Impact of OpenTelemetry Configuration on Observability and Telemetry Storage Cost of Microservices-Based Applications

Francisco Gomes, Vinicius Gabriel, Paulo Rego, Fernando Trinta, José de Souza

Impact of OpenTelemetry Configuration on Observability and Telemetry Storage Cost of Microservices-Based Applications

Francisco A. A. Gomes
Federal University of Ceará (UFC)
Crateús, Ceará, Brazil
almada@crateus.ufc.br

Vinicius B. Gabriel
Federal University of Ceará (UFC)
Fortaleza, Ceará, Brazil
vinicius.bg@alu.ufc.br

Paulo A. L. Rego
Federal University of Ceará (UFC)
Fortaleza, Ceará, Brazil
paulo@dc.ufc.br

Fernando A. M. Trinta
Federal University of Ceará (UFC)
Fortaleza, Ceará, Brazil
fernando.trinta@dc.ufc.br

José N. de Souza
Federal University of Ceará (UFC)
Fortaleza, Ceará, Brazil
neuman@ufc.br

ABSTRACT

In response to the increasing scale and complexity of modern software systems, microservices architecture has emerged as a fundamental solution, enabling continuous scalability and surpassing intricate monolithic codebases. Despite its adoption to enhance performance and reduce labor costs, the transition to a microservices architecture introduces new challenges for developers. As cloud-native applications become more distributed and complex, potential failure modes increase, and performance may degrade, rendering traditional monitoring solutions inadequate. Observability encompasses practices and tools that extend beyond monitoring, offering insights into a system's internal state by analyzing its external outcomes. Observability gathers telemetry data about the application for future analysis by operators. Despite that, the volume of generated data can be detrimental to costs. The more data are monitored, the more storage is required, and thus, the higher the cost in the cloud. This study explores the process of integrating a microservices-based benchmark application using an observability solution called OpenTelemetry (OTel) and evalute the impact on the cost of storing the telemetry generated. The findings reveal the benefits of using OTel for application observability, but users monitoring the application need to assess what is crucial to know about the application to avoid unnecessary expenses.

CCS CONCEPTS

• **General and reference** → *Performance*; *Evaluation*; • **Computer systems organization** → *Cloud computing*.

KEYWORDS

Microservices, Observability, OpenTelemetry

1 INTRODUCTION

In recent years, rapid advancements in Internet technology have led to the growth of modern software systems in scale and complexity, posing increasing challenges in incorporating new functionalities. The emergence of microservices architectures [8], coupled with the Development and Operations (DevOps) [11] design philosophy, has brought about significant transformations in the development, deployment, and management of user applications. Unlike the conventional monolithic approach, where the entire application is constructed as a unified system, a microservices architecture breaks down the application into multiple independent executable components that seamlessly collaborate to deliver specific application functionalities. Practices such as the widespread adoption of lightweight RESTful Application Programming Interfaces (APIs) [4] are employed to facilitate real-time communication among microservices. With IT infrastructures expanding and becoming more decentralized [1], driven by the promise of accelerated innovation, cost efficiency, and heightened agility, numerous enterprises are transitioning their traditional monolithic applications to microservices [17].

Although monitoring has been a central function in IT for decades, it has begun to show limitations due to factors such as agile development methodologies, native cloud deployments, and new DevOps practices. These factors have changed how the entire IT ecosystem (e.g., infrastructures, systems, and applications) should be monitored to respond promptly to incidents. For instance, most edge infrastructures are orchestrated through Kubernetes [6], aiming to abstract much of the underlying complexity. However, the platform consists of various independent projects and interconnected components. Therefore, simply deploying traditional monitoring tools doesn't work immediately for several reasons, such as various influencing components being hosted outside the containerized application, the underlying environment being more dynamic, and applications having faster deployment cycles. Observability is an emerging set of practices combined with tools that go beyond monitoring to provide insights into the internal state of a system by analyzing its external outcomes [17].

Observability can be defined as the capacity to infer the status of a complex system based on its outputs. It also refers to a system's property that can be observed and compared with the expected state throughout the software development life cycle [10]. Existing observability tools identify three data sources to collect and process information: (i) metrics, which are values describing the status of processes and resources of a system, such as per-process memory consumption; (ii) logs, which are reports of software execution, and (iii) traces, which are composed of causally-related data representing the flow of requests in the system. OpenTelemetry (OTel) is an open-source observability framework that provides a set of standardized, vendor-independent SDKs, APIs, and tools for ingesting, transforming, and sending data to an observability backend [5]. The more data that is monitored, the more storage is

required, thus, the higher the cost in the cloud. No study was found that examines the cost impact of storing telemetry generated by OTel.

