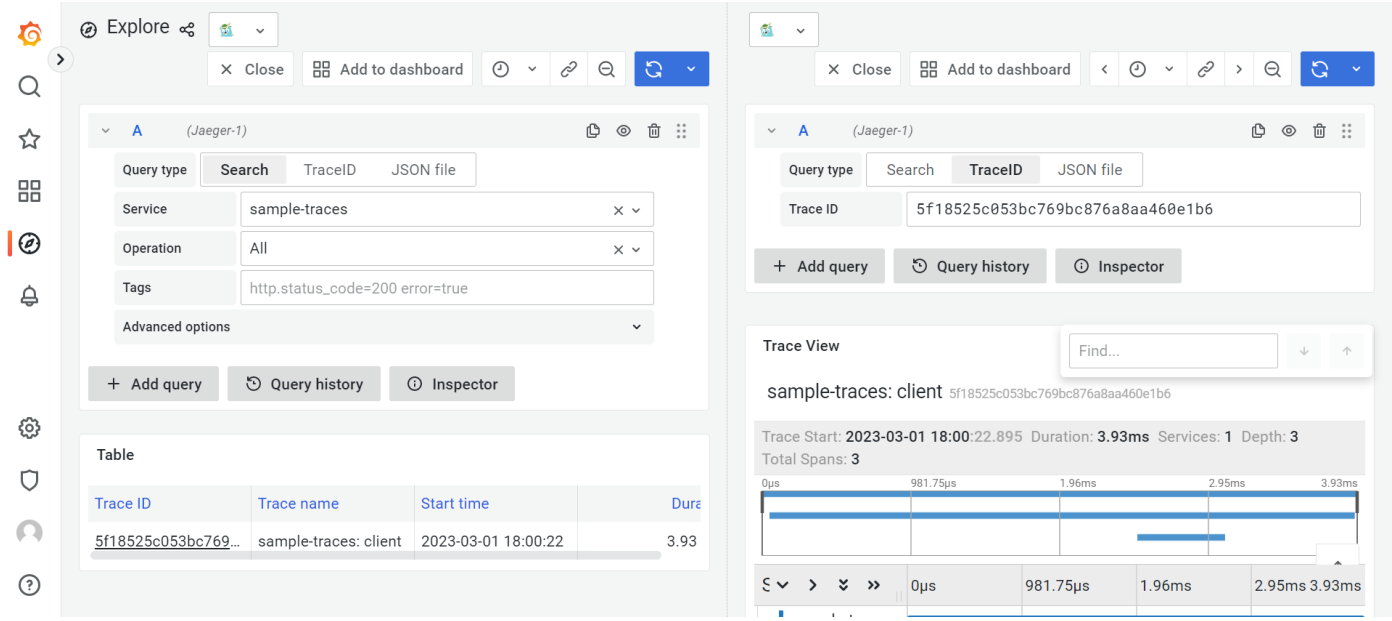

Figure 4: Trace View in Grafana

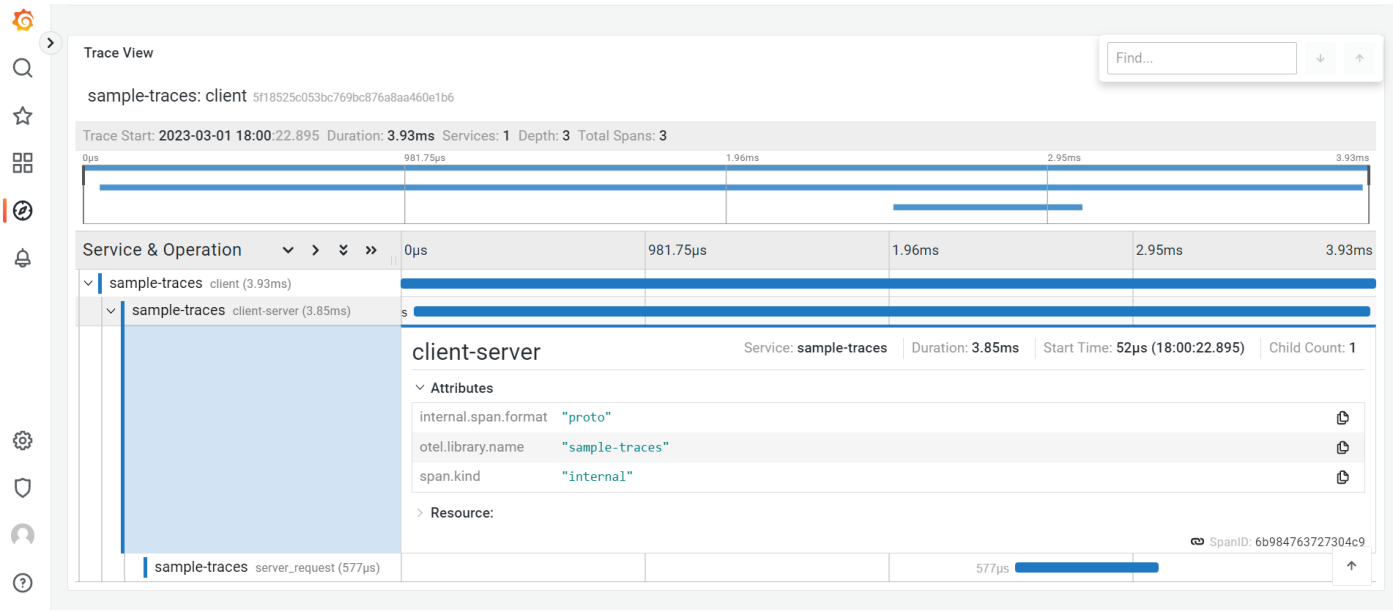

Figure 5: In Distributed system particular service Trace View in Grafana

G. SSL Certificates for Secure Connection

To ensure secure communication between the different components of our system, we have implemented SSL [12] certificates using Python code and scripts. Our approach involves using the cryptography library to generate both the root certificate and tool certificates. To generate the root certificate, we have written a generate-ca.py script that uses the cryptography.x509 [21] module to create a self-signed root certificate.

We specify the subject name of the root certificate using the NameOID class and set the basicConstraints extension to indicate that the certificate is a CA. To generate certificates for each tool, we use the generate-certs.py script, which generates a certificate signing request (CSR) for each tool and signs it is using the root certificate. We set the subject name of the certificate using the NameOID class and set the subjectAltName extension to include the hostname or IP address of the tool. To streamline the process for users, we have created a generate-certificates.sh script that automates the process of generating SSL certificates. All the required tools and dependencies are stored in a dedicated folder for easy access. By implementing SSL certificates, we can ensure that communication between the different components of our system is encrypted and secure thus providing zero-trust security model in our observability stack.

H. Docker Containerization

In our Linux-based system, we are utilizing Docker [8] containerization to deploy various tools for monitoring and analysis. To facilitate this, we are utilizing images for each tool, apart from Telegraf. The images we are using for Opentelemetry [6], Grafana [1], Prometheus [2], Jaeger [3], Loki [4], and Xpumanager [10] are obtained from Docker Hub. For Telegraf, we created a custom image using a Docker file to incorporate a custom-built GPU input plugin. Docker provides us with an effortless way to build and distribute images across different systems, ensuring consistency and portability. To ensure that the configuration files and necessary files for each tool are available within the container, we used Docker volumes to mount these files with read-only access. By mounting the files using volumes, we can ensure that the data persists even when the container is destroyed, and we can keep the configuration files separate from the container image. Additionally, we mounted all the certificates required for secure connections with read-only access, ensuring that the security of the system is maintained. One of the key benefits of using Docker for containerization is that it allows us to package an entire application or service into a single container. This makes it easy to move the application between different environments or hosts, without worrying about compatibility issues. In our implementation, we used Docker Compose [11] (docker-compose.yaml) to define and run the containers for all the tools required. Docker Compose provides a straightforward way to define multi-container applications, with the ability to scale up or down as required. By using Docker Compose [11], we can start and stop all the containers with a single command, simplifying the management and deployment of the system. We are also providing two versions of our solution, one with security and one without security.