This study describes the process of integrating a microservices-based benchmark application with OTel. The main objective is to evaluate the cost impact of storing telemetry generated by OTel when observing an application in use by users. Our key finding suggests advantages to using OTel for application observability, but users need to assess the cost implications. Depending on their needs, this observation may result in an increase in the cost of storing telemetry in the cloud. The following research question guided the study:

RQ *What is the cost of storing telemetry from a microservices-based application when using OTel?*

The contributions of the article are as follows: We use OTel and discuss its use, emphasizing management and observability. We defined a workload for experimentation and conducted experiments to evaluate the cost of storing the telemetry generated by OTel to perform the observability of a microservices-based benchmarking application.

The rest of the paper is organized as follows. Section 2 presents the theoretical foundation, focusing on key concepts for understanding the work. Section 3 discusses related work. The benchmarking application used and OTel integration are explained in Section 4. Section 5 contains the experiments and results. Finally, Section 6 concludes the study along with suggestions for future research.

2 BACKGROUND

2.1 Microservices

The microservices has empirically emerged from architectural patterns used in the real world, where systems are composed of services that collaborate to achieve their objectives, communicating through lightweight mechanisms (such as Web APIs) [9]. These services are built around a specific part of the business and are deployed in a fully automated manner. They can be written in different programming languages and use different database technologies.

The microservices architecture addresses the complexity issue by decomposing the monolithic application into a set of microservices. Each microservice has a well-defined boundary, ensuring a practical level of modularity that is challenging to achieve in a monolithic application. Consequently, individual services are easier to develop, understand, and maintain. Independent teams work on each microservice, employing technologies they deem most suitable (both in terms of software and hardware), following the organization's rules. Each microservice is deployed independently, facilitating the updating of new versions without impacting the deployment of other microservices [2].

Although it is recommended that microservices be small, the goal is for them to correspond to decompositions of the application to facilitate agile development and deployment. A microservices architecture corresponds to a distributed system with interprocess communication between them, which is more complex than invoking methods at the language level and is also more challenging to test, for example, requiring integration tests between microservices. Database partitioning among microservices brings the challenge of ensuring data integrity distributed among them. There is a greater number of parts to be configured, deployed, scaled, and monitored, requiring significant control and a high level of automation [14].

2.2 Observability

Observability is a term borrowed from control theory. In computer science, it is defined as a feature of software and systems related to the information they generate, allowing for more comprehensive monitoring and understanding, including at runtime. Beyond simple black-box monitoring, Observability can provide greater insight into the correctness and performance of services. One of its goals is to reduce the time needed to understand why something is not functioning as it should. It is an inherently data-intensive and time-sensitive process [10]. Observability is an emerging concept that has been used to refer to advanced monitoring functions in the context of microservices-based applications. Observability can be considered a superset of monitoring, expanding its concept and applying data analysis techniques to monitoring data [13]. Observability is more closely related to white-box monitoring and the generation and consumption of traces as the primary basis for information and decision-making [7]. This process enables not only the analysis of the current behavior but also the inference of the future behavior of a system. The observability of a distributed system can be enhanced by employing what is commonly referred to as the three pillars (signals) of observability: logs, metrics, and tracing.

2.2.1 Logs. are immutable records of events, which can be either unstructured or structured. Collected from various sources, such as applications and operating systems, they are processed by an aggregation layer, stored, and analyzed. They expose highly granular information with rich local context. This flexibility makes logs crucial to understanding why unexpected behavior occurred in a service. Despite their usefulness for auditing and issue resolution, the increase in the quantity and granularity of logs has a linear impact on the performance of the service under high demand [3].

2.2.2 Metrics. are periodically collected numbers forming a time series. Metrics are generated by aggregating events from an entity, allowing them to reveal trends in the behavior of a system. Typically, metrics are used to generate alerts. Alerts are actions triggered by events when a predefined condition is met. For instance, if a system produces numerous error responses in a short period, an alert can be generated for immediate intervention, either automatically or through manual notification. In contrast to logs, the overhead of metrics is constant, depending solely on the quantity monitored. It increases only with the addition of new monitored elements, such as services or additional metrics. Metrics and logs provide information about services individually but are insufficient for applications with multiple services distributed across an infrastructure. To fully understand the behavior of an application with distributed services and identify the information flow, it is necessary to utilize tracing [3].

2.2.3 Tracing. can reveal the causality of events in a system, showing how one event interacts with another. Understanding the cause-and-effect relationship in a trace is crucial to comprehend how an event propagates through the system. To infer the causality of events in a trace, each service must contribute by propagating a trace context alongside the usual payload. This context is then sent

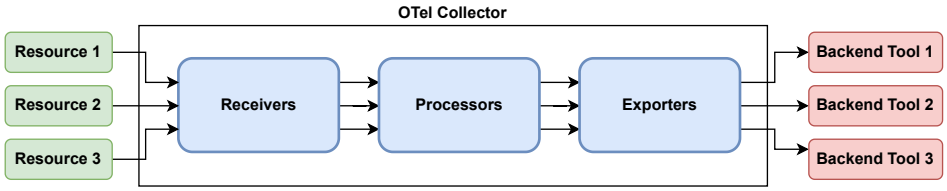

Figure 1: OTel Collector Architecture.

directly to a tool that will later be used to correlate the chain of events. This context may also contain troubleshooting data, such as stack traces and other application-specific annotations [3]. The overhead of tracing is similar to that of logs, meaning the cost of tracing increases linearly with the number of events being produced.