IV. EXPERIMENTAL SETUP AND ANALYZING DATA

A. Experimental Setup

To comprehensively evaluate the effectiveness of our solution, we designed and implemented a range of sample applications, covering metrics, traces, and logs. We also instrumented a media application that is specifically focused on video inferencing. This application is used to create optimized end-to-end pipelines across multiple media frameworks, including GStreamer, FFMPEG, and Windows Media Foundation. With the help of our library, we were able to trace the fps and latency of this video inferencing application.

We have successfully integrated our library with the FPS data recorder and Latency data recorder code, enabling them to send data to our observability stack, as illustrated in Figure 6 in Linux environment. By instrumenting these components with our library, we have enhanced the monitoring capabilities of our solution and enabled the tracking of critical performance metrics such as FPS and latency in real-time.

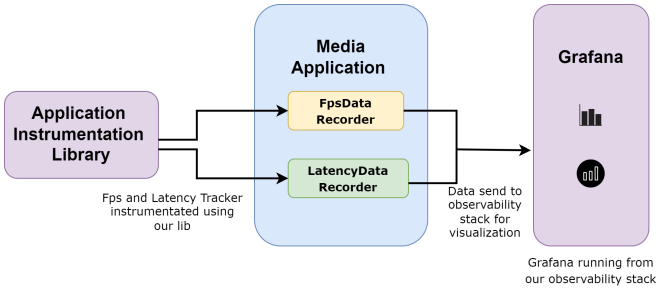

Figure 6 : Media Application Instrumentation Diagram

In addition to these sample applications, we conducted benchmark tests with multiple pipelines, measuring performance metrics such as CPU time, allocation size, module running, and microstructure usage. These tests allowed us to identify areas for improvement and ensure that our solution can handle enormous amounts of data and scale effectively in production environments.

Our benchmark tests also included testing the efficiency of our media application. We ran tests to determine how much load the application can handle before the fps and latency begins to drop. These tests have provided us with valuable insights into the capabilities of our solution and how it can be utilized for monitoring and troubleshooting applications.

Overall, our sample applications and benchmark tests have demonstrated the effectiveness and efficiency of our solution for collecting, analyzing, and visualizing metrics, traces, and logs data in a containerized [8] environment. These results have important implications for organizations that are looking to implement similar solutions, particularly those that require video inferencing capabilities.

B. Analyzing Data

After integrating our library with the media application's FPS Data Recorder and Latency Data Recorder code, we were able to receive real-time data for both metrics. Specifically, we received data for the current FPS and Latency, as well as average FPS and Latency data. To gain further insights and make the data more interpretable, we utilized Grafana [1] to visualize the data as shown in Figure 7 and Figure 8.

We created an observability dashboard that consists of multiple charts for each metric. For FPS, we created four different charts that display the current FPS data as a time-series, the current FPS data in a histogram format, the average FPS Data as time-series and the average FPS Data is bar-chart format. These chart in Figure 7 provide a comprehensive overview of the FPS data and enable