2.3 OpenTelemetry

OTel is an observability framework and toolkit designed to create and manage telemetry data, such as traces, metrics, and logs. OTel is vendor-agnostic, meaning it can be used with a wide variety of observability backends, such as Jaeger and Prometheus, as well as commercial offerings. OTel is a Cloud Native Computing Foundation (CNCF) project [5].

OTel achieves two crucial goals: (i) it allows you to own the generated data, rather than being tied to a proprietary data format or tool; and (ii) it enables you to learn a single set of APIs and conventions. The combination of these two aspects provides teams and organizations with the necessary flexibility in today's modern computing world. The mission of OTel is to enable effective observability by making high-quality portable telemetry ubiquitous, with consistent and high-performance instrumentation. To ensure that telemetry data is transmitted as efficiently and reliably as possible, OTel has defined OpenTelemetry protocol (OTLP). The protocol itself is defined through protobuf files. This means that any client or server interested in sending or receiving OTLP only needs to implement these definitions to support it. OTLP is the recommended protocol for transmitting telemetry data in OTel and is supported as a core component of the collector.

The *OTel Collector* is a component that provides a vendor-agnostic implementation of how to receive, process, and export telemetry data. It eliminates the need to run, operate, and maintain multiple agents/collectors. The *Collector* acts as an intermediary between the telemetry source, such as applications or nodes, and the backend that will store the data for analysis.

With the *Collector*, it's possible to decouple the origin of telemetry data from its destination. Additionally, it can provide a single destination for various types of data. It can also reduce latency when sending data to a backend. Finally, it's possible to modify telemetry data to address compliance and security issues. It's important to note that despite the benefits, additional resources are required for the execution, operation, and monitoring of the *Collector*.

The *Collector* allows users to configure pipelines for each signal separately, combining any number of receivers, processors, and exporters, as shown in Figure 1. This gives the collector a lot of flexibility on how and where it can be used. The component name that receives the data is *Receiver*. A receiver is how data arrives at the *Collector*. Receivers can support one or more data sources, receive data in various supported formats, and convert this data into an internal data format within the *Collector*. Typically, a receiver registers a listener that exposes a port on the *Collector* for the protocols it supports. Once telemetry data is received through a receiver, it can be further processed through *Processors*. It can be beneficial to perform additional tasks, such as filtering unwanted telemetry or injecting additional attributes into the data before passing it to the exporter. The order of components in the configuration is important for processors, as data is passed serially from one processor to another. The component name that provides the data is *Exporter*. An exporter is how you send data to one or more backends/destinations. Exporters can support one or more data sources. Multiple exporters of the same type can be configured for different destinations, as needed.

3 RELATED WORKS

Li *et al.* conduct a survey on microservices and observability, emphasizing the complexity of industrial microservices systems operating in intricate cloud infrastructures with dynamically created and destroyed service instances [12]. The challenge lies in understanding and diagnosing issues within this architecture, such as failed requests and high latency, given the involvement of numerous services and the diverse runtime environment. Observability is recognized as crucial in such systems. Distributed tracing, widely adopted in the industry, plays a pivotal role in achieving observability by tracking requests across services. The research highlights that microservices tracing and analysis introduce a new big data challenge for software engineering, presenting opportunities and challenges like adaptive log sampling, data fusion for tracing analysis, smarter tracing analysis, and business intelligence through tracing analysis.

Usman *et al.* conduct a comprehensive survey on distributed IT environments and microservices observability, aiming to explore challenges, requirements, best practices, and current solutions in observing distributed systems [18]. The researchers compile relevant research and articles, presenting various types of telemetry data crucial for resolving operational issues. These telemetry data (including signals like latency, traffic, errors, and saturation) are the foundation for alerts, troubleshooting, and capacity planning. The paper identifies and discusses essential functionalities for observability while addressing open research issues tied to heterogeneous infrastructures and microservices architectures across different use cases. Additionally, the study notes the influence of organizational

culture and individual mindsets on adopting new observability tools and practices.

Picoreti *et al.* claim that automated orchestration tools are based on the observability of the infrastructure itself, but this is not sufficient in some cases [15]. Certain applications have specific requirements that are challenging to meet, such as low latency, high bandwidth, and high computational power. To adequately address these requirements, orchestration must be based on multi-level observability, which means collecting data from both the application level and the infrastructure. They developed a platform with the goal of demonstrating how multi-level observability can be implemented and how it can be used to enhance automated orchestration in cloud environments, integrating Prometheus with Kubernetes. Kubernetes has native support for automated orchestration based on infrastructure metrics such as CPU and memory usage. Kubernetes has an API aggregation layer that allows its core functionality to be extended. The authors used the application metric "total request rate" for these scaling operations.