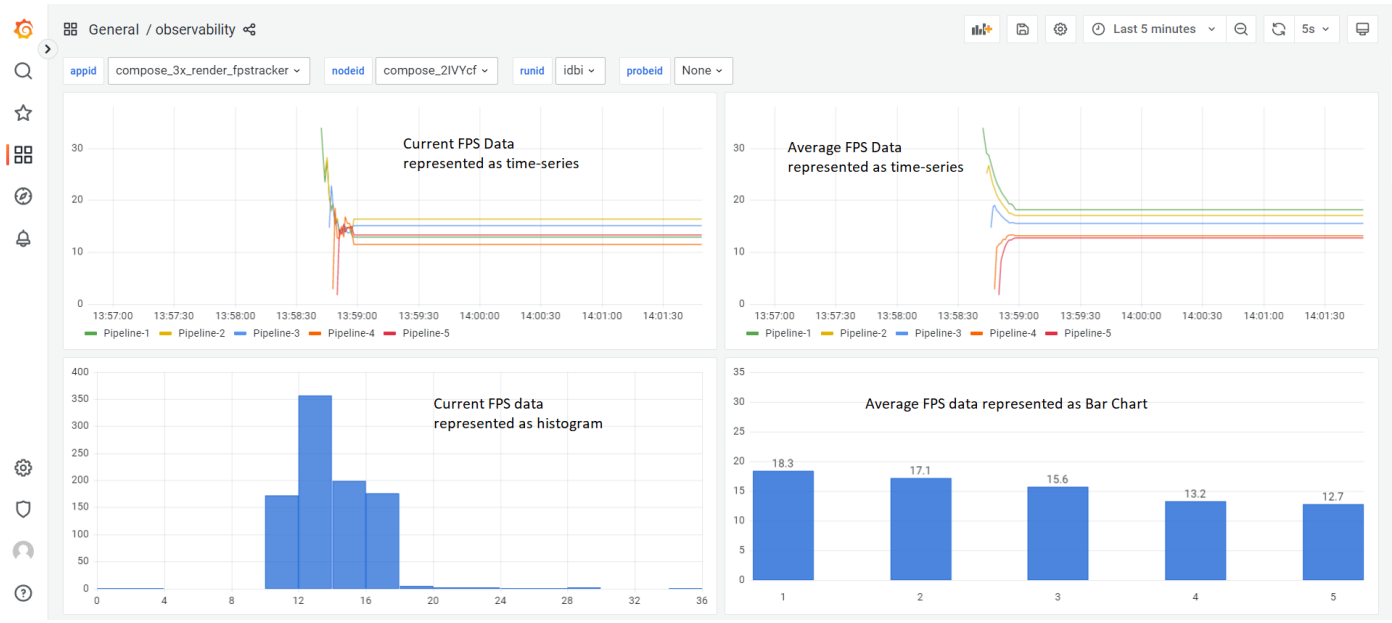

Figure 7: FPS Data as Visualized in Grafana

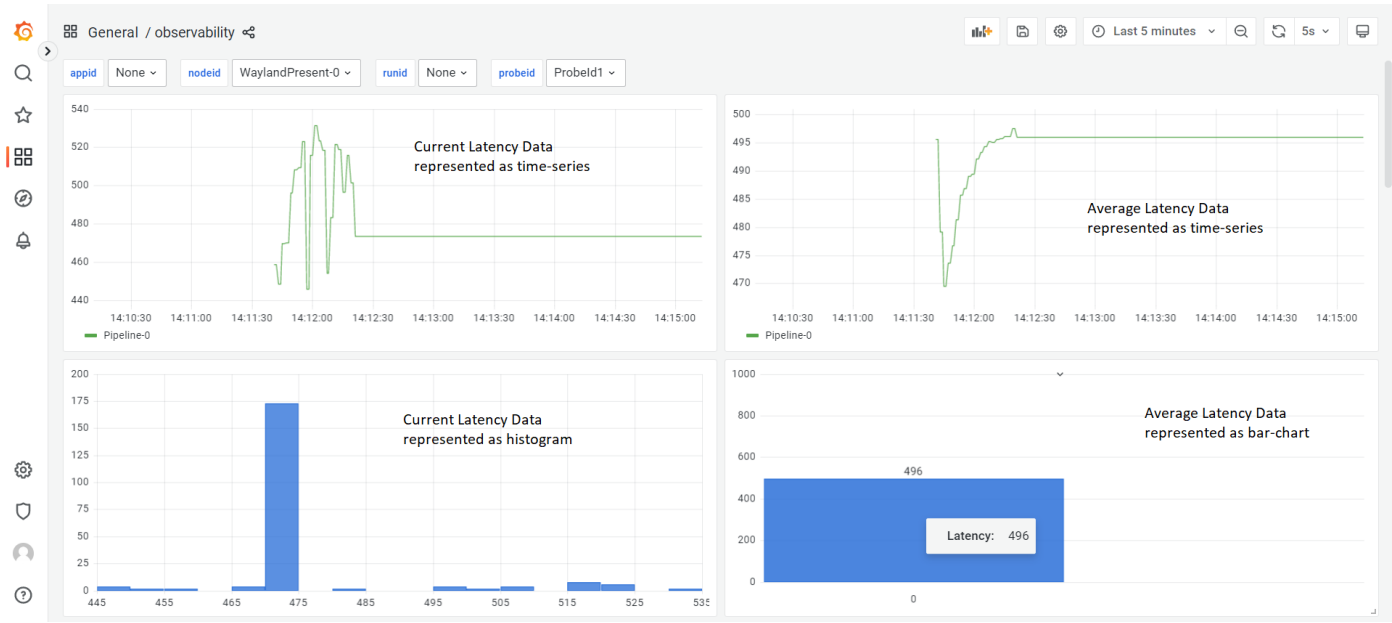

Figure 8: Latency Data as Visualized in Grafana

The chart of current FPS as time-series depicts the variation of FPS over time, showing how the FPS changes during various parts of the media application execution. This chart is useful in identifying any trends or patterns in the FPS data, such as spikes or drops in FPS during operations or phases of the application. On the other hand, the histogram of current FPS data provides a distribution of FPS values, which could help in identifying any outliers or abnormalities in the data. It provides insight into the most common FPS values and their frequency of occurrence, which are useful in optimizing the application's performance. The chart of Average FPS as a time-series depicts the average FPS of the media application over a period. It provides insight into the overall performance of the application and how it fluctuates over time. The bar chart of Average FPS is used to compare the average FPS of the application across different runs or configurations. It is useful for identifying performance bottlenecks or improvements in the application. The decreasing trend in average FPS values indicates that the application is experiencing some performance issues, such as high CPU usage or memory constraints, as it processes more videos which developers need to check.

The charts of current Latency as time-series and histogram depict the distribution and variation of the latency values observed during the media application's execution as shown in Figure 8. The time-series chart shows how the latency values change over time, which can help identify trends and patterns

in the data. The histogram, on the other hand, shows the frequency of different latency values, providing insight into the distribution of the latency values and any outliers that may exist. Overall, these charts can help developers understand the performance of the media application and identify areas for improvement, such as reducing latency or addressing any outliers.

The charts of Average Latency as time-series and bar-chart depict the performance of the media application in terms of how much time it takes to process each frame. The time-series chart shows the trend of the average latency over time, which helps in identifying any spikes or drops in performance. The bar-chart shows the distribution of the latency values, which gives an idea about the overall latency of the application. A lower value in the latency charts would indicate that the media application is performing well, while a higher value would indicate that there is room for improvement.

V. RESULTS

In this section, we will present the results of our performance monitoring of two applications: our sample examples and the media application. We instrumented both applications using our application instrumentation library, which allowed us to collect data on CPU time and memory usage during their execution. By measuring the extra CPU time or memory usage taken by each application due to the instrumentation, we were able to assess the impact of our monitoring on the performance of the applications.

We collected data, during which time we captured and analyzed various performance metrics, such as response times, resource usage, CPU time and memory usage. To ensure the accuracy and reliability of our measurements, we used the vtune profiler [22], a performance profiling tool that allowed us to capture detailed data on the applications' resource usage.