Tzanettis *et al.* address the challenge of merging data from orchestration platforms, distributed tracing, and logging tools [16]. The main objective is to reduce complexity for system administrators and software developers who must sift through a myriad of monitoring data and logs to assess the performance of distributed applications. In this perspective, the primary contribution of the work relates to the specification of a data linkage schema to support the collection and merging of data related to container resource usage, performance metrics, distributed tracing metrics, and logs. To achieve this, the proposed data schema is capable of merging and correlating data from different types of signals. The solution is based on a set of open-source tools. These tools are based on the Prometheus monitoring engine, which is supported by the Kubernetes orchestration platform, the Zipkin distributed tracing tool, the Fluentd logging software and the adoption of the Prometheus Python instrumentation library for the definition of exemplars in the source code.

In summary, [12] and [18] theoretically addressed observability as a whole, discussing the benefits and challenges for adoption, focusing solely on microservices. [15] used observability to present alternatives in managing orchestration platform scaling, and [16] addresses the data fusion challenge, but none of the studies evaluated the impact on the cost of storing the generated telemetry data. Our study applies these concepts through OTel and focus on evaluate the cost of storing the telemetry generated to perform the observability of microservices-based application.

4 PROPOSAL

The objective of this work is to evaluate the cost of storing telemetry data generated by OTel to perform the observability of a microservices-based benchmarking application. This section presents the application and how it was integrated with OTel in this solution. In addition, the scripts created for both the deployment of the solution on AWS and the configuration of OTel Collector to handle telemetry are presented. The repository used is the following: https://github.com/viniciusbrg/TeaStore-Microservices/tree/otel.

4.1 TeaStore Overview

TeaStore [19] is a microservice reference and test application to be used in benchmarks and tests. The TeaStore[1] emulates a basic web store for automatically generated tea and tea supplies. As it is primarily a test application, it features User Interface (UI) elements for database generation and service resetting in addition to the store itself. It is composed of five microservices and a registry service that communicate with each other using the REST protocol.

The application comprises several key services. The **Registry** is a custom implementation of a simplified version of Netflix Eureka, maintaining a repository of all service instance locations. All other services rely on Netflix Ribbon for client-side load balancing. The **WebUI** service, based on Servlets, serves as the user entry point, making API REST calls to other services and displaying results in the user interface. The **Auth** handles user authentication, while the **Image-Provider** service manages product images, including storage and resizing, with the option to use caches for optimization. The **Recommender** service offers product recommendations based on user interactions, and the **Persistence** service provides access to the product database, allowing for database size adjustments to impact application performance.

4.2 OTel Integration

In the next step of the study, we configured and integrated OTel tools with the TeaStore. In Figure 2, we provide an overview of the proposal's solution architecture, which operates in a cluster environment with multiple machines. The cluster is managed by Kubernetes, that is an open-source container orchestration system, ensuring the application scales as needed. The application employed here is TeaStore, and we utilize the *OTel Collector*. Observability backend tools are set up for telemetry display and analysis.

OTel Collector is configured to both receive and export environmental telemetry. In total, ten components were configured, five receivers and five exporters. The following components are introduced: **OTLP Receiver**, which collects application data via gRPC or HTTP in OTLP format; **MySQL Receiver**, responsible for retrieving metrics from the database by querying MySQL's global status and InnoDB tables; **Host Metrics Receiver**, which generates a wide range of host system metrics, including CPU utilization, Disk I/O, CPU load, File System utilization, Memory utilization, Network interface I/O, TCP connection metrics, Paging/Swap space utilization, Process count, and per-process CPU, Memory, and Disk I/O metrics; **Kubelet Stats Receiver**, which pulls node, pod, container, and volume metrics from the kubelet's API server; **Filelog Receiver**, used for collecting logs from files, primarily for Loki; **OTLP Exporter**, for exporting data in OTLP format via gRPC. Jaeger[2] is a distributed tracing system was created by Uber and provides Adaptive Sampling. Jaeger has native support for OTLP; **Zipkin Exporter**, responsible for sending data to a Zipkin[3] back-end, that is another distributed tracing system; **Prometheus Exporter**, which exports data in Prometheus format for scraping by a Prometheus[4]

[1] https://github.com/DescartesResearch/TeaStore
[2] https://github.com/jaegertracing/jaeger
[3] https://github.com/openzipkin/zipkin
[4] https://github.com/prometheus/prometheus

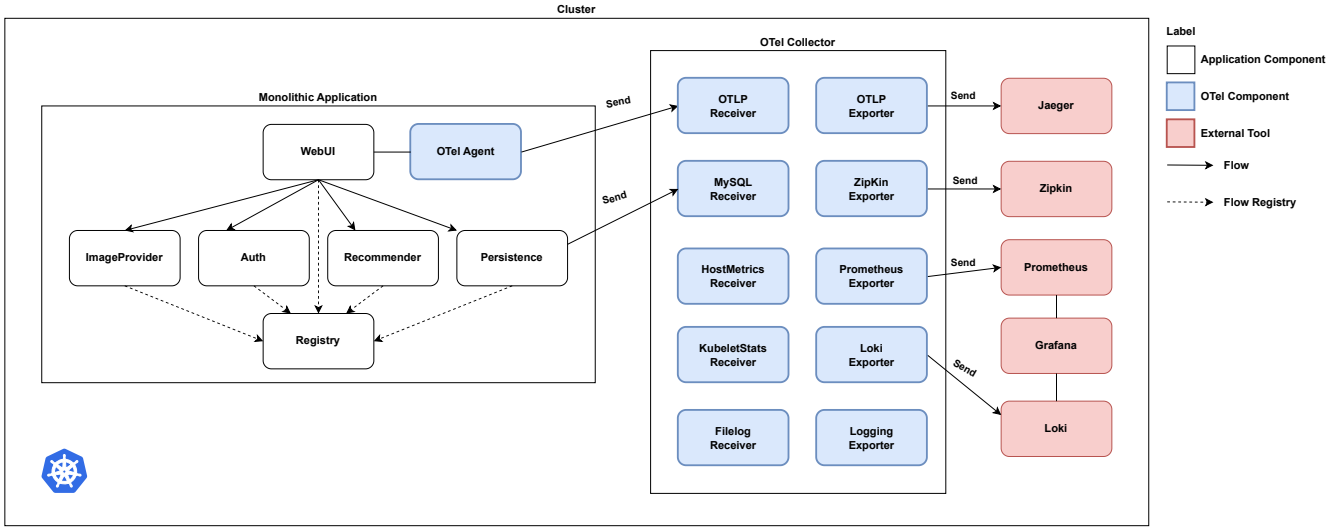

Figure 2: Proposal Architecture.

server. Prometheus is an open-source monitoring system; **Loki Exporter**, exporting data via HTTP to Loki[5], that is a log aggregation system designed to store and query logs from applications and infrastructure. Loki using Grafana[6], that is a platform for visualizing and analyzing metrics through graphs; and **Logging Exporter**, which sends data to the console via zap.Logger and supports traces, metrics, and logs in the pipeline.

To have a general idea of the application's performance, it is necessary to configure receivers to capture observability telemetry as a whole. Thus, we set up receivers that capture telemetry from the application, host infrastructure, Kubernetes cluster, and the database. As mentioned earlier (Section 2.2.2), if there is an addition of new monitored elements, more data is generated, requiring an assessment of this cost for the user.

4.3 CloudFormation Scripts to Facilitate Reproducibility

In addition to the application integration with OTel for enabling modern technology for observability, we also leveraged an infrastructure as code tool, CloudFormation, to create scripts to facilitate the implementation, experimentation, visualization, and analysis of telemetry data collected in the microservices of the TeaStore application.

Our goal is to fill the gap of having an environment that facilitates comparison between these types of architecture, as well as having all required tools already configured to collect, visualize, and analyze the data. This solution will facilitate the reproducibility of works that desire to use any version of the TeaStore application and OTel. Besides that, researchers may easily deploy the environment in the AWS Cloud since the CloudFormation service provides and configures those resources. The CloudFormation configuration can be seen at the link: https://github.com/viniciusbrg/TeaStore-Microservices/blob/otel/cloudformation.yml. The Listing 1 has both the CloudFormation configuration call and the parameters that are passed in the deployment. Note that it is possible to define the number and type of instances of both the application with the integration (TeastoreInstances), as well as the load test (LocustInstances).

Listing 1: CloudFormation Configuration

```
1  aws cloudformation create-stack --stack-name teastorestack
2  --template-body file://cloudformation.yml \
3  --parameters ParameterKey=ClustersSubnetIds,
4  ParameterValue=$SUBNET_IDS \
5  ParameterKey=TeastoreInstanceType,
6  ParameterValue=$TEASTORE_INSTANCE_TYPE \
7  ParameterKey=TeastoreInstances,
8  ParameterValue=$TEASTORE_INSTANCES \
9  ParameterKey=LocustInstances,
10 ParameterValue=$LOCUST_INSTANCES \
11 ParameterKey=LocustInstanceType,
12 ParameterValue=$LOCUST_INSTANCE_TYPE \
13 ParameterKey=ClustersARN,
14 ParameterValue=$ARN
```

4.4 OTel Config

To set up the collector, it is necessary to create a file that indicates what will be collected, processed, and where it will be exported. The Listing 2 below presents the code used in this solution.

From lines 1 to 40, the Receivers are defined. Note that all the components we presented earlier have been configured. If the user does not want some of the metrics, they can simply remove them. The collection_interval parameter is used to specify the interval for data collection. This allows configuring the time and adjusting the amount of data to be collected based on the application context. Prometheus is configured to perform this sampling. From lines 42 to 57, the processors used are configured. With them, it is possible to filter what the user wants or does not want to receive as telemetry, among other operations. From lines 59 to 75, the exporters are configured, specifying the backend tools that should receive the data.