VTune Profiler [22] is a performance profiling tool developed by Intel Corporation. It allows developers to analyze the performance of their applications at various levels, including CPU, memory, I/O, and threading. The tool can identify performance bottlenecks and provide recommendations for optimization. VTune Profiler [22] supports a wide range of platforms, including Windows, Linux, and macOS. It can be used to profile applications written in various programming languages, such as C, C++, Fortran, and Python. The tool provides several profiling modes, including hotspots analysis, which identifies the most time-consuming parts of the application, and concurrency analysis, which identifies threading issues and synchronization problems. It also provides a memory analysis mode that can help identify memory leaks and inefficient memory usage.

Before proceeding with the actual testing, it is important to provide information about the Linux CPU on which these tests were performed. The Linux system used in our experiments had an x86_64 architecture and supported both 32-bit and 64-bit CPU op-modes. The system had a Little-Endian byte order and supported 39 bits of physical address space and 48 bits of virtual address space. The CPU used in our system was an Intel(R) Xeon(R) CPU E3-1275 v5 @ 3.60GHz, with 8 physical cores and 16 logical threads (2 threads per core). Each physical core had access to a 128 KiB L1 data cache and a 128 KiB L1 instruction cache, while the entire CPU had a shared 1 MiB L2 cache and an 8 MiB L3 cache. The CPU also supported virtualization through VT-x technology. To ensure consistent and reliable testing results, we ran our experiments on all 8 physical cores of the CPU, which were available for testing. We also used the vtune profiler [22] to monitor the CPU usage and performance metrics during the experiments. To ensure consistent and reliable testing results, we ran our experiments on all 8 physical cores of the CPU, which were available for testing. We also used the vtune profiler to monitor the CPU usage and other performance metrics during the experiments. By providing this information, we can ensure that the reader has a clear understanding of the CPU environment in which the tests were performed and can better interpret the results obtained.

Now let's move to the results section. To evaluate the effectiveness of our instrumentation approach, we conducted experiments on three sample applications. The first sample application consisted of basic instrumentation of logs, the second sample application consisted of instrumentation of metrics, and the third sample application consisted of instrumentation of traces which also includes multithreaded traces.

We conducted CPU and memory usage analyses on our logs sample application using instrumentation and without instrumentation. With instrumentation, we found that the CPU time was 24 microseconds out of a total elapsed time of 162 microseconds, indicating a microarchitecture usage of 37.5% with a CPI (Cycle per instruction) of 1.143.

Function/ Function Stack	Allocation/Deallocation Delta	Allocation Size	Deallocation Size	Allocations	Module
Instrumentation API-1	321.6 KB	643.1 KB	321.5 KB	21	sample_logs
Instrumentation API-2	240.7 KB	345.4 KB	104.7 KB	2689	sample_logs
Func@0xa6a64	72.7 KB	72.7 KB		1	libstdc++.so.6
_registerframe	528 B	528 B		11	libgcc_s.so.1
[Unknown]				1	

Table 1 : Memory Consumption for logs_sample

Additionally, the memory consumption analysis showed an allocation size of 1.1 MB and a deallocation size of 426.2 KB. Memory consumption w.r.t. to our application instrumentation library APIs which are given in below table (Table 1). These APIs are instrumentation initialization APIs responsible for initializing our instrumentation process.

On the other hand, without instrumentation, we observed a CPU time of only 2 microseconds out of a total elapsed time of 5 microseconds, indicating a microarchitecture usage of 0.00% with a CPI of 2. The memory consumption analysis showed an allocation size of 73.2 KB. These results suggest that instrumentation can have a significant impact on both CPU time and memory consumption.

Similarly, In our metrics sample, we conducted the same analysis as we did for the logs sample. We found that with

instrumentation, the CPU time was 23 microseconds out of a total elapsed time of 155 microseconds. The microarchitecture usage was 0.00% with a CPI of 1.143. Additionally, the memory consumption analysis showed an allocation size of 1.1 MB and a deallocation size of 426.2 KB.