[5]https://github.com/grafana/loki
[6]https://github.com/grafana/grafana

Finally, all the components that users want to use are configured in the services (from line 77 onward). Users can configure what they want to receive, process, and export, such as metrics, logs, and traces. Similarly, if the user does not need to receive traces, they simply omit the exporter in the service configuration.

Listing 2: Example of OTel Collector Configuration

```
receivers:
  kubeletstats:
    collection_interval: 5s
    auth_type: "serviceAccount"
    endpoint: "${env:K8S_NODE_NAME}:10250"
    insecure_skip_verify: true

  hostmetrics:
    root_path: /hostfs
    collection_interval: 5s
    scrapers:
      cpu:
      memory:
      load:
      disk:
      filesystem:
      memory:
      network:
      paging:

  mysql:
    endpoint: teastore-db:3306
    username: teauser
    password: teapassword
    database: teadb
    collection_interval: 5s

  otlp:
    protocols:
      grpc:
      http:

  filelog:
    include:
    - /var/log/pods/*/*/*.log
    exclude:
    - /var/log/pods/*/otel-collector/*.log
    start_at: beginning
    include_file_path: true
    include_file_name: false

processors:
  filter:
    metrics:
      exclude:
        match_type: strict
        metric_names:
          - queueSize

  attributes:
    actions:
      - action: insert
        key: log_file_name
        from_attribute: log.file.name
      - action: insert
        key: loki.attribute.labels
        value: log_file_name

exporters:
  logging:
    verbosity: normal

  zipkin:
    endpoint: http://zipkin:9411/api/v2/spans

  loki:
    endpoint: http://loki:3100/loki/api/v1/push

  prometheus:
    endpoint: "0.0.0.0:8889"

  otlp:
    endpoint: "jaeger:4317"
    tls:
      insecure: true

service:
  pipelines:
    metrics:
      receivers: [otlp, mysql, kubeletstats, hostmetrics]
      exporters: [prometheus]
    traces:
      receivers: [otlp]
      exporters: [zipkin, otlp]
    logs:
      receivers: [otlp, filelog]
      processors: [attributes]
      exporters: [logging, loki]
```

4.5 Efficient Sampling and Integrate with Tracing

Sampling is an important aspect of metric collection. It involves selecting a subset of data from a larger data set for analysis. Efficient sampling can help you reduce the data volume and computational overhead associated with metric collection, without sacrificing the quality of your insights. In the context of OTel, efficient sampling might involve using probabilistic sampling methods, which are designed to provide a representative sample of a data set with minimal computational overhead. In our solution, we use this approach to collect traces, with a probability of 0.5. Thus, half of the samples can be collected.

Integrating metrics, logging, and tracing provides a comprehensive perspective on your system's behavior and performance. For example, metrics can help identify a latency spike, while traces can pinpoint the specific operations causing the delay.

Our solution supports log storage (as mentioned earlier with Loki); however, we choose not to store logs using this application because it doesn't generate logs about user actions. Therefore, our focus is solely on metrics and traces.

5 EXPERIMENTS AND RESULTS

We planned and conducted experiments to address the research question. Thus, the experiment's goal was to assess the storage cost of telemetry provided by OTel from monitoring the TeaStore application and gain insights into the overhead of employing different strategies for collecting metrics and traces.

5.1 Experiments Setup

We used the CloudFormation scripts to create the application execution environment, deploying a Kubernetes cluster in the AWS Cloud through EKS. The application was run with OTel and the backend tools residing in the same cluster as the application. In relation to the replicas, we used 4 WebUI and 4 Persistence, as they are the components that receive the most requests.

We deployed two distributed tracing environments (Jaeger and Zipkin), providing system administrators the flexibility to analyze data using the platform they are most familiar with. For the analysis of metrics received by the configured receivers, Prometheus

was set up. Additionally, Loki was configured for logs, with results displayed in Grafana. Grafana, capable of receiving data from Prometheus, facilitates the creation of visualizations. The Logging exporter was configured to receive all telemetry signals. Therefore, users analyzing this data have various tool options for each signal and can select the one that aligns with their preferences.

In this experiment, sampling was defined as: Fast and Slow. "Fast" sampling involves collecting data every 5 seconds, while "Slow" sampling uses a 1-minute interval. The quantity of metrics to be collected was also defined as: All and Few. In the "All", it includes all metrics collected by the receivers. In the "Few", only CPU/memory metrics are used, providing a basic understanding of the application's performance. These configurations were applied to the metrics. As for traces, their collection was fixed, considering whether or not to use them for telemetry storage. In total, 8 scenarios were defined for execution

- *Scenario 1* consists of fast sampling and all metrics.
- *Scenario 2* consists of fast sampling and few metrics.
- *Scenario 3* consists of slow sampling and all metrics.
- *Scenario 4* consists of slow sampling and few metrics.
- *Scenario 5* consists of fast sampling and all metrics with traces.
- *Scenario 6* consists of fast sampling and few metrics with traces.
- *Scenario 7* consists of slow sampling and all metrics with traces.
- *Scenario 8* consists of slow sampling and few metrics with traces.

We leveraged Locust[7] to generate the workload. With such a tool, it is possible to document the execution history and create a coherent flow. The following operations were used to simulate user behavior: visit the landing page; User login with random userid between 1 and 90; simulate random browsing behavior; simulates to buy products in the cart with sample user data; visits user profile; and user logout. The execution of these operations is carried out by varying the number of users, using ramp-up and down, to simulate a real application usage scenario. Empirically, the number of users for the test was chosen and defined as varying from 100 users to 200 users simultaneously. We used the same workload for one hour in all scenarios.

5.2 Results and Discussion

From the results and studies with OTel, it is evident that it covers all three aspects of monitoring software: logs, metrics, and distributed tracing. All of this is not tied to providers. Development teams do not need to manually create monitoring solutions if they want a provider-agnostic solution. Furthermore, there is no need to use the instrumentation libraries of the solution's provider to meet client specifications, reducing the effort and cost of building software. Being an open standard, OTel encourages broad collaboration and promotes better coverage, flexibility, and ubiquity as engineers worldwide contribute to instrumentation. Another advantage of OTel is interoperability, meaning that various systems can use this standard form of instrumentation across all services to exchange information.

[7]https://locust.io/

Additionally, the flexibility to change backend observability tools is another advantage. Most proprietary monitoring solutions require the installation of specific instrumentation agents to gather telemetry data. If the user decides to switch to a different tool, he/she will need to spend more time and resources installing different proprietary agents or reinstrumenting the code. With OTel, users can change the backend tool without changing the instrumentation. Also, if emerging technologies are created with OTel specifications, they can automatically integrate with the solution. In summary, OTel offers several significant benefits to organizations looking to improve their application observability capabilities. It provides flexibility, scalability, and the ability to provide detailed and comprehensive performance data for an application.

Despite these benefits, the study managed to analyze the impact on the storage cost of telemetry data generated by application observability through OTel. As shown in Table 1, the required storage to run this experiment for one hour is presented in the first column (Storage in MB). The second column (Cost in USD) corresponds to the cost of storing this data using the AWS S3 (Simple Storage Service) during one month. The cost used was 0.023 USD per GB for a month (S3 Standard - general-purpose storage for any type of data, typically used for frequently accessed data) [source](https://aws.amazon.com/s3/pricing/). This provides an idea of how much a user would need to spend to have this data for future analysis of application states.

The storage spent on traces is fixed, collecting around 160 MB of distributed tracing in this collection. As mentioned earlier, scenarios 1 to 4 are without traces, so they require less data and consequently have a lower storage service cost. Scenarios 5 to 8 include traces and therefore have the highest storage service cost, adding the size of traces to the metrics collected in the first scenarios. Scenario 5 has the greatest impact on storage cost (2.875359375 USD), as expected, as it has fast sampling and all metrics, including all traces. Scenario 4 has the lowest impact (0.0005175 USD), as expected, as it has slow sampling and few metrics and no traces. These results address the research question, demonstrating the impact of storage cost in various observability scenarios.

The configuration of sampling is a task that requires an assessment by the application observability operator, as well as choices about which metrics to collect. The faster the sampling and the more metrics collected, the more disk space is needed. Conversely, with lower sampling and fewer metrics collected, less space is required, but costs increase proportionally with storage. Additionally, concerning application observability, having fewer telemetry data means less knowledge about the application's state, making it less observable and reducing the operator's insight into the application's condition. This, in turn, can impact decision-making regarding actions such as increasing CPU/memory, scaling the most overloaded microservice, adding more nodes to the cluster, adjusting the database, among others. Furthermore, relying solely on metrics without integration with traces compromises observability, given the lack of correlation between them. For example, it is impossible to determine why the CPU metric is increasing without information about the requests that users are making in the application.

The data generated by OTel has been placed into a dataset for future analysis by the community and use in other researches. The dataset includes metrics collected from the application, database,

	Storage in MB	Cost in USD
Scenario 1	17,8	0,287859375
Scenario 2	0,15	0,00242578125
Scenario 3	3,8	0,061453125
Scenario 4	0,032	0,0005175
Scenario 5	177	2,875359375
Scenario 6	163	2,589925781
Scenario 7	164	2,648953125
Scenario 8	160,032	2,5880175

Table 1: Storage and Cost impact

host (CPU, memory, disk, network), and the Kubernetes cluster (nodes, pods, volumes). Traces collected during the experiment are also provided in the dataset. Researchers and practitioners can leverage this dataset to conduct various studies. For instance, they can delve into performance analyses. Moreover, the dataset provides an interesting resource for researchers to formulate and assess models, such as Petri nets, and conduct simulations, enabling a deeper understanding of the system's dynamics and behavior under varying conditions. The dataset can be downloaded in the following link: https://tinyurl.com/advance2024.