Function/Function Stack	Allocation/Deallocation Delta	Allocation Size	Deallocation Size	Allocations	Module
Instrumentation API-1	321.6 KB	643.1 KB	321.5 KB	21	sample_metrics
Instrumentation API-2	240.7 KB	345.4 KB	104.7 KB	2689	sample_metrics
Func@0xa6a64	72.7 KB	72.7 KB		1	libstdc++.so.6
__register_frame	528 B	528 B		11	libgcc_s.so.1
[Unknown]				1	

Table 2 : Memory Consumption for metrics_sample

Memory consumption w.r.t. to our application instrumentation library APIs which are given in below table (Table 2). These APIs are instrumentation initialization APIs responsible for initializing our instrumentation process.

However, without instrumentation, we observed that the CPU time was only 1 microsecond out of a total elapsed time of 2 microseconds. This indicates that the microarchitecture usage was also 0.00%, but the CPI was 1. The memory consumption analysis showed an allocation size of 73.2 KB.

Function/Function Stack	Allocation/Deallocation Delta	Allocation Size	Deallocation Size	Allocations	Module
Instrumentation API-1	321.6 KB	643.1 KB	321.5 KB	21	sample_traces
Instrumentation API-2	240.7 KB	345.4 KB	104.7 KB	2689	sample_traces
Func@0xa6a64	72.7 KB	72.7 KB		1	libstdc++.so.6
main	4.1 KB	4.1 KB		1	sample_traces
__register_frame	528 B	528 B		11	libgcc_s.so.1
[Unknown]				1	

Table 3 : Memory Consumption for trace_sample

Similarly, In our traces sample, we conducted the same analysis as we did for the logs sample. We found that with instrumentation, the CPU time was 23 microseconds out of a total elapsed time of 172 microseconds. The microarchitecture usage was 13.9% with a CPI of 1.2. Additionally, the memory consumption analysis showed an allocation size of 1.1 MB and a deallocation size of 426.2 KB. Memory consumption w.r.t. to our application instrumentation library APIs which are given in table (Table 3). These APIs are instrumentation initialization APIs responsible for initializing our instrumentation process.

However, without instrumentation, we observed that the CPU time was only 1 microsecond out of a total elapsed time of 2 microseconds. The microarchitecture usage was also 0.00%, but the CPI was 2. The memory consumption analysis showed an allocation size of 73.2 KB.

Moreover, for multithreaded traces sample with instrumentation, the CPU time was 32 microseconds which is quite greater than as compared to simple traces. This could be due to the overhead of managing multiple threads and the additional synchronization required to ensure correct execution. Microarchitecture usage was also higher at 36.5%, indicating that the processor was working harder to execute the code with a CPI of 1.6. The CPI of 1.6 suggests that more instructions were being executed per cycle, possibly due to more efficient use of the processor's resources. The memory consumption analysis showed an allocation size of 418.6KB and a deallocation size of 104.7KB, indicating that more memory was being freed up during the execution of the program.

Overall, these results suggest that the use of multithreading in combination with instrumentation can have a significant impact on both CPU time and memory consumption and careful consideration must be taken when designing and implementing multithreaded applications to ensure optimal performance.

To further analyze the impact of instrumentation on the media application, we can look at the specific numbers obtained from the profiling results. With instrumentation, the CPU time taken by the application is 8.1 seconds out of a total elapsed time of 49.842 seconds. Additionally, the microarchitecture usage is reported to be 9.8% and CPI value of 3.876.

Module/Function/Call Stack	CPU time	Instruction Retired	Microarchitecture Usage	CPI Rate
otlp_recordable.so	0.053s	80500000	46.0%	1.783
Instrumentation module	0.084s	10500000	15.2%	20333
Exporter_otlp_http_client.so	0.054s	63000000	34.3%	1.944

Table 4 : CPU analysis of media Application

To get a more detailed view of the impact of instrumentation on the media application, Table 4 is included. This snapshot shows the microarchitecture usage and CPU time taken by the application instrumentation library. By analyzing this data, we can identify specific areas of the code that may be causing performance bottlenecks and potentially optimize them to improve overall application performance.

Comparing the results of all three scenarios, we can see that instrumentation can have a significant impact on both CPU time and memory consumption. This is because when we instrument an application, it adds additional code that needs to be executed, which takes up CPU time and memory. However, the benefits of instrumentation are also clear - it allows us to gather valuable performance metrics that can help us optimize and improve our applications. Therefore, it is important to carefully consider the trade-offs between the benefits and costs of instrumentation when performing performance analysis.

VI. CONCLUSION

To conclude, our paper has highlighted the challenges faced by developers and system administrators when it comes to observability of complex systems. The complexity of modern distributed systems makes it difficult to collect, analyze, and visualize telemetry data. We have proposed a solution NEOS to these challenges by providing an easy-to-use instrumentation library that simplifies the process of collecting telemetry data, as well as a pre-configured observability stack that allows for real-time insights and data aggregation. Furthermore, our solution offers a streamlined deployment process for both Linux and Windows, and all configurations of the tools are optimized to work well with one another. This makes the user experience seamless and efficient, especially in the complex distributed systems. In our testing, we have found that our instrumentation library is efficient in terms of execution time, CPU time, and allocation time. Overall, our observability solution simplifies the process of collecting, analyzing, and visualizing telemetry data, and enables developers and system administrators to focus on analyzing data and troubleshooting issues. Our solution is a powerful tool for improving observability and troubleshooting issues in complex systems while maintaining a secure and non-intrusive environment.

Disclaimer: *By submitting this paper, the authors confirm that the work presented is original and has not been previously published, nor is it under consideration for publication elsewhere. The authors acknowledge that Intel and its affiliates are not responsible for the content of this paper, including any errors or omissions, and make no warranties, express or implied, as to the accuracy, reliability, or completeness of the information presented.*

REFERENCES

[1] "Grafana," in *https://grafana.com/docs/grafana/latest/*.

[2] "Prometheus," in *https://prometheus.io/*.

[3] "Jaeger," in *https://www.jaegertracing.io/docs/1.22/*.

[4] "Loki," in *https://grafana.com/docs/loki/latest/*.

[5] "mtls," in *https://en.wikipedia.org/wiki/Mutual*$_a uthentication$.

[6] "Opentelemetry," in *https://opentelemetry.io/*.

[7] "Opentelemetry collector," in *https://opentelemetry.io/docs/collector/getting-started/*.

[8] "Docker," in *https://www.docker.com/what-docker*.

[9] "Telegraf," in *https://www.influxdata.com/time-series-platform/telegraf/*.

[10] "Intel xpumanager," in *https://www.intel.in/content/www/in/en/software/xpu-manager.html*.

[11] "Docker-compose," in *https://docs.docker.com/compose/*.

[12] "Ssl," in *https://en.wikipedia.org/wiki/SSL*.

[13] "Prometheus grafana datasource," in *https://grafana.com/docs/grafana/latest/datasources/prometheus/*.

[14] "Jaeger grafana datasource," in *https://grafana.com/docs/grafana/latest/datasources/jaeger/*.

[15] "Loki grafana datasource," in *https://grafana.com/docs/grafana/latest/datasources/loki/*.

[16] "Protobuf," in *https://protobuf.dev/*.

[17] "Tenserflow," in *https://www.tensorflow.org/*.

[18] "Nginx," in *https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/r*.

[19] "Promql language," in *https://grafana.com/blog/2020/02/04/introduction-to-promql-the-prometheus-query-language/*.

[20] "Logql," in *https://grafana.com/docs/loki/latest/logql/*.

[21] "Cryptography," in *https://cryptography.io/en/latest/x509/index.html*.

[22] "Vtune profiler," in *https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.htmlgs.qvwoi0*.