6 CONCLUSION AND FUTURE WORKS

This work discussed the process of integrating a microservices-based benchmark application with OTel. It was presented how the deployment was carried out on AWS and how the OTel Collector was configured to perform telemetry of application metrics, logs, and traces. The experiments evaluated the cost of storing telemetry data in AWS in various scenarios. The results showed that the more the user needs the data, obviously, the cost becomes higher. Therefore, it is concluded that the use of OTel brings benefits, but users need to be careful with the issue of storing application metrics and traces. As future work, we plan to use machine learning algorithms on the data collected through OTel to suggest improvements in the application code, identify bottlenecks, and create components for OTel with customized metrics for a better understanding of the application.

REFERENCES

[1] Ahmad Alkhatib, Abeer Al Sabbagh, and Randa Maraqa. 2021. Pubic Cloud Computing: Big Three Vendors. In *2021 International Conference on Information Technology (ICIT)*. IEEE, 230–237.

[2] Saša Baškarada, Vivian Nguyen, and Andy Koronios. 2018. Architecting microservices: Practical opportunities and challenges. *Journal of Computer Information Systems* (2018).

[3] Andre Bento, Jaime Correia, Ricardo Filipe, Filipe Araujo, and Jorge Cardoso. 2021. Automated analysis of distributed tracing: Challenges and research directions. *Journal of Grid Computing* 19 (2021), 1–15.

[4] Matthias Biehl. 2016. *RESTful Api Design*. Vol. 3. API-University Press.

[5] A. Boten and C. Majors. 2022. *Cloud-Native Observability with OpenTelemetry: Learn to gain visibility into systems by combining tracing, metrics, and logging with OpenTelemetry*. Packt Publishing. https://books.google.com.br/books?id=YZVsEAAAQBAJ

[6] Brendan Burns, Joe Beda, Kelsey Hightower, and Lachlan Evenson. 2022. *Kubernetes: up and running*. " O'Reilly Media, Inc.".

[7] Breno Costa, Joao Bachiega Jr, Leonardo Rebouças Carvalho, Michel Rosa, and Aleteia Araujo. 2022. Monitoring fog computing: A review, taxonomy and open challenges. *Computer Networks* 215 (2022), 109189.

[8] Paolo Di Francesco. 2017. Architecting microservices. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, 224–229.

[9] Martin Fowler and James Lewis. 2014. Microservices, 2014. *URL: http://martinfowler. com/articles/microservices. html* 1, 1 (2014), 1–1.

[10] Suman Karumuri, Franco Solleza, Stan Zdonik, and Nesime Tatbul. 2021. Towards observability data management at scale. *ACM SIGMOD Record* 49, 4 (2021), 18–23.

[11] Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles. 2019. A survey of DevOps concepts and challenges. *ACM Computing Surveys (CSUR)* 52, 6 (2019), 1–35.

[12] Bowen Li, Xin Peng, Qilin Xiang, Hanzhang Wang, Tao Xie, Jun Sun, and Xuanzhe Liu. 2022. Enjoy your observability: an industrial survey of microservice tracing and analysis. *Empirical Software Engineering* 27 (2022), 1–28.

[13] Nicolas Marie-Magdelaine, Toufik Ahmed, and Gauthier Astruc-Amato. 2019. Demonstration of an observability framework for cloud native microservices. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IEEE, 722–724.

[14] Claus Pahl and Pooyan Jamshidi. 2016. Microservices: A Systematic Mapping Study. *CLOSER (1)* (2016), 137–146.

[15] Rodolfo Picoreti, Alexandre Pereira do Carmo, Felippe Mendonca de Queiroz, Anilton Salles Garcia, Raquel Frizera Vassallo, and Dimitra Simeonidou. 2018. Multilevel observability in cloud orchestration. In *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*. IEEE, 776–784.

[16] Ioannis Tzanettis, Christina-Maria Androna, Anastasios Zafeiropoulos, Eleni Fotopoulou, and Symeon Papavassiliou. 2022. Data Fusion of Observability Signals for Assisting Orchestration of Distributed Applications. *Sensors* 22, 5 (2022), 2061.

[17] Muhammad Usman, Simone Ferlin, Anna Brunstrom, and Javid Taheri. 2022. A survey on observability of distributed edge & container-based microservices. *IEEE Access* (2022).

[18] Muhammad Usman, Simone Ferlin, Anna Brunstrom, and Javid Taheri. 2022. A Survey on Observability of Distributed Edge & Container-Based Microservices. *IEEE Access* 10 (2022), 86904–86919. https://doi.org/10.1109/ACCESS.2022.3193102

[19] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. 2018. TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research. In *Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems* (Milwaukee, WI, USA) *(MASCOTS '18)*.